⟨*Version*⟩≡

```
2.4.1
```

⟨*Date*⟩≡

```
Mar 24 2017
```

WHIZARD[1]

Wolfgang Kilian,[2] Thorsten Ohl,[3] Jürgen Reuter[4]

Version 2.4.1, Mar 24 2017
with contributions from: Fabian Bach, Felix Braam, Hans-Werner Boschmann, Bijan Chokoufé Nejad, Christian Fleper, David Gordo Gomez, Sebastian Schmidt, Marco Sekulla, So Young Shim, Christian Speckner, Christian Weiss, Daniel Wiesler, Zhijie Zhao

[1]The original meaning of the acronym is *W, Higgs, Z, And Respective Decays.* The current program is much more than that, however.

[2]e-mail: `kilian@physik.uni-siegen.de`
[3]e-mail: `ohl@physik.uni-wuerzburg.de`
[4]e-mail: `juergen.reuter@desy.de`

Abstract

`WHIZARD` is an application of the `VAMP` algorithm: Adaptive multi-channel integration and event generation. The bare `VAMP` library is augmented by modules for Lorentz algebra, particles, phase space, etc., such that physical processes with arbitrary complex final states [well, in principle...] can be integrated and *unweighted* events be generated.

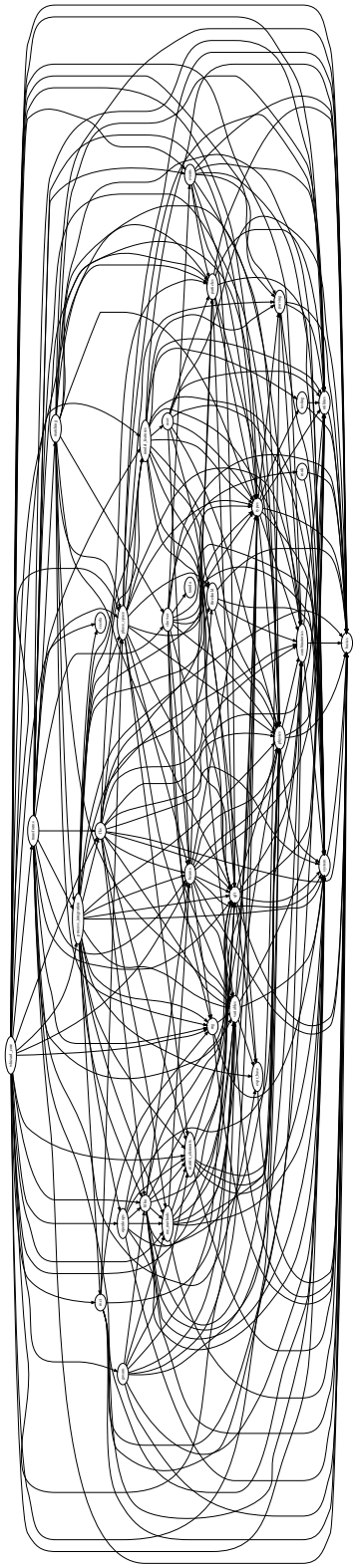

Figure 1: Overall folder structure

Contents

1 Changes **26**

2 Preliminaries **27**

3 Utilities **29**
3.1 File Utilities . . . 29
3.1.1 Deleting a file . . . 30
3.2 File Registries . . . 30
3.2.1 File handle . . . 31
3.2.2 File handles registry . . . 33
3.3 String Utilities . . . 35
3.3.1 Upper and Lower Case . . . 36
3.3.2 C-compatible Output . . . 37
3.3.3 Number Conversion . . . 37
3.3.4 String splitting . . . 39
3.4 Format Utilities . . . 40
3.4.1 Line Output . . . 40
3.4.2 Array Output . . . 41
3.4.3 TEX-compatible Output . . . 42
3.4.4 Metapost-compatible Output . . . 43
3.4.5 Conditional Formatting . . . 43
3.4.6 Compressed output of integer arrays . . . 44
3.5 Format Definitions . . . 45
3.6 Numeric Utilities . . . 45

4 Testing **52**
4.1 Unit tests . . . 52
4.1.1 Parameters . . . 53
4.1.2 Type for storing test results . . . 53
4.1.3 Wrapup . . . 55
4.1.4 Tool for Unit Tests . . . 56

5 System: Interfaces and Handlers **58**
5.1 Constants . . . 58
5.1.1 Version . . . 59
5.1.2 Text Buffer . . . 59
5.1.3 IOSTAT Codes . . . 59
5.1.4 Character Codes . . . 59

5.2 C wrapper for sigaction . . . 60
5.3 C wrapper for printf . . . 62
5.4 Error, Message and Signal Handling . . . 63
5.4.1 Logfile . . . 80
5.4.2 Checking values . . . 82
5.4.3 Suppression of numerical noise . . . 85
5.4.4 Signal handling . . . 85
5.5 Operating-system interface . . . 88
5.5.1 Path variables . . . 89
5.5.2 System dependencies . . . 90
5.5.3 Dynamic linking . . . 97
5.5.4 Predicates . . . 100
5.5.5 Shell access . . . 100
5.5.6 Querying for a directory . . . 101
5.5.7 Fortran compiler and linker . . . 102
5.5.8 Controlling OpenMP . . . 105
5.5.9 Unit tests . . . 106
5.6 Interface for formatted I/O . . . 109
5.6.1 Parsing a C format string . . . 110
5.6.2 API . . . 117
5.6.3 Unit tests . . . 117
5.7 CPU timing . . . 119
5.7.1 Timer . . . 124
5.7.2 Unit tests . . . 126

6 Combinatorics 132
6.1 Bytes and such . . . 132
6.1.1 8-bit words: bytes . . . 134
6.1.2 32-bit words . . . 135
6.1.3 Operations on 32-bit words . . . 138
6.1.4 64-bit words . . . 141
6.2 Hashtables . . . 143
6.2.1 The hash function . . . 143
6.2.2 The hash table . . . 144
6.2.3 Hashtable insertion . . . 146
6.2.4 Hashtable lookup . . . 147
6.3 MD5 Checksums . . . 148
6.3.1 Blocks . . . 149
6.3.2 Messages . . . 151
6.3.3 Message I/O . . . 153
6.3.4 Auxiliary functions . . . 155
6.3.5 Auxiliary stuff . . . 155
6.3.6 MD5 algorithm . . . 156
6.3.7 User interface . . . 158
6.3.8 Unit tests . . . 159
6.4 Permutations . . . 161
6.4.1 Permutations . . . 162
6.4.2 Operations on binary codes . . . 167
6.5 Sorting . . . 168
6.5.1 Implementation . . . 168

6.5.2 Concatenating arrays . . . 172
6.5.3 Unit tests . . . 172
6.6 Grids . . . 175
6.6.1 Initializer and finalizer . . . 177
6.6.2 Segment finding and memory mapping . . . 178
6.6.3 Grid manipulations . . . 179
6.6.4 Input and Output to screen and disk . . . 180
6.6.5 Unit tests . . . 183
6.6.6 Unit tests . . . 192

7 Text handling 195
7.1 Internal files . . . 197
7.1.1 The line type . . . 197
7.1.2 The ifile type . . . 198
7.1.3 I/O on ifiles . . . 199
7.1.4 Ifile tools . . . 202
7.1.5 Line pointers . . . 202
7.1.6 Access lines via pointers . . . 203
7.2 Lexer . . . 205
7.2.1 Input streams . . . 206
7.2.2 Keyword list . . . 210
7.2.3 Lexeme templates . . . 212
7.2.4 The lexer setup . . . 216
7.2.5 The lexeme type . . . 219
7.2.6 The lexer object . . . 222
7.2.7 The lexer routine . . . 223
7.2.8 Diagnostics . . . 226
7.2.9 Unit tests . . . 227
7.3 Syntax rules . . . 230
7.3.1 Syntax rules . . . 230
7.3.2 I/O . . . 233
7.3.3 Completing syntax rules . . . 235
7.3.4 Accessing rules . . . 237
7.3.5 Syntax tables . . . 239
7.3.6 Accessing the syntax table . . . 245
7.3.7 I/O . . . 246
7.4 The parser . . . 248
7.4.1 The token type . . . 248
7.4.2 Retrieve token contents . . . 253
7.4.3 The parse tree: nodes . . . 255
7.4.4 Filling nodes . . . 257
7.4.5 Accessing nodes . . . 261
7.4.6 The parse tree . . . 265
7.4.7 Access the parser . . . 271
7.4.8 Tools . . . 272
7.4.9 Applications . . . 273
7.4.10 Unit tests . . . 274
7.5 XML Parser . . . 277
7.5.1 Cached Stream . . . 277
7.5.2 Attributes . . . 278

7.5.3 The Tag Type . . . 280
7.5.4 Unit tests . . . 287
7.5.5 Auxiliary Routines . . . 288
7.5.6 Basic Tag I/O . . . 288
7.5.7 Optional Tag . . . 291
7.5.8 Optional Tag . . . 293
7.5.9 Basic Tag I/O . . . 295

8 Random-Number Generator 297
8.1 Generic Random-Number Generator . . . 299
8.1.1 Generator type . . . 299
8.1.2 RNG Factory . . . 301
8.1.3 Unit tests . . . 302
8.2 Select from a weighted sample . . . 309
8.2.1 Selector type . . . 309
8.2.2 Unit tests . . . 312
8.3 TAO Random-Number Generator . . . 314
8.3.1 Generator type . . . 314
8.3.2 Unit tests . . . 317
8.4 RNG Stream . . . 321
8.4.1 Factory . . . 327
8.4.2 Unit tests . . . 330
8.5 Dispatch . . . 335
8.5.1 Unit tests . . . 337

9 Physics 341
9.1 Physics Constants . . . 343
9.1.1 Units . . . 343
9.1.2 SM and QCD constants . . . 343
9.1.3 Parameter Reference values . . . 344
9.1.4 Particle codes . . . 344
9.1.5 Spin codes . . . 346
9.1.6 NLO status codes . . . 346
9.1.7 Threshold . . . 348
9.2 C-compatible Particle Type . . . 349
9.3 Lorentz algebra . . . 351
9.3.1 Three-vectors . . . 351
9.3.2 Four-vectors . . . 357
9.3.3 Conversions . . . 363
9.3.4 Angles . . . 369
9.3.5 More kinematical functions (some redundant) . . . 375
9.3.6 Lorentz transformations . . . 379
9.3.7 Functions of Lorentz transformations . . . 381
9.3.8 Invariants . . . 381
9.3.9 Boosts . . . 387
9.3.10 Rotations . . . 389
9.3.11 Composite Lorentz transformations . . . 391
9.3.12 Applying Lorentz transformations . . . 392
9.3.13 Special Lorentz transformations . . . 393
9.3.14 Special functions . . . 395

9.4 Special Physics functions . . . 400
9.4.1 Running α_s . . . 400
9.4.2 Catani-Seymour Parameters . . . 403
9.4.3 Mathematical Functions . . . 404
9.4.4 Loop Integrals . . . 405
9.4.5 More on α_s . . . 407
9.4.6 Functions for Catani-Seymour dipoles . . . 408
9.4.7 Distributions for integrated dipoles and such . . . 410
9.4.8 Splitting Functions . . . 419
9.4.9 Top width . . . 420
9.4.10 Unit tests . . . 425
9.5 QCD Coupling . . . 429
9.5.1 Coupling: Abstract Data Type . . . 429
9.5.2 Fixed Coupling . . . 430
9.5.3 Running Coupling . . . 431
9.5.4 Running Coupling, determined by Λ_{QCD} . . . 432
9.5.5 Wrapper type . . . 433
9.5.6 Unit tests . . . 434
9.6 Shower algorithms . . . 438
9.6.1 Unit tests . . . 439

10 Quantum Field Theory Concepts 441
10.1 Model Data . . . 443
10.1.1 Physics Parameters . . . 443
10.1.2 Field Data . . . 447
10.1.3 Vertex data . . . 463
10.1.4 Vertex lookup table . . . 466
10.1.5 Model Data Record . . . 470
10.1.6 Toy Models . . . 486
10.2 Model Testbed . . . 492
10.2.1 Abstract Model Handlers . . . 492
10.3 Helicities . . . 494
10.3.1 Helicity types . . . 494
10.3.2 Predicates . . . 497
10.3.3 Accessing contents . . . 498
10.3.4 Comparisons . . . 498
10.3.5 Tools . . . 499
10.4 Colors . . . 500
10.4.1 The color type . . . 500
10.4.2 Predicates . . . 506
10.4.3 Accessing contents . . . 507
10.4.4 Comparisons . . . 509
10.4.5 Tools . . . 510
10.4.6 Unit tests . . . 519
10.4.7 The Madgraph color model . . . 522
10.5 Flavors: Particle properties . . . 525
10.5.1 The flavor type . . . 525
10.6 Quantum numbers . . . 541
10.6.1 The quantum number type . . . 541
10.6.2 I/O . . . 544

10.6.3 Accessing contents 546
10.6.4 Predicates 548
10.6.5 Comparisons 549
10.6.6 Operations 551
10.6.7 The quantum number mask 555
10.6.8 Setting mask components 557
10.6.9 Mask predicates 558
10.6.10 Operators 559
10.6.11 Mask comparisons 559
10.6.12 Apply a mask 560

11 Transition Matrices and Evaluation 562
11.1 State matrices 563
11.1.1 Nodes of the quantum state trie 563
11.1.2 State matrix 568
11.1.3 State iterators 580
11.1.4 Operations on quantum states 588
11.1.5 Factorization 595
11.1.6 Unit tests 602
11.2 Interactions 613
11.2.1 External interaction links 614
11.2.2 Internal relations 615
11.2.3 The interaction type 617
11.2.4 Methods inherited from the state matrix member 624
11.2.5 Accessing contents 633
11.2.6 Modifying contents 639
11.2.7 Handling Linked interactions 641
11.2.8 Recovering connections 647
11.2.9 Unit tests 649
11.3 Matrix element evaluation 652
11.3.1 Array of pairings 653
11.3.2 The evaluator type 653
11.3.3 Auxiliary structures for evaluator creation 656
11.3.4 Creating an evaluator: Matrix multiplication 664
11.3.5 Creating an evaluator: square 675
11.3.6 Creating an evaluator: identity 691
11.3.7 Creating an evaluator: quantum number sum 692
11.3.8 Evaluation 694
11.3.9 Unit tests 695

12 Sindarin Built-In Types 706
12.1 Particle Specifiers 706
12.1.1 Base type 707
12.1.2 Wrapper type 708
12.1.3 The atomic type 710
12.1.4 List 716
12.1.5 Sum 718
12.1.6 Expression Expansion 720
12.1.7 Unit Tests 721
12.2 PDG arrays 726

12.2.1 Type definition . . . 727
12.2.2 Basic operations . . . 728
12.2.3 Matching . . . 731
12.2.4 Sorting . . . 737
12.2.5 PDG array list . . . 739
12.2.6 Unit tests . . . 747
12.3 Jets . . . 756
12.3.1 Re-exported symbols . . . 756
12.3.2 Unit tests . . . 757
12.4 Subevents . . . 760
12.4.1 Particles . . . 760
12.4.2 subevents . . . 768
12.4.3 Eliminate numerical noise . . . 783
12.5 Analysis tools . . . 784
12.5.1 Output formats . . . 784
12.5.2 Graph options . . . 785
12.5.3 Drawing options . . . 790
12.5.4 Observables . . . 794
12.5.5 Output . . . 798
12.5.6 Histograms . . . 800
12.5.7 Plots . . . 810
12.5.8 Graphs . . . 816
12.5.9 Analysis objects . . . 822
12.5.10 Analysis object iterator . . . 828
12.5.11 Analysis store . . . 831
12.5.12 LaTeX driver file . . . 834
12.5.13 API . . . 835
12.5.14 Unit tests . . . 844

13 Matrix Element Handling 848
13.1 Process data block . . . 851
13.2 Process library interface . . . 857
13.2.1 Overview . . . 857
13.2.2 Workflow . . . 857
13.2.3 The module . . . 858
13.2.4 Writers . . . 858
13.2.5 Process records in the library driver . . . 863
13.2.6 The process library driver object . . . 865
13.2.7 Write makefile . . . 870
13.2.8 Write driver file . . . 872
13.2.9 Interface bodies for informational functions . . . 875
13.2.10 Interfaces for C-library matrix element . . . 890
13.2.11 Retrieving the tables . . . 892
13.2.12 Returning a procedure pointer . . . 894
13.2.13 Hooks . . . 895
13.2.14 Make source, compile, link . . . 896
13.2.15 Clean up generated files . . . 897
13.2.16 Further Tools . . . 899
13.2.17 Load the library . . . 899
13.2.18 MD5 sums . . . 902

13.2.19 Unit Test 904
13.3 Abstract process core configuration 933
13.3.1 Process core definition type 933
13.3.2 Process core template 936
13.3.3 Process driver 937
13.3.4 Process driver for intrinsic process 937
13.3.5 Process driver for user-defined processes 938
13.4 Process library access 938
13.4.1 Auxiliary stuff 939
13.4.2 Process definition objects 939
13.4.3 Process library 966
13.4.4 Use the library 982
13.4.5 Collect model-specific libraries 986
13.4.6 Unit Test 988
13.5 Process Library Stacks 1010
13.5.1 The stack entry type 1011
13.5.2 The prclib stack type 1011
13.5.3 Operating on Stacks 1012
13.5.4 Accessing Contents 1013
13.5.5 Unit tests 1014
13.6 Trivial matrix element for tests 1016
13.6.1 Process definition 1017
13.6.2 Driver 1019
13.6.3 Shortcut 1021
13.6.4 Unit Test 1022

14 Particles **1031**
14.1 $su(N)$ Algebra 1033
14.1.1 $su(N)$ fundamental representation 1033
14.1.2 Mapping between helicity and matrix index 1034
14.1.3 Generator Basis: Cartan Generators 1035
14.1.4 Roots (Off-Diagonal Generators) 1039
14.1.5 Unit tests 1041
14.2 Bloch Representation 1049
14.2.1 Preliminaries 1050
14.2.2 The basic polarization type 1051
14.2.3 Direct Access 1051
14.2.4 Raw I/O 1052
14.2.5 Properties 1053
14.2.6 Diagonal density matrix 1055
14.2.7 Massless density matrix 1056
14.2.8 Arbitrary density matrix 1057
14.2.9 Unit tests 1059
14.3 Polarization 1073
14.3.1 The polarization type 1074
14.3.2 Basic initializer and finalizer 1074
14.3.3 I/O 1076
14.3.4 Accessing contents 1077
14.3.5 Mapping between polarization and state matrix 1078
14.3.6 Specific initializers 1080

14.3.7 Operations . . . 1083
14.3.8 Polarization Iterator . . . 1084
14.3.9 Sparse Matrix . . . 1088
14.3.10 Polarization Matrix . . . 1090
14.3.11 Data Transformation . . . 1094
14.3.12 Unit tests . . . 1095
14.4 Particles . . . 1105
14.4.1 The particle type . . . 1105
14.4.2 Particle sets . . . 1122
14.4.3 I/O formats . . . 1134
14.4.4 Expression interface . . . 1156
14.4.5 Unit tests . . . 1160

15 Beams **1181**
15.1 Beam structure . . . 1183
15.1.1 Beam structure elements . . . 1183
15.1.2 Beam structure type . . . 1184
15.1.3 Polarization . . . 1189
15.1.4 Beam momenta . . . 1190
15.1.5 Get contents . . . 1191
15.1.6 Unit Tests . . . 1195
15.2 Beams for collisions and decays . . . 1202
15.2.1 Beam data . . . 1203
15.2.2 Initializers: beam structure . . . 1209
15.2.3 Initializers: collisions . . . 1210
15.2.4 Initializers: decays . . . 1212
15.2.5 Sanity check . . . 1213
15.2.6 The beams type . . . 1213
15.2.7 Inherited procedures . . . 1215
15.2.8 Accessing contents . . . 1216
15.2.9 Unit tests . . . 1216
15.3 Tools . . . 1227
15.3.1 Momentum splitting . . . 1228
15.3.2 Constant data . . . 1230
15.3.3 Sampling recoil . . . 1231
15.3.4 Splitting . . . 1233
15.3.5 Recovering the splitting . . . 1235
15.3.6 Extract data . . . 1236
15.3.7 Unit tests . . . 1236
15.4 Mappings for structure functions . . . 1255
15.4.1 Base type . . . 1255
15.4.2 Methods for self-tests . . . 1258
15.4.3 Implementation: standard mapping . . . 1260
15.4.4 Implementation: resonance pair mapping . . . 1262
15.4.5 Implementation: resonance single mapping . . . 1263
15.4.6 Implementation: on-shell mapping . . . 1265
15.4.7 Implementation: on-shell single mapping . . . 1267
15.4.8 Implementation: endpoint mapping . . . 1269
15.4.9 Implementation: endpoint mapping with resonance . . . 1271
15.4.10 Implementation: endpoint mapping for on-shell particle . 1273

15.4.11 Implementation: ISR endpoint mapping 1275
15.4.12 Implementation: ISR endpoint mapping, resonant 1277
15.4.13 Implementation: ISR on-shell mapping 1280
15.4.14 Implementation: Endpoint + ISR power mapping 1282
15.4.15 Implementation: Endpoint + ISR + resonance 1284
15.4.16 Implementation: Endpoint + ISR power mapping, on-shell 1286
15.4.17 Basic formulas 1289
15.4.18 Unit tests 1308
15.5 Structure function base 1331
15.5.1 Abstract structure-function data type 1332
15.5.2 Structure-function chain configuration 1334
15.5.3 Structure-function instance 1335
15.5.4 Accessing the structure function 1348
15.5.5 Direct calculations 1349
15.5.6 Structure-function instance 1350
15.5.7 Structure-function chain 1351
15.5.8 Chain instances 1356
15.5.9 Access to the chain instance 1370
15.5.10 Unit tests 1372
15.5.11 Test implementation: structure function 1373
15.5.12 Test implementation: pair spectrum 1379
15.5.13 Test implementation: generator spectrum 1384
15.6 Photon radiation: ISR 1429
15.6.1 Physics 1430
15.6.2 Implementation 1432
15.6.3 The ISR data block 1432
15.6.4 The ISR object 1437
15.6.5 Kinematics 1438
15.6.6 ISR application 1441
15.6.7 Unit tests 1443
15.7 EPA 1457
15.7.1 Physics 1457
15.7.2 The EPA data block 1458
15.7.3 The EPA object 1462
15.7.4 Kinematics 1465
15.7.5 EPA application 1467
15.7.6 Unit tests 1468
15.8 EWA 1480
15.8.1 Physics 1480
15.8.2 The EWA data block 1482
15.8.3 The EWA object 1487
15.8.4 Kinematics 1491
15.8.5 EWA application 1494
15.8.6 Unit tests 1495
15.9 Energy-scan spectrum 1507
15.9.1 Data type 1508
15.9.2 The Energy-scan object 1510
15.9.3 Kinematics 1512
15.9.4 Energy scan application 1513
15.9.5 Unit tests 1513

15.10 Gaussian beam spread . . . 1518
15.10.1 The beam-data file registry . . . 1518
15.10.2 Data type . . . 1519
15.10.3 The gaussian object . . . 1521
15.10.4 Kinematics . . . 1523
15.10.5 gaussian application . . . 1524
15.10.6 Unit tests . . . 1525
15.11 Using beam event data . . . 1530
15.11.1 The beam-data file registry . . . 1530
15.11.2 Data type . . . 1531
15.11.3 The beam events object . . . 1534
15.11.4 Kinematics . . . 1536
15.11.5 Beam events application . . . 1538
15.11.6 Unit tests . . . 1538
15.12 Lepton collider beamstrahlung: CIRCE1 . . . 1545
15.12.1 Physics . . . 1545
15.12.2 The CIRCE1 data block . . . 1546
15.12.3 Random Number Generator for CIRCE . . . 1550
15.12.4 The CIRCE1 object . . . 1550
15.12.5 Kinematics . . . 1555
15.12.6 CIRCE1 application . . . 1557
15.12.7 Unit tests . . . 1558
15.13 Lepton Collider Beamstrahlung and Photon collider: CIRCE2 . . . 1566
15.13.1 Physics . . . 1566
15.13.2 The CIRCE2 data block . . . 1566
15.13.3 Random Number Generator for CIRCE . . . 1571
15.13.4 The CIRCE2 object . . . 1571
15.13.5 Kinematics . . . 1574
15.13.6 CIRCE2 application . . . 1576
15.13.7 Unit tests . . . 1577
15.14 HOPPET interface . . . 1584
15.15 Builtin PDF sets . . . 1585
15.15.1 The module . . . 1585
15.15.2 Codes for default PDF sets . . . 1585
15.15.3 The PDF builtin data block . . . 1586
15.15.4 The PDF object . . . 1589
15.15.5 Kinematics . . . 1591
15.15.6 Structure function . . . 1593
15.15.7 Strong Coupling . . . 1594
15.15.8 Unit tests . . . 1595
15.16 LHAPDF . . . 1601
15.16.1 The module . . . 1601
15.16.2 Codes for default PDF sets . . . 1602
15.16.3 LHAPDF library interface . . . 1602
15.16.4 The LHAPDF status . . . 1604
15.16.5 LHAPDF initialization . . . 1605
15.16.6 Kinematics . . . 1606
15.16.7 The LHAPDF data block . . . 1608
15.16.8 The LHAPDF object . . . 1613
15.16.9 Structure function . . . 1615

15.16.10 Strong Coupling . . . 1617
15.16.11 Unit tests . . . 1618
15.17 Easy PDF Access . . . 1624
15.18 Dispatch . . . 1627
15.18.1 QCD coupling . . . 1636

16 Interface for Matrix Element Objects 1639
16.1 Abstract process core . . . 1639
16.1.1 The process core . . . 1641
16.1.2 Storage for intermediate results . . . 1645
16.1.3 Helicity selection data . . . 1645
16.2 Test process type . . . 1646
16.3 Template matrix elements . . . 1650
16.3.1 Process definition . . . 1650
16.3.2 The Template Matrix element writer . . . 1654
16.3.3 Driver . . . 1666
16.3.4 High-level process definition . . . 1666
16.3.5 The `prc_template_me_t` wrapper . . . 1667
16.3.6 Unit Test . . . 1672
16.4 `O'MEGA` Interface . . . 1680
16.4.1 Process definition . . . 1681
16.4.2 The `O'MEGA` writer . . . 1685
16.4.3 Driver . . . 1695
16.4.4 High-level process definition . . . 1695
16.4.5 The `prc_omega_t` wrapper . . . 1696
16.4.6 Unit Test . . . 1703
16.5 User-defined matrix elements squared . . . 1724
16.5.1 Handling of structure functions . . . 1725
16.5.2 Abstract user defined processes . . . 1727
16.5.3 User defined test . . . 1741
16.5.4 Threshold . . . 1745

17 Generic Event Handling 1758
17.1 Generic Event Handling . . . 1759
17.1.1 generic event type . . . 1760
17.1.2 Initialization . . . 1760
17.1.3 Access particle set . . . 1761
17.1.4 Access sqme and weight . . . 1762
17.1.5 Pure Virtual Methods . . . 1766
17.1.6 Utilities . . . 1770
17.1.7 Event normalization . . . 1771
17.1.8 Callback container . . . 1772
17.2 Event Sample Data . . . 1774
17.2.1 Event Sample Data . . . 1774
17.2.2 Unit tests . . . 1776
17.3 Abstract I/O Handler . . . 1781
17.3.1 Type . . . 1782
17.3.2 Unit tests . . . 1787
17.4 Event Generation Checkpoints . . . 1793
17.4.1 Type . . . 1794

17.4.2 Specific Methods 1794
17.4.3 Common Methods 1795
17.4.4 Message header 1798
17.4.5 Unit tests 1799
17.5 Event Generation Callback 1801
17.5.1 Type 1802
17.5.2 Specific Methods 1802
17.5.3 Common Methods 1802
17.6 Event Weight Output 1805
17.6.1 Type 1805
17.6.2 Specific Methods 1806
17.6.3 Common Methods 1806
17.6.4 Unit tests 1809
17.7 Event Dump Output 1813
17.7.1 Type 1813
17.7.2 Specific Methods 1814
17.7.3 Common Methods 1814
17.7.4 Unit tests 1818
17.8 ASCII File Formats 1821
17.8.1 Type 1821
17.8.2 Specific Methods 1823
17.8.3 Common Methods 1824
17.8.4 Unit tests 1830
17.9 HEP Common Blocks 1849
17.9.1 Event characteristics 1850
17.9.2 Particle characteristics 1851
17.9.3 The HEPRUP common block 1851
17.9.4 Run parameter output (verbose) 1859
17.9.5 Run parameter output (other formats) 1860
17.9.6 The HEPEUP common block 1861
17.9.7 The HEPEVT and HEPEV4 common block 1865
17.9.8 Event output 1870
17.9.9 Event output in various formats 1872
17.9.10 Event input in various formats 1875
17.9.11 Data Transfer: particle sets 1875
17.10 HepMC events 1879
17.10.1 Interface check 1880
17.10.2 FourVector 1880
17.10.3 Polarization 1883
17.10.4 GenParticle 1886
17.10.5 GenVertex 1894
17.10.6 Vertex-particle-in iterator 1897
17.10.7 Vertex-particle-out iterator 1899
17.10.8 GenEvent 1902
17.10.9 Event-particle iterator 1910
17.10.10 I/O streams 1912
17.10.11 Unit tests 1914
17.11 LCIO events 1919
17.11.1 Interface check 1919
17.11.2 LCIO Run Header 1920

17.11.3 LCIO Event and LC Collection 1921
17.11.4 LCIO Particle 1926
17.11.5 Polarization 1930
17.11.6 LCIO Writer type 1936
17.11.7 LCIO Reader type 1937
17.11.8 Unit tests 1941
17.12 HEP Common and Events 1944
17.12.1 Data Transfer: events 1945
17.12.2 Unit tests 1958
17.13 LHEF Input/Output 1963
17.13.1 Type 1963
17.13.2 Specific Methods 1964
17.13.3 Common Methods 1965
17.13.4 Les Houches Event File: header/footer 1975
17.13.5 Version-Specific Code: 1.0 1977
17.13.6 Version-Specific Code: 2.0 1978
17.13.7 Version-Specific Code: 3.0 1981
17.13.8 Unit tests 1984
17.14 STDHEP File Formats 1998
17.14.1 Type 1999
17.14.2 Specific Methods 2000
17.14.3 Common Methods 2001
17.14.4 Variables 2008
17.14.5 Unit tests 2008
17.15 HepMC Output 2018
17.15.1 Type 2019
17.15.2 Specific Methods 2019
17.15.3 Common Methods 2020
17.15.4 Unit tests 2025
17.16 LCIO Output 2030
17.16.1 Type 2031
17.16.2 Specific Methods 2031
17.16.3 Common Methods 2032
17.16.4 Unit tests 2037

18 Phase Space **2044**
18.1 Abstract phase-space module 2047
18.1.1 Phase-space channels 2047
18.1.2 Property collection 2052
18.1.3 Kinematics configuration 2055
18.1.4 Extract data 2060
18.1.5 Phase-space point instance 2062
18.1.6 Auxiliary stuff 2072
18.1.7 Unit tests 2072
18.2 Single-particle phase space 2087
18.2.1 Configuration 2087
18.2.2 Kinematics implementation 2089
18.2.3 Unit tests 2093
18.3 Resonance Handler 2102
18.3.1 Decay products (contributors) 2102

18.3.2 Resonance info object 2103
18.3.3 Resonance history object 2105
18.3.4 O'Mega restriction strings 2109
18.3.5 Resonance history set 2110
18.3.6 Unit tests 2112
18.4 Mappings 2119
18.4.1 Default parameters 2119
18.4.2 The Mapping type 2121
18.4.3 Screen output 2121
18.4.4 Define a mapping 2122
18.4.5 Retrieve contents 2124
18.4.6 Compare mappings 2126
18.4.7 Mappings of the invariant mass 2126
18.4.8 Step mapping 2132
18.4.9 Mappings of the polar angle 2135
18.5 Phase-space trees 2138
18.5.1 Particles 2139
18.5.2 The phase-space tree type 2141
18.5.3 PHS tree setup 2145
18.5.4 Phase-space evaluation 2157
18.5.5 Unit tests 2167
18.6 The phase-space forest 2172
18.6.1 Phase-space setup parameters 2173
18.6.2 Equivalences 2175
18.6.3 Groves 2177
18.6.4 The forest type 2179
18.6.5 Screen output 2181
18.6.6 Accessing contents 2184
18.6.7 Read the phase space setup from file 2187
18.6.8 Preparation 2193
18.6.9 Accessing the particle arrays 2195
18.6.10 Find equivalences among phase-space trees 2198
18.6.11 Interface for channel equivalences 2199
18.6.12 Phase-space evaluation 2200
18.6.13 Unit tests 2203
18.7 Finding phase space parameterizations 2208
18.7.1 The mapping modes 2209
18.7.2 The cascade type 2210
18.7.3 Creating new cascades 2217
18.7.4 Tools 2218
18.7.5 Hash entries for cascades 2219
18.7.6 The cascade set 2222
18.7.7 Adding cascades 2232
18.7.8 External particles 2235
18.7.9 Cascade combination I: flavor assignment 2237
18.7.10 Cascade combination II: kinematics setup and check . . . 2238
18.7.11 Cascade combination III: node connections and tree fusion 2244
18.7.12 Cascade set generation 2247
18.7.13 Groves 2250
18.7.14 Generate the phase space file 2251

18.7.15 Return the resonance histories for subtraction 2253
18.7.16 Unit tests 2255
18.8 WOOD phase space 2259
18.8.1 Configuration 2259
18.8.2 Phase-space generation 2262
18.8.3 Phase-space configuration 2265
18.8.4 Kinematics implementation 2271
18.8.5 Evaluation 2273
18.8.6 Unit tests 2275
18.9 The FKS phase space 2292
18.9.1 Creation of the real phase space - FSR 2320
18.9.2 Creation of the real phase space - ISR 2334
18.9.3 Unit tests 2355
18.10 Dispatch 2369
18.10.1 Unit tests 2378

19 VEGAS Integration 2383
19.1 Integration modes 2383
19.2 Type: vegas_func_t 2384
19.3 Type: vegas_config_t 2384
19.4 Type: vegas_grid_t 2385
19.5 Type: vegas_result_t 2387
19.6 Type: vegas_t 2389
19.7 Get-/Set-methods 2390
19.8 Grid resize- and refinement 2394
19.9 Integration 2398
19.10 I/0 operation 2404
19.11 Unit tests 2407
19.12 VAMP2 2421
19.12.1 Type: vamp2_func_t 2421
19.12.2 Type: vamp2_config_t 2424
19.12.3 Type: vamp2_result_t 2425
19.12.4 Type: vamp2_t 2426
19.13 Unit tests 2437

20 Multi-Channel Integration 2445
20.1 Generic Integrator 2447
20.1.1 MCI: integrator 2447
20.1.2 MCI instance 2457
20.1.3 MCI state 2462
20.1.4 MCI sampler 2463
20.1.5 Results record 2465
20.1.6 Unit tests 2466
20.2 Iterations 2491
20.2.1 The iterations list 2491
20.2.2 Tools 2494
20.2.3 Iteration Multipliers 2495
20.2.4 Unit tests 2495
20.3 Integration results 2498
20.3.1 Integration results entry 2499

20.3.2 Combined integration results . . . 2505
20.3.3 Access results . . . 2515
20.3.4 Results display . . . 2519
20.4 Simple midpoint integration . . . 2525
20.4.1 Integrator . . . 2525
20.4.2 Integrator instance . . . 2532
20.4.3 Unit tests . . . 2536
20.5 `VAMP` interface . . . 2553
20.5.1 Grid parameters . . . 2554
20.5.2 History parameters . . . 2555
20.5.3 Integration pass . . . 2555
20.5.4 Integrator . . . 2562
20.5.5 Sampler as Workspace . . . 2583
20.5.6 Integrator instance . . . 2583
20.5.7 Sampling function . . . 2593
20.5.8 Integrator instance: evaluation . . . 2595
20.5.9 VAMP exceptions . . . 2596
20.5.10 Unit tests . . . 2597
20.6 Multi-channel integration with VAMP2 . . . 2637
20.6.1 Type: mci_vamp2_func_t . . . 2638
20.6.2 Type: mci_vamp2_config_t . . . 2640
20.6.3 Integration pass . . . 2640
20.6.4 Integrator . . . 2647
20.6.5 Event generation . . . 2662
20.6.6 Integrator instance . . . 2664
20.6.7 Unit tests . . . 2668
20.7 Dispatch . . . 2679
20.7.1 Unit tests . . . 2682

21 Parton shower and interface to PYTHIA6 2686
21.1 Basics of the shower . . . 2686
21.1.1 Shower implementations . . . 2687
21.1.2 Shower settings . . . 2688
21.1.3 Abstract Shower Type . . . 2692
21.1.4 Additional parameters . . . 2694
21.1.5 Unit tests . . . 2698
21.2 Parton module for the shower . . . 2700
21.2.1 The basic type defintions . . . 2701
21.2.2 Routines . . . 2701
21.2.3 The analytic FSR . . . 2717
21.3 Main shower module . . . 2723
21.4 Interface to PYTHIA . . . 2802

22 Multiple Interactions (MPI) Code 2837
22.1 Basic types . . . 2837
22.2 Transverse momenta . . . 2902
22.3 Multi parton interactions . . . 2910
22.4 VEGAS and CUBA integration routines . . . 2947
22.5 Trapezoidal integration routines . . . 2966
22.6 Fibonacci trees . . . 3000

22.7 Adaptive Quadrature . . . 3041
22.8 Integrands for Multiple Interactions . . . 3055
22.9 MC Integrations for QCD $2 \to 2$ processes . . . 3059
22.10 Proton remnants . . . 3089
22.11 The Multiple Interactions main module . . . 3124
22.11.1 The main Multiple Interactions type . . . 3131

23 User Plugin Support 3147
23.1 User Code Interface . . . 3149
23.1.1 User Code Management . . . 3149
23.1.2 Interfaces for user-defined functions . . . 3152
23.1.3 Interfaces for user-defined interactions . . . 3154
23.2 User Plugin for Structure Functions . . . 3156
23.2.1 The module . . . 3156
23.2.2 The user structure function data block . . . 3157
23.2.3 The interaction . . . 3159
23.2.4 The user structure function . . . 3162
23.2.5 Kinematics . . . 3163

24 BLHA Interface 3166
24.1 Module definition . . . 3166
24.2 Configuration . . . 3199
24.2.1 Unit tests . . . 3220

25 GoSam Interface 3225
25.1 Gosam Interface . . . 3226

26 OpenLoops Interface 3239

27 FKS Subtraction Scheme 3252
27.1 Brief outline of FKS subtraction . . . 3252
27.2 Identifying singular regions . . . 3255
27.3 FKS Regions . . . 3257
27.3.1 Unit tests . . . 3340
27.4 Virtual contribution to the cross section . . . 3353
27.5 Real Subtraction . . . 3370
27.5.1 Soft mismatch . . . 3378
27.5.2 Collinear and soft-collinear subtraction terms . . . 3381
27.5.3 Real Subtraction . . . 3384
27.5.4 The real contribution to the cross section . . . 3387
27.6 Combining the FKS Pieces . . . 3409
27.6.1 Unit tests . . . 3432
27.6.2 Putting it together . . . 3438
27.7 Contribution of divergencies due to PDF Evolution . . . 3442
27.8 Dispatch . . . 3451

28 Model Handling and Features **3453**
28.1 Automatic generation of process components . . . 3455
28.1.1 Constraints: Abstract types . . . 3455
28.1.2 Specific constraints . . . 3458
28.1.3 Tables of states . . . 3466
28.1.4 Top-level methods . . . 3472
28.1.5 Splitting algorithm . . . 3473
28.1.6 Tools . . . 3477
28.1.7 Access results . . . 3478
28.1.8 Unit tests . . . 3480
28.2 Creating the real flavor structure . . . 3488
28.2.1 Unit tests . . . 3511
28.3 Sindarin Expression Implementation . . . 3521
28.3.1 Tree nodes . . . 3522
28.3.2 Operation types . . . 3539
28.3.3 Specific operators . . . 3544
28.3.4 Compiling the parse tree . . . 3572
28.3.5 Auxiliary functions for the compiler . . . 3632
28.3.6 Evaluation . . . 3633
28.3.7 Evaluation syntax . . . 3640
28.3.8 Set up appropriate parse trees . . . 3649
28.3.9 The evaluation tree . . . 3651
28.3.10 Direct evaluation . . . 3660
28.3.11 Factory Type . . . 3664
28.3.12 Unit tests . . . 3666
28.4 Physics Models . . . 3672
28.4.1 Module . . . 3672
28.4.2 Physics Parameters . . . 3673
28.4.3 Model Object . . . 3676
28.4.4 Model Access via Variables . . . 3687
28.4.5 Scheme handling . . . 3689
28.4.6 Reading models from file . . . 3691
28.4.7 Test models . . . 3705
28.4.8 Model list . . . 3705
28.4.9 Model instances . . . 3709
28.4.10 Unit tests . . . 3710
28.5 The SUSY Les Houches Accord . . . 3723
28.5.1 Preprocessor . . . 3723
28.5.2 Lexer and syntax . . . 3725
28.5.3 Interpreter . . . 3727
28.5.4 Auxiliary function . . . 3733
28.5.5 Parser . . . 3745
28.5.6 API . . . 3746
28.5.7 Dispatch . . . 3747
28.5.8 Unit tests . . . 3747

29 Infrastructure for threshold processes **3752**
29.0.1 Unit tests . . . 3983

30 Integration and Process Objects **3990**
30.1 Process observables . . . 3992
30.1.1 Abstract base type . . . 3993
30.1.2 Initialization . . . 3994
30.1.3 Evaluation . . . 3995
30.1.4 Implementation for partonic events . . . 3996
30.1.5 Implementation for full events . . . 4003
30.2 Parton states . . . 4009
30.2.1 Abstract base type . . . 4010
30.2.2 Common Initialization . . . 4014
30.2.3 Evaluator initialization: isolated state . . . 4014
30.2.4 Evaluator initialization: connected state . . . 4017
30.2.5 Cuts and expressions . . . 4020
30.2.6 Evaluation . . . 4022
30.2.7 Accessing the state . . . 4025
30.2.8 Unit tests . . . 4026
30.3 Process component management . . . 4030
30.3.1 Abstract base type . . . 4030
30.3.2 Manager instance . . . 4032
30.4 Manager configuration . . . 4033
30.5 The process object . . . 4037
30.5.1 Default iterations . . . 4080
30.5.2 Constant process data . . . 4082
30.5.3 Compute an amplitude . . . 4089
30.5.4 NLO specifics . . . 4094
30.6 Process config . . . 4101
30.6.1 Generic configuration data . . . 4102
30.6.2 Metadata . . . 4106
30.6.3 Beam configuration . . . 4111
30.6.4 Process components . . . 4117
30.6.5 Process terms . . . 4122
30.7 Process call statistics . . . 4129
30.8 Multi-channel integration . . . 4131
30.8.1 Process MCI entry . . . 4132
30.8.2 MC parameter set and MCI instance . . . 4142
30.9 Process component manager . . . 4145
30.9.1 Default process component manager . . . 4146
30.9.2 NLO process component manager . . . 4147
30.10 Kinematics instance . . . 4163
30.11 Instances . . . 4173
30.11.1 Term instance . . . 4174
30.11.2 The process instance . . . 4203
30.12 Unit tests . . . 4240
30.13 Process Stacks . . . 4278
30.13.1 The process entry type . . . 4279
30.13.2 The process stack type . . . 4280
30.13.3 Link . . . 4282
30.13.4 Push . . . 4283
30.13.5 Data Access . . . 4284
30.13.6 Unit tests . . . 4284

31 Matching **4291**
31.1 Abstract Matching Type 4292
31.1.1 Matching implementations 4294
31.2 MLM Matching 4295
31.3 CKKW matching 4305
31.4 POWHEG 4315
31.4.1 Base types for settings and data 4316
31.4.2 Upper bounding functions and `sudakovs` 4323
31.4.3 Main POWHEG class 4340
31.4.4 α_s and its reweighting 4355
31.4.5 Unit tests 4358

32 Event Implementation **4363**
32.1 Abstract Event Transforms 4363
32.1.1 Abstract base type 4365
32.1.2 Implementation: Trivial transform 4370
32.1.3 Unit tests 4372
32.2 Hadronization interface 4376
32.2.1 Hadronization implementations 4376
32.2.2 Hadronization settings 4377
32.2.3 Abstract Hadronization Type 4378
32.2.4 `WHIZARD` Hadronization Type 4380
32.2.5 `PYTHIA6` Hadronization Type 4383
32.2.6 `Pythia8` Hadronization Type 4385
32.2.7 Hadronization Event Transform 4386
32.3 Tau decays 4389
32.3.1 Tau Decays Event Transform 4389
32.4 Decays 4391
32.4.1 Final-State Particle Configuration 4392
32.4.2 Final-State Particle 4393
32.4.3 Decay Term Configuration 4394
32.4.4 Decay Term 4396
32.4.5 Decay Root Configuration 4399
32.4.6 Decay Root Instance 4404
32.4.7 Decay Configuration 4408
32.4.8 Decay Instance 4410
32.4.9 Stable Particles 4413
32.4.10 Unstable Particles 4415
32.4.11 Decay Chain 4421
32.4.12 Decay as Event Transform 4426
32.4.13 Unit tests 4430
32.5 Shower 4443
32.5.1 Configuration Parameters 4444
32.5.2 Event Transform 4445
32.5.3 Unit tests 4450
32.6 Fixed Order NLO Events 4458
32.7 Complete Events 4471
32.7.1 Event configuration 4472
32.7.2 The event type 4473
32.7.3 Initialization 4478

32.7.4 Evaluation 4481
32.7.5 Reset to empty state 4484
32.7.6 Squared Matrix Element and Weight 4485
32.7.7 Generation 4487
32.7.8 Recovering an event 4488
32.7.9 Access content 4491
32.7.10 Unit tests 4495
32.8 Raw Event I/O 4506
32.8.1 File Format Version 4506
32.8.2 Type 4507
32.8.3 Unit tests 4515
32.9 Dispatch 4521
32.9.1 Unit tests 4530

33 Integration and Simulation 4536
33.1 User-controlled File I/O 4536
33.1.1 The file type 4536
33.1.2 The file list 4539
33.2 Runtime data 4543
33.2.1 Strategy for models and variables 4543
33.2.2 Container for parse nodes 4544
33.2.3 The data type 4547
33.2.4 Output 4547
33.2.5 Clear 4550
33.2.6 Initialization 4551
33.2.7 Local copies 4551
33.2.8 Finalization 4555
33.2.9 Model Management 4555
33.2.10 Managing Variables 4559
33.2.11 Further Content 4564
33.2.12 Miscellaneous 4565
33.2.13 Unit Tests 4569
33.3 Select implementations 4586
33.3.1 Process Core Definition 4586
33.3.2 Process core allocation 4589
33.3.3 Process core update and restoration 4590
33.3.4 Unit Tests 4591
33.4 Process Configuration 4604
33.4.1 Data Type 4604
33.4.2 Unit Tests 4608
33.5 Compilation 4614
33.5.1 The data type 4614
33.5.2 API for library compilation and loading 4616
33.5.3 Compiling static executable 4617
33.5.4 API for executable compilation 4621
33.5.5 Unit Tests 4622
33.5.6 Test static build 4627
33.6 Integration 4631
33.6.1 The integration type 4632
33.6.2 Initialization 4632

33.6.3 Integration 4641
33.6.4 API for integration objects 4643
33.6.5 Unit Tests 4649
33.7 Event Streams 4669
33.7.1 Event Stream Array 4669
33.7.2 Unit Tests 4675
33.8 Simulation 4683
33.8.1 Event counting 4684
33.8.2 Simulation: component sets 4687
33.8.3 Process-core Safe 4688
33.8.4 Process Object 4688
33.8.5 Simulation entry 4690
33.8.6 Entries for alternative environment 4704
33.8.7 The simulation type 4706
33.8.8 Event Stream I/O 4723
33.8.9 Event Stream Array 4724
33.8.10 Recover event 4726
33.8.11 Extract contents 4727
33.8.12 Auxiliary 4729
33.8.13 Unit tests 4730

34 More Unit Tests **4766**
34.1 Expression Testing 4766
34.1.1 Test 4767

35 Top Level **4783**
35.1 Commands 4784
35.1.1 The command type 4785
35.1.2 Options 4788
35.1.3 Specific command types 4790
35.1.4 User-controlled output to data files 4865
35.1.5 Print custom-formatted values 4867
35.1.6 The command list 4916
35.1.7 Compiling the parse tree 4918
35.1.8 Executing the command list 4918
35.1.9 Command list syntax 4919
35.1.10 Unit Tests 4926
35.2 Toplevel module WHIZARD 4988
35.2.1 Options 4988
35.2.2 Parse tree stack 4989
35.2.3 The `whizard` object 4990
35.2.4 Initialization and finalization 4990
35.2.5 Initialization and finalization (old version) 4993
35.2.6 Execute command lists 4994
35.2.7 The WHIZARD shell 4996
35.3 Tools for the command line 4998
35.4 Query Feature Support 5000
35.4.1 Output 5000
35.4.2 Query function 5000
35.4.3 Basic configuration 5001

35.4.4 Optional features case by case 5001
35.5 Driver program 5005
35.6 Driver program for the unit tests 5015
35.6.1 Self-tests 5018
35.6.2 Unit test references 5022
35.6.3 BLHA 5026
35.7 Whizard-C-Interface 5045

36 Cross References **5053**
36.1 Identifiers 5053
36.2 Chunks 5053

Chapter 1

Changes

For a comprehensive list of changes confer the ChangeLog file or the subversion log.

Chapter 2

Preliminaries

The WHIZARD file header:

⟨*File header*⟩≡

```
! WHIZARD ⟨Version⟩ ⟨Date⟩
!
! Copyright (C) 1999-2017 by
!     Wolfgang Kilian <kilian@physik.uni-siegen.de>
!     Thorsten Ohl <ohl@physik.uni-wuerzburg.de>
!     Juergen Reuter <juergen.reuter@desy.de>
!
!     with contributions from
!     Fabian Bach <fabian.bach@t-online.de>
!     Bijan Chokoufe <bijan.chokoufe@desy.de>
!     Christian Speckner <cnspeckn@googlemail.com>
!     So Young Shim <soyoung.shim@desy.de>
!     Florian Staub <florian.staub@cern.ch>
!     Christian Weiss <christian.weiss@desy.de>
!     and Hans-Werner Boschmann, Felix Braam,
!     Sebastian Schmidt, So-young Shim, Daniel Wiesler
!
! WHIZARD is free software; you can redistribute it and/or modify it
! under the terms of the GNU General Public License as published by
! the Free Software Foundation; either version 2, or (at your option)
! any later version.
!
! WHIZARD is distributed in the hope that it will be useful, but
! WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
! GNU General Public License for more details.
!
! You should have received a copy of the GNU General Public License
! along with this program; if not, write to the Free Software
! Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
!
!!!
```

⟨*Standard module head*⟩≡

```
implicit none
private
```

This is the way to envoke the kinds module (not contained in this source)

⟨*Use kinds*⟩≡

```
use kinds, only: default
```

⟨*Use kinds with double*⟩≡

```
use kinds, only: default, double
```

And we make heavy use of variable-length strings

⟨*Use strings*⟩≡

```
use iso_varying_string, string_t => varying_string
```

Chapter 3

Utilities

These modules are intended as part of WHIZARD, but in fact they are generic and could be useful for any purpose.

The modules depend only on modules from the `basics` set.

file_utils Procedures that deal with external files, if not covered by Fortran built-ins.

file_registries Manage files that are accessed by their name.

string_utils Some string-handling utilities. Includes conversion to C string.

format_utils Utilities for pretty-printing.

format_defs Predefined format strings.

numeric_utils Utilities for comparing numerical values.

3.1 File Utilities

This module provides miscellaneous tools associated with named external files. Currently only:

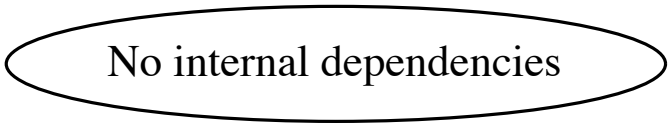

Figure 3.1: Module dependencies in `src/utilities`.

- Delete a named file

⟨file_utils.f90⟩≡
⟨*File header*⟩

```
module file_utils

  use io_units
```

⟨*Standard module head*⟩

⟨*File utils: public*⟩

```
contains
```

⟨*File utils: procedures*⟩

```
end module file_utils
```

3.1.1 Deleting a file

Fortran does not contain a command for deleting a file. Here, we provide a subroutine that deletes a file if it exists. We do not handle the subtleties, so we assume that it is writable if it exists.

⟨*File utils: public*⟩≡

```
  public :: delete_file
```

⟨*File utils: procedures*⟩≡

```
  subroutine delete_file (name)
    character(*), intent(in) :: name
    logical :: exist
    integer :: u
    inquire (file = name, exist = exist)
    if (exist) then
       u = free_unit ()
       open (unit = u, file = name)
       close (u, status = "delete")
    end if
  end subroutine delete_file
```

3.2 File Registries

This module provides a file-registry facility. We can open and close files multiple times without inadvertedly accessing a single file by two different I/O unit numbers. Opening a file the first time enters it into the registry. Opening again just returns the associated I/O unit. The registry maintains a reference count, so closing a file does not actually complete until the last reference is released.

File access will always be sequential, however. The file can't be opened at different positions simultaneously.

⟨file_registries.f90⟩≡
⟨*File header*⟩

```
module file_registries

⟨Use strings⟩
  use io_units

⟨Standard module head⟩

⟨File registries: public⟩

⟨File registries: types⟩

contains

⟨File registries: procedures⟩

end module file_registries
```

3.2.1 File handle

This object holds a filename (fully qualified), the associated unit, and a reference count. The idea is that the object should be deleted when the reference count drops to zero.

⟨*File registries: types*⟩≡

```
type :: file_handle_t
   type(string_t) :: file
   integer :: unit = 0
   integer :: refcount = 0
 contains
 ⟨File registries: file handle: TBP⟩
end type file_handle_t
```

Debugging output:

⟨*File registries: file handle: TBP*⟩≡

```
procedure :: write => file_handle_write
```

⟨*File registries: procedures*⟩≡

```
subroutine file_handle_write (handle, u, show_unit)
  class(file_handle_t), intent(in) :: handle
  integer, intent(in) :: u
  logical, intent(in), optional :: show_unit
  logical :: show_u
  show_u = .false.;  if (present (show_unit))  show_u = show_unit
  if (show_u) then
     write (u, "(3x,A,1x,I0,1x,'(',I0,')')")  &
          char (handle%file), handle%unit, handle%refcount
  else
     write (u, "(3x,A,1x,'(',I0,')')")  &
          char (handle%file), handle%refcount
  end if
end subroutine file_handle_write
```

Initialize with a file name, don't open the file yet:

⟨*File registries: file handle: TBP*⟩+≡

```
procedure :: init => file_handle_init
```

⟨*File registries: procedures*⟩+≡

```
subroutine file_handle_init (handle, file)
  class(file_handle_t), intent(out) :: handle
  type(string_t), intent(in) :: file
  handle%file = file
end subroutine file_handle_init
```

We check the `refcount` before actually opening the file.

⟨*File registries: file handle: TBP*⟩+≡

```
procedure :: open => file_handle_open
```

⟨*File registries: procedures*⟩+≡

```
subroutine file_handle_open (handle)
  class(file_handle_t), intent(inout) :: handle
  if (handle%refcount == 0) then
     handle%unit = free_unit ()
     open (unit = handle%unit, file = char (handle%file), action = "read", &
          status = "old")
  end if
  handle%refcount = handle%refcount + 1
end subroutine file_handle_open
```

Analogously, close if the refcount drops to zero. The caller may then delete the object.

⟨*File registries: file handle: TBP*⟩+≡

```
procedure :: close => file_handle_close
```

⟨*File registries: procedures*⟩+≡

```
subroutine file_handle_close (handle)
  class(file_handle_t), intent(inout) :: handle
  handle%refcount = handle%refcount - 1
  if (handle%refcount == 0) then
     close (handle%unit)
     handle%unit = 0
  end if
end subroutine file_handle_close
```

The I/O unit will be nonzero when the file is open.

⟨*File registries: file handle: TBP*⟩+≡

```
procedure :: is_open => file_handle_is_open
```

⟨*File registries: procedures*⟩+≡

```
function file_handle_is_open (handle) result (flag)
  class(file_handle_t), intent(in) :: handle
  logical :: flag
  flag = handle%unit /= 0
end function file_handle_is_open
```

Return the filename, so we can identify the entry.

⟨*File registries: file handle: TBP*⟩+≡
```
procedure :: get_file => file_handle_get_file
```

⟨*File registries: procedures*⟩+≡
```
function file_handle_get_file (handle) result (file)
  class(file_handle_t), intent(in) :: handle
  type(string_t) :: file
  file = handle%file
end function file_handle_get_file
```

For debugging, return the I/O unit number.

⟨*File registries: file handle: TBP*⟩+≡
```
procedure :: get_unit => file_handle_get_unit
```

⟨*File registries: procedures*⟩+≡
```
function file_handle_get_unit (handle) result (unit)
  class(file_handle_t), intent(in) :: handle
  integer :: unit
  unit = handle%unit
end function file_handle_get_unit
```

3.2.2 File handles registry

This is implemented as a doubly-linked list. The list exists only once in the program, as a private module variable.

Extend the handle type to become a list entry:

⟨*File registries: types*⟩+≡
```
type, extends (file_handle_t) :: file_entry_t
   type(file_entry_t), pointer :: prev => null ()
   type(file_entry_t), pointer :: next => null ()
end type file_entry_t
```

The actual registry. We need only the pointer to the first entry.

⟨*File registries: public*⟩≡
```
public :: file_registry_t
```

⟨*File registries: types*⟩+≡
```
type :: file_registry_t
   type(file_entry_t), pointer :: first => null ()
 contains
 ⟨File registries: file registry: TBP⟩
end type file_registry_t
```

Debugging output.

⟨*File registries: file registry: TBP*⟩≡
```
procedure :: write => file_registry_write
```

⟨*File registries: procedures*⟩+≡

```
subroutine file_registry_write (registry, unit, show_unit)
  class(file_registry_t), intent(in) :: registry
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: show_unit
  type(file_entry_t), pointer :: entry
  integer :: u
  u = given_output_unit (unit)
  if (associated (registry%first)) then
     write (u, "(1x,A)")  "File registry:"
     entry => registry%first
     do while (associated (entry))
        call entry%write (u, show_unit)
        entry => entry%next
     end do
  else
     write (u, "(1x,A)")  "File registry: [empty]"
  end if
end subroutine file_registry_write
```

Open a file: find the appropriate entry. Create a new entry and add to the list if necessary. The list is extended at the beginning. Return the I/O unit number for the records.

⟨*File registries: file registry: TBP*⟩+≡

```
procedure :: open => file_registry_open
```

⟨*File registries: procedures*⟩+≡

```
subroutine file_registry_open (registry, file, unit)
  class(file_registry_t), intent(inout) :: registry
  type(string_t), intent(in) :: file
  integer, intent(out), optional :: unit
  type(file_entry_t), pointer :: entry
  entry => registry%first
  FIND_ENTRY: do while (associated (entry))
     if (entry%get_file () == file)  exit FIND_ENTRY
     entry => entry%next
  end do FIND_ENTRY
  if (.not. associated (entry)) then
     allocate (entry)
     call entry%init (file)
     if (associated (registry%first)) then
        registry%first%prev => entry
        entry%next => registry%first
     end if
     registry%first => entry
  end if
  call entry%open ()
  if (present (unit))  unit = entry%get_unit ()
end subroutine file_registry_open
```

Close a file: find the appropriate entry. Delete the entry if there is no file connected to it anymore.

⟨*File registries: file registry: TBP*⟩+≡

```
    procedure :: close => file_registry_close
```

⟨*File registries: procedures*⟩+≡

```
    subroutine file_registry_close (registry, file)
      class(file_registry_t), intent(inout) :: registry
      type(string_t), intent(in) :: file
      type(file_entry_t), pointer :: entry
      entry => registry%first
      FIND_ENTRY: do while (associated (entry))
         if (entry%get_file () == file)  exit FIND_ENTRY
         entry => entry%next
      end do FIND_ENTRY
      if (associated (entry)) then
         call entry%close ()
         if (.not. entry%is_open ()) then
            if (associated (entry%prev)) then
               entry%prev%next => entry%next
            else
               registry%first => entry%next
            end if
            if (associated (entry%next)) then
               entry%next%prev => entry%prev
            end if
            deallocate (entry)
         end if
      end if
    end subroutine file_registry_close
```

3.3 String Utilities

This module provides tools associated with strings (built-in and variable). Currently:

- Upper and lower case for strings
- Convert to null-terminated C string

⟨`string_utils.f90`⟩≡

```
  ⟨File header⟩

  module string_utils

    use, intrinsic :: iso_c_binding

  ⟨Use kinds⟩
  ⟨Use strings⟩

  ⟨Standard module head⟩

  ⟨String utils: public⟩

  ⟨String utils: interfaces⟩
```

```
contains

⟨String utils: procedures⟩

end module string_utils
```

3.3.1 Upper and Lower Case

These are, unfortunately, not part of Fortran.

⟨*String utils: public*⟩≡
```
public :: upper_case
public :: lower_case
```

⟨*String utils: interfaces*⟩≡
```
interface upper_case
   module procedure upper_case_char, upper_case_string
end interface
interface lower_case
   module procedure lower_case_char, lower_case_string
end interface
```

⟨*String utils: procedures*⟩≡
```
function upper_case_char (string) result (new_string)
  character(*), intent(in) :: string
  character(len(string)) :: new_string
  integer :: pos, code
  integer, parameter :: offset = ichar('A')-ichar('a')
  do pos = 1, len (string)
     code = ichar (string(pos:pos))
     select case (code)
     case (ichar('a'):ichar('z'))
        new_string(pos:pos) = char (code + offset)
     case default
        new_string(pos:pos) = string(pos:pos)
     end select
  end do
end function upper_case_char

function lower_case_char (string) result (new_string)
  character(*), intent(in) :: string
  character(len(string)) :: new_string
  integer :: pos, code
  integer, parameter :: offset = ichar('a')-ichar('A')
  do pos = 1, len (string)
     code = ichar (string(pos:pos))
     select case (code)
     case (ichar('A'):ichar('Z'))
        new_string(pos:pos) = char (code + offset)
     case default
        new_string(pos:pos) = string(pos:pos)
     end select
  end do
end function lower_case_char
```

```
function upper_case_string (string) result (new_string)
  type(string_t), intent(in) :: string
  type(string_t) :: new_string
  new_string = upper_case_char (char (string))
end function upper_case_string

function lower_case_string (string) result (new_string)
  type(string_t), intent(in) :: string
  type(string_t) :: new_string
  new_string = lower_case_char (char (string))
end function lower_case_string
```

3.3.2 C-compatible Output

Convert a FORTRAN string into a zero terminated C string.

⟨*String utils: public*⟩+≡

```
public :: string_f2c
```

⟨*String utils: interfaces*⟩+≡

```
interface string_f2c
   module procedure string_f2c_char, string_f2c_var_str
end interface string_f2c
```

⟨*String utils: procedures*⟩+≡

```
pure function string_f2c_char (i) result (o)
  character(*), intent(in) :: i
  character(kind=c_char, len=len (i) + 1) :: o
  o = i // c_null_char
end function string_f2c_char

pure function string_f2c_var_str (i) result (o)
  type(string_t), intent(in) :: i
  character(kind=c_char, len=len (i) + 1) :: o
  o = char (i) // c_null_char
end function string_f2c_var_str
```

3.3.3 Number Conversion

Create a string from a number. We use fixed format for the reals and variable format for integers.

⟨*String utils: public*⟩+≡

```
public :: str
```

⟨*String utils: interfaces*⟩+≡

```
interface str
   module procedure str_log, str_logs, str_int, str_ints, &
          str_real, str_reals, str_complex, str_complexs
end interface
```

⟨*String utils: procedures*⟩+≡

```
function str_log (l) result (s)
  logical, intent(in) :: l
  type(string_t) :: s
  if (l) then
     s = "True"
  else
     s = "False"
  end if
end function str_log

function str_logs (x) result (s)
  logical, dimension(:), intent(in) :: x
  ⟨concatenate strings⟩
end function str_logs

function str_int (i) result (s)
  integer, intent(in) :: i
  type(string_t) :: s
  character(32) :: buffer
  write (buffer, "(I0)")  i
  s = var_str (trim (adjustl (buffer)))
end function str_int

function str_ints (x) result (s)
  integer, dimension(:), intent(in) :: x
  ⟨concatenate strings⟩
end function str_ints

function str_real (x) result (s)
  real(default), intent(in) :: x
  type(string_t) :: s
  character(32) :: buffer
  write (buffer, "(ES17.10)")  x
  s = var_str (trim (adjustl (buffer)))
end function str_real

function str_reals (x) result (s)
  real(default), dimension(:), intent(in) :: x
  ⟨concatenate strings⟩
end function str_reals

function str_complex (x) result (s)
  complex(default), intent(in) :: x
  type(string_t) :: s
  s = str_real (real (x)) // " + i " // str_real (aimag (x))
end function str_complex

function str_complexs (x) result (s)
  complex(default), dimension(:), intent(in) :: x
  ⟨concatenate strings⟩
end function str_complexs
```

⟨*concatenate strings*⟩≡

```
type(string_t) :: s
integer :: i
s = '['
do i = 1, size(x) - 1
   s = s // str(x(i)) // ', '
end do
s = s // str(x(size(x))) // ']'
```

Auxiliary: Read real, integer, string value.

⟨*String utils: public*⟩+≡

```
public :: read_rval
public :: read_ival
```

⟨*String utils: procedures*⟩+≡

```
function read_rval (s) result (rval)
  real(default) :: rval
  type(string_t), intent(in) :: s
  character(80) :: buffer
  buffer = s
  read (buffer, *)  rval
end function read_rval

function read_ival (s) result (ival)
  integer :: ival
  type(string_t), intent(in) :: s
  character(80) :: buffer
  buffer = s
  read (buffer, *)  ival
end function read_ival
```

3.3.4 String splitting

⟨*String utils: public*⟩+≡

```
public :: string_contains_word
```

⟨*String utils: procedures*⟩+≡

```
function string_contains_word (str, word) result (val)
  logical :: val
  type(string_t), intent(in) :: str, word
  type(string_t) :: str_tmp, str_out
  str_tmp = str
  call split (str_tmp, str_out, word)
  val = str_out /= ""
end function string_contains_word
```

Create an array of strings using a separator.

⟨*String utils: public*⟩+≡

```
public :: split_string
```

⟨*String utils: procedures*⟩+≡

```
subroutine split_string (str, separator, str_array)
  type(string_t), dimension(:), allocatable, intent(out) :: str_array
  type(string_t), intent(in) :: str, separator
  type(string_t) :: str_tmp, str_out
```

```
    integer :: n_str
    n_str = 0; str_tmp = str
    do while (string_contains_word (str_tmp, separator))
       n_str = n_str + 1
       call split (str_tmp, str_out, separator)
    end do
    allocate (str_array (n_str))
    n_str = 1; str_tmp = str
    do while (string_contains_word (str_tmp, separator))
       call split (str_tmp, str_array (n_str), separator)
       n_str = n_str + 1
    end do
  end subroutine split_string
```

3.4 Format Utilities

This module provides miscellaneous tools associated with formatting and pretty-printing.

- Horizontal separator lines in output
- Indenting an output line
- Formatting a number for TEX output.
- Formatting a number for MetaPost output.
- Alternate numeric formats.

⟨format_utils.f90⟩≡

```
⟨File header⟩

module format_utils

⟨Use kinds⟩
⟨Use strings⟩
  use string_utils, only: lower_case
  use io_units, only: given_output_unit

⟨Standard module head⟩

⟨Format utils: public⟩

contains

⟨Format utils: procedures⟩

end module format_utils
```

3.4.1 Line Output

Write a separator line.

⟨*Format utils: public*⟩≡

```
  public :: write_separator
```

⟨*Format utils: procedures*⟩≡

```
subroutine write_separator (u, mode)
  integer, intent(in) :: u
  integer, intent(in), optional :: mode
  integer :: m
  m = 1;  if (present (mode))  m = mode
  select case (m)
  case default
     write (u, "(A)")  repeat ("-", 72)
  case (1)
     write (u, "(A)")  repeat ("-", 72)
  case (2)
     write (u, "(A)")  repeat ("=", 72)
  end select
end subroutine write_separator
```

Indent the line with given number of blanks.

⟨*Format utils: public*⟩+≡

```
public :: write_indent
```

⟨*Format utils: procedures*⟩+≡

```
subroutine write_indent (unit, indent)
  integer, intent(in) :: unit
  integer, intent(in), optional :: indent
  if (present (indent)) then
     write (unit, "(1x,A)", advance="no")  repeat ("  ", indent)
  end if
end subroutine write_indent
```

3.4.2 Array Output

Write an array of integers.

⟨*Format utils: public*⟩+≡

```
public :: write_integer_array
```

⟨*Format utils: procedures*⟩+≡

```
subroutine write_integer_array (array, unit, n_max)
  integer, intent(in), dimension(:) :: array
  integer, intent(in), optional :: unit
  integer, intent(in), optional :: n_max
  integer :: u, i, n
  u = given_output_unit (unit)
  if (present (n_max)) then
     n = n_max
  else
     n = size (array)
  end if
  do i = 1, n
     if (i < n) then
        write (u, "(I0, A)", advance = "no") array(i), ", "
     else
        write (u, "(I0)") array(i)
     end if
```

```
    end do
  end subroutine write_integer_array
```

3.4.3 TEX-compatible Output

Quote underscore characters for use in TEX output.

⟨*Format utils: public*⟩+≡

```
  public :: quote_underscore
```

⟨*Format utils: procedures*⟩+≡

```
  function quote_underscore (string) result (quoted)
    type(string_t) :: quoted
    type(string_t), intent(in) :: string
    type(string_t) :: part
    type(string_t) :: buffer
    buffer = string
    quoted = ""
    do
      call split (part, buffer, "_")
      quoted = quoted // part
      if (buffer == "")  exit
      quoted = quoted // "\_"
    end do
  end function quote_underscore
```

Format a number with n significant digits for use in TEX documents.

⟨*Format utils: public*⟩+≡

```
  public :: tex_format
```

⟨*Format utils: procedures*⟩+≡

```
  function tex_format (rval, n_digits) result (string)
    type(string_t) :: string
    real(default), intent(in) :: rval
    integer, intent(in) :: n_digits
    integer :: e, n, w, d
    real(default) :: absval
    real(default) :: mantissa
    character :: sign
    character(20) :: format
    character(80) :: cstr
    n = min (abs (n_digits), 16)
    if (rval == 0) then
       string = "0"
    else
       absval = abs (rval)
       e = int (log10 (absval))
       if (rval < 0) then
          sign = "-"
       else
          sign = ""
       end if
       select case (e)
       case (:-3)
```

```
        d = max (n - 1, 0)
        w = max (d + 2, 2)
        write (format, "('(F',I0,'.',I0,',A,I0,A)')")  w, d
        mantissa = absval * 10._default ** (1 - e)
        write (cstr, fmt=format)  mantissa, "\times 10^{", e - 1, "}"
     case (-2:0)
        d = max (n - e, 1 - e)
        w = max (d + e + 2, d + 2)
        write (format, "('(F',I0,'.',I0,')')")  w, d
        write (cstr, fmt=format)  absval
     case (1:2)
        d = max (n - e - 1, -e, 0)
        w = max (d + e + 2, d + 2, e + 2)
        write (format, "('(F',I0,'.',I0,')')")  w, d
        write (cstr, fmt=format)  absval
     case default
        d = max (n - 1, 0)
        w = max (d + 2, 2)
        write (format, "('(F',I0,'.',I0,',A,I0,A)')")  w, d
        mantissa = absval * 10._default ** (- e)
        write (cstr, fmt=format)  mantissa, "\times 10^{", e, "}"
     end select
     string = sign // trim (cstr)
  end if
end function tex_format
```

3.4.4 Metapost-compatible Output

Write a number for use in Metapost code:

⟨*Format utils: public*⟩+≡

```
public :: mp_format
```

⟨*Format utils: procedures*⟩+≡

```
function mp_format (rval) result (string)
  type(string_t) :: string
  real(default), intent(in) :: rval
  character(16) :: tmp
  write (tmp, "(G16.8)")  rval
  string = lower_case (trim (adjustl (trim (tmp))))
end function mp_format
```

3.4.5 Conditional Formatting

Conditional format string, intended for switchable numeric precision.

⟨*Format utils: public*⟩+≡

```
public :: pac_fmt
```

⟨*Format utils: procedures*⟩+≡

```
subroutine pac_fmt (fmt, fmt_orig, fmt_pac, pacify)
  character(*), intent(in) :: fmt_orig, fmt_pac
  character(*), intent(out) :: fmt
```

```
    logical, intent(in), optional :: pacify
    logical :: pacified
    pacified = .false.
    if (present (pacify))  pacified = pacify
    if (pacified) then
       fmt = fmt_pac
    else
       fmt = fmt_orig
    end if
  end subroutine pac_fmt
```

3.4.6 Compressed output of integer arrays

⟨*Format utils: public*⟩+≡

```
  public :: write_compressed_integer_array
```

⟨*Format utils: procedures*⟩+≡

```
  subroutine write_compressed_integer_array (chars, array)
    character(len=*), intent(out) :: chars
    integer, intent(in), allocatable, dimension(:) :: array
    logical, dimension(:), allocatable :: used
    character(len=16) :: tmp
    type(string_t) :: string
    integer :: i, j, start_chain, end_chain
    chars = '[none]'
    string = ""
    if (allocated (array)) then
       if (size (array) > 0) then
          allocate (used (size (array)))
          used = .false.
          do i = 1, size (array)
             if (.not. used(i)) then
                start_chain = array(i)
                end_chain = array(i)
                used(i) = .true.
                EXTEND: do
                   do j = 1, size (array)
                      if (array(j) == end_chain + 1) then
                         end_chain = array(j)
                         used(j) = .true.
                         cycle EXTEND
                      end if
                      if (array(j) == start_chain - 1) then
                         start_chain = array(j)
                         used(j) = .true.
                         cycle EXTEND
                      end if
                   end do
                   exit
                end do EXTEND
                if (end_chain - start_chain > 0) then
                   write (tmp, "(I0,A,I0)") start_chain, "-", end_chain
                else
```

```
                write (tmp, "(I0)") start_chain
             end if
             string = string // trim (tmp)
             if (any (.not. used)) then
                string = string // ','
             end if
          end if
       end do
       chars = string
    end if
  end if
  chars = adjustr (chars)
end subroutine write_compressed_integer_array
```

3.5 Format Definitions

This module provides named integer parameters that specify certain format strings, used for numerical output.

⟨`format_defs.f90`⟩≡

```
⟨File header⟩

module format_defs

⟨Standard module head⟩

⟨Format defs: public parameters⟩

end module format_defs
```

We collect format strings for various numerical output formats here.

⟨*Format defs: public parameters*⟩≡

```
character(*), parameter, public :: FMT_19 = "ES19.12"
character(*), parameter, public :: FMT_18 = "ES18.11"
character(*), parameter, public :: FMT_17 = "ES17.10"
character(*), parameter, public :: FMT_16 = "ES16.9"
character(*), parameter, public :: FMT_15 = "ES15.8"
character(*), parameter, public :: FMT_14 = "ES14.7"
character(*), parameter, public :: FMT_13 = "ES13.6"
character(*), parameter, public :: FMT_12 = "ES12.5"
character(*), parameter, public :: FMT_11 = "ES11.4"
character(*), parameter, public :: FMT_10 = "ES10.3"
```

3.6 Numeric Utilities

⟨`numeric_utils.f90`⟩≡

```
⟨File header⟩

module numeric_utils

⟨Use kinds⟩
```

```
⟨Use strings⟩
  use string_utils
  use constants
  use format_defs

⟨Standard module head⟩

⟨Numeric utils: public⟩

⟨Numeric utils: parameters⟩

⟨Numeric utils: types⟩

⟨Numeric utils: interfaces⟩

contains

⟨Numeric utils: procedures⟩

end module numeric_utils
```

⟨*Numeric utils: public*⟩≡

```
  public :: assert
```

⟨*Numeric utils: procedures*⟩≡

```
  subroutine assert (unit, ok, description, exit_on_fail)
    integer, intent(in) :: unit
    logical, intent(in) :: ok
    character(*), intent(in), optional :: description
    logical, intent(in), optional :: exit_on_fail
    logical :: ef
    ef = .false.;  if (present (exit_on_fail)) ef = exit_on_fail
    if (.not. ok) then
       if (present(description)) then
          write (unit, "(A)") "* FAIL: " // description
       else
          write (unit, "(A)") "* FAIL: Assertion error"
       end if
       if (ef)  stop 1
    end if
  end subroutine assert
```

Compare numbers and output error message if not equal.

⟨*Numeric utils: public*⟩+≡

```
  public:: assert_equal
  interface assert_equal
     module procedure assert_equal_integer, assert_equal_integers, &
            assert_equal_real, assert_equal_reals, &
            assert_equal_complex, assert_equal_complexs
  end interface
```

⟨*Numeric utils: procedures*⟩+≡

```
  subroutine assert_equal_integer (unit, lhs, rhs, description, exit_on_fail)
    integer, intent(in) :: unit
```

```
    integer, intent(in) :: lhs, rhs
    character(*), intent(in), optional :: description
    logical, intent(in), optional :: exit_on_fail
    type(string_t) :: desc
    logical :: ok
    ok = lhs == rhs
    desc = '';  if (present (description)) desc = var_str(description) // ": "
    call assert (unit, ok, char(desc // str (lhs) // " /= " // str (rhs)), exit_on_fail)
  end subroutine assert_equal_integer
```

⟨*Numeric utils: procedures*⟩+≡

```
  subroutine assert_equal_integers (unit, lhs, rhs, description, exit_on_fail)
    integer, intent(in) :: unit
    integer, dimension(:), intent(in) :: lhs, rhs
    character(*), intent(in), optional :: description
    logical, intent(in), optional :: exit_on_fail
    type(string_t) :: desc
    logical :: ok
    ok = all(lhs == rhs)
    desc = '';  if (present (description)) desc = var_str(description) // ": "
    call assert (unit, ok, char(desc // str (lhs) // " /= " // str (rhs)), exit_on_fail)
  end subroutine assert_equal_integers
```

⟨*Numeric utils: procedures*⟩+≡

```
  subroutine assert_equal_real (unit, lhs, rhs, description, &
                                abs_smallness, rel_smallness, exit_on_fail)
    integer, intent(in) :: unit
    real(default), intent(in) :: lhs, rhs
    character(*), intent(in), optional :: description
    real(default), intent(in), optional :: abs_smallness, rel_smallness
    logical, intent(in), optional :: exit_on_fail
    type(string_t) :: desc
    logical :: ok
    ok = nearly_equal (lhs, rhs, abs_smallness, rel_smallness)
    desc = '';  if (present (description)) desc = var_str(description) // ": "
    call assert (unit, ok, char(desc // str (lhs) // " /= " // str (rhs)), exit_on_fail)
  end subroutine assert_equal_real
```

⟨*Numeric utils: procedures*⟩+≡

```
  subroutine assert_equal_reals (unit, lhs, rhs, description, &
                                 abs_smallness, rel_smallness, exit_on_fail)
    integer, intent(in) :: unit
    real(default), dimension(:), intent(in) :: lhs, rhs
    character(*), intent(in), optional :: description
    real(default), intent(in), optional :: abs_smallness, rel_smallness
    logical, intent(in), optional :: exit_on_fail
    type(string_t) :: desc
    logical :: ok
    ok = all(nearly_equal (lhs, rhs, abs_smallness, rel_smallness))
    desc = '';  if (present (description)) desc = var_str(description) // ": "
    call assert (unit, ok, char(desc // str (lhs) // " /= " // str (rhs)), exit_on_fail)
  end subroutine assert_equal_reals
```

⟨*Numeric utils: procedures*⟩+≡

```
subroutine assert_equal_complex (unit, lhs, rhs, description, &
                                 abs_smallness, rel_smallness, exit_on_fail)
  integer, intent(in) :: unit
  complex(default), intent(in) :: lhs, rhs
  character(*), intent(in), optional :: description
  real(default), intent(in), optional :: abs_smallness, rel_smallness
  logical, intent(in), optional :: exit_on_fail
  type(string_t) :: desc
  logical :: ok
  ok = nearly_equal (real(lhs), real(rhs), abs_smallness, rel_smallness) &
       .and. nearly_equal (aimag(lhs), aimag(rhs), abs_smallness, rel_smallness)
  desc = '';  if (present (description)) desc = var_str(description) // ": "
  call assert (unit, ok, char(desc // str (lhs) // " /= " // str (rhs)), exit_on_fail)
end subroutine assert_equal_complex
```

⟨*Numeric utils: procedures*⟩+≡

```
subroutine assert_equal_complexs (unit, lhs, rhs, description, &
                                  abs_smallness, rel_smallness, exit_on_fail)
  integer, intent(in) :: unit
  complex(default), dimension(:), intent(in) :: lhs, rhs
  character(*), intent(in), optional :: description
  real(default), intent(in), optional :: abs_smallness, rel_smallness
  logical, intent(in), optional :: exit_on_fail
  type(string_t) :: desc
  logical :: ok
  ok = all (nearly_equal (real(lhs), real(rhs), abs_smallness, rel_smallness)) &
       .and. all (nearly_equal (aimag(lhs), aimag(rhs), abs_smallness, rel_smallness))
  desc = '';  if (present (description)) desc = var_str(description) // ": "
  call assert (unit, ok, char(desc // str (lhs) // " /= " // str (rhs)), exit_on_fail)
end subroutine assert_equal_complexs
```

Note that this poor man's check will be disabled if someone compiles with `-ffast-math` or similar optimizations.

⟨*Numeric utils: procedures*⟩+≡

```
elemental function ieee_is_nan (x) result (yorn)
  logical :: yorn
  real(default), intent(in) :: x
  yorn = (x /= x)
end function ieee_is_nan
```

This is still not perfect but should work in most cases. Usually one wants to compare to a relative epsilon `rel_smallness`, except for numbers close to zero defined by `abs_smallness`. Both might need adaption to specific use cases but have reasonable defaults.

⟨*Numeric utils: public*⟩+≡

```
public :: nearly_equal
```

⟨*Numeric utils: interfaces*⟩≡

```
interface nearly_equal
   module procedure nearly_equal_real
   module procedure nearly_equal_complex
end interface nearly_equal
```

⟨*Numeric utils: procedures*⟩+≡

```
elemental function nearly_equal_real (a, b, abs_smallness, rel_smallness) result (r)
  logical :: r
  real(default), intent(in) :: a, b
  real(default), intent(in), optional :: abs_smallness, rel_smallness
  real(default) :: abs_a, abs_b, diff, abs_small, rel_small
  abs_a = abs (a)
  abs_b = abs (b)
  diff = abs (a - b)
  ! shortcut, handles infinities and nans
  if (a == b) then
     r = .true.
     return
  else if (ieee_is_nan (a) .or. ieee_is_nan (b) .or. ieee_is_nan (diff)) then
     r = .false.
     return
  end if
  abs_small = tiny_13; if (present (abs_smallness)) abs_small = abs_smallness
  rel_small = tiny_10; if (present (rel_smallness)) rel_small = rel_smallness
  if (abs_a < abs_small .and. abs_b < abs_small) then
     r = diff < abs_small
  else
     r = diff / max (abs_a, abs_b) < rel_small
  end if
end function nearly_equal_real
```

⟨*Numeric utils: procedures*⟩+≡

```
elemental function nearly_equal_complex (a, b, abs_smallness, rel_smallness) result (r)
  logical :: r
  complex(default), intent(in) :: a, b
  real(default), intent(in), optional :: abs_smallness, rel_smallness
  r = nearly_equal_real (real (a), real (b), abs_smallness, rel_smallness) .and. &
      nearly_equal_real (aimag (a), aimag(b), abs_smallness, rel_smallness)
end function nearly_equal_complex
```

Often we will need to check whether floats vanish:

⟨*Numeric utils: public*⟩+≡

```
public:: vanishes
interface vanishes
   module procedure vanishes_real, vanishes_complex
end interface
```

⟨*Numeric utils: procedures*⟩+≡

```
elemental function vanishes_real (x, abs_smallness, rel_smallness) result (r)
  logical :: r
  real(default), intent(in) :: x
  real(default), intent(in), optional :: abs_smallness, rel_smallness
  r = nearly_equal (x, zero, abs_smallness, rel_smallness)
end function vanishes_real

elemental function vanishes_complex (x, abs_smallness, rel_smallness) result (r)
  logical :: r
```

```
    complex(default), intent(in) :: x
    real(default), intent(in), optional :: abs_smallness, rel_smallness
    r = vanishes_real (abs (x), abs_smallness, rel_smallness)
  end function vanishes_complex
```

⟨*Numeric utils: public*⟩+≡

```
  public :: expanded_amp2
```

⟨*Numeric utils: procedures*⟩+≡

```
  pure function expanded_amp2 (amp_tree, amp_blob) result (amp2)
    real(default) :: amp2
    complex(default), dimension(:), intent(in) :: amp_tree, amp_blob
    amp2 = sum (amp_tree * conjg (amp_tree) + &
                amp_tree * conjg (amp_blob) + &
                amp_blob * conjg (amp_tree))
  end function expanded_amp2
```

⟨*Numeric utils: public*⟩+≡

```
  public :: abs2
```

⟨*Numeric utils: procedures*⟩+≡

```
  elemental function abs2 (c) result (c2)
    real(default) :: c2
    complex(default), intent(in) :: c
    c2 = real (c * conjg(c))
  end function abs2
```

Remove all duplicates from a list of integers. This method does not really fit into this module. It could be part of a larger module which deals with list manipulations.

⟨*Numeric utils: public*⟩+≡

```
  public :: remove_duplicates_from_list
```

⟨*Numeric utils: procedures*⟩+≡

```
  function remove_duplicates_from_list (list) result (list_clean)
    integer, dimension(:), allocatable :: list_clean
    integer, intent(in), dimension(:) :: list
    integer, parameter :: N_MAX = 20
    integer, dimension(N_MAX) :: buf
    integer :: i_buf, i_list, n_buf
    buf = -1; i_buf = 1
    if (size (list) > N_MAX) return
    do i_list = 1, size (list)
       if (.not. any (list(i_list) == buf)) then
          buf(i_buf) = list(i_list)
          i_buf = i_buf + 1
       end if
    end do
    n_buf = count (buf >= 0)
    allocate (list_clean (n_buf))
    list_clean = buf (1 : n_buf)
  end function remove_duplicates_from_list
```

⟨*Numeric utils: public*⟩+≡

```
public :: extend_integer_array
```

⟨*Numeric utils: procedures*⟩+≡

```
subroutine extend_integer_array (list, incr, initial_value)
  integer, intent(inout), dimension(:), allocatable :: list
  integer, intent(in) :: incr
  integer, intent(in), optional :: initial_value
  integer, dimension(:), allocatable :: list_store
  integer :: n, ini
  ini = 0; if (present (initial_value)) ini = initial_value
  n = size (list)
  allocate (list_store (n))
  list_store = list
  deallocate (list)
  allocate (list (n+incr))
  list(1:n) = list_store
  list(1+n : n+incr) = ini
  deallocate (list_store)
end subroutine extend_integer_array
```

⟨*Numeric utils: public*⟩+≡

```
public :: crop_integer_array
```

⟨*Numeric utils: procedures*⟩+≡

```
subroutine crop_integer_array (list, i_crop)
  integer, intent(inout), dimension(:), allocatable :: list
  integer, intent(in) :: i_crop
  integer, dimension(:), allocatable :: list_store
  allocate (list_store (i_crop))
  list_store = list(1:i_crop)
  deallocate (list)
  allocate (list (i_crop))
  list = list_store
  deallocate (list_store)
end subroutine crop_integer_array
```

Chapter 4

Testing

This part contains tools for automatic testing.

unit_tests A handler that executes test procedures and compares and collects results.

4.1 Unit tests

We provide functionality for automated unit tests. Each test is required to produce output which is compared against a reference file. If the two are identical, we signal success. Otherwise, we signal failure and write the output to a file.

⟨`unit_tests.f90`⟩≡

```
⟨File header⟩

module unit_tests

⟨Use kinds⟩
  use constants, only: zero
⟨Use strings⟩
  use format_defs
  use io_units
```

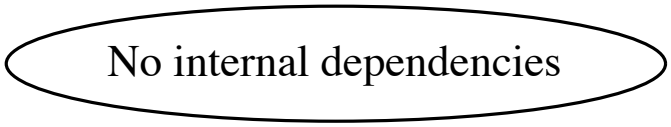

Figure 4.1: Module dependencies in `src/testing`.

⟨*Standard module head*⟩

⟨*Tests: public*⟩

⟨*Tests: parameters*⟩

⟨*Tests: types*⟩

⟨*Tests: interfaces*⟩

```
contains
```

⟨*Tests: procedures*⟩

```
end module unit_tests
```

4.1.1 Parameters

Building blocks of file names. The directory names and suffixes are hard-coded here, and they must reflect actual Makefile targets where applicable.

⟨*Tests: parameters*⟩≡

```
character(*), parameter :: ref_prefix = "ref-output/"
character(*), parameter :: ref = ".ref"

character(*), parameter :: err_prefix = "err-output/"
character(*), parameter :: err = ".out"
```

4.1.2 Type for storing test results

We store the results of the individual unit tests in a linked list. Here is the entry:

⟨*Tests: public*⟩≡

```
public :: test_results_t
```

⟨*Tests: types*⟩≡

```
type :: test_result_t
   logical :: success = .false.
   type(string_t) :: name
   type(string_t) :: description
   type(test_result_t), pointer :: next => null ()
end type test_result_t

type :: test_results_t
   private
   type(test_result_t), pointer :: first => null ()
   type(test_result_t), pointer :: last => null ()
   integer :: n_success = 0
   integer :: n_failure = 0
 contains
 ⟨Tests: test results: TBP⟩
end type test_results_t
```

Add a test result.

⟨*Tests: test results: TBP*⟩≡

```
procedure, private :: add => test_results_add
```

⟨*Tests: procedures*⟩≡

```
subroutine test_results_add (list, name, description, success)
  class(test_results_t), intent(inout) :: list
  character(len=*), intent(in) :: name
  character(len=*), intent(in) :: description
  logical, intent(in) :: success
  type(test_result_t), pointer :: result
  allocate (result)
  result%success = success
  result%name = name
  result%description = description
  if (associated (list%first)) then
     list%last%next => result
  else
     list%first => result
  end if
  list%last => result
  if (success) then
     list%n_success = list%n_success + 1
  else
     list%n_failure = list%n_failure + 1
  end if
end subroutine test_results_add
```

Display the current state.

⟨*Tests: test results: TBP*⟩+≡

```
procedure, private :: write => test_results_write
```

⟨*Tests: procedures*⟩+≡

```
subroutine test_results_write (list, u)
  class(test_results_t), intent(in) :: list
  integer, intent(in) :: u
  type(test_result_t), pointer :: result
  write (u, "(A)")  "*** Test Summary ***"
  if (list%n_success > 0) then
     write (u, "(2x,A)")  "Success:"
     result => list%first
     do while (associated (result))
        if (result%success)  write (u, "(4x,A,': ',A)")  &
             char (result%name), char (result%description)
        result => result%next
     end do
  end if
  if (list%n_failure > 0) then
     write (u, "(2x,A)")  "Failure:"
     result => list%first
     do while (associated (result))
        if (.not. result%success)  write (u, "(4x,A,': ',A)")  &
             char (result%name), char (result%description)
```

```
        result => result%next
      end do
    end if
    write (u, "(A,I0)")  "Total   = ", list%n_success + list%n_failure
    write (u, "(A,I0)")  "Success = ", list%n_success
    write (u, "(A,I0)")  "Failure = ", list%n_failure
    write (u, "(A)")  "*** End of test Summary ***"
  end subroutine test_results_write
```

Return true if all tests were successful (or no test).

⟨*Tests: test results: TBP*⟩+≡

```
  procedure, private :: report => test_results_report
```

⟨*Tests: procedures*⟩+≡

```
  subroutine test_results_report (list, success)
    class(test_results_t), intent(in) :: list
    logical, intent(out) :: success
    success = list%n_failure == 0
  end subroutine test_results_report
```

Delete the list.

⟨*Tests: test results: TBP*⟩+≡

```
  procedure, private :: final => test_results_final
```

⟨*Tests: procedures*⟩+≡

```
  subroutine test_results_final (list)
    class(test_results_t), intent(inout) :: list
    type(test_result_t), pointer :: result
    do while (associated (list%first))
       result => list%first
       list%first => result%next
       deallocate (result)
    end do
    list%last => null ()
    list%n_success = 0
    list%n_failure = 0
  end subroutine test_results_final
```

4.1.3 Wrapup

This will write results, report status, and finalize. This is the only method which we need to access from outside.

⟨*Tests: test results: TBP*⟩+≡

```
  procedure :: wrapup => test_results_wrapup
```

⟨*Tests: procedures*⟩+≡

```
  subroutine test_results_wrapup (list, u, success)
    class(test_results_t), intent(inout) :: list
    integer, intent(in) :: u
    logical, intent(out), optional :: success
    call list%write (u)
    if (present (success))  call list%report (success)
```

```
    call list%final ()
  end subroutine test_results_wrapup
```

4.1.4 Tool for Unit Tests

This procedure takes a test routine as an argument. It runs the test, output directed to a temporary file. Then, it compares the file against a reference file.

The test routine must take the output unit as argument. We export this abstract interface, so the test drivers can reference it for declaring the actual test routines.

⟨*Tests: public*⟩+≡

```
  public :: unit_test
```

⟨*Tests: interfaces*⟩≡

```
  abstract interface
     subroutine unit_test (u)
       integer, intent(in) :: u
     end subroutine unit_test
  end interface
```

The test routine can print to screen and, optionally, to a logging unit.

⟨*Tests: public*⟩+≡

```
  public :: test
```

⟨*Tests: procedures*⟩+≡

```
  subroutine test (test_proc, name, description, u_log, results)
    procedure(unit_test) :: test_proc
    character(*), intent(in) :: name
    character(*), intent(in) :: description
    integer, intent(in) :: u_log
    type(test_results_t), intent(inout) :: results
    integer :: u_test, u_ref, u_err
    logical :: exist
    character(256) :: buffer1, buffer2
    integer :: iostat1, iostat2
    logical :: success
    write (*, "(A)", advance="no")  "Running test: " // name
    write (u_log, "(A)")  "Test: " // name
    u_test = free_unit ()
    open (u_test, status="scratch", action="readwrite")
    call test_proc (u_test)
    rewind (u_test)
    inquire (file=ref_prefix//name//ref, exist=exist)
    if (exist) then
       u_ref = free_unit ()
       open (u_ref, file=ref_prefix//name//ref, status="old", action="read")
       COMPARE_FILES: do
          read (u_test, "(A)", iostat=iostat1)  buffer1
          read (u_ref, "(A)", iostat=iostat2)  buffer2
          if (iostat1 /= iostat2) then
             success = .false.
             exit COMPARE_FILES
```

```
         else if (iostat1 < 0) then
            success = .true.
            exit COMPARE_FILES
         else if (buffer1 /= buffer2) then
            success = .false.
            exit COMPARE_FILES
         end if
      end do COMPARE_FILES
      close (u_ref)
   else
      write (*, "(A)", advance="no") " ... no reference output available"
      write (u_log, "(A)") "  No reference output available."
      success = .false.
   end if
   if (success) then
      write (*, "(A)") " ... success."
      write (u_log, "(A)")  "  Success."
   else
      write (*, "(A)") " ... failure.  See: " // err_prefix//name//err
      write (u_log, "(A)")  "  Failure."
      rewind (u_test)
      u_err = free_unit ()
      open (u_err, file=err_prefix//name//err, &
           action="write", status="replace")
      WRITE_OUTPUT: do
         read (u_test, "(A)", end=1) buffer1
         write (u_err, "(A)")  trim (buffer1)
      end do WRITE_OUTPUT
1     close (u_err)
   end if
   close (u_test)
   call results%add (name, description, success)
 end subroutine test
```

Chapter 5

System: Interfaces and Handlers

Here, we collect modules that deal with the "system": operating-system interfaces, error handlers and diagnostics.

system_defs Constants relevant for the modules in this set.

diagnostics Error and diagnostic message handling. Any messages and errors issued by WHIZARD functions are handled by the subroutines in this module, if possible.

os_interface Execute system calls, build and link external object files and libraries.

cputime Timer data type and methods, for measuring performance.

5.1 Constants

The parameters here are used in various parts of the program, starting from the modules in the current chapter. Some of them may be modified if the need arises.

⟨`system_defs.f90`⟩≡

```
⟨File header⟩

module system_defs

  use, intrinsic :: iso_fortran_env, only: iostat_end, iostat_eor !NODEP!

⟨Standard module head⟩

⟨System defs: public parameters⟩

end module system_defs
```

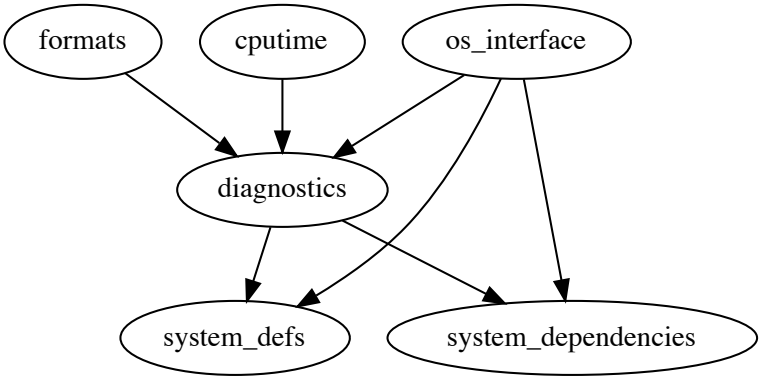

Figure 5.1: Module dependencies in `src/system`.

5.1.1 Version

The version string is used for checking files. Note that the string length MUST NOT be changed, because reading binary files relies on it.

⟨*System defs: public parameters*⟩≡

```
integer, parameter, public :: VERSION_STRLEN = 255
character(len=VERSION_STRLEN), parameter, public :: &
     & VERSION_STRING = "WHIZARD version ⟨Version⟩ (⟨Date⟩)"
```

5.1.2 Text Buffer

There is a hard limit on the line length which we should export. This buffer size is used both by the message handler, the lexer, and some further modules.

⟨*System defs: public parameters*⟩+≡

```
integer, parameter, public :: BUFFER_SIZE = 1000
```

5.1.3 IOSTAT Codes

Defined in `iso_fortran_env`, but we would like to use shorthands.

⟨*System defs: public parameters*⟩+≡

```
integer, parameter, public :: EOF = iostat_end,  EOR = iostat_eor
```

5.1.4 Character Codes

Single-character constants.

⟨*System defs: public parameters*⟩+≡

```
character, parameter, public :: BLANK = ' '
character, parameter, public :: TAB = achar(9)
character, parameter, public :: CR = achar(13)
character, parameter, public :: LF = achar(10)
character, parameter, public :: BACKSLASH = achar(92)
```

Character strings that indicate character classes.

⟨*System defs: public parameters*⟩+≡

```
character(*), parameter, public :: WHITESPACE_CHARS = BLANK// TAB // CR // LF
character(*), parameter, public :: LCLETTERS = "abcdefghijklmnopqrstuvwxyz"
character(*), parameter, public :: UCLETTERS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
character(*), parameter, public :: DIGITS = "0123456789"
```

5.2 C wrapper for sigaction

This implements calls to `sigaction` and the appropriate signal handlers in C.
The functionality is needed for the `diagnostics` module.

⟨`signal_interface.c`⟩≡

```
/*
⟨File header⟩
*/
#include <signal.h>
#include <stdlib.h>

extern int wo_sigint;
extern int wo_sigterm;
extern int wo_sigxcpu;
extern int wo_sigxfsz;

static void wo_handler_sigint (int sig) {
  wo_sigint = sig;
}

static void wo_handler_sigterm (int sig) {
  wo_sigterm = sig;
}

static void wo_handler_sigxcpu (int sig) {
  wo_sigxcpu = sig;
}

static void wo_handler_sigxfsz (int sig) {
  wo_sigxfsz = sig;
}

int wo_mask_sigint () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
```

```
  sa.sa_handler = wo_handler_sigint;
  return sigaction(SIGINT, &sa, NULL);
}

int wo_mask_sigterm () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = wo_handler_sigterm;
  return sigaction(SIGTERM, &sa, NULL);
}

int wo_mask_sigxcpu () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = wo_handler_sigxcpu;
  return sigaction(SIGXCPU, &sa, NULL);
}

int wo_mask_sigxfsz () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = wo_handler_sigxfsz;
  return sigaction(SIGXFSZ, &sa, NULL);
}

int wo_release_sigint () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = SIG_DFL;
  return sigaction(SIGINT, &sa, NULL);
}

int wo_release_sigterm () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = SIG_DFL;
  return sigaction(SIGTERM, &sa, NULL);
}
```

```
int wo_release_sigxcpu () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = SIG_DFL;
  return sigaction(SIGXCPU, &sa, NULL);
}

int wo_release_sigxfsz () {
  struct sigaction sa;
  sigset_t blocks;
  sigfillset (&blocks);
  sa.sa_flags = 0;
  sa.sa_mask = blocks;
  sa.sa_handler = SIG_DFL;
  return sigaction(SIGXFSZ, &sa, NULL);
}
```

5.3 C wrapper for printf

The `printf` family of functions is implemented in C with an undefined number of arguments. This is not supported by the `bind(C)` interface. We therefore write wrappers for the versions of `sprintf` that we will actually use.

This is used by the `formats` module.

⟨`sprintf_interface.c`⟩≡

```
/*
⟨File header⟩
*/
#include <stdio.h>

int sprintf_none(char* str, const char* format) {
  return sprintf(str, format);
}

int sprintf_int(char* str, const char* format, int val) {
  return sprintf(str, format, val);
}

int sprintf_double(char* str, const char* format, double val) {
  return sprintf(str, format, val);
}

int sprintf_str(char* str, const char* format, const char* val) {
  return sprintf(str, format, val);
}
```

⟨*sprintf interfaces*⟩≡

```
  interface
    function sprintf_none (str, fmt) result (stat) bind(C)
      use iso_c_binding !NODEP!
```

```
    integer(c_int) :: stat
    character(c_char), dimension(*), intent(inout) :: str
    character(c_char), dimension(*), intent(in) :: fmt
  end function sprintf_none
end interface

interface
  function sprintf_int (str, fmt, val) result (stat) bind(C)
    use iso_c_binding !NODEP!
    integer(c_int) :: stat
    character(c_char), dimension(*), intent(inout) :: str
    character(c_char), dimension(*), intent(in) :: fmt
    integer(c_int), value :: val
  end function sprintf_int
end interface

interface
  function sprintf_double (str, fmt, val) result (stat) bind(C)
    use iso_c_binding !NODEP!
    integer(c_int) :: stat
    character(c_char), dimension(*), intent(inout) :: str
    character(c_char), dimension(*), intent(in) :: fmt
    real(c_double), value :: val
  end function sprintf_double
end interface

interface
  function sprintf_str(str, fmt, val) result (stat) bind(C)
    use iso_c_binding !NODEP!
    integer(c_int) :: stat
    character(c_char), dimension(*), intent(inout) :: str
    character(c_char), dimension(*), intent(in) :: fmt
    character(c_char), dimension(*), intent(in) :: val
  end function sprintf_str
end interface
```

5.4 Error, Message and Signal Handling

We are not so ambitious as to do proper exception handling in WHIZARD, but at least it may be useful to have a common interface for diagnostics: Results, messages, warnings, and such. As module variables we keep a buffer where the current message may be written to and a level indicator which tells which messages should be written on screen and which ones should be skipped. Alternatively, a string may be directly supplied to the message routine: this overrides the buffer, avoiding the necessety of formatted I/O in trivial cases.

⟨diagnostics.f90⟩≡
  ⟨*File header*⟩

```
module diagnostics

  use, intrinsic :: iso_c_binding !NODEP!
```

```
  use, intrinsic :: iso_fortran_env, only: output_unit !NODEP!

⟨Use kinds⟩
⟨Use strings⟩
  use string_utils, only: str
  use io_units

  use system_dependencies
  use system_defs, only: BUFFER_SIZE, MAX_ERRORS

⟨Standard module head⟩

⟨Diagnostics: public⟩

⟨Diagnostics: parameters⟩

⟨Diagnostics: types⟩

⟨Diagnostics: variables⟩

⟨Diagnostics: interfaces⟩

contains

⟨Diagnostics: procedures⟩

end module diagnostics
```

Diagnostics levels:

⟨*Diagnostics: public*⟩≡

```
  public :: RESULT, DEBUG, DEBUG2
```

⟨*Diagnostics: parameters*⟩≡

```
  integer, parameter :: TERMINATE=-2, BUG=-1, FATAL=1, &
       ERROR=2, WARNING=3, MESSAGE=4, RESULT=5, &
       DEBUG=6, DEBUG2=7
```

Diagnostics areas:

⟨*Diagnostics: public*⟩+≡

```
  public :: d_area
```

⟨*Diagnostics: interfaces*⟩≡

```
  interface d_area
     module procedure d_area_of_string
     module procedure d_area_to_string
  end interface
```

⟨*Diagnostics: procedures*⟩≡

```
  function d_area_of_string (string) result (i)
    integer :: i
    type(string_t), intent(in) :: string
    select case (char (string))
    case ("particles")
       i = D_PARTICLES
    case ("events")
       i = D_EVENTS
    case ("shower")
```

```
      i = D_SHOWER
   case ("model_features")
      i = D_MODEL_F
   case ("matching")
      i = D_MATCHING
   case ("transforms")
      i = D_TRANSFORMS
   case ("subtraction")
      i = D_SUBTRACTION
   case ("virtual")
      i = D_VIRTUAL
   case ("threshold")
      i = D_THRESHOLD
   case ("phasespace")
      i = D_PHASESPACE
   case ("mismatch")
      i = D_MISMATCH
   case ("me_methods")
      i = D_ME_METHODS
   case ("process_integration")
      i = D_PROCESS_INTEGRATION
   case ("tauola")
      i = D_TAUOLA
   case ("core")
      i = D_CORE
   case default
      print "(A)", "Possible values for --debug are:"
      do i = 1, D_LAST
         print "(A)", char ('  ' // d_area_to_string(i))
      end do
      call msg_fatal ("Please use one of the listed areas")
   end select
end function d_area_of_string

elemental function d_area_to_string (i) result (string)
   type(string_t) :: string
   integer, intent(in) :: i
   select case (i)
   case (D_PARTICLES)
      string = "particles"
   case (D_EVENTS)
      string = "events"
   case (D_SHOWER)
      string = "shower"
   case (D_MODEL_F)
      string = "model_features"
   case (D_MATCHING)
      string = "matching"
   case (D_TRANSFORMS)
      string = "transforms"
   case (D_SUBTRACTION)
      string = "subtraction"
   case (D_VIRTUAL)
      string = "virtual"
```

```
      case (D_THRESHOLD)
         string = "threshold"
      case (D_PHASESPACE)
         string = "phasespace"
      case (D_MISMATCH)
         string = "mismatch"
      case (D_ME_METHODS)
         string = "me_methods"
      case (D_PROCESS_INTEGRATION)
         string = "process_integration"
      case (D_TAUOLA)
         string = "tauola"
      case (D_CORE)
         string = "core"
      case default
         string = "undefined"
      end select
   end function d_area_to_string
```

⟨*Diagnostics: public*⟩+≡

```
   public :: D_PARTICLES, D_EVENTS, D_SHOWER, D_MODEL_F, &
        D_MATCHING, D_TRANSFORMS, D_SUBTRACTION, D_VIRTUAL, D_THRESHOLD, &
        D_PHASESPACE, D_MISMATCH, D_ME_METHODS, D_PROCESS_INTEGRATION, &
        D_TAUOLA, D_CORE
```

⟨*Diagnostics: parameters*⟩+≡

```
   integer, parameter :: D_ALL=0, D_PARTICLES=1, D_EVENTS=2, &
        D_SHOWER=3, D_MODEL_F=4, &
        D_MATCHING=5, D_TRANSFORMS=6, &
        D_SUBTRACTION=7, D_VIRTUAL=8, D_THRESHOLD=9, D_PHASESPACE=10, &
        D_MISMATCH=11, D_ME_METHODS=12, D_PROCESS_INTEGRATION=13, &
        D_TAUOLA=14, D_CORE=15, &
        D_LAST=15
```

⟨*Diagnostics: public*⟩+≡

```
   public :: msg_level
```

⟨*Diagnostics: variables*⟩≡

```
   integer, save, dimension(D_ALL:D_LAST) :: msg_level = RESULT
```

⟨*Diagnostics: parameters*⟩+≡

```
   integer, parameter, public :: COL_UNDEFINED = -1
   integer, parameter, public :: COL_GREY = 90, COL_PEACH = 91, COL_LIGHT_GREEN = 92, &
      COL_LIGHT_YELLOW = 93, COL_LIGHT_BLUE = 94, COL_PINK = 95, &
      COL_LIGHT_AQUA = 96, COL_PEARL_WHITE = 97, COL_BLACK = 30, &
      COL_RED = 31, COL_GREEN = 32, COL_YELLOW = 33, COL_BLUE = 34, &
      COL_PURPLE = 35, COL_AQUA = 36
```

⟨*Diagnostics: types*⟩≡

```
   type :: terminal_color_t
      integer :: color = COL_UNDEFINED
   contains
   ⟨Diagnostics: terminal color: TBP⟩
   end type terminal_color_t
```

⟨*Diagnostics: public*⟩+≡

```
public :: term_col
```

⟨*Diagnostics: interfaces*⟩+≡

```
interface term_col
   module procedure term_col_int
   module procedure term_col_char
end interface term_col
```

⟨*Diagnostics: procedures*⟩+≡

```
function term_col_int (col_int) result (color)
  type(terminal_color_t) :: color
  integer, intent(in) :: col_int
  color%color = col_int
end function term_col_int

function term_col_char (col_char) result (color)
  type(terminal_color_t) :: color
  character(len=*), intent(in) :: col_char
  type(string_t) :: buf
  select case (col_char)
  case ('Grey')
     color%color = COL_GREY
  case ('Peach')
     color%color = COL_PEACH
  case ('Light Green')
     color%color = COL_LIGHT_GREEN
  case ('Light Yellow')
     color%color = COL_LIGHT_YELLOW
  case ('Light Blue')
     color%color = COL_LIGHT_BLUE
  case ('Pink')
     color%color = COL_PINK
  case ('Light Aqua')
     color%color = COL_LIGHT_AQUA
  case ('Pearl White')
     color%color = COL_PEARL_WHITE
  case ('Black')
     color%color = COL_BLACK
  case ('Red')
     color%color = COL_RED
  case ('Green')
     color%color = COL_GREEN
  case ('Yellow')
     color%color = COL_YELLOW
  case ('Blue')
     color%color = COL_BLUE
  case ('Purple')
     color%color = COL_PURPLE
  case ('Aqua')
     color%color = COL_AQUA
  case default
     buf = var_str ('Color ') // var_str (col_char) // var_str (' is not defined')
     call msg_warning (char (buf))
```

```
      color%color = COL_UNDEFINED
    end select
  end function term_col_char
```

Mask fatal errors so that are treated as normal errors. Useful for interactive mode.

⟨*Diagnostics: public*⟩+≡
```
  public :: mask_fatal_errors
```

⟨*Diagnostics: variables*⟩+≡
```
  logical, save :: mask_fatal_errors = .false.
```

How to handle bugs and unmasked fatal errors. Either execute a normal stop statement, or call the C `exit()` function, or try to cause a program crash by dereferencing a null pointer.

⟨*Diagnostics: parameters*⟩+≡
```
  integer, parameter :: TERM_STOP = 0, TERM_EXIT = 1, TERM_CRASH = 2
```

⟨*Diagnostics: variables*⟩+≡
```
  integer, save :: handle_fatal_errors = TERM_EXIT
```

Keep track of errors. This might be used for exception handling, later. The counter is incremented only for screen messages, to avoid double counting.

⟨*Diagnostics: public*⟩+≡
```
  public :: msg_count
```

⟨*Diagnostics: variables*⟩+≡
```
  integer, dimension(TERMINATE:WARNING), save :: msg_count = 0
```

Keep a list of all errors and warnings. Since we do not know the number of entries beforehand, we use a linked list.

⟨*Diagnostics: types*⟩+≡
```
  type :: string_list
     character(len=BUFFER_SIZE) :: string
     type(string_list), pointer :: next
  end type string_list
  type :: string_list_pointer
     type(string_list), pointer :: first, last
  end type string_list_pointer
```

⟨*Diagnostics: variables*⟩+≡
```
  type(string_list_pointer), dimension(TERMINATE:WARNING), save :: &
       & msg_list = string_list_pointer (null(), null())
```

Create a format string indicating color
Add the current message buffer contents to the internal list.

⟨*Diagnostics: procedures*⟩+≡
```
  subroutine msg_add (level)
    integer, intent(in) :: level
    type(string_list), pointer :: message
    select case (level)
    case (TERMINATE:WARNING)
       allocate (message)
       message%string = msg_buffer
       nullify (message%next)
       if (.not.associated (msg_list(level)%first)) &
```

```
            & msg_list(level)%first => message
         if (associated (msg_list(level)%last)) &
            & msg_list(level)%last%next => message
         msg_list(level)%last => message
         msg_count(level) = msg_count(level) + 1
      end select
   end subroutine msg_add
```

Initialization:

⟨*Diagnostics: public*⟩+≡

```
   public :: msg_list_clear
```

⟨*Diagnostics: procedures*⟩+≡

```
   subroutine msg_list_clear
      integer :: level
      type(string_list), pointer :: message
      do level = TERMINATE, WARNING
         do while (associated (msg_list(level)%first))
            message => msg_list(level)%first
            msg_list(level)%first => message%next
            deallocate (message)
         end do
         nullify (msg_list(level)%last)
      end do
      msg_count = 0
   end subroutine msg_list_clear
```

Display the summary of errors and warnings (no need to count fatals...)

⟨*Diagnostics: public*⟩+≡

```
   public :: msg_summary
```

⟨*Diagnostics: procedures*⟩+≡

```
   subroutine msg_summary (unit)
      integer, intent(in), optional :: unit
      call expect_summary (unit)
  1   format (A,1x,I2,1x,A,I2,1x,A)
      if (msg_count(ERROR) > 0 .and. msg_count(WARNING) > 0) then
         write (msg_buffer, 1) "There were", &
            & msg_count(ERROR), "error(s) and  ", &
            & msg_count(WARNING), "warning(s)."
         call msg_message (unit=unit)
      else if (msg_count(ERROR) > 0) then
         write (msg_buffer, 1) "There were", &
            & msg_count(ERROR), "error(s) and no warnings."
         call msg_message (unit=unit)
      else if (msg_count(WARNING) > 0) then
         write (msg_buffer, 1) "There were no errors and  ", &
            & msg_count(WARNING), "warning(s)."
         call msg_message (unit=unit)
      end if
   end subroutine msg_summary
```

Print the list of all messages of a given level.

⟨*Diagnostics: public*⟩+≡
```
public :: msg_listing
```

⟨*Diagnostics: procedures*⟩+≡
```
subroutine msg_listing (level, unit, prefix)
  integer, intent(in) :: level
  integer, intent(in), optional :: unit
  character(len=*), intent(in), optional :: prefix
  type(string_list), pointer :: message
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  if (present (unit))  u = unit
  message => msg_list(level)%first
  do while (associated (message))
     if (present (prefix)) then
        write (u, "(A)") prefix // trim (message%string)
     else
        write (u, "(A)") trim (message%string)
     end if
     message => message%next
  end do
  flush (u)
end subroutine msg_listing
```

The message buffer:

⟨*Diagnostics: public*⟩+≡
```
public :: msg_buffer
```

⟨*Diagnostics: variables*⟩+≡
```
character(len=BUFFER_SIZE), save :: msg_buffer = " "
```

After a message is issued, the buffer should be cleared:

⟨*Diagnostics: procedures*⟩+≡
```
subroutine buffer_clear
  msg_buffer = " "
end subroutine buffer_clear
```

⟨*Diagnostics: procedures*⟩+≡
```
function create_col_string (color) result (col_string)
   type(string_t) :: col_string
   integer, intent(in) :: color
   character(2) :: buf
   write (buf, '(I2)') color
   col_string = var_str ("[") // var_str (buf) // var_str ("m")
end function create_col_string
```

The generic handler for messages. If the unit is omitted (or = 6), the message is written to standard output if the precedence if sufficiently high (as determined by the value of `msg_level`). If the string is omitted, the buffer is used. In any case, the buffer is cleared after printing. In accordance with FORTRAN custom, the first column in the output is left blank. For messages and warnings,

an additional exclamation mark and a blank is prepended. Furthermore, each message is appended to the internal message list (without prepending anything).

⟨*Diagnostics: procedures*⟩+≡

```
subroutine message_print (level, string, str_arr, unit, logfile, area, color)
  integer, intent(in) :: level
  character(len=*), intent(in), optional :: string
  type(string_t), dimension(:), intent(in), optional :: str_arr
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: logfile
  integer, intent(in), optional :: area
  integer, intent(in), optional :: color
  type(string_t) :: col_string, prep_string, aux_string, head_footer, app_string
  integer :: lu, i, ar
  logical :: severe, is_error
  ar = D_ALL; if (present (area))  ar = area
  severe = .false.
  head_footer  = "*******************************************************************************
  aux_string = ""
  is_error = .false.
  app_string = ""
  select case (level)
  case (TERMINATE)
     prep_string = ""
  case (BUG)
     prep_string   = "*** WHIZARD BUG: "
     aux_string    = "***              "
     severe = .true.
     is_error = .true.
  case (FATAL)
     prep_string   = "*** FATAL ERROR: "
     aux_string    = "***              "
     severe = .true.
     is_error = .true.
  case (ERROR)
     prep_string = "*** ERROR: "
     aux_string  = "***        "
     is_error = .true.
  case (WARNING)
     prep_string = "Warning: "
  case (MESSAGE)
     prep_string = "| "
  case (DEBUG, DEBUG2)
     prep_string = "D: "
  case default
     prep_string = ""
  end select
  if (present (color)) then
     if (color > COL_UNDEFINED) then
        col_string = create_col_string (color)
        prep_string = achar(27) // col_string // prep_string
        app_string = app_string // achar(27) // "[0m"
     end if
  end if
  if (present(string))  msg_buffer = string
```

```
lu = log_unit
if (present(unit)) then
   if (unit /= output_unit) then
      if (severe) write (unit, "(A)") char(head_footer)
      if (is_error) write (unit, "(A)") char(head_footer)
      write (unit, "(A,A,A)") char(prep_string), trim(msg_buffer), &
           char(app_string)
      if (present (str_arr)) then
         do i = 1, size(str_arr)
            write (unit, "(A,A)") char(aux_string), char(trim(str_arr(i)))
         end do
      end if
      if (is_error) write (unit, "(A)") char(head_footer)
      if (severe) write (unit, "(A)") char(head_footer)
      flush (unit)
      lu = -1
   else if (level <= msg_level(ar)) then
      if (severe) print "(A)", char(head_footer)
      if (is_error) print "(A)", char(head_footer)
      print "(A,A,A)", char(prep_string), trim(msg_buffer), &
           char(app_string)
      if (present (str_arr)) then
         do i = 1, size(str_arr)
            print "(A,A)", char(aux_string), char(trim(str_arr(i)))
         end do
      end if
      if (is_error) print "(A)", char(head_footer)
      if (severe) print "(A)", char(head_footer)
      flush (output_unit)
      if (unit == log_unit)  lu = -1
   end if
else if (level <= msg_level(ar)) then
   if (severe) print "(A)", char(head_footer)
   if (is_error) print "(A)", char(head_footer)
   print "(A,A,A)", char(prep_string), trim(msg_buffer), &
           char(app_string)
      if (present (str_arr)) then
         do i = 1, size(str_arr)
            print "(A,A)", char(aux_string), char(trim(str_arr(i)))
         end do
      end if
   if (is_error) print "(A)", char(head_footer)
   if (severe) print "(A)", char(head_footer)
   flush (output_unit)
end if
if (present (logfile)) then
   if (.not. logfile)  lu = -1
end if
if (logging .and. lu >= 0) then
   if (severe) write (lu, "(A)") char(head_footer)
   if (is_error) write (lu, "(A)") char(head_footer)
   write (lu, "(A,A,A)")  char(prep_string), trim(msg_buffer), &
           char(app_string)
   if (present (str_arr)) then
```

```
        do i = 1, size(str_arr)
           write (lu, "(A,A)") char(aux_string), char(trim(str_arr(i)))
        end do
     end if
     if (is_error) write (lu, "(A)") char(head_footer)
     if (severe) write (lu, "(A)") char(head_footer)
     flush (lu)
  end if
  call msg_add (level)
  call buffer_clear
end subroutine message_print
```

The number of non-fatal errors that we allow before stopping the program. We might trade this later for an adjustable number.

⟨*System defs: public parameters*⟩+≡

```
integer, parameter, public :: MAX_ERRORS = 10
```

The specific handlers. In the case of fatal errors, bugs (failed assertions) and normal termination execution is stopped. For non-fatal errors a message is printed to standard output if no unit is given. Only if the number of MAX_ERRORS errors is reached, we abort the program. There are no further actions in the other cases, but this may change.

⟨*Diagnostics: public*⟩+≡

```
public :: msg_terminate
public :: msg_bug, msg_fatal, msg_error, msg_warning
public :: msg_message, msg_result
```

⟨*Diagnostics: procedures*⟩+≡

```
subroutine msg_terminate (string, unit, quit_code)
  integer, intent(in), optional :: unit
  character(len=*), intent(in), optional :: string
  integer, intent(in), optional :: quit_code
  integer(c_int) :: return_code
  call release_term_signals ()
  if (present (quit_code)) then
     return_code = quit_code
  else
     return_code = 0
  end if
  if (present (string)) &
       call message_print (MESSAGE, string, unit=unit)
  call msg_summary (unit)
  if (return_code == 0 .and. expect_failures /= 0) then
     return_code = 5
     call message_print (MESSAGE, &
          "WHIZARD run finished with 'expect' failure(s).", unit=unit)
  else if (return_code == 7) then
     call message_print (MESSAGE, &
          "WHIZARD run finished with failed self-test.", unit=unit)
  else
     call message_print (MESSAGE, "WHIZARD run finished.", unit=unit)
  end if
  call message_print (0, &
       "|==============================================================================|", unit=u
```

```
  call logfile_final ()
  call msg_list_clear ()
  if (return_code /= 0) then
     call exit (return_code)
  else
     !!! Should implement WHIZARD exit code (currently only via C)
     ! stop
     call exit (0)
  end if
end subroutine msg_terminate

subroutine msg_bug (string, arr, unit)
  integer, intent(in), optional :: unit
  character(len=*), intent(in), optional :: string
  type(string_t), dimension(:), intent(in), optional :: arr
  logical, pointer :: crash_ptr
  call message_print (BUG, string, arr, unit)
  call msg_summary (unit)
  select case (handle_fatal_errors)
  case (TERM_EXIT)
     call message_print (TERMINATE, "WHIZARD run aborted.", unit=unit)
     call exit (-1_c_int)
  case (TERM_CRASH)
     print *, "*** Intentional crash ***"
     crash_ptr => null ()
     print *, crash_ptr
  end select
  stop "WHIZARD run aborted."
end subroutine msg_bug

recursive subroutine msg_fatal (string, arr, unit)
  integer, intent(in), optional :: unit
  character(len=*), intent(in), optional :: string
  type(string_t), dimension(:), intent(in), optional :: arr
  logical, pointer :: crash_ptr
  if (mask_fatal_errors) then
     call msg_error (string, arr, unit)
  else
     call message_print (FATAL, string, arr, unit)
     call msg_summary (unit)
     select case (handle_fatal_errors)
     case (TERM_EXIT)
        call message_print (TERMINATE, "WHIZARD run aborted.", unit=unit)
        call exit (1_c_int)
     case (TERM_CRASH)
        print *, "*** Intentional crash ***"
        crash_ptr => null ()
        print *, crash_ptr
     end select
     stop "WHIZARD run aborted."
  end if
end subroutine msg_fatal

subroutine msg_error (string, arr, unit)
```

```
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    call message_print (ERROR, string, arr, unit)
    if (msg_count(ERROR) >= MAX_ERRORS) then
       mask_fatal_errors = .false.
       call msg_fatal (" Too many errors encountered.")
    else if (.not.present(unit) .and. .not.mask_fatal_errors)  then
       call message_print (MESSAGE, "            (WHIZARD run continues)")
    end if
  end subroutine msg_error

  subroutine msg_warning (string, arr, unit, color)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    type(terminal_color_t), intent(in), optional :: color
    integer :: cl
    cl = COL_UNDEFINED; if (present (color)) cl = color%color
    call message_print (level = WARNING, string = string, &
       str_arr = arr, unit = unit, color = cl)
  end subroutine msg_warning

  subroutine msg_message (string, unit, arr, logfile, color)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    logical, intent(in), optional :: logfile
    type(terminal_color_t), intent(in), optional :: color
    integer :: cl
    cl = COL_UNDEFINED; if (present (color)) cl = color%color
    call message_print (level = MESSAGE, &
       string = string, str_arr = arr, unit = unit, &
       logfile = logfile, color = cl)
  end subroutine msg_message

  subroutine msg_result (string, arr, unit, logfile, color)
    integer, intent(in), optional :: unit
    character(len=*), intent(in), optional :: string
    type(string_t), dimension(:), intent(in), optional :: arr
    logical, intent(in), optional :: logfile
    type(terminal_color_t), intent(in), optional :: color
    integer :: cl
    cl = COL_UNDEFINED; if (present (color)) cl = color%color
    call message_print (level = RESULT, string = string, &
       str_arr = arr, unit = unit, logfile = logfile, color = cl)
  end subroutine msg_result
```

⟨*Diagnostics: public*⟩+≡

```
  public :: msg_debug
```

⟨*Diagnostics: interfaces*⟩+≡

```
  interface msg_debug
     module procedure msg_debug_none
```

```
    module procedure msg_debug_logical
    module procedure msg_debug_integer
    module procedure msg_debug_real
    module procedure msg_debug_complex
    module procedure msg_debug_string
  end interface
```

⟨*Diagnostics: procedures*⟩+≡

```
  subroutine msg_debug_none (area, string, color)
    integer, intent(in) :: area
    character(len=*), intent(in), optional :: string
    type(terminal_color_t), intent(in), optional :: color
    integer :: cl
    cl = COL_BLUE; if (present (color)) cl = color%color
    call message_print (DEBUG, string, unit = output_unit, &
         area = area, logfile = .false., color = cl)
  end subroutine msg_debug_none

  subroutine msg_debug_logical (area, string, value, color)
    logical, intent(in) :: value
  ⟨msg debug implementation⟩
  end subroutine msg_debug_logical

  subroutine msg_debug_integer (area, string, value, color)
    integer, intent(in) :: value
  ⟨msg debug implementation⟩
  end subroutine msg_debug_integer

  subroutine msg_debug_real (area, string, value, color)
    real(default), intent(in) :: value
  ⟨msg debug implementation⟩
  end subroutine msg_debug_real

  subroutine msg_debug_complex (area, string, value, color)
    complex(default), intent(in) :: value
  ⟨msg debug implementation⟩
  end subroutine msg_debug_complex

  subroutine msg_debug_string (area, string, value, color)
    type(string_t), intent(in) :: value
    integer, intent(in) :: area
    character(len=*), intent(in) :: string
    type(terminal_color_t), intent(in), optional :: color
    call msg_debug_none (area, char (string // " = " // value), &
         color = color)
  end subroutine msg_debug_string
```

⟨*msg debug implementation*⟩≡

```
  integer, intent(in) :: area
  character(len=*), intent(in) :: string
  type(terminal_color_t), intent(in), optional :: color
  call msg_debug_none (area, char (string // " = " // str (value)), &
       color = color)
```

⟨*Diagnostics: public*⟩+≡

```
public :: msg_print_color
```

⟨*Diagnostics: interfaces*⟩+≡

```
interface msg_print_color
   module procedure msg_print_color_none
   module procedure msg_print_color_logical
   module procedure msg_print_color_integer
   module procedure msg_print_color_real
end interface
```

⟨*Diagnostics: procedures*⟩+≡

```
subroutine msg_print_color_none (string, color)
  character(len=*), intent(in) :: string
  !!!type(terminal_color_t), intent(in) :: color
  integer, intent(in) :: color
  call message_print (0, string, color = color)
end subroutine msg_print_color_none

subroutine msg_print_color_logical (string, value, color)
  character(len=*), intent(in) :: string
  logical, intent(in) :: value
  integer, intent(in) :: color
  call msg_print_color_none (char (string // " = " // str (value)), &
     color = color)
end subroutine msg_print_color_logical

subroutine msg_print_color_integer (string, value, color)
  character(len=*), intent(in) :: string
  integer, intent(in) :: value
  integer, intent(in) :: color
  call msg_print_color_none (char (string // " = " // str (value)), &
     color = color)
end subroutine msg_print_color_integer

subroutine msg_print_color_real (string, value, color)
  character(len=*), intent(in) :: string
  real(default), intent(in) :: value
  integer, intent(in) :: color
  call msg_print_color_none (char (string // " = " // str (value)), &
     color = color)
end subroutine msg_print_color_real
```

⟨*Diagnostics: public*⟩+≡

```
public :: msg_debug2
```

⟨*Diagnostics: interfaces*⟩+≡

```
interface msg_debug2
   module procedure msg_debug2_none
   module procedure msg_debug2_logical
   module procedure msg_debug2_integer
   module procedure msg_debug2_real
   module procedure msg_debug2_complex
   module procedure msg_debug2_string
end interface
```

⟨*Diagnostics: procedures*⟩+≡

```
subroutine msg_debug2_none (area, string, color)
  integer, intent(in) :: area
  character(len=*), intent(in), optional :: string
  type(terminal_color_t), intent(in), optional :: color
  integer :: cl
  cl = COL_BLUE; if (present (color)) cl = color%color
  call message_print (DEBUG2, string, unit = output_unit, &
       area = area, logfile = .false., color = cl)
end subroutine msg_debug2_none

subroutine msg_debug2_logical (area, string, value, color)
  logical, intent(in) :: value
⟨msg debug2 implementation⟩
end subroutine msg_debug2_logical

subroutine msg_debug2_integer (area, string, value, color)
  integer, intent(in) :: value
⟨msg debug2 implementation⟩
end subroutine msg_debug2_integer

subroutine msg_debug2_real (area, string, value, color)
  real(default), intent(in) :: value
⟨msg debug2 implementation⟩
end subroutine msg_debug2_real

subroutine msg_debug2_complex (area, string, value, color)
  complex(default), intent(in) :: value
⟨msg debug2 implementation⟩
end subroutine msg_debug2_complex

subroutine msg_debug2_string (area, string, value, color)
  type(string_t), intent(in) :: value
  integer, intent(in) :: area
  character(len=*), intent(in) :: string
  type(terminal_color_t), intent(in), optional :: color
  call msg_debug2_none (area, char (string // " = " // value), &
     color = color)
end subroutine msg_debug2_string
```

⟨*msg debug2 implementation*⟩≡

```
integer, intent(in) :: area
character(len=*), intent(in) :: string
type(terminal_color_t), intent(in), optional :: color
call msg_debug2_none (area, char (string // " = " // str (value)), &
   color = color)
```

⟨*Diagnostics: public*⟩+≡

```
public :: debug_active
```

⟨*Diagnostics: procedures*⟩+≡

```
elemental function debug_active (area) result (active)
  logical :: active
  integer, intent(in) :: area
  active = msg_level(area) >= DEBUG
```

```
    end function debug_active
```

⟨*Diagnostics: public*⟩+≡

```
    public :: debug2_active
```

⟨*Diagnostics: procedures*⟩+≡

```
    elemental function debug2_active (area) result (active)
      logical :: active
      integer, intent(in) :: area
      active = msg_level(area) >= DEBUG2
    end function debug2_active
```

Show the progress of a loop in steps of 10 %. Could be generalized to other step sizes with an optional argument.

⟨*Diagnostics: public*⟩+≡

```
    public :: msg_show_progress
```

⟨*Diagnostics: procedures*⟩+≡

```
    subroutine msg_show_progress (i_call, n_calls)
      integer, intent(in) :: i_call, n_calls
      real(default) :: progress
      integer, save :: next_check
      if (i_call == 1) next_check = 10
      progress = (i_call * 100._default) / n_calls
      if (progress >= next_check) then
         write (msg_buffer, "(F5.1,A)") progress, "%"
         call msg_message ()
         next_check = next_check + 10
      end if
    end subroutine msg_show_progress
```

Interface to the standard clib exit function

⟨*Diagnostics: interfaces*⟩+≡

```
    interface
       subroutine exit (status) bind (C)
         use iso_c_binding !NODEP!
         integer(c_int), value :: status
       end subroutine exit
    end interface
```

Print the WHIZARD banner:

⟨*Diagnostics: public*⟩+≡

```
    public :: msg_banner
```

⟨*Diagnostics: procedures*⟩+≡

```
    subroutine msg_banner (unit)
      integer, intent(in), optional :: unit
      call message_print (0, "|=====================================================================
      call message_print (0, "|
      call message_print (0, "|    WW              WW  WW   WW  WW  WWWWWW       WW      WWWWW    WWWW
      call message_print (0, "|     WW    WW      WW   WW   WW  WW      WW       WWWW     WW  WW   WW
      call message_print (0, "|      WW  WW WW  WW    WWWWWWW  WW     WW       WW  WW    WWWWW    WW
      call message_print (0, "|       WWWW   WWWW     WW   WW  WW   WW      WWWWWWW   WW  WW   WW
```

```
  call message_print (0, "|        WW     WW      WW   WW  WW  WWWWWW  WW      WW  WW   WW  WWWW
  call message_print (0, "|
  call message_print (0, "|
  call message_print (0, "|                                        W
  call message_print (0, "|                                       sW
  call message_print (0, "|                                       WW
  call message_print (0, "|                                      sWW
  call message_print (0, "|                                      WWW
  call message_print (0, "|                                     wWWW
  call message_print (0, "|                                    wWWWW
  call message_print (0, "|                                    WW WW
  call message_print (0, "|                                    WW WW
  call message_print (0, "|                                   wWW WW
  call message_print (0, "|                                  wWW  WW
  call message_print (0, "|                                  WW   WW
  call message_print (0, "|                                  WW   WW
  call message_print (0, "|                                 WW    WW
  call message_print (0, "|                                 WW    WW
  call message_print (0, "|                                WW     WW
  call message_print (0, "|                                WW     WW
  call message_print (0, "|           wwwwww               WW      WW
  call message_print (0, "|              WWWWWww           WW      WW
  call message_print (0, "|                 WWWWWwwwww   WW        WW
  call message_print (0, "|                     wWWWwwwwwWW        WW
  call message_print (0, "|                 wWWWWWWWWWWwWWW        WW
  call message_print (0, "|                wWWWWW        wW         WWWWWWWW
  call message_print (0, "|                  WWWW        wW         WW  wWWWWWWWWwww
  call message_print (0, "|                   WWWW                      wWWWWWWWWwwww
  call message_print (0, "|                     WWWW                      WWWW     WWw
  call message_print (0, "|                       WWWWww                   WWWW
  call message_print (0, "|                           WWWwwww              WWWW
  call message_print (0, "|                               wWWWWwww       wWWWWW
  call message_print (0, "|                                     WwwwwwwwwWWW
  call message_print (0, "|
  call message_print (0, "|
  call message_print (0, "|
  call message_print (0, "|  by:   Wolfgang Kilian, Thorsten Ohl, Juergen Reuter
  call message_print (0, "|        with contributions from Christian Speckner
  call message_print (0, "|        Contact: <whizard@desy.de>
  call message_print (0, "|
  call message_print (0, "|  if you use WHIZARD please cite:
  call message_print (0, "|        W. Kilian, T. Ohl, J. Reuter,  Eur.Phys.J.C71 (2011) 1742
  call message_print (0, "|                                          [arXiv: 0708.4233 [hep-ph]]
  call message_print (0, "|        M. Moretti, T. Ohl, J. Reuter, arXiv: hep-ph/0102195
  call message_print (0, "|
  call message_print (0, "|=====================================================================
  call message_print (0, "|                              WHIZARD " // WHIZARD_VERSION, unit=uni
  call message_print (0, "|=====================================================================
end subroutine msg_banner
```

5.4.1 Logfile

All screen output should be duplicated in the logfile, unless requested otherwise.

⟨*Diagnostics: public*⟩+≡
```
public :: logging
```
⟨*Diagnostics: variables*⟩+≡
```
integer, save :: log_unit = -1
logical, target, save :: logging = .false.
```
⟨*Diagnostics: public*⟩+≡
```
public :: logfile_init
```
⟨*Diagnostics: procedures*⟩+≡
```
subroutine logfile_init (filename)
  type(string_t), intent(in) :: filename
  call msg_message ("Writing log to '" // char (filename) // "'")
  if (.not. logging)  call msg_message ("(Logging turned off.)")
  log_unit = free_unit ()
  open (file = char (filename), unit = log_unit, &
        action = "write", status = "replace")
end subroutine logfile_init
```

⟨*Diagnostics: public*⟩+≡
```
public :: logfile_final
```
⟨*Diagnostics: procedures*⟩+≡
```
subroutine logfile_final ()
  if (log_unit >= 0) then
     close (log_unit)
     log_unit = -1
  end if
end subroutine logfile_final
```

This returns the valid logfile unit only if the default is write to screen, and if `logfile` is not set false.

⟨*Diagnostics: public*⟩+≡
```
public :: logfile_unit
```
⟨*Diagnostics: procedures*⟩+≡
```
function logfile_unit (unit, logfile)
  integer :: logfile_unit
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: logfile
  if (logging) then
     if (present (unit)) then
        if (unit == output_unit) then
           logfile_unit = log_unit
        else
           logfile_unit = -1
        end if
     else if (present (logfile)) then
        if (logfile) then
           logfile_unit = log_unit
        else
           logfile_unit = -1
        end if
     else
        logfile_unit = log_unit
```

```
      end if
    else
      logfile_unit = -1
    end if
  end function logfile_unit
```

5.4.2 Checking values

The `expect` function does not just check a value for correctness (actually, it checks if a logical expression is true); it records its result here. If failures are present when the program terminates, the exit code is nonzero.

⟨*Diagnostics: variables*⟩+≡

```
  integer, save :: expect_total = 0
  integer, save :: expect_failures = 0
```

⟨*Diagnostics: public*⟩+≡

```
  public :: expect_record
```

⟨*Diagnostics: procedures*⟩+≡

```
  subroutine expect_record (success)
    logical, intent(in) :: success
    expect_total = expect_total + 1
    if (.not. success)  expect_failures = expect_failures + 1
  end subroutine expect_record
```

⟨*Diagnostics: public*⟩+≡

```
  public :: expect_clear
```

⟨*Diagnostics: procedures*⟩+≡

```
  subroutine expect_clear ()
    expect_total = 0
    expect_failures = 0
  end subroutine expect_clear
```

⟨*Diagnostics: public*⟩+≡

```
  public :: expect_summary
```

⟨*Diagnostics: procedures*⟩+≡

```
  subroutine expect_summary (unit, force)
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: force
    logical :: force_output
    force_output = .false.;  if (present (force))  force_output = force
    if (expect_total /= 0 .or. force_output) then
       call msg_message ("Summary of value checks:", unit)
       write (msg_buffer, "(2x,A,1x,I0,1x,A,1x,A,1x,I0)") &
            "Failures:", expect_failures, "/", "Total:", expect_total
       call msg_message (unit=unit)
    end if
  end subroutine expect_summary
```

Helpers for converting integers into strings with minimal length.

⟨*Diagnostics: public*⟩+≡

```
public :: int2string
public :: int2char
public :: int2fixed
```

⟨*Diagnostics: procedures*⟩+≡

```
pure function int2fixed (i) result (c)
  integer, intent(in) :: i
  character(200) :: c
  c = ""
  write (c, *) i
  c = adjustl (c)
end function int2fixed

pure function int2string (i) result (s)
  integer, intent(in) :: i
  type (string_t) :: s
  s = trim (int2fixed (i))
end function int2string

pure function int2char (i) result (c)
  integer, intent(in) :: i
  character(len (trim (int2fixed (i)))) :: c
  c = int2fixed (i)
end function int2char
```

Dito for reals.

⟨*Diagnostics: public*⟩+≡

```
public :: real2string
public :: real2char
public :: real2fixed
```

⟨*Diagnostics: interfaces*⟩+≡

```
interface real2string
   module procedure real2string_list, real2string_fmt
end interface
interface real2char
   module procedure real2char_list, real2char_fmt
end interface
```

⟨*Diagnostics: procedures*⟩+≡

```
pure function real2fixed (x, fmt) result (c)
  real(default), intent(in) :: x
  character(*), intent(in), optional :: fmt
  character(200) :: c
  c = ""
  write (c, *) x
  c = adjustl (c)
end function real2fixed

pure function real2fixed_fmt (x, fmt) result (c)
  real(default), intent(in) :: x
  character(*), intent(in) :: fmt
  character(200) :: c
```

```
  c = ""
  write (c, fmt)  x
  c = adjustl (c)
end function real2fixed_fmt

pure function real2string_list (x) result (s)
  real(default), intent(in) :: x
  type(string_t) :: s
  s = trim (real2fixed (x))
end function real2string_list

pure function real2string_fmt (x, fmt) result (s)
  real(default), intent(in) :: x
  character(*), intent(in) :: fmt
  type(string_t) :: s
  s = trim (real2fixed_fmt (x, fmt))
end function real2string_fmt

pure function real2char_list (x) result (c)
  real(default), intent(in) :: x
  character(len_trim (real2fixed (x))) :: c
  c = real2fixed (x)
end function real2char_list

pure function real2char_fmt (x, fmt) result (c)
  real(default), intent(in) :: x
  character(*), intent(in) :: fmt
  character(len_trim (real2fixed_fmt (x, fmt))) :: c
  c = real2fixed_fmt (x, fmt)
end function real2char_fmt
```

Dito for complex values; we do not use the slightly ugly FORTRAN output form here but instead introduce our own. Ifort and Portland seem to have problems with this, therefore temporarily disable it.

⟨*CCC Diagnostics: public*⟩≡

```
public :: cmplx2string
public :: cmplx2char
```

⟨*CCC Diagnostics: procedures*⟩≡

```
pure function cmplx2string (x) result (s)
  complex(default), intent(in) :: x
  type(string_t) :: s
  s = real2string (real (x, default))
  if (aimag (x) /= 0) s = s // " + " // real2string (aimag (x)) // " I"
end function cmplx2string

pure function cmplx2char (x) result (c)
  complex(default), intent(in) :: x
  character(len (char (cmplx2string (x)))) :: c
  c = char (cmplx2string (x))
end function cmplx2char
```

5.4.3 Suppression of numerical noise

⟨*Diagnostics: public*⟩+≡
```
public :: pacify
```

⟨*Diagnostics: interfaces*⟩+≡
```
interface pacify
   module procedure pacify_real_default
   module procedure pacify_complex_default
end interface pacify
```

⟨*Diagnostics: procedures*⟩+≡
```
elemental subroutine pacify_real_default (x, tolerance)
  real(default), intent(inout) :: x
  real(default), intent(in) :: tolerance
  if (abs (x) < tolerance)  x = 0._default
end subroutine pacify_real_default

elemental subroutine pacify_complex_default (x, tolerance)
  complex(default), intent(inout) :: x
  real(default), intent(in) :: tolerance
  if (abs (real (x)) < tolerance)   &
       x = cmplx (0._default, aimag (x), kind=default)
  if (abs (aimag (x)) < tolerance)  &
       x = cmplx (real (x), 0._default, kind=default)
end subroutine pacify_complex_default
```

5.4.4 Signal handling

Killing the program by external signals may leave the files written by it in an undefined state. This can be avoided by catching signals and deferring program termination. Instead of masking only critical sections, we choose to mask signals globally (done in the main program) and terminate the program at predefined checkpoints only. Checkpoints are after each command, within the sampling function (so the program can be terminated after each event), and after each iteration in the phase-space generation algorithm.

Signal handling is done via a C interface to the `sigaction` system call. When a signal is raised that has been masked by the handler, the corresponding variable is set to the value of the signal. The variables are visible from the C signal handler.

The signal SIGINT is for keyboard interrupt (ctrl-C), SIGTERM is for system interrupt, e.g., at shutdown. The SIGXCPU and SIGXFSZ signals may be issued by batch systems.

⟨*Diagnostics: public*⟩+≡
```
public :: wo_sigint
public :: wo_sigterm
public :: wo_sigxcpu
public :: wo_sigxfsz
```

⟨*Diagnostics: variables*⟩+≡

```
integer(c_int), bind(C), volatile :: wo_sigint = 0
integer(c_int), bind(C), volatile :: wo_sigterm = 0
integer(c_int), bind(C), volatile :: wo_sigxcpu = 0
integer(c_int), bind(C), volatile :: wo_sigxfsz = 0
```

Here are the interfaces to the C functions. The routine `mask_term_signals` forces termination signals to be delayed. `release_term_signals` restores normal behavior. However, the program can be terminated anytime by calling `terminate_now_if_signal` which inspects the signals and terminates the program if requested..

⟨*Diagnostics: public*⟩+≡

```
public :: mask_term_signals
```

⟨*Diagnostics: procedures*⟩+≡

```
subroutine mask_term_signals ()
  logical :: ok
  wo_sigint = 0
  ok = wo_mask_sigint () == 0
  if (.not. ok)  call msg_error ("Masking SIGINT failed")
  wo_sigterm = 0
  ok = wo_mask_sigterm () == 0
  if (.not. ok)  call msg_error ("Masking SIGTERM failed")
  wo_sigxcpu = 0
  ok = wo_mask_sigxcpu () == 0
  if (.not. ok)  call msg_error ("Masking SIGXCPU failed")
  wo_sigxfsz = 0
  ok = wo_mask_sigxfsz () == 0
  if (.not. ok)  call msg_error ("Masking SIGXFSZ failed")
end subroutine mask_term_signals
```

⟨*Diagnostics: interfaces*⟩+≡

```
interface
   integer(c_int) function wo_mask_sigint () bind(C)
     import
   end function wo_mask_sigint
end interface
interface
   integer(c_int) function wo_mask_sigterm () bind(C)
     import
   end function wo_mask_sigterm
end interface
interface
   integer(c_int) function wo_mask_sigxcpu () bind(C)
     import
   end function wo_mask_sigxcpu
end interface
interface
   integer(c_int) function wo_mask_sigxfsz () bind(C)
     import
   end function wo_mask_sigxfsz
end interface
```

⟨*Diagnostics: public*⟩+≡

```
public :: release_term_signals
```

⟨*Diagnostics: procedures*⟩+≡

```
subroutine release_term_signals ()
  logical :: ok
  ok = wo_release_sigint () == 0
  if (.not. ok)  call msg_error ("Releasing SIGINT failed")
  ok = wo_release_sigterm () == 0
  if (.not. ok)  call msg_error ("Releasing SIGTERM failed")
  ok = wo_release_sigxcpu () == 0
  if (.not. ok)  call msg_error ("Releasing SIGXCPU failed")
  ok = wo_release_sigxfsz () == 0
  if (.not. ok)  call msg_error ("Releasing SIGXFSZ failed")
end subroutine release_term_signals
```

⟨*Diagnostics: interfaces*⟩+≡

```
interface
   integer(c_int) function wo_release_sigint () bind(C)
     import
   end function wo_release_sigint
end interface
interface
   integer(c_int) function wo_release_sigterm () bind(C)
     import
   end function wo_release_sigterm
end interface
interface
   integer(c_int) function wo_release_sigxcpu () bind(C)
     import
   end function wo_release_sigxcpu
end interface
interface
   integer(c_int) function wo_release_sigxfsz () bind(C)
     import
   end function wo_release_sigxfsz
end interface
```

⟨*Diagnostics: public*⟩+≡

```
public :: signal_is_pending
```

⟨*Diagnostics: procedures*⟩+≡

```
function signal_is_pending () result (flag)
  logical :: flag
  flag = &
       wo_sigint /= 0 .or. &
       wo_sigterm /= 0 .or. &
       wo_sigxcpu /= 0 .or. &
       wo_sigxfsz /= 0
end function signal_is_pending
```

⟨*Diagnostics: public*⟩+≡

```
public :: terminate_now_if_signal
```

⟨*Diagnostics: procedures*⟩+≡

```
subroutine terminate_now_if_signal ()
  if (wo_sigint /= 0) then
     call msg_terminate ("Signal SIGINT (keyboard interrupt) received.", &
        quit_code=int (wo_sigint))
  else if (wo_sigterm /= 0) then
     call msg_terminate ("Signal SIGTERM (termination signal) received.", &
        quit_code=int (wo_sigterm))
  else if (wo_sigxcpu /= 0) then
     call msg_terminate ("Signal SIGXCPU (CPU time limit exceeded) received.", &
        quit_code=int (wo_sigxcpu))
  else if (wo_sigxfsz /= 0) then
     call msg_terminate ("Signal SIGXFSZ (file size limit exceeded) received.", &
        quit_code=int (wo_sigxfsz))
  end if
end subroutine terminate_now_if_signal
```

⟨*Diagnostics: public*⟩+≡

```
public :: single_event
```

⟨*Diagnostics: variables*⟩+≡

```
logical :: single_event = .false.
```

⟨*Diagnostics: public*⟩+≡

```
public :: terminate_now_if_single_event
```

⟨*Diagnostics: procedures*⟩+≡

```
subroutine terminate_now_if_single_event ()
  integer, save :: n_calls = 0
  n_calls = n_calls + 1
  if (single_event .and. n_calls > 1) then
     call msg_terminate ("Stopping after one event", quit_code=0)
  end if
end subroutine terminate_now_if_single_event
```

5.5 Operating-system interface

For specific purposes, we need direct access to the OS (system calls). This is, of course, system dependent. The current version is valid for GNU/Linux; we expect to use a preprocessor for this module if different OSs are to be supported.

The current implementation lacks error handling.

⟨`os_interface.f90`⟩≡

```
⟨File header⟩

module os_interface

  use, intrinsic :: iso_c_binding !NODEP!

⟨Use strings⟩
  use io_units
  use diagnostics
  use system_defs, only: DLERROR_LEN, ENVVAR_LEN
```

```
  use system_dependencies
```

⟨*Standard module head*⟩

⟨*OS interface: public*⟩

⟨*OS interface: types*⟩

⟨*OS interface: interfaces*⟩

```
contains
```

⟨*OS interface: procedures*⟩

```
end module os_interface
```

5.5.1 Path variables

This is a transparent container for storing user-defined path variables.

⟨*OS interface: public*⟩≡
```
  public :: paths_t
```

⟨*OS interface: types*⟩≡
```
  type :: paths_t
     type(string_t) :: prefix
     type(string_t) :: exec_prefix
     type(string_t) :: bindir
     type(string_t) :: libdir
     type(string_t) :: includedir
     type(string_t) :: datarootdir
     type(string_t) :: localprefix
     type(string_t) :: libtool
     type(string_t) :: lhapdfdir
  end type paths_t
```

⟨*OS interface: public*⟩+≡
```
  public :: paths_init
```

⟨*OS interface: procedures*⟩≡
```
  subroutine paths_init (paths)
    type(paths_t), intent(out) :: paths
    paths%prefix = ""
    paths%exec_prefix = ""
    paths%bindir = ""
    paths%libdir = ""
    paths%includedir = ""
    paths%datarootdir = ""
    paths%localprefix = ""
    paths%libtool = ""
    paths%lhapdfdir = ""
  end subroutine paths_init
```

5.5.2 System dependencies

We store all potentially system- and user/run-dependent data in a transparent container. This includes compiler/linker names and flags, file extensions, etc. There are actually two different possibilities for extensions of shared libraries, depending on whether the Fortran compiler or the system linker (usually the C compiler) has been used for linking. The default for the Fortran compiler on most systems is `.so`.

⟨*OS interface: public*⟩+≡
```
public :: os_data_t
```

⟨*OS interface: types*⟩+≡
```
type :: os_data_t
   logical :: use_libtool
   logical :: use_testfiles
   type(string_t) :: fc
   type(string_t) :: fcflags
   type(string_t) :: fcflags_pic
   type(string_t) :: fc_src_ext
   type(string_t) :: cc
   type(string_t) :: cflags
   type(string_t) :: cflags_pic
   type(string_t) :: obj_ext
   type(string_t) :: ld
   type(string_t) :: ldflags
   type(string_t) :: ldflags_so
   type(string_t) :: ldflags_static
   type(string_t) :: ldflags_hepmc
   type(string_t) :: ldflags_lcio
   type(string_t) :: ldflags_hoppet
   type(string_t) :: ldflags_looptools
   type(string_t) :: shrlib_ext
   type(string_t) :: fc_shrlib_ext
   type(string_t) :: makeflags
   type(string_t) :: prefix
   type(string_t) :: exec_prefix
   type(string_t) :: bindir
   type(string_t) :: libdir
   type(string_t) :: includedir
   type(string_t) :: datarootdir
   type(string_t) :: whizard_omega_binpath
   type(string_t) :: whizard_includes
   type(string_t) :: whizard_ldflags
   type(string_t) :: whizard_libtool
   type(string_t) :: whizard_modelpath
   type(string_t) :: whizard_models_libpath
   type(string_t) :: whizard_susypath
   type(string_t) :: whizard_gmlpath
   type(string_t) :: whizard_cutspath
   type(string_t) :: whizard_texpath
   type(string_t) :: whizard_sharepath
   type(string_t) :: whizard_testdatapath
   type(string_t) :: whizard_modelpath_local
   type(string_t) :: whizard_models_libpath_local
```

```
    type(string_t) :: whizard_omega_binpath_local
    type(string_t) :: whizard_circe2path
    type(string_t) :: whizard_beamsimpath
    type(string_t) :: whizard_mulipath
    type(string_t) :: pdf_builtin_datapath
    logical :: event_analysis = .false.
    logical :: event_analysis_ps  = .false.
    logical :: event_analysis_pdf = .false.
    type(string_t) :: latex
    type(string_t) :: mpost
    type(string_t) :: gml
    type(string_t) :: dvips
    type(string_t) :: ps2pdf
    type(string_t) :: gosampath
    type(string_t) :: golempath
    type(string_t) :: formpath
    type(string_t) :: qgrafpath
    type(string_t) :: ninjapath
    type(string_t) :: samuraipath
end type os_data_t
```

Since all are allocatable strings, explicit initialization is necessary.

⟨*System defs: public parameters*⟩+≡

```
integer, parameter, public :: ENVVAR_LEN = 1000
```

⟨*OS interface: public*⟩+≡

```
public :: os_data_init
```

⟨*OS interface: procedures*⟩+≡

```
subroutine os_data_init (os_data, paths)
  type(os_data_t), intent(out) :: os_data
  type(paths_t), intent(in), optional :: paths
  character(len=ENVVAR_LEN) :: home
  type(string_t) :: localprefix, local_includes
  os_data%use_libtool = .true.
  inquire (file = "TESTFLAG", exist = os_data%use_testfiles)
  call get_environment_variable ("HOME", home)
  if (present(paths)) then
     if (paths%localprefix == "") then
        localprefix = trim (home) // "/.whizard"
     else
        localprefix = paths%localprefix
     end if
  else
     localprefix = trim (home) // "/.whizard"
  end if
  local_includes = localprefix // "/lib/whizard/mod/models"
  os_data%whizard_modelpath_local = localprefix // "/share/whizard/models"
  os_data%whizard_models_libpath_local = localprefix // "/lib/whizard/models"
  os_data%whizard_omega_binpath_local = localprefix // "/bin"
  os_data%fc             = DEFAULT_FC
  os_data%fcflags        = DEFAULT_FCFLAGS
  os_data%fcflags_pic    = DEFAULT_FCFLAGS_PIC
  os_data%fc_src_ext     = DEFAULT_FC_SRC_EXT
  os_data%cc             = DEFAULT_CC
```

```
os_data%cflags         = DEFAULT_CFLAGS
os_data%cflags_pic     = DEFAULT_CFLAGS_PIC
os_data%obj_ext        = DEFAULT_OBJ_EXT
os_data%ld             = DEFAULT_LD
os_data%ldflags        = DEFAULT_LDFLAGS
os_data%ldflags_so     = DEFAULT_LDFLAGS_SO
os_data%ldflags_static = DEFAULT_LDFLAGS_STATIC
os_data%ldflags_hepmc  = DEFAULT_LDFLAGS_HEPMC
os_data%ldflags_lcio   = DEFAULT_LDFLAGS_LCIO
os_data%ldflags_hoppet = DEFAULT_LDFLAGS_HOPPET
os_data%ldflags_looptools = DEFAULT_LDFLAGS_LOOPTOOLS
os_data%shrlib_ext     = DEFAULT_SHRLIB_EXT
os_data%fc_shrlib_ext  = DEFAULT_FC_SHRLIB_EXT
os_data%makeflags      = DEFAULT_MAKEFLAGS
os_data%prefix      = PREFIX
os_data%exec_prefix = EXEC_PREFIX
os_data%bindir      = BINDIR
os_data%libdir      = LIBDIR
os_data%includedir  = INCLUDEDIR
os_data%datarootdir = DATAROOTDIR
if (present (paths)) then
   if (paths%prefix      /= "")  os_data%prefix      = paths%prefix
   if (paths%exec_prefix /= "")  os_data%exec_prefix = paths%exec_prefix
   if (paths%bindir      /= "")  os_data%bindir      = paths%bindir
   if (paths%libdir      /= "")  os_data%libdir      = paths%libdir
   if (paths%includedir  /= "")  os_data%includedir  = paths%includedir
   if (paths%datarootdir /= "")  os_data%datarootdir = paths%datarootdir
end if
if (os_data%use_testfiles) then
   os_data%whizard_omega_binpath  = WHIZARD_TEST_OMEGA_BINPATH
   os_data%whizard_includes       = WHIZARD_TEST_INCLUDES
   os_data%whizard_ldflags        = WHIZARD_TEST_LDFLAGS
   os_data%whizard_libtool        = WHIZARD_LIBTOOL_TEST
   os_data%whizard_modelpath      = WHIZARD_TEST_MODELPATH
   os_data%whizard_models_libpath = WHIZARD_TEST_MODELS_LIBPATH
   os_data%whizard_susypath       = WHIZARD_TEST_SUSYPATH
   os_data%whizard_gmlpath        = WHIZARD_TEST_GMLPATH
   os_data%whizard_cutspath       = WHIZARD_TEST_CUTSPATH
   os_data%whizard_texpath        = WHIZARD_TEST_TEXPATH
   os_data%whizard_sharepath      = WHIZARD_TEST_SHAREPATH
   os_data%whizard_testdatapath   = WHIZARD_TEST_TESTDATAPATH
   os_data%whizard_circe2path     = WHIZARD_TEST_CIRCE2PATH
   os_data%whizard_beamsimpath    = WHIZARD_TEST_BEAMSIMPATH
   os_data%whizard_mulipath       = WHIZARD_TEST_MULIPATH
   os_data%pdf_builtin_datapath   = PDF_BUILTIN_TEST_DATAPATH
else
   if (os_dir_exist (local_includes)) then
      os_data%whizard_includes = "-I" // local_includes // " "// &
         WHIZARD_INCLUDES
   else
      os_data%whizard_includes = WHIZARD_INCLUDES
   end if
   os_data%whizard_omega_binpath  = WHIZARD_OMEGA_BINPATH
   os_data%whizard_ldflags        = WHIZARD_LDFLAGS
```

```
      os_data%whizard_libtool        = WHIZARD_LIBTOOL
      if(present(paths)) then
         if (paths%libtool /= "")  os_data%whizard_libtool = paths%libtool
      end if
      os_data%whizard_modelpath      = WHIZARD_MODELPATH
      os_data%whizard_models_libpath = WHIZARD_MODELS_LIBPATH
      os_data%whizard_susypath       = WHIZARD_SUSYPATH
      os_data%whizard_gmlpath        = WHIZARD_GMLPATH
      os_data%whizard_cutspath       = WHIZARD_CUTSPATH
      os_data%whizard_texpath        = WHIZARD_TEXPATH
      os_data%whizard_sharepath      = WHIZARD_SHAREPATH
      os_data%whizard_testdatapath   = WHIZARD_TESTDATAPATH
      os_data%whizard_circe2path     = WHIZARD_CIRCE2PATH
      os_data%whizard_beamsimpath    = WHIZARD_BEAMSIMPATH
      os_data%whizard_mulipath       = WHIZARD_MULIPATH
      os_data%pdf_builtin_datapath   = PDF_BUILTIN_DATAPATH
   end if
   os_data%event_analysis     = EVENT_ANALYSIS     == "yes"
   os_data%event_analysis_ps  = EVENT_ANALYSIS_PS  == "yes"
   os_data%event_analysis_pdf = EVENT_ANALYSIS_PDF == "yes"
   os_data%latex  = PRG_LATEX // " " // OPT_LATEX
   os_data%mpost  = PRG_MPOST // " " // OPT_MPOST
   if (os_data%use_testfiles) then
      os_data%gml    = os_data%whizard_gmlpath // "/whizard-gml" // " " // &
           OPT_MPOST // " " // "--gmldir " // os_data%whizard_gmlpath
   else
      os_data%gml    = os_data%bindir // "/whizard-gml" // " " // OPT_MPOST &
        // " " // "--gmldir " // os_data%whizard_gmlpath
   end if
   os_data%dvips  = PRG_DVIPS
   os_data%ps2pdf = PRG_PS2PDF
   call os_data_expand_paths (os_data)
   os_data%gosampath = GOSAM_DIR
   os_data%golempath = GOLEM_DIR
   os_data%formpath = FORM_DIR
   os_data%qgrafpath = QGRAF_DIR
   os_data%ninjapath = NINJA_DIR
   os_data%samuraipath = SAMURAI_DIR
end subroutine os_data_init
```

Replace occurences of GNU path variables (such as `${prefix}`) by their values. Do this for all strings that could depend on them, and do the replacement in reverse order, since the path variables may be defined in terms of each other.

⟨*OS interface: procedures*⟩+≡

```
subroutine os_data_expand_paths (os_data)
   type(os_data_t), intent(inout) :: os_data
   integer, parameter :: N_VARIABLES = 6
   type(string_t), dimension(N_VARIABLES) :: variable, value
   variable(1) = "${prefix}";       value(1) = os_data%prefix
   variable(2) = "${exec_prefix}";  value(2) = os_data%exec_prefix
   variable(3) = "${bindir}";       value(3) = os_data%bindir
   variable(4) = "${libdir}";       value(4) = os_data%libdir
   variable(5) = "${includedir}";   value(5) = os_data%includedir
```

```
    variable(6) = "${datarootdir}";  value(6) = os_data%datarootdir
    call expand_paths (os_data%whizard_omega_binpath)
    call expand_paths (os_data%whizard_includes)
    call expand_paths (os_data%whizard_ldflags)
    call expand_paths (os_data%whizard_libtool)
    call expand_paths (os_data%whizard_modelpath)
    call expand_paths (os_data%whizard_models_libpath)
    call expand_paths (os_data%whizard_susypath)
    call expand_paths (os_data%whizard_gmlpath)
    call expand_paths (os_data%whizard_cutspath)
    call expand_paths (os_data%whizard_texpath)
    call expand_paths (os_data%whizard_sharepath)
    call expand_paths (os_data%whizard_testdatapath)
    call expand_paths (os_data%whizard_circe2path)
    call expand_paths (os_data%whizard_beamsimpath)
    call expand_paths (os_data%whizard_mulipath)
    call expand_paths (os_data%whizard_models_libpath_local)
    call expand_paths (os_data%whizard_modelpath_local)
    call expand_paths (os_data%whizard_omega_binpath_local)
    call expand_paths (os_data%pdf_builtin_datapath)
    call expand_paths (os_data%latex)
    call expand_paths (os_data%mpost)
    call expand_paths (os_data%gml)
    call expand_paths (os_data%dvips)
    call expand_paths (os_data%ps2pdf)
  contains
    subroutine expand_paths (string)
      type(string_t), intent(inout) :: string
      integer :: i
      do i = N_VARIABLES, 1, -1
         string = replace (string, variable(i), value(i), every=.true.)
      end do
    end subroutine expand_paths
  end subroutine os_data_expand_paths
```

Write contents

⟨*OS interface: public*⟩+≡

```
  public :: os_data_write
```

⟨*OS interface: procedures*⟩+≡

```
  subroutine os_data_write (os_data, unit)
    type(os_data_t), intent(in) :: os_data
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(A)")  "OS data:"
    write (u, *) "use_libtool    = ", os_data%use_libtool
    write (u, *) "use_testfiles  = ", os_data%use_testfiles
    write (u, *) "fc             = ", char (os_data%fc)
    write (u, *) "fcflags        = ", char (os_data%fcflags)
    write (u, *) "fcflags_pic    = ", char (os_data%fcflags_pic)
    write (u, *) "fc_src_ext     = ", char (os_data%fc_src_ext)
    write (u, *) "cc             = ", char (os_data%cc)
    write (u, *) "cflags         = ", char (os_data%cflags)
```

```
write (u, *) "cflags_pic     = ", char (os_data%cflags_pic)
write (u, *) "obj_ext        = ", char (os_data%obj_ext)
write (u, *) "ld             = ", char (os_data%ld)
write (u, *) "ldflags        = ", char (os_data%ldflags)
write (u, *) "ldflags_so     = ", char (os_data%ldflags_so)
write (u, *) "ldflags_static = ", char (os_data%ldflags_static)
write (u, *) "ldflags_hepmc  = ", char (os_data%ldflags_hepmc)
write (u, *) "ldflags_lcio   = ", char (os_data%ldflags_lcio)
write (u, *) "ldflags_hoppet = ", char (os_data%ldflags_hoppet)
write (u, *) "ldflags_looptools = ", char (os_data%ldflags_looptools)
write (u, *) "shrlib_ext     = ", char (os_data%shrlib_ext)
write (u, *) "fc_shrlib_ext  = ", char (os_data%fc_shrlib_ext)
write (u, *) "makeflags      = ", char (os_data%makeflags)
write (u, *) "prefix         = ", char (os_data%prefix)
write (u, *) "exec_prefix    = ", char (os_data%exec_prefix)
write (u, *) "bindir         = ", char (os_data%bindir)
write (u, *) "libdir         = ", char (os_data%libdir)
write (u, *) "includedir     = ", char (os_data%includedir)
write (u, *) "datarootdir    = ", char (os_data%datarootdir)
write (u, *) "whizard_omega_binpath  = ", &
     char (os_data%whizard_omega_binpath)
write (u, *) "whizard_includes       = ", char (os_data%whizard_includes)
write (u, *) "whizard_ldflags        = ", char (os_data%whizard_ldflags)
write (u, *) "whizard_libtool        = ", char (os_data%whizard_libtool)
write (u, *) "whizard_modelpath      = ", &
     char (os_data%whizard_modelpath)
write (u, *) "whizard_models_libpath = ", &
     char (os_data%whizard_modelpath)
write (u, *) "whizard_susypath       = ", char (os_data%whizard_susypath)
write (u, *) "whizard_gmlpath        = ", char (os_data%whizard_gmlpath)
write (u, *) "whizard_cutspath       = ", char (os_data%whizard_cutspath)
write (u, *) "whizard_texpath        = ", char (os_data%whizard_texpath)
write (u, *) "whizard_circe2path     = ", char (os_data%whizard_circe2path)
write (u, *) "whizard_beamsimpath    = ", char (os_data%whizard_beamsimpath)
write (u, *) "whizard_mulipath    = ", char (os_data%whizard_mulipath)
write (u, *) "whizard_sharepath  = ", &
     char (os_data%whizard_sharepath)
write (u, *) "whizard_testdatapath  = ", &
     char (os_data%whizard_testdatapath)
write (u, *) "whizard_modelpath_local      = ", &
     char (os_data%whizard_modelpath_local)
write (u, *) "whizard_models_libpath_local = ", &
     char (os_data%whizard_models_libpath_local)
write (u, *) "whizard_omega_binpath_local  = ", &
     char (os_data%whizard_omega_binpath_local)
write (u, *) "event_analysis     = ", os_data%event_analysis
write (u, *) "event_analysis_ps  = ", os_data%event_analysis_ps
write (u, *) "event_analysis_pdf = ", os_data%event_analysis_pdf
write (u, *) "latex  = ", char (os_data%latex)
write (u, *) "mpost  = ", char (os_data%mpost)
write (u, *) "gml    = ", char (os_data%gml)
write (u, *) "dvips  = ", char (os_data%dvips)
write (u, *) "ps2pdf = ", char (os_data%ps2pdf)
if (os_data%gosampath /= "") then
```

```
      write (u, *) "gosam   = ", char (os_data%gosampath)
      write (u, *) "golem   = ", char (os_data%golempath)
      write (u, *) "form    = ", char (os_data%formpath)
      write (u, *) "qgraf   = ", char (os_data%qgrafpath)
      write (u, *) "ninja   = ", char (os_data%ninjapath)
      write (u, *) "samurai = ", char (os_data%samuraipath)
    end if
  end subroutine os_data_write
```

⟨*OS interface: public*⟩+≡

```
  public :: os_data_build_latex_file
```

⟨*OS interface: procedures*⟩+≡

```
  subroutine os_data_build_latex_file (os_data, filename, stat_out)
    type(os_data_t), intent(in) :: os_data
    type(string_t), intent(in) :: filename
    integer, intent(out), optional :: stat_out
    type(string_t) :: setenv_tex, pipe, pipe_dvi
    integer :: unit_dev, status
    status = -1
    if (os_data%event_analysis_ps) then
       !!! Check if our OS has a /dev/null
       unit_dev = free_unit ()
       open (file = "/dev/null", unit = unit_dev, &
            action = "write", iostat = status)
       close (unit_dev)
       if (status /= 0) then
          pipe = ""
          pipe_dvi = ""
       else
          pipe = " > /dev/null"
          pipe_dvi = " 2>/dev/null 1>/dev/null"
       end if
       if (os_data%whizard_texpath /= "") then
          setenv_tex = "TEXINPUTS=" // &
               os_data%whizard_texpath // ":$TEXINPUTS "
       else
          setenv_tex = ""
       end if
       call os_system_call (setenv_tex // &
            os_data%latex // " " // filename // ".tex " // pipe, &
            verbose = .true., status = status)
       call os_system_call (os_data%dvips // " -o " // filename // &
            ".ps " // filename // ".dvi" // pipe_dvi, verbose = .true., &
            status = status)
       call os_system_call (os_data%ps2pdf // " " // filename // ".ps", &
            verbose = .true., status = status)
    end if
    if (present (stat_out)) stat_out = status
  end subroutine os_data_build_latex_file
```

5.5.3 Dynamic linking

We define a type that holds the filehandle for a dynamically linked library (shared object), together with functions to open and close the library, and to access functions in this library.

⟨*OS interface: public*⟩+≡
```
public :: dlaccess_t
```

⟨*OS interface: types*⟩+≡
```
type :: dlaccess_t
   private
   type(string_t) :: filename
   type(c_ptr) :: handle = c_null_ptr
   logical :: is_open = .false.
   logical :: has_error = .false.
   type(string_t) :: error
 contains
 ⟨OS interface: dlaccess: TBP⟩
end type dlaccess_t
```

Output. This is called by the output routine for the process library.

⟨*OS interface: dlaccess: TBP*⟩≡
```
procedure :: write => dlaccess_write
```

⟨*OS interface: procedures*⟩+≡
```
subroutine dlaccess_write (object, unit)
  class(dlaccess_t), intent(in) :: object
  integer, intent(in) :: unit
  write (unit, "(1x,A)")  "DL access info:"
  write (unit, "(3x,A,L1)")   "is open   = ", object%is_open
  if (object%has_error) then
     write (unit, "(3x,A,A,A)")  "error     = '", char (object%error), "'"
  else
     write (unit, "(3x,A)")      "error     = [none]"
  end if
end subroutine dlaccess_write
```

The interface to the library functions:

⟨*OS interface: interfaces*⟩≡
```
interface
   function dlopen (filename, flag) result (handle) bind(C)
     import
     character(c_char), dimension(*) :: filename
     integer(c_int), value :: flag
     type(c_ptr) :: handle
   end function dlopen
end interface

interface
   function dlclose (handle) result (status) bind(C)
     import
     type(c_ptr), value :: handle
     integer(c_int) :: status
   end function dlclose
```

```
end interface

interface
   function dlerror () result (str) bind(C)
     import
     type(c_ptr) :: str
   end function dlerror
end interface

interface
   function dlsym (handle, symbol) result (fptr) bind(C)
     import
     type(c_ptr), value :: handle
     character(c_char), dimension(*) :: symbol
     type(c_funptr) :: fptr
   end function dlsym
end interface
```

This reads an error string and transforms it into a `string_t` object, if an error has occured. If not, set the error flag to false and return an empty string.

⟨*System defs: public parameters*⟩+≡

```
integer, parameter, public :: DLERROR_LEN = 160
```

⟨*OS interface: procedures*⟩+≡

```
subroutine read_dlerror (has_error, error)
  logical, intent(out) :: has_error
  type(string_t), intent(out) :: error
  type(c_ptr) :: err_cptr
  character(len=DLERROR_LEN, kind=c_char), pointer :: err_fptr
  integer :: str_end
  err_cptr = dlerror ()
  if (c_associated (err_cptr)) then
     call c_f_pointer (err_cptr, err_fptr)
     has_error = .true.
     str_end = scan (err_fptr, c_null_char)
     if (str_end > 0) then
        error = err_fptr(1:str_end-1)
     else
        error = err_fptr
     end if
  else
     has_error = .false.
     error = ""
  end if
end subroutine read_dlerror
```

This is the Fortran API. Init/final open and close the file, i.e., load and unload the library.

Note that a library can be opened more than once, and that for an ultimate close as many `dlclose` calls as `dlopen` calls are necessary. However, we assume that it is opened and closed only once.

⟨*OS interface: public*⟩+≡

```
public :: dlaccess_init
```

```
public :: dlaccess_final
```

⟨*OS interface: dlaccess: TBP*⟩+≡

```
procedure :: init => dlaccess_init
procedure :: final => dlaccess_final
```

⟨*OS interface: procedures*⟩+≡

```
subroutine dlaccess_init (dlaccess, prefix, libname, os_data)
  class(dlaccess_t), intent(out) :: dlaccess
  type(string_t), intent(in) :: prefix, libname
  type(os_data_t), intent(in), optional :: os_data
  type(string_t) :: filename
  logical :: exist
  dlaccess%filename = libname
  filename = prefix // "/" // libname
  inquire (file=char(filename), exist=exist)
  if (.not. exist) then
     filename = prefix // "/.libs/" // libname
     inquire (file=char(filename), exist=exist)
     if (.not. exist) then
        dlaccess%has_error = .true.
        dlaccess%error = "Library '" // filename // "' not found"
        return
     end if
  end if
  dlaccess%handle = dlopen (char (filename) // c_null_char, ior ( &
     RTLD_LAZY, RTLD_LOCAL))
  dlaccess%is_open = c_associated (dlaccess%handle)
  call read_dlerror (dlaccess%has_error, dlaccess%error)
end subroutine dlaccess_init

subroutine dlaccess_final (dlaccess)
  class(dlaccess_t), intent(inout) :: dlaccess
  integer(c_int) :: status
  if (dlaccess%is_open) then
     status = dlclose (dlaccess%handle)
     dlaccess%is_open = .false.
     call read_dlerror (dlaccess%has_error, dlaccess%error)
  end if
end subroutine dlaccess_final
```

Return true if an error has occured.

⟨*OS interface: public*⟩+≡

```
public :: dlaccess_has_error
```

⟨*OS interface: procedures*⟩+≡

```
function dlaccess_has_error (dlaccess) result (flag)
  logical :: flag
  type(dlaccess_t), intent(in) :: dlaccess
  flag = dlaccess%has_error
end function dlaccess_has_error
```

Return the error string currently stored in the `dlaccess` object.

⟨*OS interface: public*⟩+≡

```
public :: dlaccess_get_error
```

⟨*OS interface: procedures*⟩+≡

```
function dlaccess_get_error (dlaccess) result (error)
  type(string_t) :: error
  type(dlaccess_t), intent(in) :: dlaccess
  error = dlaccess%error
end function dlaccess_get_error
```

The symbol handler returns the C address of the function with the given string name. (It is a good idea to use `bind(C)` for all functions accessed by this, such that the name string is well-defined.) Call `c_f_procpointer` to cast this into a Fortran procedure pointer with an appropriate interface.

⟨*OS interface: public*⟩+≡

```
public :: dlaccess_get_c_funptr
```

⟨*OS interface: procedures*⟩+≡

```
function dlaccess_get_c_funptr (dlaccess, fname) result (fptr)
  type(c_funptr) :: fptr
  type(dlaccess_t), intent(inout) :: dlaccess
  type(string_t), intent(in) :: fname
  fptr = dlsym (dlaccess%handle, char (fname) // c_null_char)
  call read_dlerror (dlaccess%has_error, dlaccess%error)
end function dlaccess_get_c_funptr
```

5.5.4 Predicates

Return true if the library is loaded. In particular, this is false if loading was unsuccessful.

⟨*OS interface: public*⟩+≡

```
public :: dlaccess_is_open
```

⟨*OS interface: procedures*⟩+≡

```
function dlaccess_is_open (dlaccess) result (flag)
  logical :: flag
  type(dlaccess_t), intent(in) :: dlaccess
  flag = dlaccess%is_open
end function dlaccess_is_open
```

5.5.5 Shell access

This is the standard system call for executing a shell command, such as invoking a compiler.

In F2008 there will be the equivalent built-in command `execute_command_line`.

⟨*OS interface: public*⟩+≡

```
public :: os_system_call
```

⟨*OS interface: procedures*⟩+≡

```
subroutine os_system_call (command_string, status, verbose)
  type(string_t), intent(in) :: command_string
  integer, intent(out), optional :: status
  logical, intent(in), optional :: verbose
  logical :: verb
```

```
    integer :: stat
    verb = .false.;  if (present (verbose))  verb = verbose
    if (verb) &
         call msg_message ("command: " // char (command_string))
    stat = system (char (command_string) // c_null_char)
    if (present (status)) then
       status = stat
    else if (stat /= 0) then
       if (.not. verb) &
            call msg_message ("command: " // char (command_string))
       write (msg_buffer, "(A,I0)")  "Return code = ", stat
       call msg_message ()
       call msg_fatal ("System command returned with nonzero status code")
    end if
  end subroutine os_system_call
```

⟨*OS interface: interfaces*⟩+≡

```
  interface
     function system (command) result (status) bind(C)
       import
       integer(c_int) :: status
       character(c_char), dimension(*) :: command
     end function system
  end interface
```

5.5.6 Querying for a directory

This queries for the existence of a directory. There is no standard way to achieve this in FORTRAN, and if we were to call into `libc`, we would need access to C macros for evaluating the result, so we resort to calling `test` as a system call.

⟨*OS interface: public*⟩+≡

```
  public :: os_dir_exist
```

⟨*OS interface: procedures*⟩+≡

```
  function os_dir_exist (name) result (res)
    type(string_t), intent(in) :: name
    logical :: res
    integer :: status
    call os_system_call ('test -d "' // name // '"', status=status)
    res = status == 0
  end function os_dir_exist
```

⟨*OS interface: public*⟩+≡

```
  public :: os_file_exist
```

⟨*OS interface: procedures*⟩+≡

```
  function os_file_exist (name) result (exist)
    type(string_t), intent(in) :: name
!    logical, intent(in), optional :: verb
    logical :: exist
!    integer :: status
!    call  os_system_call ('test -f "' // name // '"', status=status, verbose=verb)
!    res = (status == 0)
```

```
    inquire (file = char (name), exist=exist)
  end function os_file_exist
```

5.5.7 Fortran compiler and linker

Compile a single module for use in a shared library, but without linking.

⟨*OS interface: public*⟩+≡

```
  public :: os_compile_shared
```

⟨*OS interface: procedures*⟩+≡

```
  subroutine os_compile_shared (src, os_data, status)
    type(string_t), intent(in) :: src
    type(os_data_t), intent(in) :: os_data
    integer, intent(out), optional :: status
    type(string_t) :: command_string
    if (os_data%use_libtool) then
       command_string = &
            os_data%whizard_libtool // " --mode=compile " // &
            os_data%fc // " " // &
            "-c " // &
            os_data%whizard_includes // " " // &
            os_data%fcflags // " " // &
            "'" // src // os_data%fc_src_ext // "'"
    else
       command_string = &
            os_data%fc // " " // &
            "-c  " // &
            os_data%fcflags_pic // " " // &
            os_data%whizard_includes // " " // &
            os_data%fcflags // " " // &
            "'" // src // os_data%fc_src_ext // "'"
    end if
    call os_system_call (command_string, status)
  end subroutine os_compile_shared
```

Link an array of object files to build a shared object library. In the libtool case, we have to specify a `-rpath`, otherwise only a static library can be built. However, since the library is never installed, this rpath is irrelevant.

⟨*OS interface: public*⟩+≡

```
  public :: os_link_shared
```

⟨*OS interface: procedures*⟩+≡

```
  subroutine os_link_shared (objlist, lib, os_data, status)
    type(string_t), intent(in) :: objlist, lib
    type(os_data_t), intent(in) :: os_data
    integer, intent(out), optional :: status
    type(string_t) :: command_string
    if (os_data%use_libtool) then
       command_string = &
            os_data%whizard_libtool // " --mode=link " // &
            os_data%fc // " " // &
            "-module " // &
```

```
            "-rpath /usr/local/lib" // " " // &
            os_data%fcflags // " " // &
            os_data%whizard_ldflags // " " // &
            os_data%ldflags // " " // &
            "-o '" // lib // ".la' " // &
            objlist
      else
         command_string = &
            os_data%ld // " " // &
            os_data%ldflags_so // " " // &
            os_data%fcflags // " " // &
            os_data%whizard_ldflags // " " // &
            os_data%ldflags // " " // &
            "-o '" // lib // "." // os_data%fc_shrlib_ext // "' " // &
            objlist
      end if
      call os_system_call (command_string, status)
   end subroutine os_link_shared
```

Link an array of object files / libraries to build a static executable.

⟨*OS interface: public*⟩+≡

```
   public :: os_link_static
```

⟨*OS interface: procedures*⟩+≡

```
   subroutine os_link_static (objlist, exec_name, os_data, status)
      type(string_t), intent(in) :: objlist, exec_name
      type(os_data_t), intent(in) :: os_data
      integer, intent(out), optional :: status
      type(string_t) :: command_string
      if (os_data%use_libtool) then
         command_string = &
            os_data%whizard_libtool // " --mode=link " // &
            os_data%fc // " " // &
            "-static-libtool-libs " // &
            os_data%fcflags // " " // &
            os_data%whizard_ldflags // " " // &
            os_data%ldflags // " " // &
            os_data%ldflags_static // " " // &
            "-o '" // exec_name // "' " // &
            objlist // " " // &
            os_data%ldflags_hepmc // " " // &
            os_data%ldflags_lcio // " " // &
            os_data%ldflags_hoppet // " " // &
            os_data%ldflags_looptools
      else
         command_string = &
            os_data%ld // " " // &
            os_data%ldflags_so // " " // &
            os_data%fcflags // " " // &
            os_data%whizard_ldflags // " " // &
            os_data%ldflags // " " // &
            os_data%ldflags_static // " " // &
            "-o '" // exec_name // "' " // &
            objlist // " " // &
```

```
            os_data%ldflags_hepmc // " " // &
            os_data%ldflags_lcio // " " // &
            os_data%ldflags_hoppet // " " // &
            os_data%ldflags_looptools
      end if
      call os_system_call (command_string, status)
    end subroutine os_link_static
```

Determine the name of the shared library to link. If libtool is used, this is encoded in the `.la` file which resides in place of the library itself.

⟨*OS interface: public*⟩+≡

```
    public :: os_get_dlname
```

⟨*OS interface: procedures*⟩+≡

```
    function os_get_dlname (lib, os_data, ignore, silent) result (dlname)
      type(string_t) :: dlname
      type(string_t), intent(in) :: lib
      type(os_data_t), intent(in) :: os_data
      logical, intent(in), optional :: ignore, silent
      type(string_t) :: filename
      type(string_t) :: buffer
      logical :: exist, required, quiet
      integer :: u
      u = free_unit ()
      if (present (ignore)) then
         required = .not. ignore
      else
         required = .true.
      end if
           if (present (silent)) then
         quiet = silent
      else
         quiet = .false.
      end if
      if (os_data%use_libtool) then
         filename = lib // ".la"
         inquire (file=char(filename), exist=exist)
         if (exist) then
            open (unit=u, file=char(filename), action="read", status="old")
            SCAN_LTFILE: do
               call get (u, buffer)
               if (extract (buffer, 1, 7) == "dlname=") then
                  dlname = extract (buffer, 9)
                  dlname = remove (dlname, len (dlname))
                  exit SCAN_LTFILE
               end if
            end do SCAN_LTFILE
            close (u)
         else if (required) then
            if (.not. quiet) call msg_fatal (" Library '" // char (lib) &
                 // "': libtool archive not found")
            dlname = ""
         else
            if (.not. quiet) call msg_message ("[No compiled library '" &
```

```
            // char (lib) // "']")
         dlname = ""
      end if
   else
      dlname = lib // "." // os_data%fc_shrlib_ext
      inquire (file=char(dlname), exist=exist)
      if (.not. exist) then
         if (required) then
            if (.not. quiet) call msg_fatal (" Library '" // char (lib) &
                 // "' not found")
         else
            if (.not. quiet) call msg_message &
               ("[No compiled process library '" // char (lib) // "']")
            dlname = ""
         end if
      end if
   end if
end function os_get_dlname
```

5.5.8 Controlling OpenMP

OpenMP is handled automatically by the library for the most part. Here is a convenience routine for setting the number of threads, with some diagnostics.

⟨*OS interface: public*⟩+≡

```
public :: openmp_set_num_threads_verbose
```

⟨*OS interface: procedures*⟩+≡

```
subroutine openmp_set_num_threads_verbose (num_threads, openmp_logging)
  integer, intent(in) :: num_threads
  integer :: n_threads
  logical, intent(in), optional :: openmp_logging
  logical :: logging
  if (present (openmp_logging)) then
     logging = openmp_logging
  else
     logging = .true.
  end if
  n_threads = num_threads
  if (openmp_is_active ()) then
     if (num_threads == 1) then
        if (logging) then
           write (msg_buffer, "(A,I0,A)")  "OpenMP: Using ", num_threads, &
                " thread"
           call msg_message
        end if
        n_threads = num_threads
     else if (num_threads > 1) then
        if (logging) then
           write (msg_buffer, "(A,I0,A)")  "OpenMP: Using ", num_threads, &
                " threads"
           call msg_message
        end if
        n_threads = num_threads
```

```
      else
         if (logging) then
            write (msg_buffer, "(A,I0,A)")  "OpenMP: " &
                 // "Illegal value of openmp_num_threads (", num_threads, &
              ") ignored"
            call msg_error
         end if
         n_threads = openmp_get_default_max_threads ()
         if (logging) then
            write (msg_buffer, "(A,I0,A)")  "OpenMP: Using ", &
                 n_threads, " threads"
            call msg_message
         end if
      end if
      if (n_threads > openmp_get_default_max_threads ()) then
         if (logging) then
            write (msg_buffer, "(A,I0)")  "OpenMP: " &
                 // "Number of threads is greater than library default of ", &
                 openmp_get_default_max_threads ()
            call msg_warning
         end if
      end if
      call openmp_set_num_threads (n_threads)
   else if (num_threads /= 1) then
      if (logging) then
         write (msg_buffer, "(A,I0,A)")  "openmp_num_threads set to ", &
              num_threads, ", but OpenMP is not active: ignored"
         call msg_warning
      end if
   end if
end subroutine openmp_set_num_threads_verbose
```

5.5.9 Unit tests

Test module, followed by the corresponding implementation module.

⟨os_interface_ut.f90⟩≡

```
⟨File header⟩

module os_interface_ut
  use unit_tests
  use os_interface_uti

⟨Standard module head⟩

⟨OS interface: public test⟩

contains

⟨OS interface: test driver⟩

end module os_interface_ut
```

⟨os_interface_uti.f90⟩≡

```
⟨File header⟩

module os_interface_uti

  use, intrinsic :: iso_c_binding !NODEP!

⟨Use strings⟩
  use io_units

  use os_interface

⟨Standard module head⟩

⟨OS interface: test declarations⟩

contains

⟨OS interface: tests⟩

end module os_interface_uti
```

API: driver for the unit tests below.

⟨*OS interface: public test*⟩≡

```
  public :: os_interface_test
```

⟨*OS interface: test driver*⟩≡

```
  subroutine os_interface_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨OS interface: execute tests⟩
  end subroutine os_interface_test
```

Write a Fortran source file, compile it to a shared library, load it, and execute the contained function.

⟨*OS interface: execute tests*⟩≡

```
  call test (os_interface_1, "os_interface_1", &
       "check OS interface routines", &
       u, results)
```

⟨*OS interface: test declarations*⟩≡

```
  public :: os_interface_1
```

⟨*OS interface: tests*⟩≡

```
  subroutine os_interface_1 (u)
    integer, intent(in) :: u
    type(dlaccess_t) :: dlaccess
    type(string_t) :: fname, libname, ext
    type(os_data_t) :: os_data
    type(string_t) :: filename_src, filename_obj
    abstract interface
       function so_test_proc (i) result (j) bind(C)
         import c_int
         integer(c_int), intent(in) :: i
         integer(c_int) :: j
```

```
   end function so_test_proc
end interface
procedure(so_test_proc), pointer :: so_test => null ()
type(c_funptr) :: c_fptr
integer :: unit
integer(c_int) :: i
call os_data_init (os_data)
fname = "so_test"
filename_src = fname // os_data%fc_src_ext
if (os_data%use_libtool) then
   ext = ".lo"
else
   ext = os_data%obj_ext
end if
filename_obj = fname // ext
libname = fname // '.' // os_data%fc_shrlib_ext

write (u, "(A)")  "* Test output: OS interface"
write (u, "(A)")  "*   Purpose: check os_interface routines"
write (u, "(A)")

write (u, "(A)")  "* write source file 'so_test.f90'"
write (u, "(A)")
unit = free_unit ()
open (unit=unit, file=char(filename_src), action="write")
write (unit, "(A)")  "function so_test (i) result (j) bind(C)"
write (unit, "(A)")  "  use iso_c_binding"
write (unit, "(A)")  "  integer(c_int), intent(in) :: i"
write (unit, "(A)")  "  integer(c_int) :: j"
write (unit, "(A)")  "  j = 2 * i"
write (unit, "(A)")  "end function so_test"
close (unit)
write (u, "(A)")  "* compile and link as 'so_test.so/dylib'"
write (u, "(A)")
call os_compile_shared (fname, os_data)
call os_link_shared (filename_obj, fname, os_data)
write (u, "(A)")  "* load library 'so_test.so/dylib'"
write (u, "(A)")
call dlaccess_init (dlaccess, var_str ("."), libname, os_data)
if (dlaccess_is_open (dlaccess)) then
   write (u, "(A)") "     success"
else
   write (u, "(A)") "     failure"
end if
write (u, "(A)")  "* load symbol 'so_test'"
write (u, "(A)")
c_fptr = dlaccess_get_c_funptr (dlaccess, fname)
if (c_associated (c_fptr)) then
   write (u, "(A)") "     success"
else
   write (u, "(A)") "     failure"
end if
call c_f_procpointer (c_fptr, so_test)
write (u, "(A)") "* Execute function from 'so_test.so/dylib'"
```

```
    i = 7
    write (u, "(A,1x,I1)")  "     input  = ", i
    write (u, "(A,1x,I1)")  "     result = ", so_test(i)
    if (so_test(i) / i .ne. 2) then
       write (u, "(A)")  "* Compiling and linking ISO C functions failed."
    else
       write (u, "(A)")  "* Successful."
    end if
    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"
    call dlaccess_final (dlaccess)
  end subroutine os_interface_1
```

5.6 Interface for formatted I/O

For access to formatted printing (possibly input), we interface the C `printf` family of functions. There are two important issues here:

1. `printf` takes an arbitrary number of arguments, relying on the C stack. This is not interoperable. We interface it with C wrappers that output a single integer, real or string and restrict the allowed formats accordingly.

2. Restricting format strings is essential also for preventing format string attacks. Allowing arbitrary format string would create a real security hole in a Fortran program.

3. The string returned by `sprintf` must be allocated to the right size.

⟨`formats.f90`⟩≡
  ⟨*File header*⟩

```
  module formats

    use, intrinsic :: iso_c_binding
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩

```
    use io_units
    use diagnostics
```

  ⟨*Standard module head*⟩

  ⟨*Formats: public*⟩

  ⟨*Formats: parameters*⟩

  ⟨*Formats: types*⟩

  ⟨*Formats: interfaces*⟩

```
  contains
```

⟨*Formats: procedures*⟩

```
end module formats
```

5.6.1 Parsing a C format string

The C format string contains characters and format conversion specifications. The latter are initiated by a `%` sign. If the next letter is also a `%`, a percent sign is printed and no conversion is done. Otherwise, a conversion is done and applied to the next argument in the argument list. First comes an optional flag (`#`, `0`, `-`, `+`, or space), an optional field width (decimal digits starting not with zero), an optional precision (period, then another decimal digit string), a length modifier (irrelevant for us, therefore not supported), and a conversion specifier: `d` or `i` for integer; `e`, `f`, `g` (also upper case) for double-precision real, `s` for a string.

We explicitly exclude all other conversion specifiers, and we check the specifiers against the actual arguments.

A type for passing arguments

This is a polymorphic type that can hold integer, real (double), and string arguments.

⟨*Formats: parameters*⟩≡

```
integer, parameter, public :: ARGTYPE_NONE = 0
integer, parameter, public :: ARGTYPE_LOG = 1
integer, parameter, public :: ARGTYPE_INT = 2
integer, parameter, public :: ARGTYPE_REAL = 3
integer, parameter, public :: ARGTYPE_STR = 4
```

The integer and real entries are actually scalars, but we avoid relying on the allocatable-scalar feature and make them one-entry arrays. The character entry is a real array which is a copy of the string.

Logical values are mapped to strings (true or false), so this type parameter value is mostly unused.

⟨*Formats: public*⟩≡

```
public :: sprintf_arg_t
```

⟨*Formats: types*⟩≡

```
type :: sprintf_arg_t
  private
  integer :: type = ARGTYPE_NONE
  integer(c_int), dimension(:), allocatable :: ival
  real(c_double), dimension(:), allocatable :: rval
  character(c_char), dimension(:), allocatable :: sval
end type sprintf_arg_t
```

⟨*Formats: public*⟩+≡

```
public :: sprintf_arg_init
```

⟨*Formats: interfaces*⟩≡

```
interface sprintf_arg_init
   module procedure sprintf_arg_init_log
   module procedure sprintf_arg_init_int
```

```
   module procedure sprintf_arg_init_real
   module procedure sprintf_arg_init_str
end interface
```

⟨*Formats: procedures*⟩≡

```
subroutine sprintf_arg_init_log (arg, lval)
  type(sprintf_arg_t), intent(out) :: arg
  logical, intent(in) :: lval
  arg%type = ARGTYPE_STR
  if (lval) then
     allocate (arg%sval (5))
     arg%sval = ['t', 'r', 'u', 'e', c_null_char]
  else
     allocate (arg%sval (6))
     arg%sval = ['f', 'a', 'l', 's', 'e', c_null_char]
  end if
end subroutine sprintf_arg_init_log

subroutine sprintf_arg_init_int (arg, ival)
  type(sprintf_arg_t), intent(out) :: arg
  integer, intent(in) :: ival
  arg%type = ARGTYPE_INT
  allocate (arg%ival (1))
  arg%ival = ival
end subroutine sprintf_arg_init_int

subroutine sprintf_arg_init_real (arg, rval)
  type(sprintf_arg_t), intent(out) :: arg
  real(default), intent(in) :: rval
  arg%type = ARGTYPE_REAL
  allocate (arg%rval (1))
  arg%rval = rval
end subroutine sprintf_arg_init_real

subroutine sprintf_arg_init_str (arg, sval)
  type(sprintf_arg_t), intent(out) :: arg
  type(string_t), intent(in) :: sval
  integer :: i
  arg%type = ARGTYPE_STR
  allocate (arg%sval (len (sval) + 1))
  do i = 1, len (sval)
     arg%sval(i) = extract (sval, i, i)
  end do
  arg%sval(len (sval) + 1) = c_null_char
end subroutine sprintf_arg_init_str
```

⟨*Formats: procedures*⟩+≡

```
subroutine sprintf_arg_write (arg, unit)
  type(sprintf_arg_t), intent(in) :: arg
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  select case (arg%type)
```

```
    case (ARGTYPE_NONE)
      write (u, *) "[none]"
    case (ARGTYPE_INT)
      write (u, "(1x,A,1x)", advance = "no")  "[int]"
      write (u, *)  arg%ival
    case (ARGTYPE_REAL)
      write (u, "(1x,A,1x)", advance = "no")  "[real]"
      write (u, *)  arg%rval
    case (ARGTYPE_STR)
      write (u, "(1x,A,1x,A)", advance = "no")  "[string]", '"'
      write (u, *)  arg%rval, '"'
    end select
  end subroutine sprintf_arg_write
```

Return an upper bound for the length of the printed version; in case of strings the result is exact.

⟨*Formats: procedures*⟩+≡

```
  elemental function sprintf_arg_get_length (arg) result (length)
    integer :: length
    type(sprintf_arg_t), intent(in) :: arg
    select case (arg%type)
    case (ARGTYPE_INT)
       length = log10 (real (huge (arg%ival(1)))) + 2
    case (ARGTYPE_REAL)
       length = log10 (real (radix (arg%rval(1))) ** digits (arg%rval(1))) + 8
    case (ARGTYPE_STR)
       length = size (arg%sval)
    case default
       length = 0
    end select
  end function sprintf_arg_get_length
```

⟨*Formats: procedures*⟩+≡

```
  subroutine sprintf_arg_apply_sprintf (arg, fmt, result, actual_length)
    type(sprintf_arg_t), intent(in) :: arg
    character(c_char), dimension(:), intent(in) :: fmt
    character(c_char), dimension(:), intent(inout) :: result
    integer, intent(out) :: actual_length
    integer(c_int) :: ival
    real(c_double) :: rval
    select case (arg%type)
    case (ARGTYPE_NONE)
      actual_length = sprintf_none (result, fmt)
    case (ARGTYPE_INT)
      ival = arg%ival(1)
      actual_length = sprintf_int (result, fmt, ival)
    case (ARGTYPE_REAL)
      rval = arg%rval(1)
      actual_length = sprintf_double (result, fmt, rval)
    case (ARGTYPE_STR)
      actual_length = sprintf_str (result, fmt, arg%sval)
    case default
      call msg_bug ("sprintf_arg_apply_sprintf called with illegal type")
```

```
    end select
    if (actual_length < 0) then
       write (msg_buffer, *) "Format: '", fmt, "'"
       call msg_message ()
       write (msg_buffer, *) "Output: '", result, "'"
       call msg_message ()
       call msg_error ("I/O error in sprintf call")
       actual_length = 0
    end if
  end subroutine sprintf_arg_apply_sprintf
```

Container type for the output

There is a procedure which chops the format string into pieces that contain at most one conversion specifier. Pairing this with a `sprintf_arg` object, we get the actual input to the `sprintf` interface. The type below holds this input and can allocate the output string.

⟨*Formats: types*⟩+≡

```
  type :: sprintf_interface_t
    private
    character(c_char), dimension(:), allocatable :: input_fmt
    type(sprintf_arg_t) :: arg
    character(c_char), dimension(:), allocatable :: output_str
    integer :: output_str_len = 0
  end type sprintf_interface_t
```

⟨*Formats: procedures*⟩+≡

```
  subroutine sprintf_interface_init (intf, fmt, arg)
    type(sprintf_interface_t), intent(out) :: intf
    type(string_t), intent(in) :: fmt
    type(sprintf_arg_t), intent(in) :: arg
    integer :: fmt_len, i
    fmt_len = len (fmt)
    allocate (intf%input_fmt (fmt_len + 1))
    do i = 1, fmt_len
       intf%input_fmt(i) = extract (fmt, i, i)
    end do
    intf%input_fmt(fmt_len+1) = c_null_char
    intf%arg = arg
    allocate (intf%output_str (len (fmt) + sprintf_arg_get_length (arg) + 1))
  end subroutine sprintf_interface_init
```

⟨*Formats: procedures*⟩+≡

```
  subroutine sprintf_interface_write (intf, unit)
    type(sprintf_interface_t), intent(in) :: intf
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, *) "Format string = ", '"', intf%input_fmt, '"'
    write (u, "(1x,A,1x)", advance = "no") "Argument = "
    call sprintf_arg_write (intf%arg, unit)
```

```
    if (intf%output_str_len > 0) then
       write (u, *) "Result string = ", &
            '"', intf%output_str (1:intf%output_str_len), '"'
    end if
  end subroutine sprintf_interface_write
```

Return the output string:

⟨*Formats: procedures*⟩+≡

```
  function sprintf_interface_get_result (intf) result (string)
    type(string_t) :: string
    type(sprintf_interface_t), intent(in) :: intf
    character(kind = c_char, len = max (intf%output_str_len, 0)) :: buffer
    integer :: i
    if (intf%output_str_len > 0) then
       do i = 1, intf%output_str_len
          buffer(i:i) = intf%output_str(i)
       end do
       string = buffer(1:intf%output_str_len)
    else
       string = ""
    end if
  end function sprintf_interface_get_result
```

⟨*Formats: procedures*⟩+≡

```
  subroutine sprintf_interface_apply_sprintf (intf)
    type(sprintf_interface_t), intent(inout) :: intf
    call sprintf_arg_apply_sprintf &
         (intf%arg, intf%input_fmt, intf%output_str, intf%output_str_len)
  end subroutine sprintf_interface_apply_sprintf
```

Import the interfaces defined in the previous section:

⟨*Formats: interfaces*⟩+≡
  ⟨*sprintf interfaces*⟩

Scan the format string

Chop it into pieces that contain one conversion specifier each. The zero-th piece contains the part before the first specifier. Check the specifiers and allow only the subset that we support. Also check for an exact match between conversion specifiers and input arguments. The result is an allocated array of `sprintf_interface` object; each one contains a piece of the format string and the corresponding argument.

⟨*Formats: procedures*⟩+≡

```
  subroutine chop_and_check_format_string (fmt, arg, intf)
    type(string_t), intent(in) :: fmt
    type(sprintf_arg_t), dimension(:), intent(in) :: arg
    type(sprintf_interface_t), dimension(:), intent(out), allocatable :: intf
    integer :: n_args, i
    type(string_t), dimension(:), allocatable :: split_fmt
    type(string_t) :: word, buffer, separator
```

```
integer :: pos, length, l
logical :: ok
type(sprintf_arg_t) :: arg_null
ok = .true.
length = 0
n_args = size (arg)
allocate (split_fmt (0:n_args))
split_fmt = ""
buffer = fmt
SCAN_ARGS: do i = 1, n_args
   FIND_CONVERSION: do
      call split (buffer, word, "%", separator=separator)
      if (separator == "") then
         call msg_message ('"' // char (fmt) // '"')
         call msg_error ("C-formatting string: " &
              // "too few conversion specifiers in format string")
         ok = .false.;  exit SCAN_ARGS
      end if
      split_fmt(i-1) = split_fmt(i-1) // word
      if (extract (buffer, 1, 1) /= "%") then
         split_fmt(i) = "%"
         exit FIND_CONVERSION
      else
         split_fmt(i-1) = split_fmt(i-1) // "%"
      end if
   end do FIND_CONVERSION
   pos = verify (buffer, "#0-+ ")   ! Flag characters (zero or more)
   split_fmt(i) = split_fmt(i) // extract (buffer, 1, pos-1)
   buffer = remove (buffer, 1, pos-1)
   pos = verify (buffer, "123456890")  ! Field width
   word = extract (buffer, 1, pos-1)
   if (len (word) /= 0) then
     call read_int_from_string (word, len (word), l)
     length = length + l
   end if
   split_fmt(i) = split_fmt(i) // word
   buffer = remove (buffer, 1, pos-1)
   if (extract (buffer, 1, 1) == ".") then
      buffer = remove (buffer, 1, 1)
      pos = verify (buffer, "1234567890")   ! Precision
      split_fmt(i) = split_fmt(i) // "." // extract (buffer, 1, pos-1)
      buffer = remove (buffer, 1, pos-1)
   end if
   ! Length modifier would come here, but is not allowed
   select case (char (extract (buffer, 1, 1)))  ! conversion specifier
   case ("d", "i")
      if (arg(i)%type /= ARGTYPE_INT) then
         call msg_message ('"' // char (fmt) // '"')
         call msg_error ("C-formatting string: " &
              // "argument type mismatch: integer value expected")
         ok = .false.;  exit SCAN_ARGS
      end if
   case ("e", "E", "f", "F", "g", "G")
      if (arg(i)%type /= ARGTYPE_REAL) then
```

```
            call msg_message ('"' // char (fmt) // '"')
            call msg_error ("C-formatting string: " &
                 // "argument type mismatch: real value expected")
            ok = .false.;  exit SCAN_ARGS
         end if
      case ("s")
         if (arg(i)%type /= ARGTYPE_STR) then
            call msg_message ('"' // char (fmt) // '"')
            call msg_error ("C-formatting string: " &
                 // "argument type mismatch: logical or string value expected")
            ok = .false.;  exit SCAN_ARGS
         end if
      case default
         call msg_message ('"' // char (fmt) // '"')
         call msg_error ("C-formatting string: " &
              // "illegal or incomprehensible conversion specifier")
         ok = .false.;  exit SCAN_ARGS
      end select
      split_fmt(i) = split_fmt(i) // extract (buffer, 1, 1)
      buffer = remove (buffer, 1, 1)
   end do SCAN_ARGS
   if (ok) then
      FIND_EXTRA_CONVERSION: do
         call split (buffer, word, "%", separator=separator)
         split_fmt(n_args) = split_fmt(n_args) // word // separator
         if (separator == "")  exit FIND_EXTRA_CONVERSION
         if (extract (buffer, 1, 1) == "%") then
            split_fmt(n_args) = split_fmt(n_args) // "%"
            buffer = remove (buffer, 1, 1)
         else
            call msg_message ('"' // char (fmt) // '"')
            call msg_error ("C-formatting string: " &
                 // "too many conversion specifiers in format string")
            ok = .false.;  exit FIND_EXTRA_CONVERSION
         end if
      end do FIND_EXTRA_CONVERSION
      split_fmt(n_args) = split_fmt(n_args) // buffer
      allocate (intf (0:n_args))
      call sprintf_interface_init (intf(0), split_fmt(0), arg_null)
      do i = 1, n_args
         call sprintf_interface_init (intf(i), split_fmt(i), arg(i))
      end do
   else
      allocate (intf (0))
   end if
contains
   subroutine read_int_from_string (word, length, l)
     type(string_t), intent(in) :: word
     integer, intent(in) :: length
     integer, intent(out) :: l
     character(len=length) :: buffer
     buffer = word
     read (buffer, *) l
   end subroutine read_int_from_string
```

```
    end subroutine chop_and_check_format_string
```

5.6.2 API

⟨*Formats: public*⟩+≡

```
    public :: sprintf
```

⟨*Formats: procedures*⟩+≡

```
    function sprintf (fmt, arg) result (string)
      type(string_t) :: string
      type(string_t), intent(in) :: fmt
      type(sprintf_arg_t), dimension(:), intent(in) :: arg
      type(sprintf_interface_t), dimension(:), allocatable :: intf
      integer :: i
      string = ""
      call chop_and_check_format_string (fmt, arg, intf)
      if (size (intf) > 0) then
         do i = 0, ubound (intf, 1)
            call sprintf_interface_apply_sprintf (intf(i))
            string = string // sprintf_interface_get_result (intf(i))
         end do
      end if
    end function sprintf
```

5.6.3 Unit tests

Test module, followed by the corresponding implementation module.

⟨`formats_ut.f90`⟩≡

```
  ⟨File header⟩

  module formats_ut
    use unit_tests
    use formats_uti

  ⟨Standard module head⟩

  ⟨Formats: public test⟩

  contains

  ⟨Formats: test driver⟩

  end module formats_ut
```

⟨`formats_uti.f90`⟩≡

```
  ⟨File header⟩

  module formats_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
```

```
  use formats

⟨Standard module head⟩

⟨Formats: test declarations⟩

⟨Formats: test types⟩

contains

⟨Formats: tests⟩

end module formats_uti
```

API: driver for the unit tests below.

⟨*Formats: public test*⟩≡

```
  public :: format_test
```

⟨*Formats: test driver*⟩≡

```
  subroutine format_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨Formats: execute tests⟩
  end subroutine format_test
```

⟨*Formats: execute tests*⟩≡

```
  call test (format_1, "format_1", &
       "check formatting routines", &
       u, results)
```

⟨*Formats: test declarations*⟩≡

```
  public :: format_1
```

⟨*Formats: tests*⟩≡

```
  subroutine format_1 (u)
    integer, intent(in) :: u
    write (u, "(A)")  "*** Test 1: a string ***"
    write (u, "(A)")
    call test_run (var_str("%s"), 1, [4], ['abcdefghij'], u)
    write (u, "(A)")  "*** Test 2: two integers ***"
    write (u, "(A)")
    call test_run (var_str("%d,%d"), 2, [2, 2], ['42', '13'], u)
    write (u, "(A)")  "*** Test 3: floating point number ***"
    write (u, "(A)")
    call test_run (var_str("%8.4f"), 1, [3], ['42567.12345'], u)
    write (u, "(A)")  "*** Test 4: general expression ***"
    call test_run (var_str("%g"), 1, [3], ['3.1415'], u)
    contains
      subroutine test_run (fmt, n_args, type, buffer, unit)
        type(string_t), intent(in) :: fmt
        integer, intent(in) :: n_args, unit
        logical :: lval
        integer :: ival
        real(default) :: rval
        integer :: i
        type(string_t) :: string
```

```
      type(sprintf_arg_t), dimension(:), allocatable :: arg
      integer, dimension(n_args), intent(in) :: type
      character(*), dimension(n_args), intent(in) :: buffer
      write (unit, "(A,A)")   "Format string :", char(fmt)
      write (unit, "(A,I1)")  "Number of args:", n_args
      allocate (arg (n_args))
      do i = 1, n_args
         write (unit, "(A,I1)")  "Argument (type ) = ", type(i)
         select case (type(i))
         case (ARGTYPE_LOG)
            read (buffer(i), *)  lval
            call sprintf_arg_init (arg(i), lval)
         case (ARGTYPE_INT)
            read (buffer(i), *)  ival
            call sprintf_arg_init (arg(i), ival)
         case (ARGTYPE_REAL)
            read (buffer(i), *)  rval
            call sprintf_arg_init (arg(i), rval)
         case (ARGTYPE_STR)
            call sprintf_arg_init (arg(i), var_str (trim (buffer(i))))
         end select
       end do
       string = sprintf (fmt, arg)
       write (unit, "(A,A,A)")  "Result: '", char (string), "'"
       deallocate (arg)
     end subroutine test_run
  end subroutine format_1
```

5.7 CPU timing

The time is stored in a simple derived type which just holds a floating-point number.

⟨cputime.f90⟩≡

```
⟨File header⟩

module cputime

⟨Use kinds⟩
  use io_units
⟨Use strings⟩
  use diagnostics

⟨Standard module head⟩

⟨CPU time: public⟩

⟨CPU time: types⟩

⟨CPU time: interfaces⟩

contains
```

⟨*CPU time: procedures*⟩

```
end module cputime
```

The CPU time is a floating-point number with an arbitrary reference time. It is single precision (default real, not `real(default)`). It is measured in seconds.

⟨*CPU time: public*⟩≡

```
public :: time_t
```

⟨*CPU time: types*⟩≡

```
type :: time_t
   private
   logical :: known = .false.
   real :: value = 0
 contains
 ⟨CPU time: time: TBP⟩
end type time_t
```

⟨*CPU time: time: TBP*⟩≡

```
procedure :: write => time_write
```

⟨*CPU time: procedures*⟩≡

```
subroutine time_write (object, unit)
  class(time_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)", advance="no")  "Time in seconds ="
  if (object%known) then
     write (u, "(1x,ES10.3)")  object%value
  else
     write (u, "(1x,A)")  "[unknown]"
  end if
end subroutine time_write
```

Set the current time

⟨*CPU time: time: TBP*⟩+≡

```
procedure :: set_current => time_set_current
```

⟨*CPU time: procedures*⟩+≡

```
subroutine time_set_current (time)
  class(time_t), intent(out) :: time
  integer :: msecs
  call system_clock (msecs)
  time%value = real (msecs) / 1000.
  time%known = time%value > 0
end subroutine time_set_current
```

Assign to a `real(default` value. If the time is undefined, return zero.

⟨*CPU time: public*⟩+≡

```
public :: assignment(=)
```

⟨*CPU time: interfaces*⟩≡

```
interface assignment(=)
  module procedure real_assign_time
  module procedure real_default_assign_time
end interface
```

⟨*CPU time: procedures*⟩+≡

```
pure subroutine real_assign_time (r, time)
  real, intent(out) :: r
  class(time_t), intent(in) :: time
  if (time%known) then
     r = time%value
  else
     r = 0
  end if
end subroutine real_assign_time

pure subroutine real_default_assign_time (r, time)
  real(default), intent(out) :: r
  class(time_t), intent(in) :: time
  if (time%known) then
     r = time%value
  else
     r = 0
  end if
end subroutine real_default_assign_time
```

Assign an integer or (single precision) real value to the time object.

⟨*CPU time: time: TBP*⟩+≡

```
generic :: assignment(=) => time_assign_from_integer, time_assign_from_real
procedure, private :: time_assign_from_integer
procedure, private :: time_assign_from_real
```

⟨*CPU time: procedures*⟩+≡

```
subroutine time_assign_from_integer (time, ival)
  class(time_t), intent(out) :: time
  integer, intent(in) :: ival
  time%value = ival
  time%known = .true.
end subroutine time_assign_from_integer

subroutine time_assign_from_real (time, rval)
  class(time_t), intent(out) :: time
  real, intent(in) :: rval
  time%value = rval
  time%known = .true.
end subroutine time_assign_from_real
```

Add times and compute time differences. If any input value is undefined, the result is undefined.

⟨*CPU time: time: TBP*⟩+≡

```
generic :: operator(-) => subtract_times
generic :: operator(+) => add_times
```

```
procedure, private :: subtract_times
procedure, private :: add_times
```

⟨*CPU time: procedures*⟩+≡

```
pure function subtract_times (t_end, t_begin) result (time)
  type(time_t) :: time
  class(time_t), intent(in) :: t_end, t_begin
  if (t_end%known .and. t_begin%known) then
     time%known = .true.
     time%value = t_end%value - t_begin%value
  end if
end function subtract_times

pure function add_times (t1, t2) result (time)
  type(time_t) :: time
  class(time_t), intent(in) :: t1, t2
  if (t1%known .and. t2%known) then
     time%known = .true.
     time%value = t1%value + t2%value
  end if
end function add_times
```

Check if a time is known, so we can use it:

⟨*CPU time: time: TBP*⟩+≡

```
procedure :: is_known => time_is_known
```

⟨*CPU time: procedures*⟩+≡

```
function time_is_known (time) result (flag)
  class(time_t), intent(in) :: time
  logical :: flag
  flag = time%known
end function time_is_known
```

We define functions for converting the time into ss / mm:ss / hh:mm:ss / dd:mm:hh:ss.

⟨*CPU time: time: TBP*⟩+≡

```
generic :: expand => time_expand_s, time_expand_ms, &
     time_expand_hms, time_expand_dhms
procedure, private :: time_expand_s
procedure, private :: time_expand_ms
procedure, private :: time_expand_hms
procedure, private :: time_expand_dhms
```

⟨*CPU time: procedures*⟩+≡

```
subroutine time_expand_s (time, sec)
  class(time_t), intent(in) :: time
  integer, intent(out) :: sec
  if (time%known) then
     sec = time%value
  else
     call msg_bug ("Time: attempt to expand undefined value")
  end if
end subroutine time_expand_s
```

```
subroutine time_expand_ms (time, min, sec)
  class(time_t), intent(in) :: time
  integer, intent(out) :: min, sec
  if (time%known) then
     if (time%value >= 0) then
        sec = mod (int (time%value), 60)
     else
        sec = - mod (int (- time%value), 60)
     end if
     min = time%value / 60
  else
     call msg_bug ("Time: attempt to expand undefined value")
  end if
end subroutine time_expand_ms

subroutine time_expand_hms (time, hour, min, sec)
  class(time_t), intent(in) :: time
  integer, intent(out) :: hour, min, sec
  call time%expand (min, sec)
  hour = min / 60
  if (min >= 0) then
     min = mod (min, 60)
  else
     min = - mod (-min, 60)
  end if
end subroutine time_expand_hms

subroutine time_expand_dhms (time, day, hour, min, sec)
  class(time_t), intent(in) :: time
  integer, intent(out) :: day, hour, min, sec
  call time%expand (hour, min, sec)
  day = hour / 24
  if (hour >= 0) then
     hour = mod (hour, 24)
  else
     hour = - mod (- hour, 24)
  end if
end subroutine time_expand_dhms
```

Use the above expansions to generate a time string.

⟨*CPU time: time: TBP*⟩+≡

```
procedure :: to_string_s => time_to_string_s
procedure :: to_string_ms => time_to_string_ms
procedure :: to_string_hms => time_to_string_hms
procedure :: to_string_dhms => time_to_string_dhms
```

⟨*CPU time: procedures*⟩+≡

```
function time_to_string_s (time) result (str)
  class(time_t), intent(in) :: time
  type(string_t) :: str
  character(256) :: buffer
  integer :: s
  call time%expand (s)
  write (buffer, "(I0,'s')")  s
```

```
  str = trim (buffer)
end function time_to_string_s

function time_to_string_ms (time, blank) result (str)
  class(time_t), intent(in) :: time
  logical, intent(in), optional :: blank
  type(string_t) :: str
  character(256) :: buffer
  integer :: s, m
  logical :: x_out
  x_out = .false.
  if (present (blank))  x_out = blank
  call time%expand (m, s)
  write (buffer, "(I0,'m:',I2.2,'s')")  m, abs (s)
  str = trim (buffer)
  if (x_out) then
     str = replace (str, len(str)-1, "X")
  end if
end function time_to_string_ms

function time_to_string_hms (time) result (str)
  class(time_t), intent(in) :: time
  type(string_t) :: str
  character(256) :: buffer
  integer :: s, m, h
  call time%expand (h, m, s)
  write (buffer, "(I0,'h:',I2.2,'m:',I2.2,'s')")  h, abs (m), abs (s)
  str = trim (buffer)
end function time_to_string_hms

function time_to_string_dhms (time) result (str)
  class(time_t), intent(in) :: time
  type(string_t) :: str
  character(256) :: buffer
  integer :: s, m, h, d
  call time%expand (d, h, m, s)
  write (buffer, "(I0,'d:',I2.2,'h:',I2.2,'m:',I2.2,'s')")  &
       d, abs (h), abs (m), abs (s)
  str = trim (buffer)
end function time_to_string_dhms
```

5.7.1 Timer

A timer can measure real (wallclock) time differences. The base type corresponds to the result, i.e., time difference. The object contains two further times for start and stop time.

⟨*CPU time: public*⟩+≡

```
public :: timer_t
```

⟨*CPU time: types*⟩+≡

```
type, extends (time_t) :: timer_t
   private
   logical :: running = .false.
```

```
    type(time_t) :: t1, t2
  contains
  ⟨CPU time: timer: TBP⟩
end type timer_t
```

Output. If the timer is running, we indicate this, otherwise write just the result.

⟨*CPU time: timer: TBP*⟩≡

```
procedure :: write => timer_write
```

⟨*CPU time: procedures*⟩+≡

```
subroutine timer_write (object, unit)
  class(timer_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  if (object%running) then
     write (u, "(1x,A)")  "Time in seconds = [running]"
  else
     call object%time_t%write (u)
  end if
end subroutine timer_write
```

Start the timer: store the current time in the first entry and adapt the status. We forget any previous values.

⟨*CPU time: timer: TBP*⟩+≡

```
procedure :: start => timer_start
```

⟨*CPU time: procedures*⟩+≡

```
subroutine timer_start (timer)
  class(timer_t), intent(out) :: timer
  call timer%t1%set_current ()
  timer%running = .true.
end subroutine timer_start
```

Restart the timer: simply adapt the status, keeping the start time.

⟨*CPU time: timer: TBP*⟩+≡

```
procedure :: restart => timer_restart
```

⟨*CPU time: procedures*⟩+≡

```
subroutine timer_restart (timer)
  class(timer_t), intent(inout) :: timer
  if (timer%t1%known .and. .not. timer%running) then
     timer%running = .true.
  else
     call msg_bug ("Timer: restart attempt from wrong status")
  end if
end subroutine timer_restart
```

Stop the timer: store the current time in the second entry, adapt the status, and compute the elapsed time.

⟨*CPU time: timer: TBP*⟩+≡

```
procedure :: stop => timer_stop
```

⟨*CPU time: procedures*⟩+≡

```
subroutine timer_stop (timer)
  class(timer_t), intent(inout) :: timer
  call timer%t2%set_current ()
  timer%running = .false.
  call timer%evaluate ()
end subroutine timer_stop
```

Manually set the time (for unit test)

⟨*CPU time: timer: TBP*⟩+≡

```
procedure :: set_test_time1 => timer_set_test_time1
procedure :: set_test_time2 => timer_set_test_time2
```

⟨*CPU time: procedures*⟩+≡

```
subroutine timer_set_test_time1 (timer, t)
  class(timer_t), intent(inout) :: timer
  integer, intent(in) :: t
  timer%t1 = t
end subroutine timer_set_test_time1

subroutine timer_set_test_time2 (timer, t)
  class(timer_t), intent(inout) :: timer
  integer, intent(in) :: t
  timer%t2 = t
end subroutine timer_set_test_time2
```

This is separate, available for the unit test.

⟨*CPU time: timer: TBP*⟩+≡

```
procedure :: evaluate => timer_evaluate
```

⟨*CPU time: procedures*⟩+≡

```
subroutine timer_evaluate (timer)
  class(timer_t), intent(inout) :: timer
  timer%time_t = timer%t2 - timer%t1
end subroutine timer_evaluate
```

5.7.2 Unit tests

Test module, followed by the corresponding implementation module.

⟨`cputime_ut.f90`⟩≡

```
⟨File header⟩

module cputime_ut
  use unit_tests
  use cputime_uti

⟨Standard module head⟩

⟨CPU time: public test⟩

contains
```

```
  ⟨CPU time: test driver⟩

  end module cputime_ut
```

⟨cputime_uti.f90⟩≡

```
  ⟨File header⟩

  module cputime_uti

  ⟨Use strings⟩

    use cputime

  ⟨Standard module head⟩

  ⟨CPU time: test declarations⟩

  contains

  ⟨CPU time: tests⟩

  end module cputime_uti
```

API: driver for the unit tests below.

⟨*CPU time: public test*⟩≡

```
    public :: cputime_test
```

⟨*CPU time: test driver*⟩≡

```
    subroutine cputime_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨CPU time: execute tests⟩
    end subroutine cputime_test
```

Basic tests

Check basic functions of the time object. The part which we can't check is getting the actual time from the system clock, since the output will not be reproducible. However, we can check time formats and operations.

⟨*CPU time: execute tests*⟩≡

```
    call test (cputime_1, "cputime_1", &
         "time operations", &
         u, results)
```

⟨*CPU time: test declarations*⟩≡

```
    public :: cputime_1
```

⟨*CPU time: tests*⟩≡

```
    subroutine cputime_1 (u)
      integer, intent(in) :: u
      type(time_t) :: time, time1, time2
      real :: t
      integer :: d, h, m, s

      write (u, "(A)")  "* Test output: cputime_1"
```

```
write (u, "(A)")  "*   Purpose: check time operations"
write (u, "(A)")

write (u, "(A)") "* Undefined time"
write (u, *)

call time%write (u)

write (u, *)
write (u, "(A)") "* Set time to zero"
write (u, *)

time = 0
call time%write (u)

write (u, *)
write (u, "(A)") "* Set time to 1.234 s"
write (u, *)

time = 1.234
call time%write (u)

t = time
write (u, "(1x,A,F6.3)")  "Time as real =", t

write (u, *)
write (u, "(A)") "* Compute time difference"
write (u, *)

time1 = 5.33
time2 = 7.55
time = time2 - time1

call time1%write (u)
call time2%write (u)
call time%write (u)

write (u, *)
write (u, "(A)") "* Compute time sum"
write (u, *)

time = time2 + time1

call time1%write (u)
call time2%write (u)
call time%write (u)

write (u, *)
write (u, "(A)") "* Expand time"
write (u, *)

time1 = ((24 + 1) * 60 + 1) * 60 + 1
time2 = ((3 * 24 + 23) * 60 + 59) * 60 + 59
```

```
    call time1%expand (s)
    write (u, 1)  "s =", s
    call time1%expand (m,s)
    write (u, 1)  "ms =", m, s
    call time1%expand (h,m,s)
    write (u, 1)  "hms =", h, m, s
    call time1%expand (d,h,m,s)
    write (u, 1)  "dhms =", d, h, m, s

    call time2%expand (s)
    write (u, 1)  "s =", s
    call time2%expand (m,s)
    write (u, 1)  "ms =", m, s
    call time2%expand (h,m,s)
    write (u, 1)  "hms =", h, m, s
    call time2%expand (d,h,m,s)
    write (u, 1)  "dhms =", d, h, m, s

    write (u, *)
    write (u, "(A)") "* Expand negative time"
    write (u, *)

    time1 = - (((24 + 1) * 60 + 1) * 60 + 1)
    time2 = - (((3 * 24 + 23) * 60 + 59) * 60 + 59)

    call time1%expand (s)
    write (u, 1)  "s =", s
    call time1%expand (m,s)
    write (u, 1)  "ms =", m, s
    call time1%expand (h,m,s)
    write (u, 1)  "hms =", h, m, s
    call time1%expand (d,h,m,s)
    write (u, 1)  "dhms =", d, h, m, s

    call time2%expand (s)
    write (u, 1)  "s =", s
    call time2%expand (m,s)
    write (u, 1)  "ms =", m, s
    call time2%expand (h,m,s)
    write (u, 1)  "hms =", h, m, s
    call time2%expand (d,h,m,s)
    write (u, 1)  "dhms =", d, h, m, s

1   format (1x,A,1x,4(I0,:,':'))

    write (u, *)
    write (u, "(A)") "* String from time"
    write (u, *)

    time1 = ((24 + 1) * 60 + 1) * 60 + 1
    time2 = ((3 * 24 + 23) * 60 + 59) * 60 + 59

    write (u, "(A)")  char (time1%to_string_s ())
    write (u, "(A)")  char (time1%to_string_ms ())
```

```
    write (u, "(A)")  char (time1%to_string_hms ())
    write (u, "(A)")  char (time1%to_string_dhms ())

    write (u, "(A)")  char (time2%to_string_s ())
    write (u, "(A)")  char (time2%to_string_ms ())
    write (u, "(A)")  char (time2%to_string_hms ())
    write (u, "(A)")  char (time2%to_string_dhms ())

    write (u, "(A)")
    write (u, "(A)")  "* Blanking out the last second entry"
    write (u, "(A)")

    write (u, "(A)")  char (time1%to_string_ms ())
    write (u, "(A)")  char (time1%to_string_ms (.true.))

    write (u, *)
    write (u, "(A)") "* String from negative time"
    write (u, *)

    time1 = -(((24 + 1) * 60 + 1) * 60 + 1)
    time2 = -(((3 * 24 + 23) * 60 + 59) * 60 + 59)

    write (u, "(A)")  char (time1%to_string_s ())
    write (u, "(A)")  char (time1%to_string_ms ())
    write (u, "(A)")  char (time1%to_string_hms ())
    write (u, "(A)")  char (time1%to_string_dhms ())

    write (u, "(A)")  char (time2%to_string_s ())
    write (u, "(A)")  char (time2%to_string_ms ())
    write (u, "(A)")  char (time2%to_string_hms ())
    write (u, "(A)")  char (time2%to_string_dhms ())

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: cputime_1"

  end subroutine cputime_1
```

Timer tests

Check a timer object.

⟨*CPU time: execute tests*⟩+≡

```
  call test (cputime_2, "cputime_2", &
      "timer", &
      u, results)
```

⟨*CPU time: test declarations*⟩+≡

```
  public :: cputime_2
```

⟨*CPU time: tests*⟩+≡

```
  subroutine cputime_2 (u)
    integer, intent(in) :: u
    type(timer_t) :: timer

    write (u, "(A)")  "* Test output: cputime_2"
```

```
  write (u, "(A)")  "*   Purpose: check timer"
  write (u, "(A)")

  write (u, "(A)") "* Undefined timer"
  write (u, *)

  call timer%write (u)

  write (u, *)
  write (u, "(A)") "* Start timer"
  write (u, *)

  call timer%start ()
  call timer%write (u)

  write (u, *)
  write (u, "(A)") "* Stop timer (injecting fake timings)"
  write (u, *)

  call timer%stop ()
  call timer%set_test_time1 (2)
  call timer%set_test_time2 (5)
  call timer%evaluate ()
  call timer%write (u)

  write (u, *)
  write (u, "(A)") "* Restart timer"
  write (u, *)

  call timer%restart ()
  call timer%write (u)

  write (u, *)
  write (u, "(A)") "* Stop timer again (injecting fake timing)"
  write (u, *)

  call timer%stop ()
  call timer%set_test_time2 (10)
  call timer%evaluate ()
  call timer%write (u)

  write (u, *)
  write (u, "(A)")  "* Test output end: cputime_2"

end subroutine cputime_2
```

Chapter 6

Combinatorics

These modules implement standard algorithms (sorting, hashing, etc.) that are not available in Fortran.

Fortran doesn't support generic programming, therefore the algorithms are implemented only for specific data types.

bytes Derived types for bytes and words.

hashes Types and tools for setting up hashtables.

md5 The MD5 algorithm for message digest.

permutations Permuting an array of integers.

sorting Sorting integer and real values.

grids d-dimensional grids can be saved to disk and used for interpolation, maximum finding, etc.

6.1 Bytes and such

In a few instances we will need the notion of a byte (8-bit) and a word (32 bit), even a 64-bit word. A block of 512 bit is also needed (for MD5).

We rely on integers up to 64 bit being supported by the processor. The main difference to standard integers is the interpretation as unsigned integers.

⟨`bytes.f90`⟩≡

```
⟨File header⟩

module bytes

  use kinds, only: i8, i32, i64
  use io_units

⟨Standard module head⟩

⟨Bytes: public⟩

⟨Bytes: types⟩
```

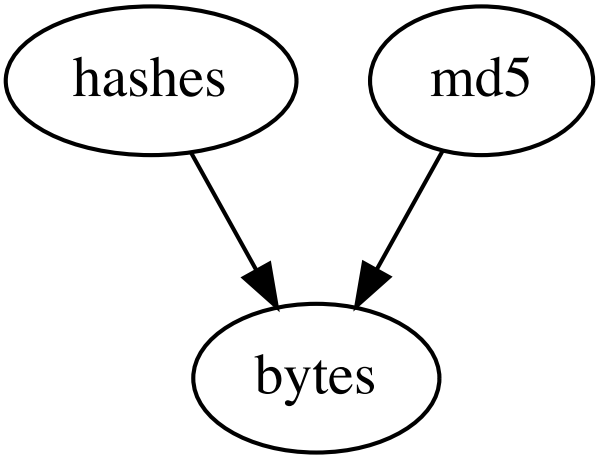

Figure 6.1: Module dependencies in `src/combinatorics`.

⟨*Bytes: parameters*⟩

⟨*Bytes: interfaces*⟩

```
contains
```

⟨*Bytes: procedures*⟩

```
end module bytes
```

6.1.1 8-bit words: bytes

This is essentially a wrapper around 8-bit integers. The wrapper emphasises their special interpretation as a sequence of bits. However, we interpret bytes as unsigned integers.

⟨*Bytes: public*⟩≡
```
    public :: byte_t
```

⟨*Bytes: types*⟩≡
```
    type :: byte_t
       private
       integer(i8) :: i
    end type byte_t
```

⟨*Bytes: public*⟩+≡
```
    public :: byte_zero
```

⟨*Bytes: parameters*⟩≡
```
    type(byte_t), parameter :: byte_zero = byte_t (0_i8)
```

Set a byte from 8-bit integer:

⟨*Bytes: public*⟩+≡
```
    public :: assignment(=)
```

⟨*Bytes: interfaces*⟩≡
```
    interface assignment(=)
       module procedure set_byte_from_i8
    end interface
```

⟨*Bytes: procedures*⟩≡
```
    subroutine set_byte_from_i8 (b, i)
      type(byte_t), intent(out) :: b
      integer(i8), intent(in) :: i
      b%i = i
    end subroutine set_byte_from_i8
```

Write a byte in one of two formats: either as a hexadecimal number (two digits, default) or as a decimal number (one to three digits). The decimal version is nontrivial because bytes are unsigned integers. Optionally append a newline.

⟨*Bytes: public*⟩+≡
```
    public :: byte_write
```

⟨*Bytes: interfaces*⟩+≡
```
interface byte_write
   module procedure byte_write_unit, byte_write_string
end interface
```

⟨*Bytes: procedures*⟩+≡
```
subroutine byte_write_unit (b, unit, decimal, newline)
  type(byte_t), intent(in), optional :: b
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: decimal, newline
  logical :: dc, nl
  type(word32_t) :: w
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  dc = .false.;  if (present (decimal))  dc = decimal
  nl = .false.;  if (present (newline))  nl = newline
  if (dc) then
     w = b
     write (u, '(I3)', advance='no')  w%i
  else
     write (u, '(z2.2)', advance='no')  b%i
  end if
  if (nl) write (u, *)
end subroutine byte_write_unit
```

The string version is hex-only

⟨*Bytes: procedures*⟩+≡
```
subroutine byte_write_string (b, s)
  type(byte_t), intent(in) :: b
  character(len=2), intent(inout) :: s
  write (s, '(z2.2)')  b%i
end subroutine byte_write_string
```

6.1.2 32-bit words

This is not exactly a 32-bit integer. A word is to be filled with bytes, and it may be partially filled. The filling is done lowest-byte first, highest-byte last. We count the bits, so `fill` should be either 0, 8, 16, 24, or 32. In printing words, we correspondingly distinguish between printing zeros and printing blanks.

⟨*Bytes: public*⟩+≡
```
public :: word32_t
```

⟨*Bytes: types*⟩+≡
```
type :: word32_t
   private
   integer(i32) :: i
   integer :: fill = 0
end type word32_t
```

Assignment: the word is filled by inserting a 32-bit integer

⟨*Bytes: interfaces*⟩+≡
```
interface assignment(=)
```

```
   module procedure word32_set_from_i32
   module procedure word32_set_from_byte
end interface
```

⟨*Bytes: procedures*⟩+≡

```
subroutine word32_set_from_i32 (w, i)
  type(word32_t), intent(out) :: w
  integer(i32), intent(in) :: i
  w%i = i
  w%fill = 32
end subroutine word32_set_from_i32
```

Reverse assignment to a 32-bit integer. We do not check the fill status.

⟨*Bytes: interfaces*⟩+≡

```
interface assignment(=)
   module procedure i32_from_word32
end interface
```

⟨*Bytes: procedures*⟩+≡

```
subroutine i32_from_word32 (i, w)
  integer(i32), intent(out) :: i
  type(word32_t), intent(in) :: w
  i = w%i
end subroutine i32_from_word32
```

Filling with a 8-bit integer is slightly tricky, because in this interpretation integers are unsigned.

⟨*Bytes: procedures*⟩+≡

```
subroutine word32_set_from_byte (w, b)
  type(word32_t), intent(out) :: w
  type(byte_t), intent(in) :: b
  if (b%i >= 0_i8) then
     w%i = b%i
  else
     w%i = 2_i32*(huge(0_i8)+1_i32) + b%i
  end if
  w%fill = 32
end subroutine word32_set_from_byte
```

Check the fill status

⟨*Bytes: public*⟩+≡

```
public :: word32_empty, word32_filled, word32_fill
```

⟨*Bytes: procedures*⟩+≡

```
function word32_empty (w)
  type(word32_t), intent(in) :: w
  logical :: word32_empty
  word32_empty = (w%fill == 0)
end function word32_empty

function word32_filled (w)
  type(word32_t), intent(in) :: w
  logical :: word32_filled
  word32_filled = (w%fill == 32)
```

```
end function word32_filled

function word32_fill (w)
  type(word32_t), intent(in) :: w
  integer :: word32_fill
  word32_fill = w%fill
end function word32_fill
```

Partial assignment: append a byte to a partially filled word. (Note: no assignment if the word is filled, so check this before if necessary.)

⟨*Bytes: public*⟩+≡

```
public :: word32_append_byte
```

⟨*Bytes: procedures*⟩+≡

```
subroutine word32_append_byte (w, b)
  type(word32_t), intent(inout) :: w
  type(byte_t), intent(in) :: b
  type(word32_t) :: w1
  if (.not. word32_filled (w)) then
     w1 = b
     call mvbits (w1%i, 0, 8, w%i, w%fill)
     w%fill = w%fill + 8
  end if
end subroutine word32_append_byte
```

Extract a byte from a word. The argument `i` is the position, which may be 0, 1, 2, or 3.

For the final assignment, we set the highest bit separately. Otherwise, we might trigger an overflow condition for a compiler with strict checking turned on.

⟨*Bytes: public*⟩+≡

```
public :: byte_from_word32
```

⟨*Bytes: procedures*⟩+≡

```
function byte_from_word32 (w, i) result (b)
  type(word32_t), intent(in) :: w
  integer, intent(in) :: i
  type(byte_t) :: b
  integer(i32) :: j
  j = 0
  if (i >= 0 .and. i*8 < w%fill) then
     call mvbits (w%i, i*8, 8, j, 0)
  end if
  b%i = int (ibclr (j, 7), kind=i8)
  if (btest (j, 7))  b%i = ibset (b%i, 7)
end function byte_from_word32
```

Write a word to file or STDOUT. We understand words as unsigned integers, therefore we cannot use the built-in routine unchanged. However, we can make use of the existence of 64-bit integers and their output routine.

In hexadecimal format, the default version prints eight hex characters, highest-first. The `bytes` version prints four bytes (two-hex characters), lowest first, with

spaces in-between. The decimal bytes version is analogous. In the `bytes` version, missing bytes are printed as whitespace.

⟨*Bytes: public*⟩+≡
```
public :: word32_write
```

⟨*Bytes: interfaces*⟩+≡
```
interface word32_write
   module procedure word32_write_unit
end interface
```

⟨*Bytes: procedures*⟩+≡
```
subroutine word32_write_unit (w, unit, bytes, decimal, newline)
  type(word32_t), intent(in) :: w
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: bytes, decimal, newline
  logical :: dc, by, nl
  type(word64_t) :: ww
  integer :: i, u
  u = given_output_unit (unit);  if (u < 0)  return
  by = .false.;  if (present (bytes))   by = bytes
  dc = .false.;  if (present (decimal)) dc = decimal
  nl = .false.;  if (present (newline)) nl = newline
  if (by) then
     do i = 0, 3
        if (i>0)  write (u, '(1x)', advance='no')
        if (8*i < w%fill) then
           call byte_write (byte_from_word32 (w, i), unit, decimal=decimal)
        else if (dc) then
           write (u, '(3x)', advance='no')
        else
           write (u, '(2x)', advance='no')
        end if
     end do
  else if (dc) then
     ww = w
     write (u, '(I10)', advance='no') ww%i
  else
     select case (w%fill)
     case ( 0)
     case ( 8);  write (6, '(1x,z8.2)', advance='no') ibits (w%i, 0, 8)
     case (16);  write (6, '(1x,z8.4)', advance='no') ibits (w%i, 0,16)
     case (24);  write (6, '(1x,z8.6)', advance='no') ibits (w%i, 0,24)
     case (32);  write (6, '(1x,z8.8)', advance='no') ibits (w%i, 0,32)
     end select
  end if
  if (nl) write (u, *)
end subroutine word32_write_unit
```

6.1.3 Operations on 32-bit words

Define the usual logical operations, as well as addition (mod 2^{32}). We assume that all operands are completely filled.

⟨*Bytes: public*⟩+≡

```
public :: not, ior, ieor, iand, ishft, ishftc
```

⟨*Bytes: interfaces*⟩+≡

```
interface not
   module procedure word_not
end interface
interface ior
   module procedure word_or
end interface
interface ieor
   module procedure word_eor
end interface
interface iand
   module procedure word_and
end interface
interface ishft
   module procedure word_shft
end interface
interface ishftc
   module procedure word_shftc
end interface
```

⟨*Bytes: procedures*⟩+≡

```
function word_not (w1) result (w2)
  type(word32_t), intent(in) :: w1
  type(word32_t) :: w2
  w2 = not (w1%i)
end function word_not

function word_or (w1, w2) result (w3)
  type(word32_t), intent(in) :: w1, w2
  type(word32_t) :: w3
  w3 = ior (w1%i, w2%i)
end function word_or

function word_eor (w1, w2) result (w3)
  type(word32_t), intent(in) :: w1, w2
  type(word32_t) :: w3
  w3 = ieor (w1%i, w2%i)
end function word_eor

function word_and (w1, w2) result (w3)
  type(word32_t), intent(in) :: w1, w2
  type(word32_t) :: w3
  w3 = iand (w1%i, w2%i)
end function word_and

function word_shft (w1, s) result (w2)
  type(word32_t), intent(in) :: w1
  integer, intent(in) :: s
  type(word32_t) :: w2
  w2 = ishft (w1%i, s)
end function word_shft

function word_shftc (w1, s) result (w2)
```

```
    type(word32_t), intent(in) :: w1
    integer, intent(in) :: s
    type(word32_t) :: w2
    w2 = ishftc (w1%i, s, 32)
  end function word_shftc
```

Addition is defined mod 2^{32}, i.e., without overflow checking. This means that we have to work around a possible overflow check enforced by the compiler.

⟨*Bytes: public*⟩+≡

```
  public :: operator(+)
```

⟨*Bytes: interfaces*⟩+≡

```
  interface operator(+)
     module procedure word_add
     module procedure word_add_i8
     module procedure word_add_i32
  end interface
```

⟨*Bytes: procedures*⟩+≡

```
  function word_add (w1, w2) result (w3)
    type(word32_t), intent(in) :: w1, w2
    type(word32_t) :: w3
    integer(i64) :: j
    j = int (ibclr (w1%i, 31), i64) + int (ibclr (w2%i, 31), i64)
    w3 = int (ibclr (j, 31), kind=i32)
    if (btest (j, 31)) then
       if (btest (w1%i, 31) .eqv. btest (w2%i, 31))  w3 = ibset (w3%i, 31)
    else
       if (btest (w1%i, 31) .neqv. btest (w2%i, 31))  w3 = ibset (w3%i, 31)
    end if
  end function word_add

  function word_add_i8 (w1, i) result (w3)
    type(word32_t), intent(in) :: w1
    integer(i8), intent(in) :: i
    type(word32_t) :: w3
    integer(i64) :: j
    j = int (ibclr (w1%i, 31), i64) + int (ibclr (i, 7), i64)
    if (btest (i, 7))  j = j + 128
    w3 = int (ibclr (j, 31), kind=i32)
    if (btest (j, 31) .neqv. btest (w1%i, 31))  w3 = ibset (w3%i, 31)
  end function word_add_i8

  function word_add_i32 (w1, i) result (w3)
    type(word32_t), intent(in) :: w1
    integer(i32), intent(in) :: i
    type(word32_t) :: w3
    integer(i64) :: j
    j = int (ibclr (w1%i, 31), i64) + int (ibclr (i, 31), i64)
    w3 = int (ibclr (j, 31), kind=i32)
    if (btest (j, 31)) then
       if (btest (w1%i, 31) .eqv. btest (i, 31))  w3 = ibset (w3%i, 31)
    else
       if (btest (w1%i, 31) .neqv. btest (i, 31))  w3 = ibset (w3%i, 31)
```

```
    end if
  end function word_add_i32
```

6.1.4 64-bit words

These objects consist of two 32-bit words. They thus can hold integer numbers larger than 2^{32} (to be exact, 2^{31} since FORTRAN integers are signed). The order is low-word, high-word.

⟨*Bytes: public*⟩+≡

```
  public :: word64_t
```

⟨*Bytes: types*⟩+≡

```
  type :: word64_t
     private
     integer(i64) :: i
  end type word64_t
```

Set a 64 bit word:

⟨*Bytes: interfaces*⟩+≡

```
  interface assignment(=)
     module procedure word64_set_from_i64
     module procedure word64_set_from_word32
  end interface
```

⟨*Bytes: procedures*⟩+≡

```
  subroutine word64_set_from_i64 (ww, i)
    type(word64_t), intent(out) :: ww
    integer(i64), intent(in) :: i
    ww%i = i
  end subroutine word64_set_from_i64
```

Filling with a 32-bit word:

⟨*Bytes: procedures*⟩+≡

```
  subroutine word64_set_from_word32 (ww, w)
    type(word64_t), intent(out) :: ww
    type(word32_t), intent(in) :: w
    if (w%i >= 0_i32) then
       ww%i = w%i
    else
       ww%i = 2_i64*(huge(0_i32)+1_i64) + w%i
    end if
  end subroutine word64_set_from_word32
```

Extract a byte from a word. The argument `i` is the position, which may be between 0 and 7.

For the final assignment, we set the highest bit separately. Otherwise, we might trigger an overflow condition for a compiler with strict checking turned on.

⟨*Bytes: public*⟩+≡

```
  public :: byte_from_word64, word32_from_word64
```

⟨*Bytes: procedures*⟩+≡

```
function byte_from_word64 (ww, i) result (b)
  type(word64_t), intent(in) :: ww
  integer, intent(in) :: i
  type(byte_t) :: b
  integer(i64) :: j
  j = 0
  if (i >= 0 .and. i*8 < 64) then
     call mvbits (ww%i, i*8, 8, j, 0)
  end if
  b%i = int (ibclr (j, 7), kind=i8)
  if (btest (j, 7))  b%i = ibset (b%i, 7)
end function byte_from_word64
```

Extract a 32-bit word from a 64-bit word. The position is either 0 or 1.

⟨*Bytes: procedures*⟩+≡

```
function word32_from_word64 (ww, i) result (w)
  type(word64_t), intent(in) :: ww
  integer, intent(in) :: i
  type(word32_t) :: w
  integer(i64) :: j
  j = 0
  select case (i)
  case (0);  call mvbits (ww%i,  0, 32, j, 0)
  case (1);  call mvbits (ww%i, 32, 32, j, 0)
  end select
  w = int (ibclr (j, 31), kind=i32)
  if (btest (j, 31))  w = ibset (w%i, 31)
end function word32_from_word64
```

Print a 64-bit word. Decimal version works up to 2^{63}. The `words` version uses the 'word32' printout, separated by two spaces. The low-word is printed first. The `bytes` version also uses the 'word32' printout. This implies that the lowest byte is first. The default version prints a hexadecimal number without spaces, highest byte first.

⟨*Bytes: public*⟩+≡

```
public :: word64_write
```

⟨*Bytes: interfaces*⟩+≡

```
interface word64_write
   module procedure word64_write_unit
end interface
```

⟨*Bytes: procedures*⟩+≡

```
subroutine word64_write_unit (ww, unit, words, bytes, decimal, newline)
  type(word64_t), intent(in) :: ww
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: words, bytes, decimal, newline
  logical :: wo, by, dc, nl
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  wo = .false.;  if (present (words))    wo = words
  by = .false.;  if (present (bytes))    by = bytes
  dc = .false.;  if (present (decimal))  dc = decimal
```

```
    nl = .false.;  if (present (newline))  nl = newline
    if (wo .or. by) then
       call word32_write_unit (word32_from_word64 (ww, 0), unit, by, dc)
       write (u, '(2x)', advance='no')
       call word32_write_unit (word32_from_word64 (ww, 1), unit, by, dc)
    else if (dc) then
       write (u, '(I19)', advance='no') ww%i
    else
       write (u, '(Z16)', advance='no') ww%i
    end if
    if (nl) write (u, *)
  end subroutine word64_write_unit
```

6.2 Hashtables

Hash tables, like lists, are not part of Fortran and must be defined on a per-case basis. In this section we define a module that contains a hash function.

Furthermore, for reference there is a complete framework of hashtable type definitions and access functions. This code is to be replicated where hash tables are used, mutatis mutandis.

⟨`hashes.f90`⟩≡

```
⟨File header⟩

module hashes

  use kinds, only: i8, i32
  use bytes

⟨Standard module head⟩

⟨Hashes: public⟩

contains

⟨Hashes: procedures⟩

end module hashes
```

6.2.1 The hash function

This is the one-at-a-time hash function by Bob Jenkins (from Wikipedia), re-implemented in Fortran. The function works on an array of bytes (8-bit integers), as could be produced by, e.g., the `transfer` function, and returns a single 32-bit integer. For determining the position in a hashtable, one can pick the lower bits of the result as appropriate to the hashtable size (which should be a power of 2). Note that we are working on signed integers, so the interpretation of values differs from the C version. This should not matter in practice, however.

⟨*Hashes: public*⟩≡

```
  public :: hash
```

⟨*Hashes: procedures*⟩≡

```
function hash (key) result (hashval)
  integer(i32) :: hashval
  integer(i8), dimension(:), intent(in) :: key
  type(word32_t) :: w
  integer :: i
  w = 0_i32
  do i = 1, size (key)
     w = w + key(i)
     w = w + ishft (w, 10)
     w = ieor (w, ishft (w, -6))
  end do
  w = w + ishft (w, 3)
  w = ieor (w, ishft (w, -11))
  w = w + ishft (w, 15)
  hashval = w
end function hash
```

6.2.2 The hash table

We define a generic hashtable type (that depends on the `hash_data_t` type) together with associated methods.

This is a template:

⟨*Hashtables: types*⟩≡

```
type :: hash_data_t
   integer :: i
end type hash_data_t
```

Associated methods:

⟨*Hashtables: procedures*⟩≡

```
subroutine hash_data_final (data)
  type(hash_data_t), intent(inout) :: data
end subroutine hash_data_final

subroutine hash_data_write (data, unit)
  type(hash_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, *) data%i
end subroutine hash_data_write
```

Each hash entry stores the unmasked hash value, the key, and points to actual data if present. Note that this could be an allocatable scalar in principle, but making it a pointer avoids deep copy when expanding the hashtable.

⟨*Hashtables: types*⟩+≡

```
type :: hash_entry_t
   integer(i32) :: hashval = 0
   integer(i8), dimension(:), allocatable :: key
   type(hash_data_t), pointer :: data => null ()
end type hash_entry_t
```

The hashtable object holds the actual table, the number of filled entries and the number of entries after which the size should be doubled. The mask is equal to the table size minus one and thus coincides with the upper bound of the table index, which starts at zero.

⟨*Hashtables: types*⟩+≡

```
type :: hashtable_t
   integer :: n_entries = 0
   real :: fill_ratio = 0
   integer :: n_entries_max = 0
   integer(i32) :: mask = 0
   type(hash_entry_t), dimension(:), allocatable :: entry
end type hashtable_t
```

Initializer: The size has to be a power of two, the fill ratio is a real (machine-default!) number between 0 and 1.

⟨*Hashtables: procedures*⟩+≡

```
subroutine hashtable_init (hashtable, size, fill_ratio)
  type(hashtable_t), intent(out) :: hashtable
  integer, intent(in) :: size
  real, intent(in) :: fill_ratio
  hashtable%fill_ratio = fill_ratio
  hashtable%n_entries_max = size * fill_ratio
  hashtable%mask = size - 1
  allocate (hashtable%entry (0:hashtable%mask))
end subroutine hashtable_init
```

Finalizer: This calls a `hash_data_final` subroutine which must exist.

⟨*Hashtables: procedures*⟩+≡

```
subroutine hashtable_final (hashtable)
  type(hashtable_t), intent(inout) :: hashtable
  integer :: i
  do i = 0, hashtable%mask
     if (associated (hashtable%entry(i)%data)) then
        call hash_data_final (hashtable%entry(i)%data)
        deallocate (hashtable%entry(i)%data)
     end if
  end do
  deallocate (hashtable%entry)
end subroutine hashtable_final
```

Output. Here, we refer to a `hash_data_write` subroutine.

⟨*Hashtables: procedures*⟩+≡

```
subroutine hashtable_write (hashtable, unit)
  type(hashtable_t), intent(in) :: hashtable
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  do i = 0, hashtable%mask
     if (associated (hashtable%entry(i)%data)) then
        write (u, *) i, "(hash =", hashtable%entry(i)%hashval, ")", &
```

```
            hashtable%entry(i)%key
         call hash_data_write (hashtable%entry(i)%data, unit)
      end if
   end do
end subroutine hashtable_write
```

6.2.3 Hashtable insertion

Insert a single entry with the hash value as trial place. If the table is filled, first expand it.

⟨*Hashtables: procedures*⟩+≡

```
subroutine hashtable_insert (hashtable, key, data)
  type(hashtable_t), intent(inout) :: hashtable
  integer(i8), dimension(:), intent(in) :: key
  type(hash_data_t), intent(in), target :: data
  integer(i32) :: h
  if (hashtable%n_entries >= hashtable%n_entries_max) &
       call hashtable_expand (hashtable)
  h = hash (key)
  call hashtable_insert_rec (hashtable, h, h, key, data)
end subroutine hashtable_insert
```

We need this auxiliary routine for doubling the size of the hashtable. We rely on the fact that default assignment copies the data pointer, not the data themselves. The temporary array must not be finalized; it is deallocated automatically together with its allocatable components.

⟨*Hashtables: procedures*⟩+≡

```
subroutine hashtable_expand (hashtable)
  type(hashtable_t), intent(inout) :: hashtable
  type(hash_entry_t), dimension(:), allocatable :: table_tmp
  integer :: i, s
  allocate (table_tmp (0:hashtable%mask))
  table_tmp = hashtable%entry
  deallocate (hashtable%entry)
  s = 2 * size (table_tmp)
  hashtable%n_entries = 0
  hashtable%n_entries_max = s * hashtable%fill_ratio
  hashtable%mask = s - 1
  allocate (hashtable%entry (0:hashtable%mask))
  do i = 0, ubound (table_tmp, 1)
     if (associated (table_tmp(i)%data)) then
        call hashtable_insert_rec (hashtable, table_tmp(i)%hashval, &
             table_tmp(i)%hashval, table_tmp(i)%key, table_tmp(i)%data)
     end if
  end do
end subroutine hashtable_expand
```

Insert a single entry at a trial place `h`, reduced to the table size. Collision resolution is done simply by choosing the next element, recursively until the place is empty. For bookkeeping, we preserve the original hash value. For a good hash function, there should be no clustering.

Note that if the new key exactly matches an existing key, nothing is done.

⟨*Hashtables: procedures*⟩+≡

```
recursive subroutine hashtable_insert_rec (hashtable, h, hashval, key, data)
  type(hashtable_t), intent(inout) :: hashtable
  integer(i32), intent(in) :: h, hashval
  integer(i8), dimension(:), intent(in) :: key
  type(hash_data_t), intent(in), target :: data
  integer(i32) :: i
  i = iand (h, hashtable%mask)
  if (associated (hashtable%entry(i)%data)) then
     if (size (hashtable%entry(i)%key) /= size (key)) then
        call hashtable_insert_rec (hashtable, h + 1, hashval, key, data)
     else if (any (hashtable%entry(i)%key /= key)) then
        call hashtable_insert_rec (hashtable, h + 1, hashval, key, data)
     end if
  else
     hashtable%entry(i)%hashval = hashval
     allocate (hashtable%entry(i)%key (size (key)))
     hashtable%entry(i)%key = key
     hashtable%entry(i)%data => data
     hashtable%n_entries = hashtable%n_entries + 1
  end if
end subroutine hashtable_insert_rec
```

6.2.4 Hashtable lookup

The lookup function has to parallel the insert function. If the place is filled, check if the key matches. Yes: return the pointer; no: increment the hash value and check again.

⟨*Hashtables: procedures*⟩+≡

```
function hashtable_lookup (hashtable, key) result (ptr)
  type(hash_data_t), pointer :: ptr
  type(hashtable_t), intent(in) :: hashtable
  integer(i8), dimension(:), intent(in) :: key
  ptr => hashtable_lookup_rec (hashtable, hash (key), key)
end function hashtable_lookup
```

⟨*Hashtables: procedures*⟩+≡

```
recursive function hashtable_lookup_rec (hashtable, h, key) result (ptr)
  type(hash_data_t), pointer :: ptr
  type(hashtable_t), intent(in) :: hashtable
  integer(i32), intent(in) :: h
  integer(i8), dimension(:), intent(in) :: key
  integer(i32) :: i
  i = iand (h, hashtable%mask)
  if (associated (hashtable%entry(i)%data)) then
     if (size (hashtable%entry(i)%key) == size (key)) then
        if (all (hashtable%entry(i)%key == key)) then
           ptr => hashtable%entry(i)%data
        else
           ptr => hashtable_lookup_rec (hashtable, h + 1, key)
        end if
```

```
      else
         ptr => hashtable_lookup_rec (hashtable, h + 1, key)
      end if
   else
      ptr => null ()
   end if
end function hashtable_lookup_rec
```

⟨*Hashtables: public*⟩≡

```
public :: hashtable_test
```

⟨*Hashtables: procedures*⟩+≡

```
subroutine hashtable_test ()
  type(hash_data_t), pointer :: data
  type(hashtable_t) :: hashtable
  integer(i8) :: i
  call hashtable_init (hashtable, 16, 0.25)
  do i = 1, 10
     allocate (data)
     data%i = i*i
     call hashtable_insert (hashtable, [i, i+i], data)
  end do
  call hashtable_insert (hashtable, [2_i8, 4_i8], data)
  call hashtable_write (hashtable)
  data => hashtable_lookup (hashtable, [5_i8, 10_i8])
  if (associated (data)) then
     print *, "lookup:", data%i
  else
     print *, "lookup: --"
  end if
  data => hashtable_lookup (hashtable, [6_i8, 12_i8])
  if (associated (data)) then
     print *, "lookup:", data%i
  else
     print *, "lookup: --"
  end if
  data => hashtable_lookup (hashtable, [4_i8, 9_i8])
  if (associated (data)) then
     print *, "lookup:", data%i
  else
     print *, "lookup: --"
  end if
  call hashtable_final (hashtable)
end subroutine hashtable_test
```

6.3 MD5 Checksums

Implementing MD5 checksums allows us to check input/file integrity on the basis of a well-known standard. The building blocks have been introduced in the `bytes` module.

⟨`md5.f90`⟩≡

⟨*File header*⟩

```
module md5

  use kinds, only: i8, i32, i64
  use io_units
  use system_defs, only: BUFFER_SIZE
  use system_defs, only: LF, EOR, EOF
  use diagnostics
  use bytes
```

⟨*Standard module head*⟩

⟨*MD5: public*⟩

⟨*MD5: types*⟩

⟨*MD5: variables*⟩

⟨*MD5: interfaces*⟩

```
contains
```

⟨*MD5: procedures*⟩

```
end module md5
```

6.3.1 Blocks

A block is a sequence of 16 words (64 bytes or 512 bits). We anticipate that blocks will be linked, so include a pointer to the next block. There is a fill status (word counter), as there is one for each word. The fill status is equal to the number of bytes that are in, so it may be between 0 and 64.

⟨*MD5: types*⟩≡

```
type :: block_t
   private
   type(word32_t), dimension(0:15) :: w
   type(block_t), pointer :: next => null ()
   integer :: fill = 0
end type block_t
```

Check if a block is completely filled or empty:

⟨*MD5: procedures*⟩≡

```
function block_is_empty (b)
  type(block_t), intent(in) :: b
  logical :: block_is_empty
  block_is_empty = (b%fill == 0 .and. word32_empty (b%w(0)))
end function block_is_empty

function block_is_filled (b)
  type(block_t), intent(in) :: b
  logical :: block_is_filled
  block_is_filled = (b%fill == 64)
```

```
end function block_is_filled
```

Append a single byte to a block. Works only if the block is not yet filled.

⟨*MD5: procedures*⟩+≡

```
subroutine block_append_byte (bl, by)
  type(block_t), intent(inout) :: bl
  type(byte_t), intent(in) :: by
  if (.not. block_is_filled (bl)) then
     call word32_append_byte (bl%w(bl%fill/4), by)
     bl%fill = bl%fill + 1
  end if
end subroutine block_append_byte
```

The printing routine allows for printing as sequences of words or bytes, decimal or hex.

⟨*MD5: interfaces*⟩≡

```
interface block_write
   module procedure block_write_unit
end interface
```

⟨*MD5: procedures*⟩+≡

```
subroutine block_write_unit (b, unit, bytes, decimal)
  type(block_t), intent(in) :: b
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: bytes, decimal
  logical :: by, dc
  integer :: i, u
  u = given_output_unit (unit);  if (u < 0)  return
  by = .false.;  if (present (bytes))    by = bytes
  dc = .false.;  if (present (decimal))  dc = decimal
  do i = 0, b%fill/4 - 1
     call newline_or_blank (u, i, by, dc)
     call word32_write (b%w(i), unit, bytes, decimal)
  end do
  if (.not. block_is_filled (b)) then
     i = b%fill/4
     if (.not. word32_empty (b%w(i))) then
        call newline_or_blank (u, i, by, dc)
        call word32_write (b%w(i), unit, bytes, decimal)
     end if
  end if
  write (u, *)
contains
  subroutine newline_or_blank (u, i, bytes, decimal)
    integer, intent(in) :: u, i
    logical, intent(in) :: bytes, decimal
    if (decimal) then
       select case (i)
       case (0)
       case (2,4,6,8,10,12,14);  write (u, *)
       case default
          write (u, '(2x)', advance='no')
       end select
```

```
      else if (bytes) then
         select case (i)
         case (0)
         case (4,8,12);  write (u, *)
         case default
            write (u, '(2x)', advance='no')
         end select
      else
         if (i == 8)  write (u, *)
      end if
    end subroutine newline_or_blank
  end subroutine block_write_unit
```

6.3.2 Messages

A message (within this module) is a linked list of blocks.

⟨*MD5: types*⟩+≡

```
  type :: message_t
     private
     type(block_t), pointer :: first => null ()
     type(block_t), pointer :: last => null ()
     integer :: n_blocks = 0
  end type message_t
```

Clear the message list

⟨*MD5: procedures*⟩+≡

```
  subroutine message_clear (m)
    type(message_t), intent(inout) :: m
    type(block_t), pointer :: b
    nullify (m%last)
    do
       b => m%first
       if (.not.(associated (b))) exit
       m%first => b%next
       deallocate (b)
    end do
    m%n_blocks = 0
  end subroutine message_clear
```

Append an empty block to the message list

⟨*MD5: procedures*⟩+≡

```
  subroutine message_append_new_block (m)
    type(message_t), intent(inout) :: m
    if (associated (m%last)) then
       allocate (m%last%next)
       m%last => m%last%next
       m%n_blocks = m%n_blocks + 1
    else
       allocate (m%first)
       m%last => m%first
       m%n_blocks = 1
```

```
    end if
  end subroutine message_append_new_block
```

Initialize: clear and allocate the first (empty) block.

⟨*MD5: procedures*⟩+≡

```
  subroutine message_init (m)
    type(message_t), intent(inout) :: m
    call message_clear (m)
    call message_append_new_block (m)
  end subroutine message_init
```

Append a single byte to a message. If necessary, allocate a new block. If the message is empty, initialize it.

⟨*MD5: procedures*⟩+≡

```
  subroutine message_append_byte (m, b)
    type(message_t), intent(inout) :: m
    type(byte_t), intent(in) :: b
    if (.not. associated (m%last)) then
       call message_init (m)
    else if (block_is_filled (m%last)) then
       call message_append_new_block (m)
    end if
    call block_append_byte (m%last, b)
  end subroutine message_append_byte
```

Append zero bytes until the current block is filled up to the required position. If we are already beyond that, append a new block and fill that one.

⟨*MD5: procedures*⟩+≡

```
  subroutine message_pad_zero (m, i)
    type(message_t), intent(inout) :: m
    integer, intent(in) :: i
    type(block_t), pointer :: b
    integer :: j
    if (associated (m%last)) then
       b => m%last
       if (b%fill > i) then
          do j = b%fill + 1, 64 + i
             call message_append_byte (m, byte_zero)
          end do
       else
          do j = b%fill + 1, i
             call message_append_byte (m, byte_zero)
          end do
       end if
    end if
  end subroutine message_pad_zero
```

This returns the number of bits within a message. We need a 64-bit word for the result since it may be more than 2^{31}. This is also required by the MD5 standard.

⟨*MD5: procedures*⟩+≡

```
function message_bits (m) result (length)
  type(message_t), intent(in) :: m
  type(word64_t) :: length
  type(block_t), pointer :: b
  integer(i64) :: n_blocks_filled, n_bytes_extra
  if (m%n_blocks > 0) then
     b => m%last
     if (block_is_filled (b)) then
        n_blocks_filled = m%n_blocks
        n_bytes_extra = 0
     else
        n_blocks_filled = m%n_blocks - 1
        n_bytes_extra = b%fill
     end if
     length = n_blocks_filled * 512 + n_bytes_extra * 8
  else
     length = 0_i64
  end if
end function message_bits
```

6.3.3 Message I/O

Append the contents of a string to a message. We first cast the character string into a 8-bit integer array and the append this byte by byte.

⟨*MD5: procedures*⟩+≡

```
subroutine message_append_string (m, s)
  type(message_t), intent(inout) :: m
  character(len=*), intent(in) :: s
  integer(i64) :: i, n_bytes
  integer(i8), dimension(:), allocatable :: buffer
  integer(i8), dimension(1) :: mold
  type(byte_t) :: b
  n_bytes = size (transfer (s, mold))
  allocate (buffer (n_bytes))
  buffer = transfer (s, mold)
  do i = 1, size (buffer)
     b = buffer(i)
     call message_append_byte (m, b)
  end do
  deallocate (buffer)
end subroutine message_append_string
```

Append the contents of a 32-bit integer to a message. We first cast the 32-bit integer into a 8-bit integer array and the append this byte by byte.

⟨*MD5: procedures*⟩+≡

```
subroutine message_append_i32 (m, x)
  type(message_t), intent(inout) :: m
  integer(i32), intent(in) :: x
  integer(i8), dimension(4) :: buffer
  type(byte_t) :: b
  integer :: i
  buffer = transfer (x, buffer, size(buffer))
```

```
  do i = 1, size (buffer)
     b = buffer(i)
     call message_append_byte (m, b)
  end do
end subroutine message_append_i32
```

Append one line from file to a message. Include the newline character.

⟨*MD5: procedures*⟩+≡

```
subroutine message_append_from_unit (m, u, iostat)
  type(message_t), intent(inout) :: m
  integer, intent(in) :: u
  integer, intent(out) :: iostat
  character(len=BUFFER_SIZE) :: buffer
  read (u, *, iostat=iostat) buffer
  call message_append_string (m, trim (buffer))
  call message_append_string (m, LF)
end subroutine message_append_from_unit
```

Fill a message from file. (Each line counts as a string.)

⟨*MD5: procedures*⟩+≡

```
subroutine message_read_from_file (m, f)
  type(message_t), intent(inout) :: m
  character(len=*), intent(in) :: f
  integer :: u, iostat
  u = free_unit ()
  open (file=f, unit=u, action='read')
  do
     call message_append_from_unit (m, u, iostat=iostat)
     if (iostat < 0) exit
  end do
  close (u)
end subroutine message_read_from_file
```

Write a message. After each block, insert an empty line.

⟨*MD5: interfaces*⟩+≡

```
interface message_write
   module procedure message_write_unit
end interface
```

⟨*MD5: procedures*⟩+≡

```
subroutine message_write_unit (m, unit, bytes, decimal)
  type(message_t), intent(in) :: m
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: bytes, decimal
  type(block_t), pointer :: b
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  b => m%first
  if (associated (b)) then
     do
        call block_write_unit (b, unit, bytes, decimal)
        b => b%next
        if (.not. associated (b))  exit
```

```
        write (u, *)
      end do
    end if
  end subroutine message_write_unit
```

6.3.4 Auxiliary functions

These four functions on three words are defined in the MD5 standard:

⟨*MD5: procedures*⟩+≡

```
  function ff (x, y, z)
    type(word32_t), intent(in) :: x, y, z
    type(word32_t) :: ff
    ff = ior (iand (x, y), iand (not (x), z))
  end function ff

  function fg (x, y, z)
    type(word32_t), intent(in) :: x, y, z
    type(word32_t) :: fg
    fg = ior (iand (x, z), iand (y, not (z)))
  end function fg

  function fh (x, y, z)
    type(word32_t), intent(in) :: x, y, z
    type(word32_t) :: fh
    fh = ieor (ieor (x, y), z)
  end function fh

  function fi (x, y, z)
    type(word32_t), intent(in) :: x, y, z
    type(word32_t) :: fi
    fi = ieor (y, ior (x, not (z)))
  end function fi
```

6.3.5 Auxiliary stuff

This defines and initializes the table of transformation constants:

⟨*MD5: variables*⟩≡

```
  type(word32_t), dimension(64), save :: t
  logical, save :: table_initialized = .false.
```

⟨*MD5: procedures*⟩+≡

```
  subroutine table_init
    type(word64_t) :: ww
    integer :: i
    if (.not.table_initialized) then
       do i = 1, 64
          ww = int (4294967296d0 * abs (sin (i * 1d0)), kind=i64)
          t(i) = word32_from_word64 (ww, 0)
       end do
       table_initialized = .true.
    end if
```

```
end subroutine table_init
```

This encodes the message digest (4 words) into a 32-character string.

⟨*MD5: procedures*⟩+≡

```
function digest_string (aa) result (s)
  type(word32_t), dimension (0:3), intent(in) :: aa
  character(len=32) :: s
  integer :: i, j
  do i = 0, 3
     do j = 0, 3
        call byte_write (byte_from_word32 (aa(i), j), s(i*8+j*2+1:i*8+j*2+2))
     end do
  end do
end function digest_string
```

6.3.6 MD5 algorithm

Pad the message with a byte `x80` and then pad zeros up to a full block minus two words; in these words, insert the message length (before padding) as a 64-bit word, low-word first.

⟨*MD5: procedures*⟩+≡

```
subroutine message_pad (m)
  type(message_t), intent(inout) :: m
  type(word64_t) :: length
  integer(i8), parameter :: ipad = -128  ! z'80'
  type(byte_t) :: b
  integer :: i
  length = message_bits (m)
  b = ipad
  call message_append_byte (m, b)
  call message_pad_zero (m, 56)
  do i = 0, 7
     call message_append_byte (m, byte_from_word64 (length, i))
  end do
end subroutine message_pad
```

Apply a series of transformations onto a state `a,b,c,d`, where the transform function uses each word of the message together with the predefined words. Finally, encode the state as a 32-character string.

⟨*MD5: procedures*⟩+≡

```
subroutine message_digest (m, s)
  type(message_t), intent(in) :: m
  character(len=32), intent(out) :: s
  integer(i32), parameter :: ia =  1732584193  ! z'67452301'
  integer(i32), parameter :: ib =  -271733879  ! z'efcdab89'
  integer(i32), parameter :: ic = -1732584194  ! z'98badcfe'
  integer(i32), parameter :: id =   271733878  ! z'10325476'
  type(word32_t) :: a, b, c, d
  type(word32_t) :: aa, bb, cc, dd
  type(word32_t), dimension(0:15) :: x
  type(block_t), pointer :: bl
```

```
call table_init
a = ia;  b = ib;  c = ic;  d = id
bl => m%first
do
   if (.not.associated (bl)) exit
   x = bl%w
   aa = a;  bb = b;  cc = c;  dd = d
   call transform (ff, a, b, c, d,  0,  7,  1)
   call transform (ff, d, a, b, c,  1, 12,  2)
   call transform (ff, c, d, a, b,  2, 17,  3)
   call transform (ff, b, c, d, a,  3, 22,  4)
   call transform (ff, a, b, c, d,  4,  7,  5)
   call transform (ff, d, a, b, c,  5, 12,  6)
   call transform (ff, c, d, a, b,  6, 17,  7)
   call transform (ff, b, c, d, a,  7, 22,  8)
   call transform (ff, a, b, c, d,  8,  7,  9)
   call transform (ff, d, a, b, c,  9, 12, 10)
   call transform (ff, c, d, a, b, 10, 17, 11)
   call transform (ff, b, c, d, a, 11, 22, 12)
   call transform (ff, a, b, c, d, 12,  7, 13)
   call transform (ff, d, a, b, c, 13, 12, 14)
   call transform (ff, c, d, a, b, 14, 17, 15)
   call transform (ff, b, c, d, a, 15, 22, 16)
   call transform (fg, a, b, c, d,  1,  5, 17)
   call transform (fg, d, a, b, c,  6,  9, 18)
   call transform (fg, c, d, a, b, 11, 14, 19)
   call transform (fg, b, c, d, a,  0, 20, 20)
   call transform (fg, a, b, c, d,  5,  5, 21)
   call transform (fg, d, a, b, c, 10,  9, 22)
   call transform (fg, c, d, a, b, 15, 14, 23)
   call transform (fg, b, c, d, a,  4, 20, 24)
   call transform (fg, a, b, c, d,  9,  5, 25)
   call transform (fg, d, a, b, c, 14,  9, 26)
   call transform (fg, c, d, a, b,  3, 14, 27)
   call transform (fg, b, c, d, a,  8, 20, 28)
   call transform (fg, a, b, c, d, 13,  5, 29)
   call transform (fg, d, a, b, c,  2,  9, 30)
   call transform (fg, c, d, a, b,  7, 14, 31)
   call transform (fg, b, c, d, a, 12, 20, 32)
   call transform (fh, a, b, c, d,  5,  4, 33)
   call transform (fh, d, a, b, c,  8, 11, 34)
   call transform (fh, c, d, a, b, 11, 16, 35)
   call transform (fh, b, c, d, a, 14, 23, 36)
   call transform (fh, a, b, c, d,  1,  4, 37)
   call transform (fh, d, a, b, c,  4, 11, 38)
   call transform (fh, c, d, a, b,  7, 16, 39)
   call transform (fh, b, c, d, a, 10, 23, 40)
   call transform (fh, a, b, c, d, 13,  4, 41)
   call transform (fh, d, a, b, c,  0, 11, 42)
   call transform (fh, c, d, a, b,  3, 16, 43)
   call transform (fh, b, c, d, a,  6, 23, 44)
   call transform (fh, a, b, c, d,  9,  4, 45)
   call transform (fh, d, a, b, c, 12, 11, 46)
   call transform (fh, c, d, a, b, 15, 16, 47)
```

```
      call transform (fh, b, c, d, a,  2, 23, 48)
      call transform (fi, a, b, c, d,  0,  6, 49)
      call transform (fi, d, a, b, c,  7, 10, 50)
      call transform (fi, c, d, a, b, 14, 15, 51)
      call transform (fi, b, c, d, a,  5, 21, 52)
      call transform (fi, a, b, c, d, 12,  6, 53)
      call transform (fi, d, a, b, c,  3, 10, 54)
      call transform (fi, c, d, a, b, 10, 15, 55)
      call transform (fi, b, c, d, a,  1, 21, 56)
      call transform (fi, a, b, c, d,  8,  6, 57)
      call transform (fi, d, a, b, c, 15, 10, 58)
      call transform (fi, c, d, a, b,  6, 15, 59)
      call transform (fi, b, c, d, a, 13, 21, 60)
      call transform (fi, a, b, c, d,  4,  6, 61)
      call transform (fi, d, a, b, c, 11, 10, 62)
      call transform (fi, c, d, a, b,  2, 15, 63)
      call transform (fi, b, c, d, a,  9, 21, 64)
      a = a + aa
      b = b + bb
      c = c + cc
      d = d + dd
      bl => bl%next
    end do
    s = digest_string ([a, b, c, d])
  contains
  ⟨MD5: Internal subroutine transform⟩
  end subroutine message_digest
```

And this is the actual transformation that depends on one of the previous functions, four words, and three integers. The implicit arguments are `x`, the word from the message to digest, and `t`, the entry in the predefined table.

⟨*MD5: Internal subroutine transform*⟩≡

```
  subroutine transform (f, a, b, c, d, k, s, i)
    interface
      function f (x, y, z)
        import word32_t
        type(word32_t), intent(in) :: x, y, z
        type(word32_t) :: f
      end function f
    end interface
    type(word32_t), intent(inout) :: a
    type(word32_t), intent(in) :: b, c, d
    integer, intent(in) :: k, s, i
    a = b + ishftc (a + f(b, c, d) + x(k) + t(i), s)
  end subroutine transform
```

6.3.7 User interface

⟨*MD5: public*⟩≡

```
  public :: md5sum
```

⟨*MD5: interfaces*⟩+≡

```
  interface md5sum
```

```
      module procedure md5sum_from_string
      module procedure md5sum_from_unit
   end interface
```

This function computes the MD5 sum of the input string and returns it as a 32-character string

⟨*MD5: procedures*⟩+≡

```
   function md5sum_from_string (s) result (digest)
     character(len=*), intent(in) :: s
     character(len=32) :: digest
     type(message_t) :: m
     call message_append_string (m, s)
     call message_pad (m)
     call message_digest (m, digest)
     call message_clear (m)
   end function md5sum_from_string
```

This funct. reads from unit u (an unformmated sequence of integers) and computes the MD5 sum.

⟨*MD5: procedures*⟩+≡

```
   function md5sum_from_unit (u) result (digest)
     integer, intent(in) :: u
     character(len=32) :: digest
     type(message_t) :: m
     character :: char
     integer :: iostat
     READ_CHARS: do
        read (u, "(A)", advance="no", iostat=iostat)  char
        select case (iostat)
        case (0)
           call message_append_string (m, char)
        case (EOR)
           call message_append_string (m, LF)
        case (EOF)
           exit READ_CHARS
        case default
           call msg_fatal &
                 ("Computing MD5 sum: I/O error while reading from scratch file")
        end select
     end do READ_CHARS
     call message_pad (m)
     call message_digest (m, digest)
     call message_clear (m)
   end function md5sum_from_unit
```

6.3.8 Unit tests

Test module, followed by the corresponding implementation module.

⟨`md5_ut.f90`⟩≡

```
  ⟨File header⟩

  module md5_ut
```

```
  use unit_tests
  use md5_uti

⟨Standard module head⟩

⟨MD5: public test⟩

contains

⟨MD5: test driver⟩

end module md5_ut
```

⟨md5_uti.f90⟩≡

```
⟨File header⟩

module md5_uti

  use diagnostics

  use md5

⟨Standard module head⟩

⟨MD5: test declarations⟩

contains

⟨MD5: tests⟩

end module md5_uti
```

API: driver for the unit tests below.

⟨*MD5: public test*⟩≡

```
  public :: md5_test
```

⟨*MD5: test driver*⟩≡

```
  subroutine md5_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨MD5: execute tests⟩
  end subroutine md5_test
```

This function checks the implementation by computing the checksum of certain strings and comparing them with the known values.

⟨*MD5: execute tests*⟩≡

```
  call test (md5_1, "md5_1", &
       "check MD5 sums", &
       u, results)
```

⟨*MD5: test declarations*⟩≡

```
  public :: md5_1
```

⟨*MD5: tests*⟩≡

```
  subroutine md5_1 (u)
    integer, intent(in) :: u
```

```
    character(32) :: s
    integer, parameter :: n = 7
    integer :: i
    character(80), dimension(n) :: teststring
    data teststring(1) /""/
    data teststring(2) /"a"/
    data teststring(3) /"abc"/
    data teststring(4) /"message digest"/
    data teststring(5) /"abcdefghijklmnopqrstuvwxyz"/
    data teststring(6) /"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"/
    data teststring(7) /"12345678901234567890123456789012345678901234567890123456789012345678901234567890123
    character(32), dimension(n) :: result
    data result(1) /"D41D8CD98F00B204E9800998ECF8427E"/
    data result(2) /"0CC175B9C0F1B6A831C399E269772661"/
    data result(3) /"900150983CD24FB0D6963F7D28E17F72"/
    data result(4) /"F96B697D7CB7938D525A2F31AAF161D0"/
    data result(5) /"C3FCD3D76192E4007DFB496CCA67E13B"/
    data result(6) /"D174AB98D277D9F5A5611C2C9F419D9F"/
    data result(7) /"57EDF4A22BE3C955AC49DA2E2107B67A"/

    write (u, "(A)")  "* Test output: MD5"
    write (u, "(A)")  "*   Purpose: test MD5 sums"
    write (u, "(A)")

    do i = 1, n
       write (u, "(A)") "MD5 test string = " // '"'// &
            trim (teststring(i)) // '"'
       s = md5sum (trim (teststring(i)))
       write (u, "(A)") "MD5 check sum   = " // trim (s)
       write (u, "(A)") "Ref check sum   = " // result(i)
       if (s == result(i)) then
          call msg_message ("=> ok", u)
       else
          call msg_message ("=> MD5 sum self-test failed", u)
       end if
    end do
    call msg_message ("==========================================================================
  end subroutine md5_1
```

6.4 Permutations

Permute arrays of integers (of specific kind).

⟨permutations.f90⟩≡

⟨*File header*⟩

```
module permutations

  use kinds, only: TC
```

⟨*Standard module head*⟩

⟨*Permutations: public*⟩

⟨*Permutations: types*⟩

⟨*Permutations: interfaces*⟩

```
contains
```

⟨*Permutations: procedures*⟩

```
end module permutations
```

6.4.1 Permutations

A permutation is an array of integers. Each integer between one and `size` should occur exactly once.

⟨*Permutations: public*⟩≡
```
public :: permutation_t
```

⟨*Permutations: types*⟩≡
```
type :: permutation_t
   private
   integer, dimension(:), allocatable :: p
end type permutation_t
```

Initialize with the identity permutation.

⟨*Permutations: public*⟩+≡
```
public :: permutation_init
public :: permutation_final
```

⟨*Permutations: procedures*⟩≡
```
elemental subroutine permutation_init (p, size)
  type(permutation_t), intent(inout) :: p
  integer, intent(in) :: size
  integer :: i
  allocate (p%p (size))
  forall (i = 1:size)
     p%p(i) = i
  end forall
end subroutine permutation_init

elemental subroutine permutation_final (p)
  type(permutation_t), intent(inout) :: p
  deallocate (p%p)
end subroutine permutation_final
```

I/O:

⟨*Permutations: public*⟩+≡
```
public :: permutation_write
```

⟨*Permutations: procedures*⟩+≡
```
subroutine permutation_write (p, u)
  type(permutation_t), intent (in) :: p
```

```
    integer, intent(in) :: u
    integer :: i
    do i = 1, size (p%p)
       if (size (p%p) < 10) then
          write (u,"(1x,I1)", advance="no") p%p(i)
       else
          write (u,"(1x,I3)", advance="no") p%p(i)
       end if
    end do
    write (u, *)
  end subroutine permutation_write
```

Administration:

⟨*Permutations: public*⟩+≡

```
  public :: permutation_size
```

⟨*Permutations: procedures*⟩+≡

```
  elemental function permutation_size (perm) result (s)
    type(permutation_t), intent(in) :: perm
    integer :: s
    s = size (perm%p)
  end function permutation_size
```

Extract an entry in a permutation.

⟨*Permutations: public*⟩+≡

```
  public :: permute
```

⟨*Permutations: procedures*⟩+≡

```
  elemental function permute (i, p) result (j)
    integer, intent(in) :: i
    type(permutation_t), intent(in) :: p
    integer :: j
    if (i > 0 .and. i <= size (p%p)) then
       j = p%p(i)
    else
       j = 0
    end if
  end function permute
```

Check whether a permutation is valid: Each integer in the range occurs exactly once.

⟨*Permutations: public*⟩+≡

```
  public :: permutation_ok
```

⟨*Permutations: procedures*⟩+≡

```
  elemental function permutation_ok (perm) result (ok)
    type(permutation_t), intent(in) :: perm
    logical :: ok
    integer :: i
    logical, dimension(:), allocatable :: set
    ok = .true.
    allocate (set (size (perm%p)))
    set = .false.
```

```
    do i = 1, size (perm%p)
       ok = (perm%p(i) > 0 .and. perm%p(i) <= size (perm%p))
       if (.not.ok) return
       set(perm%p(i)) = .true.
    end do
    ok = all (set)
  end function permutation_ok
```

Find the permutation that transforms the second array into the first one. We assume that this is possible and unique and all bounds are set correctly.

This cannot be elemental.

⟨*Permutations: public*⟩+≡

```
  public :: permutation_find
```

⟨*Permutations: procedures*⟩+≡

```
  subroutine permutation_find (perm, a1, a2)
    type(permutation_t), intent(inout) :: perm
    integer, dimension(:), intent(in) :: a1, a2
    integer :: i, j
    if (allocated (perm%p))  deallocate (perm%p)
    allocate (perm%p (size (a1)))
    do i = 1, size (a1)
       do j = 1, size (a2)
          if (a1(i) == a2(j)) then
             perm%p(i) = j
             exit
          end if
          perm%p(i) = 0
       end do
    end do
  end subroutine permutation_find
```

Find all permutations that transform an array of integers into itself. The resulting permutation list is allocated with the correct length and filled.

The first step is to count the number of different entries in `code`. Next, we scan `code` again and assign a mask to each different entry, true for all identical entries. Finally, we recursively permute the identity for each possible mask.

The permutation is done as follows: A list of all permutations of the initial one with respect to the current mask is generated, then the permutations are generated in turn for each permutation in this list with the next mask. The result is always stored back into the main list, starting from the end of the current list.

⟨*Permutations: public*⟩+≡

```
  public :: permutation_array_make
```

⟨*Permutations: procedures*⟩+≡

```
  subroutine permutation_array_make (pa, code)
    type(permutation_t), dimension(:), allocatable, intent(out) :: pa
    integer, dimension(:), intent(in) :: code
    logical, dimension(size(code)) :: mask
    logical, dimension(:,:), allocatable :: imask
    integer, dimension(:), allocatable :: n_i
    type(permutation_t) :: p_init
```

```
  type(permutation_t), dimension(:), allocatable :: p_tmp
  integer :: psize, i, j, k, n_different, n, nn_k
  psize = size (code)
  mask = .true.
  n_different = 0
  do i=1, psize
     if (mask(i)) then
        n_different = n_different + 1
        mask = mask .and. (code /= code(i))
     end if
  end do
  allocate (imask(psize, n_different), n_i(n_different))
  mask = .true.
  k = 0
  do i=1, psize
     if (mask(i)) then
        k = k + 1
        imask(:,k) = (code == code(i))
        n_i(k) = factorial (count(imask(:,k)))
        mask = mask .and. (code /= code(i))
     end if
  end do
  n = product (n_i)
  allocate (pa (n))
  call permutation_init (p_init, psize)
  pa(1) = p_init
  nn_k = 1
  do k = 1, n_different
     allocate (p_tmp (n_i(k)))
     do i = nn_k, 1, -1
        call permutation_array_with_mask (p_tmp, imask(:,k), pa(i))
        do j = n_i(k), 1, -1
           pa((i-1)*n_i(k) + j) = p_tmp(j)
        end do
     end do
     deallocate (p_tmp)
     nn_k = nn_k * n_i(k)
  end do
  call permutation_final (p_init)
  deallocate (imask, n_i)
end subroutine permutation_array_make
```

Make a list of permutations of the elements marked true in the `mask` array. The final permutation list must be allocated with the correct length ($n!$). The third argument is the initial permutation to start with, which must have the same length as the `mask` array (this is not checked).

⟨*Permutations: procedures*⟩+≡

```
subroutine permutation_array_with_mask (pa, mask, p_init)
  type(permutation_t), dimension(:), intent(inout) :: pa
  logical, dimension(:), intent(in) :: mask
  type(permutation_t), intent(in) :: p_init
  integer :: plen
  integer :: i, ii, j, fac_i, k, x
```

```
  integer, dimension(:), allocatable :: index
  plen = size (pa)
  allocate (index(count(mask)))
  ii = 0
  do i = 1, size (mask)
     if (mask(i)) then
        ii = ii + 1
        index(ii) = i
     end if
  end do
  pa = p_init
  ii = 0
  fac_i = 1
  do i = 1, size (mask)
     if (mask(i)) then
        ii = ii + 1
        fac_i = fac_i * ii
        x = permute (i, p_init)
        do j = 1, plen
           k = ii - mod (((j-1)*fac_i)/plen, ii)
           call insert (pa(j), x, k, ii, index)
        end do
     end if
  end do
  deallocate (index)
contains
  subroutine insert (p, x, k, n, index)
    type(permutation_t), intent(inout) :: p
    integer, intent(in) :: x, k, n
    integer, dimension(:), intent(in) :: index
    integer :: i
    do i = n, k+1, -1
       p%p(index(i)) = p%p(index(i-1))
    end do
    p%p(index(k)) = x
  end subroutine insert
end subroutine permutation_array_with_mask
```

The factorial function is needed for pre-determining the number of permutations that will be generated:

⟨*Permutations: public*⟩+≡

```
public :: factorial
```

⟨*Permutations: procedures*⟩+≡

```
function factorial (n) result (f)
  integer, intent(in) :: n
  integer :: f
  integer :: i
  f = 1
  do i=2, abs(n)
     f = f*i
  end do
end function factorial
```

6.4.2 Operations on binary codes

Binary codes are needed for phase-space trees. Since the permutation function uses permutations, and no other special type is involved, we put the functions here.

This is needed for phase space trees: permute bits in a tree binary code. If no permutation is given, leave as is. (We may want to access the permutation directly here if this is efficiency-critical.)

⟨*Permutations: public*⟩+≡
```
public :: tc_permute
```

⟨*Permutations: procedures*⟩+≡
```
function tc_permute (k, perm, mask_in) result (pk)
  integer(TC), intent(in) :: k, mask_in
  type(permutation_t), intent(in) :: perm
  integer(TC) :: pk
  integer :: i
  pk = iand (k, mask_in)
  do i = 1, size (perm%p)
     if (btest(k,i-1))  pk = ibset (pk, perm%p(i)-1)
  end do
end function tc_permute
```

This routine returns the number of set bits in the tree code value `k`. Hence, it is the number of externals connected to the current line. If `mask` is present, the complement of the tree code is also considered, and the smaller number is returned. This gives the true distance from the external states, taking into account the initial particles. The complement number is increased by one, since for a scattering diagram the vertex with the sum of all final-state codes is still one point apart from the initial particles.

⟨*Permutations: public*⟩+≡
```
public :: tc_decay_level
```

⟨*Permutations: interfaces*⟩≡
```
interface tc_decay_level
   module procedure decay_level_simple
   module procedure decay_level_complement
end interface
```

⟨*Permutations: procedures*⟩+≡
```
function decay_level_complement (k, mask) result (l)
  integer(TC), intent(in) :: k, mask
  integer :: l
  l = min (decay_level_simple (k), &
        &   decay_level_simple (ieor (k, mask)) + 1)
end function decay_level_complement

function decay_level_simple (k) result(l)
  integer(TC), intent(in) :: k
  integer :: l
  integer :: i
  l = 0
  do i=0, bit_size(k)-1
     if (btest(k,i)) l = l+1
```

```
    end do
  end function decay_level_simple
```

6.5 Sorting

This small module provides functions for sorting integer or real arrays.

⟨`sorting.f90`⟩≡

```
⟨File header⟩

module sorting

⟨Use kinds⟩
  use diagnostics

⟨Standard module head⟩

⟨Sorting: public⟩

⟨Sorting: interfaces⟩

contains

⟨Sorting: procedures⟩

end module sorting
```

6.5.1 Implementation

The `sort` function returns, for a given integer or real array, the array sorted by increasing value. The current implementation is *mergesort*, which has $O(n \ln n)$ behavior in all cases, and is stable for elements of equal value.

The `sort_abs` variant sorts by increasing absolute value, where for identical absolute value, the positive number comes first.

⟨*Sorting: public*⟩≡

```
public :: sort
public :: sort_abs
```

⟨*Sorting: interfaces*⟩≡

```
interface sort
   module procedure sort_int
   module procedure sort_real
end interface

interface sort_abs
   module procedure sort_int_abs
end interface
```

This variant of integer sort returns The body is identical, just the interface differs.

⟨*Sorting: procedures*⟩≡

```
function sort_int (val_in) result (val)
  integer, dimension(:), intent(in) :: val_in
  integer, dimension(size(val_in)) :: val
⟨Sorting: sort⟩
end function sort_int

function sort_real (val_in) result (val)
  real(default), dimension(:), intent(in) :: val_in
  real(default), dimension(size(val_in)) :: val
⟨Sorting: sort⟩
end function sort_real

function sort_int_abs (val_in) result (val)
  integer, dimension(:), intent(in) :: val_in
  integer, dimension(size(val_in)) :: val
⟨Sorting: sort abs⟩
end function sort_int_abs
```

⟨*Sorting: sort*⟩≡

```
val = val_in( order (val_in) )
```

⟨*Sorting: sort abs*⟩≡

```
val = val_in( order_abs (val_in) )
```

The `order` function returns, for a given integer or real array, the array of indices of the elements sorted by increasing value.

⟨*Sorting: public*⟩+≡

```
public :: order
public :: order_abs
```

⟨*Sorting: interfaces*⟩+≡

```
interface order
   module procedure order_int
   module procedure order_real
end interface

interface order_abs
   module procedure order_int_abs
end interface
```

⟨*Sorting: procedures*⟩+≡

```
function order_int (val) result (idx)
  integer, dimension(:), intent(in) :: val
  integer, dimension(size(val)) :: idx
⟨Sorting: order⟩
end function order_int

function order_real (val) result (idx)
  real(default), dimension(:), intent(in) :: val
  integer, dimension(size(val)) :: idx
⟨Sorting: order⟩
end function order_real

function order_int_abs (val) result (idx)
  integer, dimension(:), intent(in) :: val
```

```
  integer, dimension(size(val)) :: idx
⟨Sorting: order abs⟩
end function order_int_abs
```

We start by individual elements, merge them to pairs, merge those to four-element subarrays, and so on. The last subarray can extend only up to the original array bound, of course, and the second of the subarrays to merge should contain at least one element.

⟨*Sorting: order*⟩≡

```
⟨Sorting: order1⟩
        call merge (idx(b1:e2), idx(b1:e1), idx(b2:e2), val)
⟨Sorting: order2⟩
```

⟨*Sorting: order abs*⟩≡

```
⟨Sorting: order1⟩
        call merge_abs (idx(b1:e2), idx(b1:e1), idx(b2:e2), val)
⟨Sorting: order2⟩
```

⟨*Sorting: order1*⟩≡

```
integer :: n, i, s, b1, b2, e1, e2
n = size (idx)
!!! !!! !!! Work around ifort 16.0.(0/1) compiler bug
!forall (i = 1:n)
   !idx(i) = i
!end forall
do i = 1, n
   idx(i) = i
end do
s = 1
do while (s < n)
   do b1 = 1, n-s, 2*s
      b2 = b1 + s
      e1 = b2 - 1
      e2 = min (e1 + s, n)
```

⟨*Sorting: order2*⟩≡

```
   end do
   s = 2 * s
end do
```

The merging step does the actual sorting. We take two sorted array sections and merge them to a sorted result array. We are working on the indices, and comparing is done by taking the associated `val` which is real or integer.

⟨*Sorting: interfaces*⟩+≡

```
interface merge
   module procedure merge_int
   module procedure merge_real
end interface

interface merge_abs
   module procedure merge_int_abs
end interface
```

⟨*Sorting: procedures*⟩+≡

```
subroutine merge_int (res, src1, src2, val)
  integer, dimension(:), intent(out) :: res
  integer, dimension(:), intent(in) :: src1, src2
  integer, dimension(:), intent(in) :: val
  integer, dimension(size(res)) :: tmp
⟨Sorting: merge⟩
end subroutine merge_int

subroutine merge_real (res, src1, src2, val)
  integer, dimension(:), intent(out) :: res
  integer, dimension(:), intent(in) :: src1, src2
  real(default), dimension(:), intent(in) :: val
  integer, dimension(size(res)) :: tmp
⟨Sorting: merge⟩
end subroutine merge_real

subroutine merge_int_abs (res, src1, src2, val)
  integer, dimension(:), intent(out) :: res
  integer, dimension(:), intent(in) :: src1, src2
  integer, dimension(:), intent(in) :: val
  integer, dimension(size(res)) :: tmp
⟨Sorting: merge abs⟩
end subroutine merge_int_abs
```

⟨*Sorting: merge*⟩≡

```
⟨Sorting: merge1⟩
     if (val(src1(i1)) <= val(src2(i2))) then
⟨Sorting: merge2⟩
```

We keep the elements if the absolute values are strictly ordered. If they are equal in magnitude, we keep them if the larger value comes first, or if they are equal.

⟨*Sorting: merge abs*⟩≡

```
⟨Sorting: merge1⟩
     if (abs (val(src1(i1))) < abs (val(src2(i2))) .or. &
        (abs (val(src1(i1))) == abs (val(src2(i2))) .and. &
        val(src1(i1)) >= val(src2(i2)))) then
⟨Sorting: merge2⟩
```

⟨*Sorting: merge1*⟩≡

```
integer :: i1, i2, i
i1 = 1
i2 = 1
do i = 1, size (tmp)
```

⟨*Sorting: merge2*⟩≡

```
        tmp(i) = src1(i1);  i1 = i1 + 1
        if (i1 > size (src1)) then
           tmp(i+1:) = src2(i2:)
           exit
        end if
     else
        tmp(i) = src2(i2);  i2 = i2 + 1
        if (i2 > size (src2)) then
```

```
            tmp(i+1:) = src1(i1:)
            exit
         end if
      end if
   end do
   res = tmp
```

6.5.2 Concatenating arrays

Not precisely a sorting function, but useful: Concatenate two arrays.

⟨*Sorting: public*⟩+≡
```
   public :: concat
```

⟨*Sorting: interfaces*⟩+≡
```
   interface concat
      module procedure concat_int
      module procedure concat_real
   end interface
```

⟨*Sorting: procedures*⟩+≡
```
   function concat_int (val1, val2) result (val12)
     integer, dimension(:), intent(in) :: val1, val2
     integer, dimension(size(val1)+size(val2)) :: val12
   ⟨Sorting: concat⟩
   end function concat_int

   function concat_real (val1, val2) result (val12)
     real(default), dimension(:), intent(in) :: val1, val2
     integer, dimension(size(val1)+size(val2)) :: val12
   ⟨Sorting: concat⟩
   end function concat_real
```

⟨*Sorting: concat*⟩≡
```
   val12(:size(val1)) = val1
   val12(size(val1)+1:) = val2
```

6.5.3 Unit tests

Test module, followed by the corresponding implementation module.

⟨`sorting_ut.f90`⟩≡
```
  ⟨File header⟩

  module sorting_ut
    use unit_tests
    use sorting_uti

  ⟨Standard module head⟩

  ⟨Sorting: public test⟩

  contains
```

```
  ⟨Sorting: test driver⟩

  end module sorting_ut
```

⟨sorting_uti.f90⟩≡

```
  ⟨File header⟩

  module sorting_uti

  ⟨Use kinds⟩

    use sorting

  ⟨Standard module head⟩

  ⟨Sorting: test declarations⟩

  contains

  ⟨Sorting: tests⟩

  end module sorting_uti
```

API: driver for the unit tests below.

⟨*Sorting: public test*⟩≡

```
    public :: sorting_test
```

⟨*Sorting: test driver*⟩≡

```
    subroutine sorting_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Sorting: execute tests⟩
    end subroutine sorting_test
```

This checks whether the sorting routine works correctly.

⟨*Sorting: execute tests*⟩≡

```
    call test (sorting_1, "sorting_1", &
         "check sorting routines", &
         u, results)
```

⟨*Sorting: test declarations*⟩≡

```
    public :: sorting_1
```

⟨*Sorting: tests*⟩≡

```
    subroutine sorting_1 (u)
      integer, intent(in) :: u
      integer, parameter :: NMAX = 10
      real(default), dimension(NMAX) :: rval
      integer, dimension(NMAX) :: ival
      real, dimension(NMAX,NMAX) :: harvest_r
      integer, dimension(NMAX,NMAX) :: harvest_i
      integer, dimension(NMAX,NMAX) :: harvest_a
      integer :: i, j
      harvest_r(:, 1) = [0.9976, 0., 0., 0., 0., 0., 0., 0., 0., 0.]
      harvest_r(:, 2) = [0.5668, 0.9659, 0., 0., 0., 0., 0., 0., 0., 0.]
      harvest_r(:, 3) = [0.7479, 0.3674, 0.4806, 0., 0., 0., 0., 0., 0., &
```

```
     0.]
harvest_r(:, 4) = [0.0738, 0.0054, 0.3471, 0.3422, 0., 0., 0., 0., &
     0., 0.]
harvest_r(:, 5) = [0.2180, 0.1332, 0.9005, 0.3868, 0.4455, 0., 0., &
     0., 0., 0.]
harvest_r(:, 6) = [0.6619, 0.0161, 0.6509, 0.6464, 0.3230, &
     0.8557, 0., 0., 0., 0.]
harvest_r(:, 7) = [0.4013, 0.2069, 0.9685, 0.5984, 0.6730, &
     0.4569, 0.3300, 0., 0., 0.]
harvest_r(:, 8) = [0.1004, 0.7555, 0.6057, 0.7190, 0.8973, &
     0.6582, 0.1507, 0.6123, 0., 0.]
harvest_r(:, 9) = [0.9787, 0.9991, 0.2568, 0.5509, 0.6590, &
     0.5540, 0.9778, 0.9019, 0.6579, 0.]
harvest_r(:,10) = [0.7289, 0.4025, 0.9286, 0.1478, 0.6745, &
     0.7696, 0.3393, 0.1158, 0.6144, 0.8206]

harvest_i(:, 1) = [18, 0, 0, 0, 0, 0, 0, 0, 0, 0]
harvest_i(:, 2) = [14, 9, 0, 0, 0, 0, 0, 0, 0, 0]
harvest_i(:, 3) = [ 7, 8,11, 0, 0, 0, 0, 0, 0, 0]
harvest_i(:, 4) = [19,19,14,19, 0, 0, 0, 0, 0, 0]
harvest_i(:, 5) = [ 1,14,15,18,14, 0, 0, 0, 0, 0]
harvest_i(:, 6) = [16,11, 1, 9,11, 2, 0, 0, 0, 0]
harvest_i(:, 7) = [11,10,17, 6,13,13,10, 0, 0, 0]
harvest_i(:, 8) = [ 5, 1, 2,10, 7, 0,15,12, 0, 0]
harvest_i(:, 9) = [15,19, 2, 6,11, 0, 2, 4, 2, 0]
harvest_i(:,10) = [ 1, 4, 8, 4,11, 0, 8, 7,19,13]

harvest_a(:, 1) = [-6,  0,  0,  0,  0,  0,  0,  0,  0,  0]
harvest_a(:, 2) = [-8, -9,  0,  0,  0,  0,  0,  0,  0,  0]
harvest_a(:, 3) = [ 4, -3,  3,  0,  0,  0,  0,  0,  0,  0]
harvest_a(:, 4) = [-6,  6,  2, -2,  0,  0,  0,  0,  0,  0]
harvest_a(:, 5) = [ 1, -2,  0, -6,  8,  0,  0,  0,  0,  0]
harvest_a(:, 6) = [-2, -1, -8, -5,  8, -5,  0,  0,  0,  0]
harvest_a(:, 7) = [-9,  0, -6,  2,  5,  3,  2,  0,  0,  0]
harvest_a(:, 8) = [-5, -7,  6,  7, -3,  0, -7,  4,  0,  0]
harvest_a(:, 9) = [ 5,  0, -1, -7,  5,  2,  7, -3,  3,  0]
harvest_a(:,10) = [-9,  2, -6,  3, -9,  5,  5,  7,  5, -9]


write (u, "(A)")  "* Test output: Sorting"
write (u, "(A)")  "*   Purpose: test sorting routines"
write (u, "(A)")

write (u, "(A)")  "* Sorting real values:"

do i = 1, NMAX
   write (u, "(A)")
   rval(:i) = harvest_r(:i,i)
   write (u, "(10(1x,F7.4))") rval(:i)
   rval(:i) = sort (rval(:i))
   write (u, "(10(1x,F7.4))") rval(:i)
   do j = i, 2, -1
      if (rval(j)-rval(j-1) < 0) &
         write (u, "(A)") "*** Sorting failure. ***"
```

```
      end do
   end do

   write (u, "(A)")
   write (u, "(A)") "* Sorting integer values:"

   do i = 1, NMAX
      write (u, "(A)")
      ival(:i) = harvest_i(:i,i)
      write (u, "(10(1x,I2))") ival(:i)
      ival(:i) = sort (ival(:i))
      write (u, "(10(1x,I2))") ival(:i)
      do j = i, 2, -1
         if (ival(j)-ival(j-1) < 0) &
            write (u, "(A)")  "*** Sorting failure. ***"
      end do
   end do

   write (u, "(A)")
   write (u, "(A)") "* Sorting integer values by absolute value:"

   do i = 1, NMAX
      write (u, "(A)")
      ival(:i) = harvest_a(:i,i)
      write (u, "(10(1x,I2))") ival(:i)
      ival(:i) = sort_abs (ival(:i))
      write (u, "(10(1x,I2))") ival(:i)
      do j = i, 2, -1
         if (abs(ival(j))-abs(ival(j-1)) < 0 .or. &
              (abs(ival(j))==abs(ival(j-1))) .and. ival(j)>ival(j-1)) &
            write (u, "(A)")  "*** Sorting failure. ***"
      end do
   end do

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: sorting_1"

end subroutine sorting_1
```

6.6 Grids

This is not really a combinatorics module but this directory is the closest I could find. Maybe this will be moved to a seperate directory or combined with related stuff.

⟨grids.f90⟩≡

```
⟨File header⟩

module grids

⟨Use kinds⟩
  use constants, only: zero, one, tiny_07
```

```
  use io_units
  use format_defs, only: FMT_16
  use diagnostics
```

⟨*Standard module head*⟩

⟨*grids: public*⟩

⟨*grids: interfaces*⟩

⟨*grids: parameters*⟩

⟨*grids: types*⟩

```
contains
```

⟨*grids: procedures*⟩

```
end module grids
```

Grids are used in many applications and a general implementation seems useful. The relevant properties implemented so far are

- Segments of the hypercube are represented by an integer array with size d corresponding to the dimension.
- There is a mapping from the indices to the location in the continuous memory block of values.
- Given a point in the hypercube, find the corresponding segment and the value of the grid therein.
- Update the grid sequentially to represent the maximum of a function over the unit hypercube.
- The grid can be saved to and recovered from disk.

The following might be implemented in the future

- Generate a random point in the hypercube by interpreting the grid as probability distribution.

  This would most likely be solved by using projections and the `selector_t`*, which would make a move of this module higher up in the dependency tree necessary.*

- Update the grid sequentially to represent the *minimum* of a function over the unit hypercube.

⟨*grids: public*⟩≡

```
public :: grid_t
```

⟨*grids: types*⟩≡

```
type :: grid_t
   private
   real(default), dimension(:), allocatable :: values
   integer, dimension(:), allocatable :: points
```

```
contains
 ⟨grids: grid: TBP⟩
end type grid_t
```

6.6.1 Initializer and finalizer

For initialization, we expect the number of points for each dimension as an array or the the number of dimensions as a scalar whereby the default number of points is used then for each dimension.

⟨*grids: grid: TBP*⟩≡

```
generic :: init => init_base, init_simple
procedure :: init_base => grid_init_base
procedure :: init_simple => grid_init_simple
```

⟨*grids: procedures*⟩≡

```
pure subroutine grid_init_base (grid, points)
  class(grid_t), intent(inout) :: grid
  integer, dimension(:), intent(in) :: points
  allocate (grid%points (size (points)))
  allocate (grid%values (product (points)))
  grid%points = points
  grid%values = zero
end subroutine grid_init_base
```

⟨*grids: procedures*⟩+≡

```
pure subroutine grid_init_simple (grid, dimensions)
  class(grid_t), intent(inout) :: grid
  integer, intent(in) :: dimensions
  allocate (grid%points (dimensions))
  allocate (grid%values (DEFAULT_POINTS_PER_DIMENSION ** dimensions))
  grid%points = DEFAULT_POINTS_PER_DIMENSION
  grid%values = zero
end subroutine grid_init_simple
```

Manual assignment (tests)

⟨*grids: grid: TBP*⟩+≡

```
procedure :: set_values => grid_set_values
```

⟨*grids: procedures*⟩+≡

```
subroutine grid_set_values (grid, values)
  class(grid_t), intent(inout) :: grid
  real(default), dimension(:), intent(in) :: values
  grid%values = values
end subroutine grid_set_values
```

A reasonable default

⟨*grids: parameters*⟩≡

```
integer, parameter :: DEFAULT_POINTS_PER_DIMENSION = 100
```

Calling this is not mandatory, when an instance of `grid_t` goes out of scope as it will be done by Fortran automatically.

⟨*grids: grid: TBP*⟩+≡

```
procedure :: final => grid_final
```

⟨*grids: procedures*⟩+≡

```
pure subroutine grid_final (grid)
  class(grid_t), intent(inout) :: grid
  if (allocated (grid%values)) then
     deallocate (grid%values)
  end if
  if (allocated (grid%points)) then
     deallocate (grid%points)
  end if
end subroutine grid_final
```

6.6.2 Segment finding and memory mapping

The `indices` array is expected to go from 1 to d whereby the entries for the different dims are from 1 to $n_{\text{points}}(\text{dim})$.
We get the value of the grid either from given `indices` or from a point `x` in the hypercube. In the latter case, we have to find the segment first.

⟨*grids: grid: TBP*⟩+≡

```
generic :: get_value => get_value_from_x, get_value_from_indices
procedure :: get_value_from_x => grid_get_value_from_x
procedure :: get_value_from_indices => grid_get_value_from_indices
```

⟨*grids: procedures*⟩+≡

```
function grid_get_value_from_indices (grid, indices)
  real(default) :: grid_get_value_from_indices
  class(grid_t), intent(in) :: grid
  integer, dimension(:), intent(in) :: indices
  grid_get_value_from_indices = grid%values(grid%get_index(indices))
end function grid_get_value_from_indices
```

⟨*grids: procedures*⟩+≡

```
function grid_get_value_from_x (grid, x)
  real(default) :: grid_get_value_from_x
  class(grid_t), intent(in) :: grid
  real(default), dimension(:), intent(in) :: x
  grid_get_value_from_x = grid_get_value_from_indices &
       (grid, grid_get_segment (grid, x))
end function grid_get_value_from_x
```

The segment is the part of the grid that contains the point `x` and is identified by a tupel of `indices`. This is just a brute force search, for fine grids one could also implement a binary search for $\mathcal{O}(\log N)$ behavior instead of $\mathcal{O}(N)$.

⟨*grids: grid: TBP*⟩+≡

```
procedure :: get_segment => grid_get_segment
```

⟨*grids: procedures*⟩+≡

```
function grid_get_segment (grid, x, unit)
  class(grid_t), intent(in) :: grid
  real(default), dimension(:), intent(in) :: x
  integer, intent(in), optional :: unit
  integer, dimension(1:size (x)) :: grid_get_segment
  integer :: dim, i
```

```
    real(default) :: segment_width
    grid_get_segment = 0
    do dim = 1, size (grid%points)
       segment_width = one / grid%points (dim)
       SEARCH: do i = 1, grid%points (dim)
          if (x (dim) <= i * segment_width + tiny_07) then
             grid_get_segment (dim) = i
             exit SEARCH
          end if
       end do SEARCH
       if (grid_get_segment (dim) == 0) then
          do i = 1, size(x)
             write (msg_buffer, "(A," // DEFAULT_OUTPUT_PRECISION // ")") &
                  "x[i] = ", x(i)
             call msg_message ()
          end do
          call msg_error ("grid_get_segment: Did not find x in [0,1]^d", &
               unit=unit)
       end if
    end do
  end function grid_get_segment
```

This is a simple storage mapping function but more sophisticated ideas like hashing could be implemented.

$$\begin{aligned}\text{index} =& \text{indices}(1)+\\ & \text{indices}(2) * \text{size}(1)+\\ & \text{indices}(3) * \text{size}(1) * \text{size}(2) + \ldots \end{aligned} \tag{6.1}$$

⟨*grids: grid: TBP*⟩+≡

```
  procedure :: get_index => grid_get_index
```

⟨*grids: procedures*⟩+≡

```
  pure function grid_get_index (grid, indices) result (grid_index)
    integer :: grid_index
    class(grid_t), intent(in) :: grid
    integer, dimension(:), intent(in) :: indices
    integer :: dim_innerloop, dim_outerloop, multiplier
    grid_index = 1
    do dim_outerloop = 1, size(indices)
       multiplier = 1
       do dim_innerloop = 1, dim_outerloop - 1
          multiplier = multiplier * grid%points (dim_innerloop)
       end do
       grid_index = grid_index + (indices(dim_outerloop) - 1) * multiplier
    end do
  end function grid_get_index
```

6.6.3 Grid manipulations

Given a point in the hypercube `x` and its value `y`, we update the grids, such that the stepwise function f defined by the grid is $f(x_i) \geq y_i \; \forall \{x_i, y_i\}$.

⟨*grids: grid: TBP*⟩+≡

```
procedure :: update_maxima => grid_update_maxima
```

⟨*grids: procedures*⟩+≡

```
subroutine grid_update_maxima (grid, x, y)
  class(grid_t), intent(inout) :: grid
  real(default), dimension(:), intent(in) :: x
  real(default), intent(in) :: y
  integer, dimension(1:size(x)) :: indices
  indices = grid%get_segment (x)
  if (grid%get_value (indices) < y) then
     grid%values (grid%get_index (indices)) = y
  end if
end subroutine grid_update_maxima
```

More general cases have to be thought through when they are needed. *This is inefficient and non-general.*

⟨*grids: grid: TBP*⟩+≡

```
procedure :: get_maximum_in_3d => grid_get_maximum_in_3d
```

⟨*grids: procedures*⟩+≡

```
function grid_get_maximum_in_3d (grid, projected_index) result (maximum)
  real(default) :: maximum
  class(grid_t), intent(in) :: grid
  integer, intent(in) :: projected_index
  real(default) :: val
  integer :: i, j
  maximum = zero
  do i = 1, grid%points(1)
     do j = 1, grid%points(2)
        val = grid%get_value ([i, j, projected_index])
        if (val > maximum) then
           maximum = val
        end if
     end do
  end do

end function grid_get_maximum_in_3d
```

⟨*grids: grid: TBP*⟩+≡

```
procedure :: is_non_zero_everywhere => grid_is_non_zero_everywhere
```

⟨*grids: procedures*⟩+≡

```
pure function grid_is_non_zero_everywhere (grid) result (yorn)
  logical :: yorn
  class(grid_t), intent(in) :: grid
  yorn = all (abs (grid%values) > zero)
end function grid_is_non_zero_everywhere
```

6.6.4 Input and Output to screen and disk

⟨*grids: grid: TBP*⟩+≡

```
procedure :: write => grid_write
```

⟨*grids: procedures*⟩+≡

```
subroutine grid_write (grid, unit)
  class(grid_t), intent(in) :: grid
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1X,A)") "Grid"
  write (u, "(2X,A,2X)", advance='no') "Number of points per dimension:"
  if (allocated (grid%points)) then
     do i = 1, size (grid%points)
        write (u, "(I12,1X)", advance='no') &
             grid%points (i)
     end do
  end if
  write (u, *)
  write (u, "(2X,A)") "Values of the grid:"
  if (allocated (grid%values)) then
     do i = 1, size (grid%values)
        write (u, "(" // DEFAULT_OUTPUT_PRECISION // ",1X)") &
             grid%values (i)
     end do
  end if
  call grid%compute_and_write_mean_and_max (u)
end subroutine grid_write
```

⟨*grids: grid: TBP*⟩+≡

```
procedure :: compute_and_write_mean_and_max => &
     grid_compute_and_write_mean_and_max
```

⟨*grids: procedures*⟩+≡

```
subroutine grid_compute_and_write_mean_and_max (grid, unit)
  class(grid_t), intent(in) :: grid
  integer, intent(in), optional :: unit
  integer :: u, i, n_values
  real(default) :: mean, val, maximum
  u = given_output_unit (unit);  if (u < 0)  return
  mean = zero
  maximum = zero
  if (allocated (grid%values)) then
     n_values = size (grid%values)
     do i = 1, n_values
        val = grid%values (i)
        mean = mean + val / n_values
        if (val > maximum) then
           maximum = val
        end if
     end do
     write (msg_buffer, "(A," // DEFAULT_OUTPUT_PRECISION // ")") &
          "Grid: Mean value of the grid: ", mean
     call msg_message ()
     write (msg_buffer, "(A," // DEFAULT_OUTPUT_PRECISION // ")") &
          "Grid: Max value of the grid: ", maximum
     call msg_message ()
     if (maximum > zero) then
```

```
            write (msg_buffer, "(A," // DEFAULT_OUTPUT_PRECISION // ")") &
                 "Grid: Mean/Max value of the grid: ", mean / maximum
            call msg_message ()
         end if
      else
         call msg_warning ("Grid: Grid is not allocated!")
      end if
   end subroutine grid_compute_and_write_mean_and_max
```

⟨*grids: grid: TBP*⟩+≡

```
   procedure :: save_to_file => grid_save_to_file
```

⟨*grids: procedures*⟩+≡

```
   subroutine grid_save_to_file (grid, file)
      class(grid_t), intent(in) :: grid
      character(len=*), intent(in) :: file
      integer :: iostat, u, i
      u = free_unit ()
      open (file=file, unit=u, action='write')
      if (allocated (grid%points)) then
         write (u, "(I12)") size (grid%points)
         do i = 1, size (grid%points)
            write (u, "(I12,1X)", advance='no', iostat=iostat) &
                 grid%points (i)
         end do
      end if
      write (u, *)
      if (allocated (grid%values)) then
         do i = 1, size (grid%values)
            write (u, "(" // DEFAULT_OUTPUT_PRECISION // ",1X)", &
                 advance='no', iostat=iostat) grid%values (i)
         end do
      end if
      if (iostat < 0) then
         call msg_warning &
              ('grid_save_to_file: Could not save grid to file')
      end if
      close (u)
   end subroutine grid_save_to_file
```

⟨*grids: parameters*⟩+≡

```
   character(len=*), parameter :: DEFAULT_OUTPUT_PRECISION = FMT_16
```

⟨*grids: grid: TBP*⟩+≡

```
   procedure :: load_from_file => grid_load_from_file
```

⟨*grids: procedures*⟩+≡

```
   subroutine grid_load_from_file (grid, file)
      class(grid_t), intent(out) :: grid
      character(len=*), intent(in) :: file
      integer :: iostat, u, i, n_dimensions
      integer, dimension(:), allocatable :: points
      u = free_unit ()
      open (file=file, unit=u, action='read', iostat=iostat)
```

```
    read (u, "(I12)", iostat=iostat) n_dimensions
    allocate (points (n_dimensions))
    do i = 1, size (points)
       read (u, "(I12,1X)", advance='no', iostat=iostat) &
            points (i)
    end do
    read (u, *)
    call grid%init (points)
    do i = 1, size (grid%values)
       read (u, "(" // DEFAULT_OUTPUT_PRECISION // ",1X)", advance='no', iostat=iostat) &
            grid%values (i)
    end do
    if (iostat < 0) then
       call msg_warning ('grid_load_from_file: Could not load grid from file')
    end if
    close (u)
  end subroutine grid_load_from_file
```

6.6.5 Unit tests

Test module, followed by the corresponding implementation module.

⟨grids_ut.f90⟩≡

```
  ⟨File header⟩

  module grids_ut
    use unit_tests
    use grids_uti

  ⟨Standard module head⟩

  ⟨grids: public test⟩

  contains

  ⟨grids: test driver⟩

  end module grids_ut
```

⟨grids_uti.f90⟩≡

```
  ⟨File header⟩

  module grids_uti

  ⟨Use kinds⟩
    use constants, only: zero, one, two, three, four, tiny_07
    use file_utils, only: delete_file
    use numeric_utils

    use grids

  ⟨Standard module head⟩

  ⟨grids: test declarations⟩
```

```
contains

⟨grids: tests⟩

end module grids_uti
```

API: driver for the unit tests below.

⟨*grids: public test*⟩≡
```
public :: grids_test
```

⟨*grids: test driver*⟩≡
```
subroutine grids_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨grids: execute tests⟩
end subroutine grids_test
```

Test Index Function

⟨*grids: execute tests*⟩≡
```
call test(grids_1, "grids_1", &
     "Test Index Function", u, results)
```

⟨*grids: test declarations*⟩≡
```
public :: grids_1
```

⟨*grids: tests*⟩≡
```
subroutine grids_1 (u)
  integer, intent(in) :: u
  type(grid_t) :: grid
  write (u, "(A)")  "* Test output: grids_1"
  write (u, "(A)")  "*   Purpose: Test Index Function"
  write (u, "(A)")

  call grid%init ([3])
  call grid%write(u)
  call assert (u, grid%get_index([1]) == 1, "grid%get_index(1) == 1")
  call assert (u, grid%get_index([2]) == 2, "grid%get_index(2) == 2")
  call assert (u, grid%get_index([3]) == 3, "grid%get_index(3) == 3")
  call grid%final ()

  call grid%init ([3,3])
  call grid%write(u)
  call assert (u, grid%get_index([1,1]) == 1, "grid%get_index(1,1) == 1")
  call assert (u, grid%get_index([2,1]) == 2, "grid%get_index(2,1) == 2")
  call assert (u, grid%get_index([3,1]) == 3, "grid%get_index(3,1) == 3")
  call assert (u, grid%get_index([1,2]) == 4, "grid%get_index(1,2) == 4")
  call assert (u, grid%get_index([2,2]) == 5, "grid%get_index(2,2) == 5")
  call assert (u, grid%get_index([3,2]) == 6, "grid%get_index(3,2) == 6")
  call assert (u, grid%get_index([1,3]) == 7, "grid%get_index(1,3) == 7")
  call assert (u, grid%get_index([2,3]) == 8, "grid%get_index(2,3) == 8")
  call assert (u, grid%get_index([3,3]) == 9, "grid%get_index(3,3) == 9")
  call grid%final ()
```

```
    call grid%init ([3,3,2])
    call grid%write(u)
    call assert (u, grid%get_index([1,1,1]) == 1,   "grid%get_index(1,1,1) == 1")
    call assert (u, grid%get_index([2,1,2]) == 2+9, "grid%get_index(2,1,2) == 2+9")
    call assert (u, grid%get_index([3,3,1]) == 9,   "grid%get_index(3,3,1) == 3")
    call assert (u, grid%get_index([3,1,2]) == 3+9, "grid%get_index(3,1,2) == 4+9")
    call assert (u, grid%get_index([2,2,1]) == 5,   "grid%get_index(2,2,1) == 5")
    call assert (u, grid%get_index([3,2,2]) == 6+9, "grid%get_index(3,2,2) == 6+9")
    call assert (u, grid%get_index([1,3,1]) == 7,   "grid%get_index(1,3,1) == 7")
    call assert (u, grid%get_index([2,3,2]) == 8+9, "grid%get_index(2,3,2) == 8+9")
    call assert (u, grid%get_index([3,3,2]) == 9+9, "grid%get_index(3,3,2) == 9+9")
    call grid%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: grids_1"
  end subroutine grids_1
```

Saving and Loading

⟨*grids: execute tests*⟩+≡

```
  call test(grids_2, "grids_2", &
            "Saving and Loading", u, results)
```

⟨*grids: test declarations*⟩+≡

```
  public :: grids_2
```

⟨*grids: tests*⟩+≡

```
  subroutine grids_2 (u)
    integer, intent(in) :: u
    type(grid_t) :: grid
    write (u, "(A)")  "* Test output: grids_2"
    write (u, "(A)")  "*   Purpose: Saving and Loading"
    write (u, "(A)")

    call grid%init ([3])
    call grid%set_values ([one, two, three])
    call grid%save_to_file ('grids_2_test')
    call grid%final ()

    call grid%load_from_file ('grids_2_test')
    call grid%write (u)
    call assert (u, nearly_equal (grid%get_value([1]), one),   "grid%get_value(1) == 1")
    call assert (u, nearly_equal (grid%get_value([2]), two),   "grid%get_value(2) == 2")
    call assert (u, nearly_equal (grid%get_value([3]), three), "grid%get_value(3) == 3")
    call grid%final ()

    call grid%init ([3,3])
    call grid%set_values ([one, two, three, four, zero, zero, zero, zero, zero])
    call grid%save_to_file ('grids_2_test')
    call grid%final ()

    call grid%load_from_file ('grids_2_test')
    call grid%write (u)
    call assert (u, nearly_equal (grid%get_value([1,1]), one),   "grid%get_value(1,1) == 1")
```

```
    call assert (u, nearly_equal (grid%get_value([2,1]), two),   "grid%get_value(2,1) == 2")
    call assert (u, nearly_equal (grid%get_value([3,1]), three), "grid%get_value(3,1) == 3")
    call assert (u, nearly_equal (grid%get_value([1,2]), four),  "grid%get_value(1,2) == 4")
    call delete_file ('grids_2_test')

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: grids_2"
  end subroutine grids_2
```

Get Segments

⟨*grids: execute tests*⟩+≡

```
  call test(grids_3, "grids_3", &
            "Get Segments", u, results)
```

⟨*grids: test declarations*⟩+≡

```
  public :: grids_3
```

⟨*grids: tests*⟩+≡

```
  subroutine grids_3 (u)
    integer, intent(in) :: u
    type(grid_t) :: grid
    integer, dimension(2) :: fail
    write (u, "(A)")  "* Test output: grids_3"
    write (u, "(A)")  "*   Purpose: Get Segments"
    write (u, "(A)")

    call grid%init ([3])
    call assert (u, all(grid%get_segment([0.00_default]) == [1]), &
                   "all(grid%get_segment([0.00_default]) == [1])")
    call assert (u, all(grid%get_segment([0.32_default]) == [1]), &
                   "all(grid%get_segment([0.32_default]) == [1])")
    call assert (u, all(grid%get_segment([0.52_default]) == [2]), &
                   "all(grid%get_segment([0.52_default]) == [2])")
    call assert (u, all(grid%get_segment([1.00_default]) == [3]), &
                   "all(grid%get_segment([1.00_default]) == [3])")
    call grid%final ()

    call grid%init ([3,3])
    call assert (u, all(grid%get_segment([0.00_default,0.00_default]) == [1,1]), &
                   "all(grid%get_segment([0.00_default,0.00_default]) == [1,1])")
    call assert (u, all(grid%get_segment([0.32_default,0.32_default]) == [1,1]), &
                   "all(grid%get_segment([0.32_default,0.32_default]) == [1,1])")
    call assert (u, all(grid%get_segment([0.52_default,0.52_default]) == [2,2]), &
                   "all(grid%get_segment([0.52_default,0.52_default]) == [2,2])")
    call assert (u, all(grid%get_segment([1.00_default,1.00_default]) == [3,3]), &
                   "all(grid%get_segment([1.00_default,1.00_default]) == [3,3])")
    write (u, "(A)")  "* A double error is expected"
    fail = grid%get_segment([1.10_default,1.10_default], u)
    call grid%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: grids_3"
  end subroutine grids_3
```

Update Maxima

⟨*grids: execute tests*⟩+≡
```
call test(grids_4, "grids_4", &
          "Update Maxima", u, results)
```

⟨*grids: test declarations*⟩+≡
```
public :: grids_4
```

⟨*grids: tests*⟩+≡
```
subroutine grids_4 (u)
  integer, intent(in) :: u
  type(grid_t) :: grid
  write (u, "(A)")  "* Test output: grids_4"
  write (u, "(A)")  "*   Purpose: Update Maxima"
  write (u, "(A)")

  call grid%init ([4,4])
  call grid%update_maxima ([0.1_default, 0.0_default], 0.3_default)
  call grid%update_maxima ([0.9_default, 0.95_default], 1.7_default)
  call grid%write (u)
  call assert_equal (u, grid%get_value([1,1]), 0.3_default, &
             "grid%get_value([1,1]")
  call assert_equal (u, grid%get_value([2,2]), 0.0_default, &
             "grid%get_value([2,2]")
  call assert_equal (u, grid%get_value([4,4]), 1.7_default, &
             "grid%get_value([4,4]")

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: grids_4"
end subroutine grids_4
```

Finding and checking

⟨*grids: execute tests*⟩+≡
```
call test(grids_5, "grids_5", &
          "Finding and checking", u, results)
```

⟨*grids: test declarations*⟩+≡
```
public :: grids_5
```

⟨*grids: tests*⟩+≡
```
subroutine grids_5 (u)
  integer, intent(in) :: u
  type(grid_t) :: grid
  real(default) :: first, second
  write (u, "(A)")  "* Test output: grids_5"
  write (u, "(A)")  "*   Purpose: Finding and checking"
  write (u, "(A)")

  call grid%init ([2,2,2])
  first = one / two - tiny_07
  second = two / two - tiny_07
```

```
    call grid%update_maxima ([0.1_default, 0.0_default, first], 0.3_default)
    call grid%update_maxima ([0.9_default, 0.95_default, second], 1.7_default)
    call grid%write (u)
    call assert (u, .not. grid%is_non_zero_everywhere (), &
           ".not. grid%is_non_zero_everywhere (")
    call assert_equal (u, grid%get_maximum_in_3d (1), 0.3_default, &
         "grid%get_maximum_in_3d (1)")
    call assert_equal (u, grid%get_maximum_in_3d (2), 1.7_default, &
         "grid%get_maximum_in_3d (2)")

    call grid%update_maxima ([0.9_default, 0.95_default, first], 1.8_default)
    call grid%update_maxima ([0.1_default, 0.95_default, first], 1.5_default)
    call grid%update_maxima ([0.9_default, 0.15_default, first], 1.5_default)
    call grid%update_maxima ([0.1_default, 0.0_default, second], 0.2_default)
    call grid%update_maxima ([0.1_default, 0.9_default, second], 0.2_default)
    call grid%update_maxima ([0.9_default, 0.0_default, second], 0.2_default)
    call grid%write (u)
    call assert (u, grid%is_non_zero_everywhere (), &
           "grid%is_non_zero_everywhere (")
    call assert_equal (u, grid%get_maximum_in_3d (1), 1.8_default, &
         "grid%get_maximum_in_3d (1)")
    call assert_equal (u, grid%get_maximum_in_3d (2), 1.7_default, &
         "grid%get_maximum_in_3d (2)")

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: grids_5"
  end subroutine grids_5
```

One could think of multiple implementations of a generic type.

⟨solver.f90⟩≡

```
⟨File header⟩

module solver

⟨Use kinds⟩
  use constants, only: tiny_10
  use numeric_utils
  use diagnostics

⟨Standard module head⟩

⟨solver: public⟩

⟨solver: parameters⟩

⟨solver: types⟩

⟨solver: interfaces⟩

contains

⟨solver: procedures⟩
```

```
  end module solver
```

⟨*solver: public*⟩≡

```
    public :: solver_function_t
```

⟨*solver: types*⟩≡

```
    type, abstract :: solver_function_t
    contains
      procedure(solver_function_evaluate), deferred :: evaluate
    end type solver_function_t
```

⟨*solver: interfaces*⟩≡

```
    abstract interface
       function solver_function_evaluate (solver_f, x) result (f)
         import
         complex(default) :: f
         class(solver_function_t), intent(in) :: solver_f
         real(default), intent(in) :: x
       end function
    end interface
```

⟨*solver: public*⟩+≡

```
    public :: solve_secant
```

⟨*solver: procedures*⟩≡

```
    function solve_secant (func, lower_start, upper_start, success, precision) result (x0)
      class(solver_function_t), intent(in) :: func
      real(default) :: x0
      real(default), intent(in) :: lower_start, upper_start
      real(default), intent(in), optional :: precision
      logical, intent(out) :: success
      real(default) :: desired, x_curr, x_next, f_curr, f_next, x_new
      integer :: n_iter
      desired = DEFAULT_PRECISION; if (present(precision)) desired = precision
      x_curr = lower_start
      x_next = upper_start
      n_iter = 0
      success = .false.
      SEARCH: do
         n_iter = n_iter + 1
         f_curr = real( func%evaluate (x_curr) )
         f_next = real( func%evaluate (x_next) )
         ⟨Exit if close to zero and handle exceptions⟩
         x_new = x_next - (x_next - x_curr) / (f_next - f_curr) * f_next
         x_curr = x_next
         x_next = x_new
      end do SEARCH
      if (x0 < lower_start .or. x0 > upper_start) then
         call msg_warning ("solve: The root of the function is not in boundaries")
         return
      end if
      success = .true.
    end function solve_secant
```

⟨*Exit if close to zero and handle exceptions*⟩≡

```
if (abs (f_next) < desired) then
   x0 = x_next
   exit
end if
if (n_iter > MAX_TRIES) then
   call msg_warning ("solve: Couldn't find root of function")
   return
end if
if (vanishes (f_next - f_curr)) then
   x_next = x_next + (x_next - x_curr) / 10
   cycle
end if
```

⟨*solver: public*⟩+≡

```
public :: solve_interval
```

⟨*solver: procedures*⟩+≡

```
function solve_interval (func, lower_start, upper_start, success, precision) &
                                                          result (x0)
   class(solver_function_t), intent(in) :: func
   real(default) :: x0
   real(default), intent(in) :: lower_start, upper_start
   real(default), intent(in), optional :: precision
   logical, intent(out) :: success
   real(default) :: desired
   real(default) :: x_low, x_high, x_half
   real(default) :: f_low, f_high, f_half
   integer :: n_iter
   success = .false.
   desired = DEFAULT_PRECISION; if (present(precision)) desired = precision
   x0 = lower_start
   x_low = lower_start
   x_high = upper_start
   f_low = real( func%evaluate (x_low) )
   f_high = real( func%evaluate (x_high) )
   if (f_low * f_high > 0) return
   if (x_low > x_high) call msg_fatal ("Interval solver: Upper bound must be &
                                       &greater than lower bound")
   n_iter = 0
   do n_iter = 1, MAX_TRIES
      x_half = (x_high + x_low)/2
      f_half = real( func%evaluate (x_half) )
      if (abs (f_half) <= desired) then
         x0 = x_half
         exit
      end if
      if (f_low * f_half > 0._default) then
         x_low = x_half
         f_low = f_half
      else
         x_high = x_half
         f_high = f_half
      end if
   end do
```

```
    if (x0 < lower_start .or. x0 > upper_start) then
       call msg_warning ("Interval solver: The root of the function&
                         & is out of boundaries")
       return
    end if
    success = .true.
  contains
    subroutine display_solver_status ()
       print *, '================='
       print *, 'Status of interval solver: '
       print *, 'initial values: ', lower_start, upper_start
       print *, 'iteration: ', n_iter
       print *, 'x_low: ', x_low, 'f_low: ', f_low
       print *, 'x_high: ', x_high, 'f_high: ', f_high
       print *, 'x_half: ', x_half, 'f_half: ', f_half
    end subroutine display_solver_status
  end function solve_interval
```

⟨*solver: public*⟩+≡

```
  public :: solve_qgaus
```

⟨*solver: procedures*⟩+≡

```
  function solve_qgaus (integrand, grid) result (integral)
    class(solver_function_t), intent(in) :: integrand
    complex(default) :: integral
    real(default), dimension(:), intent(in) :: grid
    integer :: i, j
    real(default) :: xm, xr
    real(default), dimension(5) :: dx, &
      w = (/ 0.2955242247_default, 0.2692667193_default, &
        0.2190863625_default, 0.1494513491_default, 0.0666713443_default /), &
      x = (/ 0.1488743389_default, 0.4333953941_default, 0.6794095682_default, &
        0.8650633666_default, 0.9739065285_default /)
    integral = 0.0_default
    if ( size(grid) < 2 ) then
      call msg_warning ("solve_qgaus: size of integration grid smaller than 2.")
      return
    end if
    do i=1, size(grid)-1
      xm = 0.5_default * ( grid(i+1) + grid(i) )
      xr = 0.5_default * ( grid(i+1) - grid(i) )
      do j=1, 5
        dx(j) = xr * x(j)
        integral = integral + xr * w(j) * &
          ( integrand%evaluate (xm+dx(j)) + integrand%evaluate (xm-dx(j)) )
      end do
    end do
  end function solve_qgaus
```

⟨*solver: parameters*⟩≡

```
  real(default), parameter, public :: DEFAULT_PRECISION = tiny_10
```

⟨*solver: parameters*⟩+≡

```
  integer, parameter :: MAX_TRIES = 10000
```

6.6.6 Unit tests

Test module, followed by the corresponding implementation module.

⟨solver_ut.f90⟩≡
```
⟨File header⟩

module solver_ut
  use unit_tests
  use solver_uti

⟨Standard module head⟩

⟨solver: public test⟩

contains

⟨solver: test driver⟩

end module solver_ut
```

⟨solver_uti.f90⟩≡
```
⟨File header⟩

module solver_uti

⟨Use kinds⟩
  use constants, only: zero, one, two
  use numeric_utils

  use solver

⟨Standard module head⟩

⟨solver: test declarations⟩

⟨solver: test types⟩

contains

⟨solver: tests⟩

⟨solver: test auxiliary⟩

end module solver_uti
```

API: driver for the unit tests below.

⟨*solver: public test*⟩≡
```
  public :: solver_test
```

⟨*solver: test driver*⟩≡
```
  subroutine solver_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨solver: execute tests⟩
  end subroutine solver_test
```

Test functions

⟨*solver: test types*⟩≡
```
type, extends (solver_function_t) :: test_function_1_t
contains
  procedure :: evaluate => test_func_1
end type test_function_1_t
```

⟨*solver: test types*⟩+≡
```
type, extends (solver_function_t) :: test_function_2_t
contains
  procedure :: evaluate => test_func_2
end type test_function_2_t
```

⟨*solver: test types*⟩+≡
```
type, extends (solver_function_t) :: test_function_3_t
contains
  procedure :: evaluate => test_func_3
end type test_function_3_t
```

⟨*solver: test types*⟩+≡
```
type, extends (solver_function_t) :: test_function_4_t
contains
  procedure :: evaluate => test_func_4
end type test_function_4_t
```

⟨*solver: test auxiliary*⟩≡
```
function test_func_1 (solver_f, x) result (f)
  complex(default) :: f
  class(test_function_1_t), intent(in) :: solver_f
  real(default), intent(in) :: x
  f = x
end function test_func_1

function test_func_2 (solver_f, x) result (f)
  complex(default) :: f
  class(test_function_2_t), intent(in) :: solver_f
  real(default), intent(in) :: x
  f = x ** 2
end function test_func_2

function test_func_3 (solver_f, x) result (f)
  complex(default) :: f
  class(test_function_3_t), intent(in) :: solver_f
  real(default), intent(in) :: x
  f = x ** 3
end function test_func_3

function test_func_4 (solver_f, x) result (f)
  complex(default) :: f
  class(test_function_4_t), intent(in) :: solver_f
  real(default), intent(in) :: x
```

```
    real(default) :: s, cutoff
    s = 100.0_default
    cutoff = 1.01_default
    if (x < cutoff) then
       f = - (log (s) * log (log (s) / log(cutoff**2)) - log (s / cutoff**2)) - &
         log (one/two)
    else
       f = - (log (s) * log (log (s) / log(x**2)) - log (s / x**2)) - &
            log (one/two)
    end if
  end function test_func_4
```

Solve trivial functions

⟨*solver: execute tests*⟩≡

```
  call test(solver_1, "solver_1", &
            "Solve trivial functions", u, results)
```

⟨*solver: test declarations*⟩≡

```
  public :: solver_1
```

⟨*solver: tests*⟩≡

```
  subroutine solver_1 (u)
    integer, intent(in) :: u
    real(default) :: zero_position
    logical :: success
    type(test_function_1_t) :: test_func_1
    type(test_function_2_t) :: test_func_2
    type(test_function_3_t) :: test_func_3
    type(test_function_4_t) :: test_func_4
    write (u, "(A)")  "* Test output: solver_1"
    write (u, "(A)")  "*   Purpose: Solve trivial functions"
    write (u, "(A)")

    zero_position = solve_interval (test_func_1, -one, one, success)
    call assert (u, success, "success")
    call assert_equal (u, zero_position, zero, "test_func_1: zero_position")

    zero_position = solve_interval (test_func_4, two, 10.0_default, success)
    call assert (u, success, "success")
    call assert_equal (u, zero_position, &
         3.521667401186594028339722 4_default, &
         "test_func_4: zero_position", rel_smallness=1000*DEFAULT_PRECISION)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: solver_1"
  end subroutine solver_1
```

Chapter 7

Text handling

`WHIZARD` has to handle complex structures in input (and output) data. Doing this in a generic and transparent way requires a generic lexer and parser. The necessary modules are implemented here:

ifiles Implementation of line-oriented internal files in a more flexible way (linked lists of variable-length strings) than the Fortran builtin features.

lexers Read text and transform it into a token stream.

syntax_rules Define the rules for interpreting tokens, to be used by the WHIZARD parser.

parser Categorize tokens (keyword, string, number etc.) and use a set of syntax rules to transform the input into a parse tree.

xml Read and parse XML text, separate from the WHIZARD parser.

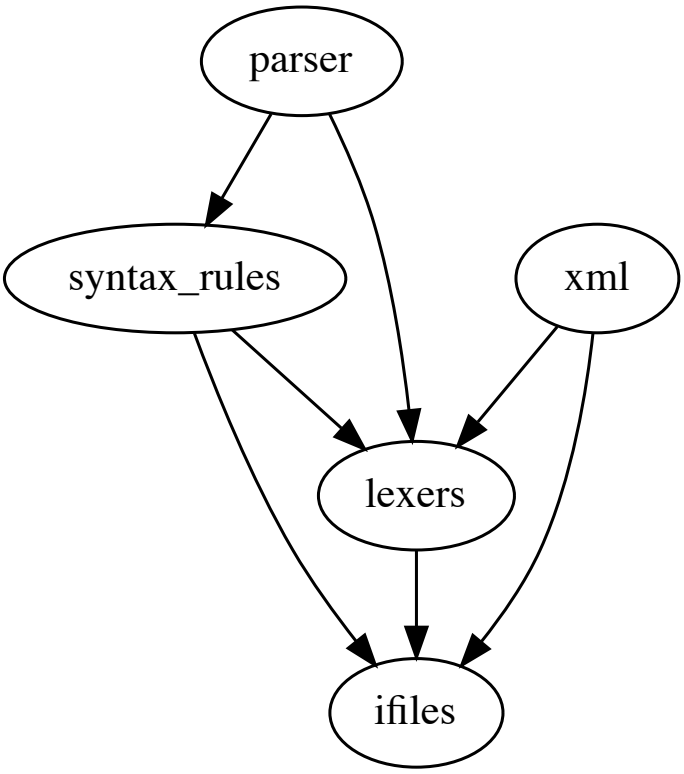

Figure 7.1: Module dependencies in `src/parsing`.

7.1 Internal files

The internal files introduced here (`ifile`) are a replacement for the built-in internal files, which are fixed-size arrays of fixed-length character strings. The `ifile` type is a doubly-linked list of variable-length character strings with line numbers.

⟨`ifiles.f90`⟩≡

```
⟨File header⟩

module ifiles

⟨Use strings⟩
  use io_units
  use system_defs, only: EOF

⟨Standard module head⟩

⟨Ifiles: public⟩

⟨Ifiles: types⟩

⟨Ifiles: interfaces⟩

contains

⟨Ifiles: subroutines⟩

end module ifiles
```

7.1.1 The line type

The line entry type is for internal use, it is the list entry to be collected in an `ifile` object.

⟨*Ifiles: types*⟩≡

```
type :: line_entry_t
   private
   type(line_entry_t), pointer :: previous => null ()
   type(line_entry_t), pointer :: next => null ()
   type(string_t) :: string
   integer :: index
end type line_entry_t
```

Create a new list entry, given a varying string as input. The line number and pointers are not set, these make sense only within an `ifile`.

⟨*Ifiles: subroutines*⟩≡

```
subroutine line_entry_create (line, string)
  type(line_entry_t), pointer :: line
  type(string_t), intent(in) :: string
  allocate (line)
  line%string = string
end subroutine line_entry_create
```

Destroy a single list entry: Since the pointer components should not be deallocated explicitly, just deallocate the object itself.

⟨*Ifiles: subroutines*⟩+≡

```
subroutine line_entry_destroy (line)
  type(line_entry_t), pointer :: line
  deallocate (line)
end subroutine line_entry_destroy
```

7.1.2 The ifile type

The internal file is a linked list of line entries.

⟨*Ifiles: public*⟩≡

```
public :: ifile_t
```

⟨*Ifiles: types*⟩+≡

```
type :: ifile_t
   private
   type(line_entry_t), pointer :: first => null ()
   type(line_entry_t), pointer :: last => null ()
   integer :: n_lines = 0
 contains
 ⟨Ifiles: ifile: TBP⟩
end type ifile_t
```

We need no explicit initializer, but a routine which recursively deallocates the contents may be appropriate. After this, existing line pointers may become undefined, so they should be nullified before the file is destroyed.

⟨*Ifiles: public*⟩+≡

```
public :: ifile_clear
```

⟨*Ifiles: subroutines*⟩+≡

```
subroutine ifile_clear (ifile)
  class(ifile_t), intent(inout) :: ifile
  type(line_entry_t), pointer :: current
  do while (associated (ifile%first))
     current => ifile%first
     ifile%first => current%next
     call line_entry_destroy (current)
  end do
  nullify (ifile%last)
  ifile%n_lines = 0
end subroutine ifile_clear
```

The finalizer is just an alias for the above.

⟨*Ifiles: public*⟩+≡

```
public :: ifile_final
```

⟨*Ifiles: ifile: TBP*⟩≡

```
procedure :: final => ifile_clear
```

⟨*Ifiles: interfaces*⟩≡

```
interface ifile_final
   module procedure ifile_clear
end interface
```

7.1.3 I/O on ifiles

Fill an ifile from an ordinary external file, i.e., I/O unit. If the ifile is not empty, the old contents will be destroyed. We can read a fixed-length character string, an ISO varying string, an ordinary internal file (character-string array), or from an external unit. In the latter case, lines are appended until EOF is reached. Finally, there is a variant which reads from another ifile, effectively copying it.

⟨*Ifiles: public*⟩+≡
```
public :: ifile_read
```

⟨*Ifiles: interfaces*⟩+≡
```
interface ifile_read
   module procedure ifile_read_from_string
   module procedure ifile_read_from_char
   module procedure ifile_read_from_unit
   module procedure ifile_read_from_char_array
   module procedure ifile_read_from_ifile
end interface
```

⟨*Ifiles: subroutines*⟩+≡
```
subroutine ifile_read_from_string (ifile, string)
  type(ifile_t), intent(inout) :: ifile
  type(string_t), intent(in) :: string
  call ifile_clear (ifile)
  call ifile_append (ifile, string)
end subroutine ifile_read_from_string

subroutine ifile_read_from_char (ifile, char)
  type(ifile_t), intent(inout) :: ifile
  character(*), intent(in) :: char
  call ifile_clear (ifile)
  call ifile_append (ifile, char)
end subroutine ifile_read_from_char

subroutine ifile_read_from_char_array (ifile, char)
  type(ifile_t), intent(inout) :: ifile
  character(*), dimension(:), intent(in) :: char
  call ifile_clear (ifile)
  call ifile_append (ifile, char)
end subroutine ifile_read_from_char_array

subroutine ifile_read_from_unit (ifile, unit, iostat)
  type(ifile_t), intent(inout) :: ifile
  integer, intent(in) :: unit
  integer, intent(out), optional :: iostat
  call ifile_clear (ifile)
  call ifile_append (ifile, unit, iostat)
end subroutine ifile_read_from_unit

subroutine ifile_read_from_ifile (ifile, ifile_in)
  type(ifile_t), intent(inout) :: ifile
  type(ifile_t), intent(in) :: ifile_in
  call ifile_clear (ifile)
  call ifile_append (ifile, ifile_in)
end subroutine ifile_read_from_ifile
```

Append to an ifile. The same as reading, but without resetting the ifile. In addition, there is a routine for appending a whole ifile.

⟨*Ifiles: public*⟩+≡
```
public :: ifile_append
```

⟨*Ifiles: ifile: TBP*⟩+≡
```
generic :: append => &
     ifile_append_from_char
procedure, private :: ifile_append_from_char
```

⟨*Ifiles: interfaces*⟩+≡
```
interface ifile_append
   module procedure ifile_append_from_string
   module procedure ifile_append_from_char
   module procedure ifile_append_from_unit
   module procedure ifile_append_from_char_array
   module procedure ifile_append_from_ifile
end interface
```

⟨*Ifiles: subroutines*⟩+≡
```
subroutine ifile_append_from_string (ifile, string)
  class(ifile_t), intent(inout) :: ifile
  type(string_t), intent(in) :: string
  type(line_entry_t), pointer :: current
  call line_entry_create (current, string)
  current%index = ifile%n_lines + 1
  if (associated (ifile%last)) then
     current%previous => ifile%last
     ifile%last%next => current
  else
     ifile%first => current
  end if
  ifile%last => current
  ifile%n_lines = current%index
end subroutine ifile_append_from_string

subroutine ifile_append_from_char (ifile, char)
  class(ifile_t), intent(inout) :: ifile
  character(*), intent(in) :: char
  call ifile_append_from_string (ifile, var_str (trim (char)))
end subroutine ifile_append_from_char

subroutine ifile_append_from_char_array (ifile, char)
  class(ifile_t), intent(inout) :: ifile
  character(*), dimension(:), intent(in) :: char
  integer :: i
  do i = 1, size (char)
     call ifile_append_from_string (ifile, var_str (trim (char(i))))
  end do
end subroutine ifile_append_from_char_array

subroutine ifile_append_from_unit (ifile, unit, iostat)
  class(ifile_t), intent(inout) :: ifile
  integer, intent(in) :: unit
```

```
    integer, intent(out), optional :: iostat
    type(string_t) :: buffer
    integer :: ios
    ios = 0
    READ_LOOP: do
       call get (unit, buffer, iostat = ios)
       if (ios == EOF .or. ios > 0)  exit READ_LOOP
       call ifile_append_from_string (ifile, buffer)
    end do READ_LOOP
    if (present (iostat)) then
       iostat = ios
    else if (ios > 0) then
       call get (unit, buffer)  ! trigger error again
    end if
  end subroutine ifile_append_from_unit

  subroutine ifile_append_from_ifile (ifile, ifile_in)
    class(ifile_t), intent(inout) :: ifile
    type(ifile_t), intent(in) :: ifile_in
    type(line_entry_t), pointer :: current
    current => ifile_in%first
    do while (associated (current))
       call ifile_append_from_string (ifile, current%string)
       current => current%next
    end do
  end subroutine ifile_append_from_ifile
```

Write the ifile contents to an external unit

⟨*Ifiles: public*⟩+≡

```
  public :: ifile_write
```

⟨*Ifiles: subroutines*⟩+≡

```
  subroutine ifile_write (ifile, unit, iostat)
    type(ifile_t), intent(in) :: ifile
    integer, intent(in), optional :: unit
    integer, intent(out), optional :: iostat
    integer :: u
    type(line_entry_t), pointer :: current
    u = given_output_unit (unit);  if (u < 0)  return
    current => ifile%first
    do while (associated (current))
       call put_line (u, current%string, iostat)
       current => current%next
    end do
  end subroutine ifile_write
```

Convert the ifile to an array of strings, which is allocated by this function:

⟨*Ifiles: public*⟩+≡

```
  public :: ifile_to_string_array
```

⟨*Ifiles: subroutines*⟩+≡

```
  subroutine ifile_to_string_array (ifile, string)
    type(ifile_t), intent(in) :: ifile
    type(string_t), dimension(:), intent(inout), allocatable :: string
```

```
    type(line_entry_t), pointer :: current
    integer :: i
    allocate (string (ifile_get_length (ifile)))
    current => ifile%first
    do i = 1, ifile_get_length (ifile)
       string(i) = current%string
       current => current%next
    end do
  end subroutine ifile_to_string_array
```

7.1.4 Ifile tools

⟨*Ifiles: public*⟩+≡
```
  public :: ifile_get_length
```

⟨*Ifiles: subroutines*⟩+≡
```
  function ifile_get_length (ifile) result (length)
    integer :: length
    type(ifile_t), intent(in) :: ifile
    length = ifile%n_lines
  end function ifile_get_length
```

7.1.5 Line pointers

Instead of the implicit pointer used in ordinary file access, we define explicit pointers, so there can be more than one at a time.

⟨*Ifiles: public*⟩+≡
```
  public :: line_p
```

⟨*Ifiles: types*⟩+≡
```
  type :: line_p
     private
     type(line_entry_t), pointer :: p => null ()
  end type line_p
```

Assign a file pointer to the first or last line in an ifile:

⟨*Ifiles: public*⟩+≡
```
  public :: line_init
```

⟨*Ifiles: subroutines*⟩+≡
```
  subroutine line_init (line, ifile, back)
    type(line_p), intent(inout) :: line
    type(ifile_t), intent(in) :: ifile
    logical, intent(in), optional :: back
    if (present (back)) then
       if (back) then
          line%p => ifile%last
       else
          line%p => ifile%first
       end if
    else
       line%p => ifile%first
```

```
    end if
  end subroutine line_init
```

Remove the pointer association:

⟨*Ifiles: public*⟩+≡
```
  public :: line_final
```

⟨*Ifiles: subroutines*⟩+≡
```
  subroutine line_final (line)
    type(line_p), intent(inout) :: line
    nullify (line%p)
  end subroutine line_final
```

Go one step forward

⟨*Ifiles: public*⟩+≡
```
  public :: line_advance
```

⟨*Ifiles: subroutines*⟩+≡
```
  subroutine line_advance (line)
    type(line_p), intent(inout) :: line
    if (associated (line%p))  line%p => line%p%next
  end subroutine line_advance
```

Go one step backward

⟨*Ifiles: public*⟩+≡
```
  public :: line_backspace
```

⟨*Ifiles: subroutines*⟩+≡
```
  subroutine line_backspace (line)
    type(line_p), intent(inout) :: line
    if (associated (line%p))  line%p => line%p%previous
  end subroutine line_backspace
```

Check whether we are accessing a valid line

⟨*Ifiles: public*⟩+≡
```
  public :: line_is_associated
```

⟨*Ifiles: subroutines*⟩+≡
```
  function line_is_associated (line) result (ok)
    logical :: ok
    type(line_p), intent(in) :: line
    ok = associated (line%p)
  end function line_is_associated
```

7.1.6 Access lines via pointers

We do not need the ifile as an argument to these functions, because the `line` type will point to an existing ifile.

⟨*Ifiles: public*⟩+≡
```
  public :: line_get_string
```

⟨*Ifiles: subroutines*⟩+≡

```
function line_get_string (line) result (string)
  type(string_t) :: string
  type(line_p), intent(in) :: line
  if (associated (line%p)) then
     string = line%p%string
  else
     string = ""
  end if
end function line_get_string
```

Variant where the line pointer is advanced after reading.

⟨*Ifiles: public*⟩+≡

```
public :: line_get_string_advance
```

⟨*Ifiles: subroutines*⟩+≡

```
function line_get_string_advance (line) result (string)
  type(string_t) :: string
  type(line_p), intent(inout) :: line
  if (associated (line%p)) then
     string = line%p%string
     call line_advance (line)
  else
     string = ""
  end if
end function line_get_string_advance
```

⟨*Ifiles: public*⟩+≡

```
public :: line_get_index
```

⟨*Ifiles: subroutines*⟩+≡

```
function line_get_index (line) result (index)
  integer :: index
  type(line_p), intent(in) :: line
  if (associated (line%p)) then
     index = line%p%index
  else
     index = 0
  end if
end function line_get_index
```

⟨*Ifiles: public*⟩+≡

```
public :: line_get_length
```

⟨*Ifiles: subroutines*⟩+≡

```
function line_get_length (line) result (length)
  integer :: length
  type(line_p), intent(in) :: line
  if (associated (line%p)) then
     length = len (line%p%string)
  else
     length = 0
  end if
end function line_get_length
```

7.2 Lexer

The lexer purpose is to read from a line-separated character input stream (usually a file) and properly chop the stream into lexemes (tokens). [The parser will transform lexemes into meaningful tokens, to be stored in a parse tree, therefore we do not use the term 'token' here.] The input is read line-by-line, but interpreted free-form, except for quotes and the comment syntax. (Fortran 2003 would allow us to use a stream type for reading.)

In an object-oriented approach, we can dynamically create and destroy lexers, including the lexer setup.

The main lexer function is to return a lexeme according to the basic lexer rules (quotes, comments, whitespace, special classes). There is also a routine to write back a lexeme to the input stream (but only once).

For the rules, we separate the possible characters into classes. Whitespace usually consists of blank, tab, and line-feed, where any number of consecutive whitespace is equivalent to one. Quoted strings are enclosed by a pair of quote characters, possibly multiline. Comments are similar to quotes, but interpreted as whitespace. Numbers are identified (not distinguishing real and integer) but not interpreted. Other character classes make up identifiers.

⟨`lexers.f90`⟩≡

```
⟨File header⟩

module lexers

⟨Use strings⟩
  use io_units
  use string_utils
  use system_defs, only: EOF, EOR
  use system_defs, only: LF
  use system_defs, only: WHITESPACE_CHARS, LCLETTERS, UCLETTERS, DIGITS
  use diagnostics
  use ifiles, only: ifile_t
  use ifiles, only: line_p, line_is_associated, line_init, line_final
  use ifiles, only: line_get_string_advance

⟨Standard module head⟩

⟨Lexer: public⟩

⟨Lexer: parameters⟩

⟨Lexer: types⟩

⟨Lexer: interfaces⟩

contains

⟨Lexer: procedures⟩

end module lexers
```

7.2.1 Input streams

For flexible input, we define a generic stream type that refers to either an external file, an external unit which is already open, a string, an `ifile` object (internal file, i.e., string list), or a line pointer to an `ifile` object. The stream type actually follows the idea of a formatted external file, which is line-oriented. Thus, the stream reader always returns a whole record (input line).

Note that only in the string version, the stream contents are stored inside the stream object. In the `ifile` version, the stream contains only the line pointer, while in the external-file case, the line pointer is implicitly created by the runtime library.

⟨*Lexer: public*⟩≡
```
public :: stream_t
```

⟨*Lexer: types*⟩≡
```
type :: stream_t
   type(string_t), pointer :: filename => null ()
   integer, pointer :: unit => null ()
   type(string_t), pointer :: string => null ()
   type(ifile_t), pointer :: ifile => null ()
   type(line_p), pointer :: line => null ()
   integer :: record = 0
   logical :: eof = .false.
 contains
 ⟨Lexer: stream: TBP⟩
end type stream_t
```

The initializers refer to the specific version. The stream should be undefined before calling this.

⟨*Lexer: public*⟩+≡
```
public :: stream_init
```

⟨*Lexer: stream: TBP*⟩≡
```
generic :: init => &
     stream_init_filename, &
     stream_init_unit, &
     stream_init_string, &
     stream_init_ifile, &
     stream_init_line
procedure, private :: stream_init_filename
procedure, private :: stream_init_unit
procedure, private :: stream_init_string
procedure, private :: stream_init_ifile
procedure, private :: stream_init_line
```

⟨*Lexer: interfaces*⟩≡
```
interface stream_init
   module procedure stream_init_filename
   module procedure stream_init_unit
   module procedure stream_init_string
   module procedure stream_init_ifile
   module procedure stream_init_line
end interface
```

⟨*Lexer: procedures*⟩≡

```
subroutine stream_init_filename (stream, filename)
  class(stream_t), intent(out) :: stream
  character(*), intent(in) :: filename
  integer :: unit
  unit = free_unit ()
  open (unit=unit, file=filename, status="old", action="read")
  call stream_init_unit (stream, unit)
  allocate (stream%filename)
  stream%filename = filename
end subroutine stream_init_filename

subroutine stream_init_unit (stream, unit)
  class(stream_t), intent(out) :: stream
  integer, intent(in) :: unit
  allocate (stream%unit)
  stream%unit = unit
  stream%eof = .false.
end subroutine stream_init_unit

subroutine stream_init_string (stream, string)
  class(stream_t), intent(out) :: stream
  type(string_t), intent(in) :: string
  allocate (stream%string)
  stream%string = string
end subroutine stream_init_string

subroutine stream_init_ifile (stream, ifile)
  class(stream_t), intent(out) :: stream
  type(ifile_t), intent(in) :: ifile
  type(line_p) :: line
  call line_init (line, ifile)
  call stream_init_line (stream, line)
  allocate (stream%ifile)
  stream%ifile = ifile
end subroutine stream_init_ifile

subroutine stream_init_line (stream, line)
  class(stream_t), intent(out) :: stream
  type(line_p), intent(in) :: line
  allocate (stream%line)
  stream%line = line
end subroutine stream_init_line
```

The finalizer restores the initial state. If an external file was opened, it is closed.

⟨*Lexer: public*⟩+≡

```
public :: stream_final
```

⟨*Lexer: stream: TBP*⟩+≡

```
procedure :: final => stream_final
```

⟨*Lexer: procedures*⟩+≡

```
subroutine stream_final (stream)
  class(stream_t), intent(inout) :: stream
  if (associated (stream%filename)) then
```

```
      close (stream%unit)
      deallocate (stream%unit)
      deallocate (stream%filename)
   else if (associated (stream%unit)) then
      deallocate (stream%unit)
   else if (associated (stream%string)) then
      deallocate (stream%string)
   else if (associated (stream%ifile)) then
      call line_final (stream%line)
      deallocate (stream%line)
      deallocate (stream%ifile)
   else if (associated (stream%line)) then
      call line_final (stream%line)
      deallocate (stream%line)
   end if
end subroutine stream_final
```

This returns the next record from the input stream. Depending on the stream type, the stream pointers are modified: Reading from external unit, the external file is advanced (implicitly). Reading from string, the string is replaced by an empty string. Reading from `ifile`, the line pointer is advanced. Note that the `iostat` argument is mandatory.

⟨*Lexer: public*⟩+≡

```
public :: stream_get_record
```

⟨*Lexer: procedures*⟩+≡

```
subroutine stream_get_record (stream, string, iostat)
  type(stream_t), intent(inout) :: stream
  type(string_t), intent(out) :: string
  integer, intent(out) :: iostat
  if (associated (stream%unit)) then
     if (stream%eof) then
        iostat = EOF
     else
        call get (stream%unit, string, iostat=iostat)
        if (iostat == EOR) then
           iostat = 0
           stream%record = stream%record + 1
        end if
        if (iostat == EOF) then
           iostat = 0
           stream%eof = .true.
           if (len (string) /= 0) stream%record = stream%record + 1
        end if
     end if
  else if (associated (stream%string)) then
     if (len (stream%string) /= 0) then
        string = stream%string
        stream%string = ""
        iostat = 0
        stream%record = stream%record + 1
     else
        string = ""
        iostat = EOF
```

```
      end if
    else if (associated (stream%line)) then
      if (line_is_associated (stream%line)) then
        string = line_get_string_advance (stream%line)
        iostat = 0
        stream%record = stream%record + 1
      else
        string = ""
        iostat = EOF
      end if
    else
      call msg_bug (" Attempt to read from uninitialized input stream")
    end if
  end subroutine stream_get_record
```

Return the current stream source as a message string.

⟨*Lexer: public*⟩+≡

```
  public :: stream_get_source_info_string
```

⟨*Lexer: procedures*⟩+≡

```
  function stream_get_source_info_string (stream) result (string)
    type(string_t) :: string
    type(stream_t), intent(in) :: stream
    character(20) :: buffer
    if (associated (stream%filename)) then
      string = "File '" // stream%filename // "' (unit = "
      write (buffer, "(I0)")  stream%unit
      string = string // trim (buffer) // ")"
    else if (associated (stream%unit)) then
      write (buffer, "(I0)")  stream%unit
      string = "Unit " // trim (buffer)
    else if (associated (stream%string)) then
      string = "Input string"
    else if (associated (stream%ifile) .or. associated (stream%line)) then
      string = "Internal file"
    else
      string = ""
    end if
  end function stream_get_source_info_string
```

Return the index of the record just read as a message string.

⟨*Lexer: public*⟩+≡

```
  public :: stream_get_record_info_string
```

⟨*Lexer: procedures*⟩+≡

```
  function stream_get_record_info_string (stream) result (string)
    type(string_t) :: string
    type(stream_t), intent(in) :: stream
    character(20) :: buffer
    string = stream_get_source_info_string (stream)
    if (string /= "")  string = string // ", "
    write (buffer, "(I0)")  stream%record
    string = string // "line " // trim (buffer)
  end function stream_get_record_info_string
```

7.2.2 Keyword list

The lexer should be capable of identifying a token as a known keyword. To this end, we store a list of keywords:

⟨*Lexer: public*⟩+≡
```
public :: keyword_list_t
```

⟨*Lexer: types*⟩+≡
```
type :: keyword_entry_t
   private
   type(string_t) :: string
   type(keyword_entry_t), pointer :: next => null ()
end type keyword_entry_t

type :: keyword_list_t
   private
   type(keyword_entry_t), pointer :: first => null ()
   type(keyword_entry_t), pointer :: last => null ()
end type keyword_list_t
```

Add a new string to the keyword list, unless it is already there:

⟨*Lexer: public*⟩+≡
```
public :: keyword_list_add
```

⟨*Lexer: procedures*⟩+≡
```
subroutine keyword_list_add (keylist, string)
  type(keyword_list_t), intent(inout) :: keylist
  type(string_t), intent(in) :: string
  type(keyword_entry_t), pointer :: k_entry_new
  if (.not. keyword_list_contains (keylist, string)) then
     allocate (k_entry_new)
     k_entry_new%string = string
     if (associated (keylist%first)) then
        keylist%last%next => k_entry_new
     else
        keylist%first => k_entry_new
     end if
     keylist%last => k_entry_new
  end if
end subroutine keyword_list_add
```

Return true if a string is a keyword.

⟨*Lexer: public*⟩+≡
```
public :: keyword_list_contains
```

⟨*Lexer: procedures*⟩+≡
```
function keyword_list_contains (keylist, string) result (found)
  type(keyword_list_t), intent(in) :: keylist
  type(string_t), intent(in) :: string
  logical :: found
  found = .false.
```

```
    call check_rec (keylist%first)
  contains
    recursive subroutine check_rec (k_entry)
      type(keyword_entry_t), pointer :: k_entry
      if (associated (k_entry)) then
         if (k_entry%string /= string) then
            call check_rec (k_entry%next)
         else
            found = .true.
         end if
      end if
    end subroutine check_rec
  end function keyword_list_contains
```

Write the keyword list

⟨*Lexer: public*⟩+≡

```
  public :: keyword_list_write
```

⟨*Lexer: interfaces*⟩+≡

```
  interface keyword_list_write
     module procedure keyword_list_write_unit
  end interface
```

⟨*Lexer: procedures*⟩+≡

```
  subroutine keyword_list_write_unit (keylist, unit)
    type(keyword_list_t), intent(in) :: keylist
    integer, intent(in) :: unit
    write (unit, "(A)") "Keyword list:"
    if (associated (keylist%first)) then
       call keyword_write_rec (keylist%first)
       write (unit, *)
    else
       write (unit, "(1x,A)") "[empty]"
    end if
  contains
    recursive subroutine keyword_write_rec (k_entry)
      type(keyword_entry_t), intent(in), pointer :: k_entry
      if (associated (k_entry)) then
         write (unit, "(1x,A)", advance="no")  char (k_entry%string)
         call keyword_write_rec (k_entry%next)
      end if
    end subroutine keyword_write_rec
  end subroutine keyword_list_write_unit
```

Clear the keyword list

⟨*Lexer: public*⟩+≡

```
  public :: keyword_list_final
```

⟨*Lexer: procedures*⟩+≡

```
  subroutine keyword_list_final (keylist)
    type(keyword_list_t), intent(inout) :: keylist
    call keyword_destroy_rec (keylist%first)
    nullify (keylist%last)
  contains
```

```
    recursive subroutine keyword_destroy_rec (k_entry)
      type(keyword_entry_t), pointer :: k_entry
      if (associated (k_entry)) then
         call keyword_destroy_rec (k_entry%next)
         deallocate (k_entry)
      end if
    end subroutine keyword_destroy_rec
  end subroutine keyword_list_final
```

7.2.3 Lexeme templates

This type is handled like a rudimentary regular expression. It determines the lexer behavior when matching a string. The actual objects made from this type and the corresponding matching routines are listed below.

⟨*Lexer: types*⟩+≡

```
  type :: template_t
     private
     integer :: type
     character(256) :: charset1, charset2
     integer :: len1, len2
  end type template_t
```

These are the types that valid lexemes can have:

⟨*Lexer: public*⟩+≡

```
  public :: T_KEYWORD, T_IDENTIFIER, T_QUOTED, T_NUMERIC
```

⟨*Lexer: parameters*⟩≡

```
  integer, parameter :: T_KEYWORD = 1
  integer, parameter :: T_IDENTIFIER = 2, T_QUOTED = 3, T_NUMERIC = 4
```

These are special types:

⟨*Lexer: parameters*⟩+≡

```
  integer, parameter :: EMPTY = 0, WHITESPACE = 10
  integer, parameter :: NO_MATCH = 11, IO_ERROR = 12, OVERFLOW = 13
  integer, parameter :: UNMATCHED_QUOTE = 14
```

In addition, we have `EOF` which is a negative integer, normally -1. Printout for debugging:

⟨*Lexer: procedures*⟩+≡

```
  subroutine lexeme_type_write (type, unit)
    integer, intent(in) :: type
    integer, intent(in) :: unit
    select case (type)
    case (EMPTY);       write(unit,"(A)",advance="no") " EMPTY      "
    case (WHITESPACE);  write(unit,"(A)",advance="no") " WHITESPACE "
    case (T_IDENTIFIER);write(unit,"(A)",advance="no") " IDENTIFIER "
    case (T_QUOTED);    write(unit,"(A)",advance="no") " QUOTED     "
    case (T_NUMERIC);   write(unit,"(A)",advance="no") " NUMERIC    "
    case (IO_ERROR);    write(unit,"(A)",advance="no") " IO_ERROR   "
    case (OVERFLOW);    write(unit,"(A)",advance="no") " OVERFLOW   "
    case (UNMATCHED_QUOTE);    write(unit,"(A)",advance="no") " UNMATCHEDQ "
    case (NO_MATCH);    write(unit,"(A)",advance="no") " NO_MATCH   "
    case (EOF);         write(unit,"(A)",advance="no") " EOF        "
```

```
    case default;       write(unit,"(A)",advance="no") " [illegal]  "
    end select
  end subroutine lexeme_type_write

  subroutine template_write (tt, unit)
    type(template_t), intent(in) :: tt
    integer, intent(in) :: unit
    call lexeme_type_write (tt%type, unit)
    write (unit, "(A)", advance="no") "'" // tt%charset1(1:tt%len1) // "'"
    write (unit, "(A)", advance="no") " '" // tt%charset2(1:tt%len2) // "'"
  end subroutine template_write
```

The matching functions all return the number of matched characters in the provided string. If this number is zero, the match has failed.

The `template` functions are declared `pure` because they appear in `forall` loops below.

A template for whitespace:

⟨*Lexer: procedures*⟩+≡

```
  pure function template_whitespace (chars) result (tt)
    character(*), intent(in) :: chars
    type(template_t) :: tt
    tt = template_t (WHITESPACE, chars, "", len (chars), 0)
  end function template_whitespace
```

Just match the string against the character set.

⟨*Lexer: procedures*⟩+≡

```
  subroutine match_whitespace (tt, s, n)
    type(template_t), intent(in) :: tt
    character(*), intent(in) :: s
    integer, intent(out) :: n
    n = verify (s, tt%charset1(1:tt%len1)) - 1
    if (n < 0)  n = len (s)
  end subroutine match_whitespace
```

A template for normal identifiers. To match, a lexeme should have a first character in class `chars1` and an arbitrary number of further characters in class `chars2`. If the latter is empty, we are looking for a single-character lexeme.

⟨*Lexer: procedures*⟩+≡

```
  pure function template_identifier (chars1, chars2) result (tt)
    character(*), intent(in) :: chars1, chars2
    type(template_t) :: tt
    tt = template_t (T_IDENTIFIER, chars1, chars2, len(chars1), len(chars2))
  end function template_identifier
```

Here, the first letter must match, the others may or may not.

⟨*Lexer: procedures*⟩+≡

```
  subroutine match_identifier (tt, s, n)
    type(template_t), intent(in) :: tt
    character(*), intent(in) :: s
    integer, intent(out) :: n
    if (verify (s(1:1), tt%charset1(1:tt%len1)) == 0) then
```

```
      n = verify (s(2:), tt%charset2(1:tt%len2))
      if (n == 0)  n = len (s)
   else
      n = 0
   end if
end subroutine match_identifier
```

A template for quoted strings. The same template applies for comments. The first character set indicates the left quote (could be a sequence of several characters), the second one the matching right quote.

⟨*Lexer: procedures*⟩+≡

```
pure function template_quoted (chars1, chars2) result (tt)
  character(*), intent(in) :: chars1, chars2
  type(template_t) :: tt
  tt = template_t (T_QUOTED, chars1, chars2, len (chars1), len (chars2))
end function template_quoted
```

Here, the beginning of the string must exactly match the first character set, then we look for the second one. If found, return. If there is a first quote but no second one, return a negative number, indicating this error condition.

⟨*Lexer: procedures*⟩+≡

```
subroutine match_quoted (tt, s, n, range)
  type(template_t), intent(in) :: tt
  character(*), intent(in) :: s
  integer, intent(out) :: n
  integer, dimension(2), intent(out) :: range
  character(tt%len1) :: ch1
  character(tt%len2) :: ch2
  integer :: i
  ch1 = tt%charset1
  if (s(1:tt%len1) == ch1) then
     ch2 = tt%charset2
     do i = tt%len1 + 1, len (s) - tt%len2 + 1
        if (s(i:i+tt%len2-1) == ch2) then
           n = i + tt%len2 - 1
           range(1) = tt%len1 + 1
           range(2) = i - 1
           return
        end if
     end do
     n = -1
     range = 0
  else
     n = 0
     range = 0
  end if
end subroutine match_quoted
```

A template for real numbers. The first character set is the set of allowed exponent letters. In accordance with the other functions we return the lexeme as a string but do not read it.

⟨*Lexer: procedures*⟩+≡

```
pure function template_numeric (chars) result (tt)
  character(*), intent(in) :: chars
  type(template_t) :: tt
  tt = template_t (T_NUMERIC, chars, "", len (chars), 0)
end function template_numeric
```

A numeric lexeme may be real or integer. We purposely do not allow for a preceding sign. If the number is followed by an exponent, this is included, otherwise the rest is ignored.

There is a possible pitfall with this behavior: while the string `1e3` will be interpreted as a single number, the analogous string `1a3` will be split into the number `1` and an identifier `a3`. There is no easy way around such an ambiguity. We should make sure that the syntax does not contain identifiers like `a3` or `e3`.

⟨*Lexer: procedures*⟩+≡

```
subroutine match_numeric (tt, s, n)
  type(template_t), intent(in) :: tt
  character(*), intent(in) :: s
  integer, intent(out) :: n
  integer :: i, n0
  character(10), parameter :: digits = "0123456789"
  character(2), parameter :: signs = "-+"
  n = verify (s, digits) - 1
  if (n < 0) then
     n = 0
     return
  else if (s(n+1:n+1) == ".") then
     i = verify (s(n+2:), digits) - 1
     if (i < 0) then
        n = len (s)
        return
     else if (i > 0 .or. n > 0) then
        n = n + 1 + i
     end if
  end if
  n0 = n
  if (n > 0) then
     if (verify (s(n+1:n+1), tt%charset1(1:tt%len1)) == 0) then
        n = n + 1
        if (verify (s(n+1:n+1), signs) == 0)  n = n + 1
        i = verify (s(n+1:), digits) - 1
        if (i < 0) then
           n = len (s)
        else if (i == 0) then
           n = n0
        else
           n = n + i
        end if
     end if
  end if
end subroutine match_numeric
```

The generic matching routine. With Fortran 2003 we would define separate types and use a SELECT TYPE instead.

⟨*Lexer: procedures*⟩+≡

```
subroutine match_template (tt, s, n, range)
  type(template_t), intent(in) :: tt
  character(*), intent(in) :: s
  integer, intent(out) :: n
  integer, dimension(2), intent(out) :: range
  select case (tt%type)
  case (WHITESPACE)
     call match_whitespace (tt, s, n)
     range = 0
  case (T_IDENTIFIER)
     call match_identifier (tt, s, n)
     range(1) = 1
     range(2) = len_trim (s)
  case (T_QUOTED)
     call match_quoted (tt, s, n, range)
  case (T_NUMERIC)
     call match_numeric (tt, s, n)
     range(1) = 1
     range(2) = len_trim (s)
  case default
     call msg_bug ("Invalid lexeme template encountered")
  end select
end subroutine match_template
```

Match against an array of templates. Return the index of the first template that matches together with the number of characters matched and the range of the relevant substring. If all fails, these numbers are zero.

⟨*Lexer: procedures*⟩+≡

```
subroutine match (tt, s, n, range, ii)
  type(template_t), dimension(:), intent(in) :: tt
  character(*), intent(in) :: s
  integer, intent(out) :: n
  integer, dimension(2), intent(out) :: range
  integer, intent(out) :: ii
  integer :: i
  do i = 1, size (tt)
     call match_template (tt(i), s, n, range)
     if (n /= 0) then
        ii = i
        return
     end if
  end do
  n = 0
  ii = 0
end subroutine match
```

7.2.4 The lexer setup

This object contains information about character classes. As said above, one class consists of quoting chars (matching left and right), another one of comment chars (similar), a class of whitespace, and several classes of characters

that make up identifiers. When creating the lexer setup, the character classes are transformed into lexeme templates which are to be matched in a certain predefined order against the input stream.

BLANK should always be taken as whitespace, some things may depend on this. TAB is also fixed by convention, but may in principle be modified. Newline (DOS!) and linefeed are also defined as whitespace. The lexer setup, containing the list of lexeme templates. No defaults yet. The type with index zero will be assigned to the `NO_MATCH` lexeme.

The keyword list is not stored, just a pointer to it. We anticipate that the keyword list is part of the syntax table, and the lexer needs not alter it. Furthermore, the lexer is typically finished before the syntax table is.

⟨*Lexer: parameters*⟩+≡

```
integer, parameter :: CASE_KEEP = 0, CASE_UP = 1, CASE_DOWN = 2
```

⟨*Lexer: types*⟩+≡

```
type :: lexer_setup_t
   private
   type(template_t), dimension(:), allocatable :: tt
   integer, dimension(:), allocatable :: type
   integer :: keyword_case = CASE_KEEP
   type(keyword_list_t), pointer :: keyword_list => null ()
end type lexer_setup_t
```

Fill the lexer setup object. Some things are hardcoded here (whitespace, alphanumeric identifiers), some are free: comment chars (but these must be single, and comments must be terminated by line-feed), quote chars and matches (must be single), characters to be read as one-character lexeme, special classes (characters of one class that should be glued together as identifiers).

⟨*Lexer: procedures*⟩+≡

```
subroutine lexer_setup_init (setup, &
     comment_chars, quote_chars, quote_match, &
     single_chars, special_class, &
     keyword_list, upper_case_keywords)
  type(lexer_setup_t), intent(inout) :: setup
  character(*), intent(in) :: comment_chars
  character(*), intent(in) :: quote_chars, quote_match
  character(*), intent(in) :: single_chars
  character(*), dimension(:), intent(in) :: special_class
  type(keyword_list_t), pointer :: keyword_list
  logical, intent(in), optional :: upper_case_keywords
  integer :: n, i
  if (present (upper_case_keywords)) then
     if (upper_case_keywords) then
        setup%keyword_case = CASE_UP
     else
        setup%keyword_case = CASE_DOWN
     end if
  else
     setup%keyword_case = CASE_KEEP
  end if
  n = 1 + len (comment_chars) + len (quote_chars) + 1 &
```

```
        + len (single_chars) + size (special_class) + 1
    allocate (setup%tt(n))
    allocate (setup%type(0:n))
    n = 0
    setup%type(n) = NO_MATCH
    n = n + 1
    setup%tt(n) = template_whitespace (WHITESPACE_CHARS)
    setup%type(n) = EMPTY
    forall (i = 1:len(comment_chars))
       setup%tt(n+i) = template_quoted (comment_chars(i:i), LF)
       setup%type(n+i) = EMPTY
    end forall
    n = n + len (comment_chars)
    forall (i = 1:len(quote_chars))
       setup%tt(n+i) = template_quoted (quote_chars(i:i), quote_match(i:i))
       setup%type(n+i) = T_QUOTED
    end forall
    n = n + len (quote_chars)
    setup%tt(n+1) = template_numeric ("EeDd")
    setup%type(n+1) = T_NUMERIC
    n = n + 1
    forall (i = 1:len (single_chars))
       setup%tt(n+i) = template_identifier (single_chars(i:i), "")
       setup%type(n+i) = T_IDENTIFIER
    end forall
    n = n + len (single_chars)
    forall (i = 1:size (special_class))
       setup%tt(n+i) = template_identifier &
            (trim (special_class(i)), trim (special_class(i)))
       setup%type(n+i) = T_IDENTIFIER
    end forall
    n = n + size (special_class)
    setup%tt(n+1) = template_identifier &
         (LCLETTERS//UCLETTERS, LCLETTERS//DIGITS//"_"//UCLETTERS)
    setup%type(n+1) = T_IDENTIFIER
    n = n + 1
    if (n /= size (setup%tt)) &
         call msg_bug ("Size mismatch in lexer setup")
    setup%keyword_list => keyword_list
  end subroutine lexer_setup_init
```

The destructor is needed only if the object is not itself part of an allocatable array

⟨*Lexer: procedures*⟩+≡

```
  subroutine lexer_setup_final (setup)
    type(lexer_setup_t), intent(inout) :: setup
    deallocate (setup%tt, setup%type)
    setup%keyword_list => null ()
  end subroutine lexer_setup_final
```

For debugging: Write the lexer setup

⟨*Lexer: procedures*⟩+≡

```
  subroutine lexer_setup_write (setup, unit)
```

```
    type(lexer_setup_t), intent(in) :: setup
    integer, intent(in) :: unit
    integer :: i
    write (unit, "(A)") "Lexer setup:"
    if (allocated (setup%tt)) then
       do i = 1, size (setup%tt)
          call template_write (setup%tt(i), unit)
          write (unit, '(A)', advance = "no")  " -> "
          call lexeme_type_write (setup%type(i), unit)
          write (unit, *)
       end do
    else
       write (unit, *) "[empty]"
    end if
    if (associated (setup%keyword_list)) then
       call keyword_list_write (setup%keyword_list, unit)
    end if
  end subroutine lexer_setup_write
```

7.2.5 The lexeme type

An object of this type is returned by the lexer. Apart from the lexeme string, it gives information about the relevant substring (first and last character index) and the lexeme type. Interpreting the string is up to the parser.

⟨*Lexer: public*⟩+≡

```
    public :: lexeme_t
```

⟨*Lexer: types*⟩+≡

```
    type :: lexeme_t
       private
       integer :: type = EMPTY
       type(string_t) :: s
       integer :: b = 0, e = 0
    end type lexeme_t
```

Debugging aid:

⟨*Lexer: public*⟩+≡

```
    public :: lexeme_write
```

⟨*Lexer: procedures*⟩+≡

```
    subroutine lexeme_write (t, unit)
      type(lexeme_t), intent(in) :: t
      integer, intent(in) :: unit
      integer :: u
      u = given_output_unit (unit);  if (u < 0)  return
      select case (t%type)
      case (T_KEYWORD)
         write (u, *) "KEYWORD:    '" // char (t%s) // "'"
      case (T_IDENTIFIER)
         write (u, *) "IDENTIFIER: '" // char (t%s) // "'"
      case (T_QUOTED)
         write (u, *) "QUOTED:     '" // char (t%s) // "'"
      case (T_NUMERIC)
```

```
      write (u, *) "NUMERIC:    '" // char (t%s) // "'"
    case (UNMATCHED_QUOTE)
      write (u, *) "Unmatched quote: "// char (t%s)
    case (OVERFLOW); write (u, *) "Overflow: "// char (t%s)
    case (EMPTY);    write (u, *) "Empty lexeme"
    case (NO_MATCH); write (u, *) "No match"
    case (IO_ERROR); write (u, *) "IO error"
    case (EOF);      write (u, *) "EOF"
    case default
      write (u, *) "Error"
    end select
  end subroutine lexeme_write
```

Store string and type in a lexeme. The range determines the beginning and end of the relevant part of the string. Check for a keyword.

⟨*Lexer: procedures*⟩+≡

```
  subroutine lexeme_set (t, keyword_list, s, range, type, keyword_case)
    type(lexeme_t), intent(out) :: t
    type(keyword_list_t), pointer :: keyword_list
    type(string_t), intent(in) :: s
    type(string_t) :: keyword
    integer, dimension(2), intent(in) :: range
    integer, intent(in) :: type
    integer, intent(in), optional :: keyword_case
    t%type = type
    if (present (keyword_case)) then
      select case (keyword_case)
      case (CASE_KEEP);   keyword = s
      case (CASE_UP);     keyword = upper_case (s)
      case (CASE_DOWN);   keyword = lower_case (s)
      end select
    else
      keyword = s
    end if
    if (type == T_IDENTIFIER) then
      if (associated (keyword_list)) then
        if (keyword_list_contains (keyword_list, keyword)) &
            t%type = T_KEYWORD
      end if
    end if
    select case (t%type)
    case (T_KEYWORD);  t%s = keyword
    case default;      t%s = s
    end select
    t%b = range(1)
    t%e = range(2)
  end subroutine lexeme_set

  subroutine lexeme_clear (t)
    type(lexeme_t), intent(out) :: t
    t%type = EMPTY
    t%s = ""
  end subroutine lexeme_clear
```

Retrieve the lexeme string, the relevant part of it, and the type. The last function returns true if there is a break condition reached (error or EOF).

⟨*Lexer: public*⟩+≡

```
public :: lexeme_get_string
public :: lexeme_get_contents
public :: lexeme_get_delimiters
public :: lexeme_get_type
```

⟨*Lexer: procedures*⟩+≡

```
function lexeme_get_string (t) result (s)
  type(string_t) :: s
  type(lexeme_t), intent(in) :: t
  s = t%s
end function lexeme_get_string

function lexeme_get_contents (t) result (s)
  type(string_t) :: s
  type(lexeme_t), intent(in) :: t
  s = extract (t%s, t%b, t%e)
end function lexeme_get_contents

function lexeme_get_delimiters (t) result (del)
  type(string_t), dimension(2) :: del
  type(lexeme_t), intent(in) :: t
  del(1) = extract (t%s, finish = t%b-1)
  del(2) = extract (t%s, start = t%e+1)
end function lexeme_get_delimiters

function lexeme_get_type (t) result (type)
  integer :: type
  type(lexeme_t), intent(in) :: t
  type = t%type
end function lexeme_get_type
```

Check for a generic break condition (error/eof) and for eof in particular.

⟨*Lexer: public*⟩+≡

```
public :: lexeme_is_break
public :: lexeme_is_eof
```

⟨*Lexer: procedures*⟩+≡

```
function lexeme_is_break (t) result (break)
  logical :: break
  type(lexeme_t), intent(in) :: t
  select case (t%type)
  case (EOF, IO_ERROR, OVERFLOW, NO_MATCH)
     break = .true.
  case default
     break = .false.
  end select
end function lexeme_is_break

function lexeme_is_eof (t) result (ok)
  logical :: ok
```

```
    type(lexeme_t), intent(in) :: t
    ok = t%type == EOF
  end function lexeme_is_eof
```

7.2.6 The lexer object

We store the current lexeme and the current line. The line buffer is set each time a new line is read from file. The working buffer has one character more, to hold any trailing blank. Pointers to line and column are for debugging, they will be used to make up readable error messages for the parser.

⟨*Lexer: public*⟩+≡

```
  public :: lexer_t
```

⟨*Lexer: types*⟩+≡

```
  type :: lexer_t
     private
     type(lexer_setup_t) :: setup
     type(stream_t), pointer :: stream => null ()
     type(lexeme_t) :: lexeme
     type(string_t) :: previous_line2
     type(string_t) :: previous_line1
     type(string_t) :: current_line
     integer :: lines_read = 0
     integer :: current_column = 0
     integer :: previous_column = 0
     type(string_t) :: buffer
     type(lexer_t), pointer :: parent => null ()
   contains
   ⟨Lexer: lexer: TBP⟩
  end type lexer_t
```

Create-setup wrapper

⟨*Lexer: public*⟩+≡

```
  public :: lexer_init
```

⟨*Lexer: lexer: TBP*⟩≡

```
  procedure :: init => lexer_init
```

⟨*Lexer: procedures*⟩+≡

```
  subroutine lexer_init (lexer, &
       comment_chars, quote_chars, quote_match, &
       single_chars, special_class, &
       keyword_list, upper_case_keywords, &
       parent)
    class(lexer_t), intent(inout) :: lexer
    character(*), intent(in) :: comment_chars
    character(*), intent(in) :: quote_chars, quote_match
    character(*), intent(in) :: single_chars
    character(*), dimension(:), intent(in) :: special_class
    type(keyword_list_t), pointer :: keyword_list
    logical, intent(in), optional :: upper_case_keywords
    type(lexer_t), target, intent(in), optional :: parent
    call lexer_setup_init (lexer%setup, &
         comment_chars = comment_chars, &
```

```
          quote_chars = quote_chars, &
          quote_match = quote_match, &
          single_chars = single_chars, &
          special_class = special_class, &
          keyword_list = keyword_list, &
          upper_case_keywords = upper_case_keywords)
      if (present (parent))  lexer%parent => parent
      call lexer_clear (lexer)
    end subroutine lexer_init
```

Clear the lexer state, but not the setup. This should be done when the lexing starts, but it is not known whether the lexer was used before.

⟨*Lexer: public*⟩+≡

```
    public :: lexer_clear
```

⟨*Lexer: lexer: TBP*⟩+≡

```
    procedure :: clear => lexer_clear
```

⟨*Lexer: procedures*⟩+≡

```
    subroutine lexer_clear (lexer)
      class(lexer_t), intent(inout) :: lexer
      call lexeme_clear (lexer%lexeme)
      lexer%previous_line2 = ""
      lexer%previous_line1 = ""
      lexer%current_line = ""
      lexer%lines_read = 0
      lexer%current_column = 0
      lexer%previous_column = 0
      lexer%buffer = ""
    end subroutine lexer_clear
```

Reset lexer state and delete setup

⟨*Lexer: public*⟩+≡

```
    public :: lexer_final
```

⟨*Lexer: lexer: TBP*⟩+≡

```
    procedure :: final => lexer_final
```

⟨*Lexer: procedures*⟩+≡

```
    subroutine lexer_final (lexer)
      class(lexer_t), intent(inout) :: lexer
      call lexer%clear ()
      call lexer_setup_final (lexer%setup)
    end subroutine lexer_final
```

7.2.7 The lexer routine

For lexing we need to associate an input stream to the lexer.

⟨*Lexer: public*⟩+≡

```
    public :: lexer_assign_stream
```

⟨*Lexer: lexer: TBP*⟩+≡

```
    procedure :: assign_stream => lexer_assign_stream
```

⟨*Lexer: procedures*⟩+≡

```
subroutine lexer_assign_stream (lexer, stream)
  class(lexer_t), intent(inout) :: lexer
  type(stream_t), intent(in), target :: stream
  lexer%stream => stream
end subroutine lexer_assign_stream
```

The lexer. The lexer function takes the lexer and returns the currently stored lexeme. If there is none, it is read from buffer, matching against the lexeme templates in the lexer setup. Empty lexemes, i.e., comments and whitespace, are discarded and the buffer is read again until we have found a nonempty lexeme (which may also be EOF or an error condition).

The initial state of the lexer contains an empty lexeme, so reading from buffer is forced. The empty state is restored after returning the lexeme. A nonempty lexeme is present in the lexer only if `lex_back` has been executed before.

The workspace is the `lexer%buffer`, treated as a sort of input stream. We chop off lexemes from the beginning, adjusting the buffer to the left. Whenever the buffer is empty, or we are matching against an open quote which has not terminated, we read a new line and append it to the right. This may result in special conditions, which for simplicity are also returned as lexemes: I/O error, buffer overflow, end of file. If the latter happens during reading a quoted string, we return an unmatched-quote lexeme. Obviously, the special-condition lexemes have to be caught by the parser.

Note that reading further lines is only necessary when reading a quoted string. Otherwise, the line-feed that ends each line is interpreted as whitespace which terminates a preceding lexeme, so there are no other valid multiline lexemes.

To enable meaningful error messages, we also keep track of the line number of the last line read, and the beginning and the end of the current lexeme with respect to this line.

The lexer is implemented as a function that returns the next lexeme (i.e., token). It uses the `lexer` setup and modifies the buffers and pointers stored within the lexer, a side effect. The lexer reads from an input stream object, which also is modified by this reading, e.g., a line pointer is advanced.

⟨*Lexer: public*⟩+≡

```
public :: lex
```

⟨*Lexer: procedures*⟩+≡

```
subroutine lex (lexeme, lexer)
  type(lexeme_t), intent(out) :: lexeme
  type(lexer_t), intent(inout) :: lexer
  integer :: iostat1, iostat2
  integer :: pos
  integer, dimension(2) :: range
  integer :: template_index, type
  if (.not. associated (lexer%stream)) &
      call msg_bug ("Lexer called without assigned stream")
  GET_LEXEME: do while (lexeme_get_type (lexer%lexeme) == EMPTY)
     if (len (lexer%buffer) /= 0) then
        iostat1 = 0
```

```
      else
         call lexer_read_line (lexer, iostat1)
      end if
      select case (iostat1)
      case (0)
         MATCH_BUFFER: do
            call match (lexer%setup%tt, char (lexer%buffer), &
                        pos, range, template_index)
            if (pos >= 0) then
               type = lexer%setup%type(template_index)
               exit MATCH_BUFFER
            else
               pos = 0
               call lexer_read_line (lexer, iostat2)
               select case (iostat2)
               case (EOF); type = UNMATCHED_QUOTE; exit MATCH_BUFFER
               case (1);   type = IO_ERROR;        exit MATCH_BUFFER
               case (2);   type = OVERFLOW;        exit MATCH_BUFFER
               end select
            end if
         end do MATCH_BUFFER
      case (EOF); type = EOF
      case (1);   type = IO_ERROR
      case (2);   type = OVERFLOW
      end select
      call lexeme_set (lexer%lexeme, lexer%setup%keyword_list, &
           extract (lexer%buffer, finish=pos), range, type, &
           lexer%setup%keyword_case)
      lexer%buffer = remove (lexer%buffer, finish=pos)
      lexer%previous_column = lexer%current_column
      lexer%current_column = lexer%current_column + pos
   end do GET_LEXEME
   lexeme = lexer%lexeme
   call lexeme_clear (lexer%lexeme)
end subroutine lex
```

Read a line and append it to the input buffer. If the input buffer overflows, return `iostat=2`. Otherwise, `iostat=1` indicates an I/O error, and `iostat=-1` the EOF.

The input stream may either be an external unit or a `ifile` object. In the latter case, a line is read and the line pointer is advanced.

Note that inserting `LF` between input lines is the Unix convention. Since we are doing this explicitly when gluing lines together, we can pattern-match against `LF` without having to worry about the system.

⟨*Lexer: procedures*⟩+≡

```
subroutine lexer_read_line (lexer, iostat)
  type(lexer_t), intent(inout) :: lexer
  integer, intent(out) :: iostat
  type(string_t) :: current_line
  current_line = lexer%current_line
  call stream_get_record (lexer%stream, lexer%current_line, iostat)
  if (iostat == 0) then
     lexer%lines_read = lexer%lines_read + 1
```

```
      lexer%previous_line2 = lexer%previous_line1
      lexer%previous_line1 = current_line
      lexer%buffer = lexer%buffer // lexer%current_line // LF
      lexer%previous_column = 0
      lexer%current_column = 0
    end if
  end subroutine lexer_read_line
```

Once in a while we have read one lexeme to many, which can be pushed back into the input stream. Do not do this more than once.

⟨*Lexer: public*⟩+≡

```
  public :: lexer_put_back
```

⟨*Lexer: procedures*⟩+≡

```
  subroutine lexer_put_back (lexer, lexeme)
    type(lexer_t), intent(inout) :: lexer
    type(lexeme_t), intent(in) :: lexeme
    if (lexeme_get_type (lexer%lexeme) == EMPTY) then
      lexer%lexeme = lexeme
    else
      call msg_bug (" Lexer: lex_back fails; probably called twice")
    end if
  end subroutine lexer_put_back
```

7.2.8 Diagnostics

For debugging: print just the setup

⟨*Lexer: public*⟩+≡

```
  public :: lexer_write_setup
```

⟨*Lexer: procedures*⟩+≡

```
  subroutine lexer_write_setup (lexer, unit)
    type(lexer_t), intent(in) :: lexer
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    call lexer_setup_write (lexer%setup, u)
  end subroutine lexer_write_setup
```

This is useful for error printing: show the current line with index and a pointer to the current column within the line.

⟨*Lexer: public*⟩+≡

```
  public :: lexer_show_location
```

⟨*Lexer: procedures*⟩+≡

```
  subroutine lexer_show_location (lexer)
    type(lexer_t), intent(in) :: lexer
    type(string_t) :: loc_str
    if (associated (lexer%parent)) then
      call lexer_show_source (lexer%parent)
      call msg_message ("[includes]")
    else
```

```
      call msg_message ()
    end if
    if (associated (lexer%stream)) then
      call msg_message &
           (char (stream_get_record_info_string (lexer%stream)) // ":")
    end if
    if (lexer%lines_read >= 4)  call msg_result ("[...]")
    if (lexer%lines_read >= 3)  call msg_result (char (lexer%previous_line2))
    if (lexer%lines_read >= 2)  call msg_result (char (lexer%previous_line1))
    if (lexer%lines_read >= 1) then
      call msg_result (char (lexer%current_line))
      loc_str = repeat (" ", lexer%previous_column)
      loc_str = loc_str // "^"
      if (lexer%current_column > lexer%previous_column) then
        loc_str = loc_str &
             // repeat ("-", max (lexer%current_column &
                                  - lexer%previous_column - 1, 0)) &
             // "^"
      end if
      call msg_result (char (loc_str))
    end if
  end subroutine lexer_show_location
```

This just prints the current stream source.

⟨*Lexer: procedures*⟩+≡

```
  recursive subroutine lexer_show_source (lexer)
    type(lexer_t), intent(in) :: lexer
    if (associated (lexer%parent)) then
      call lexer_show_source (lexer%parent)
      call msg_message ("[includes]")
    else
      call msg_message ()
    end if
    if (associated (lexer%stream)) then
      call msg_message &
           (char (stream_get_source_info_string (lexer%stream)) // ":")
    end if
  end subroutine lexer_show_source
```

7.2.9 Unit tests

Test module, followed by the corresponding implementation module.

⟨`lexers_ut.f90`⟩≡

```
  ⟨File header⟩

  module lexers_ut
    use unit_tests
    use lexers_uti

  ⟨Standard module head⟩

  ⟨Lexer: public test⟩
```

```
  contains

  ⟨Lexer: test driver⟩

  end module lexers_ut
```

⟨lexers_uti.f90⟩≡

```
  ⟨File header⟩

  module lexers_uti

  ⟨Use strings⟩

    use lexers

  ⟨Standard module head⟩

  ⟨Lexer: test declarations⟩

  contains

  ⟨Lexer: tests⟩

  end module lexers_uti
```

API: driver for the unit tests below.

⟨*Lexer: public test*⟩≡

```
    public :: lexer_test
```

⟨*Lexer: test driver*⟩≡

```
    subroutine lexer_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Lexer: execute tests⟩
    end subroutine lexer_test
```

Test the lexer by lexing and printing all lexemes from unit `u`, one per line, using preset conventions.

⟨*Lexer: execute tests*⟩≡

```
    call test (lexer_1, "lexer_1", &
         "check lexer", u, results)
```

⟨*Lexer: test declarations*⟩≡

```
    public :: lexer_1
```

⟨*Lexer: tests*⟩≡

```
    subroutine lexer_1 (u)
      integer, intent(in) :: u
      type(lexer_t), target :: lexer
      type(stream_t), target :: stream
      type(string_t) :: string
      type(lexeme_t) :: lexeme
      string = "abcdefghij"
      call lexer_init (lexer, &
         comment_chars = "", &
```

```
      quote_chars = "<'""", &
      quote_match = ">'""", &
      single_chars = "?*+|=,()", &
      special_class = [","], &
      keyword_list = null ())
   call stream_init (stream, string)
   call lexer_assign_stream (lexer, stream)
   do
      call lex (lexeme, lexer)
      call lexeme_write (lexeme, u)
      if (lexeme_is_break (lexeme))  exit
   end do
   call stream_final (stream)
   call lexer_final (lexer)
end subroutine lexer_1
```

7.3 Syntax rules

This module provides tools to handle syntax rules in an abstract way.

⟨syntax_rules.f90⟩≡

```
⟨File header⟩

module syntax_rules

⟨Use strings⟩
  use io_units
  use diagnostics
  use system_defs, only: LCLETTERS, UCLETTERS, DIGITS
  use ifiles, only: line_p, line_init, line_get_string_advance, line_final
  use ifiles, only: ifile_t, ifile_get_length
  use lexers

⟨Standard module head⟩

⟨Syntax: public⟩

⟨Syntax: parameters⟩

⟨Syntax: types⟩

⟨Syntax: interfaces⟩

contains

⟨Syntax: subroutines⟩

end module syntax_rules
```

7.3.1 Syntax rules

Syntax rules are used by the parser. They determine how to translate the stream of lexemes as returned by the lexer into the parse tree node. A rule may be terminal, i.e., replace a matching lexeme into a terminal node. The node will contain the lexeme interpreted as a recognized token:

- a keyword: unquoted fixed character string;
- a real number, to be determined at runtime;
- an integer, to be determined at runtime;
- a boolean value, to be determined at runtime;
- a quoted token (e.g., string), to be determined at runtime;
- an identifier (unquoted string that is not a recognized keyword), to be determined at runtime.

It may be nonterminal, i.e., contain a sequence of child rules. These are matched consecutively (and recursively) against the input stream; the resulting node will be a branch node.

- the file, i.e., the input stream as a whole;
- a sequence of syntax elements, where the last syntax element may be optional, or optional repetitive;

Sequences carry a flag that tells whether the last child is optional or may be repeated an arbitrary number of times, correponding to the regexp modifiers `?`, `*`, and `+`.

We also need an alternative rule; this will be replaced by the node generated by one of its children that matches; thus, it does not create a node of its own.

- an alternative of syntax elements.

We also define special types of sequences as convenience macros:

- a list: a sequence where the elements are separated by a separator keyword (e.g., commas), the separators are thrown away when parsing the list;
- a group: a sequence of three tokens, where the first and third ones are left and right delimiters, the delimiters are thrown away;
- an argument list: a delimited list, containing both delimiters and separators.

It would be great to have a polymorphic type for this purpose, but until Fortran 2003 is out we have to emulate this.

Here are the syntax element codes:

⟨*Syntax: public*⟩≡
```
public :: S_UNKNOWN
public :: S_LOGICAL, S_INTEGER, S_REAL, S_COMPLEX, S_QUOTED
public :: S_IDENTIFIER, S_KEYWORD
public :: S_SEQUENCE, S_LIST, S_GROUP, S_ARGS
public :: S_ALTERNATIVE
public :: S_IGNORE
```

⟨*Syntax: parameters*⟩≡
```
integer, parameter :: &
     S_UNKNOWN = 0, &
     S_LOGICAL = 1, S_INTEGER = 2, S_REAL = 3, S_COMPLEX = 4, &
     S_QUOTED = 5, S_IDENTIFIER = 6, S_KEYWORD = 7, &
     S_SEQUENCE = 8, S_LIST = 9, S_GROUP = 10, S_ARGS = 11, &
     S_ALTERNATIVE = 12, &
     S_IGNORE = 99
```

We need arrays of rule pointers, therefore this construct.

⟨*Syntax: types*⟩≡
```
type :: rule_p
   private
   type(syntax_rule_t), pointer :: p => null ()
end type rule_p
```

Return the association status of the rule pointer:

⟨*Syntax: subroutines*⟩≡

```
elemental function rule_is_associated (rp) result (ok)
  logical :: ok
  type (rule_p), intent(in) :: rp
  ok = associated (rp%p)
end function rule_is_associated
```

The rule type is one of the types listed above, represented by an integer code. The keyword, for a non-keyword rule, is an identifier used for the printed syntax table. The array of children is needed for nonterminal rules. In that case, there is a modifier for the last element (blank, "?", "*", or "+"), mirrored in the flags `opt` and `rep`. Then, we have the character constants used as separators and delimiters for this rule. Finally, the `used` flag can be set to indicate that this rule is the child of another rule.

Note: we separate `delimiter_left` and `delimiter_right` because the previous fixed dimension(2) array triggered an ICE in gfortran-4.9.2.

⟨*Syntax: types*⟩+≡

```
public :: syntax_rule_t
```

⟨*Syntax: types*⟩+≡

```
type :: syntax_rule_t
   private
   integer :: type = S_UNKNOWN
   logical :: used = .false.
   type(string_t) :: keyword
   type(string_t) :: separator
   type(string_t) :: delimiter_left, delimiter_right
   type(rule_p), dimension(:), allocatable :: child
   character(1) :: modifier = ""
   logical :: opt = .false., rep = .false.
 contains
 ⟨Syntax: syntax rule: TBP⟩
end type syntax_rule_t
```

Initializer: Set type and key for a rule, but do not (yet) allocate anything. Finalizer: not needed (no pointer components).

⟨*Syntax: subroutines*⟩+≡

```
subroutine syntax_rule_init (rule, key, type)
  type(syntax_rule_t), intent(inout) :: rule
  type(string_t), intent(in) :: key
  integer, intent(in) :: type
  rule%keyword = key
  rule%type = type
  select case (rule%type)
  case (S_GROUP)
     call syntax_rule_set_delimiter (rule)
  case (S_LIST)
     call syntax_rule_set_separator (rule)
  case (S_ARGS)
     call syntax_rule_set_delimiter (rule)
     call syntax_rule_set_separator (rule)
```

```
    end select
  end subroutine syntax_rule_init
```

These characters will not be enclosed in quotes when writing syntax rules:

⟨*Syntax: parameters*⟩+≡

```
  character(*), parameter :: &
          UNQUOTED = "(),|_"//LCLETTERS//UCLETTERS//DIGITS
```

7.3.2 I/O

Write an account of the rule. Setting `short` true will suppress the node type. Setting `key_only` true will suppress the definition. Setting `advance` false will suppress the trailing newline.

⟨*Syntax: public*⟩+≡

```
  public :: syntax_rule_write
```

⟨*Syntax: syntax rule: TBP*⟩≡

```
  procedure :: write => syntax_rule_write
```

⟨*Syntax: subroutines*⟩+≡

```
  subroutine syntax_rule_write (rule, unit, short, key_only, advance)
    class(syntax_rule_t), intent(in) :: rule
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: short, key_only, advance
    logical :: typ, def, adv
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    typ = .true.;  if (present (short))    typ = .not. short
    def = .true.;  if (present (key_only)) def = .not. key_only
    adv = .true.;  if (present (advance))  adv = advance
    select case (rule%type)
    case (S_UNKNOWN);    call write_atom ("???", typ)
    case (S_IGNORE);     call write_atom ("IGNORE", typ)
    case (S_LOGICAL);    call write_atom ("LOGICAL", typ)
    case (S_INTEGER);    call write_atom ("INTEGER", typ)
    case (S_REAL);       call write_atom ("REAL", typ)
    case (S_COMPLEX);    call write_atom ("COMPLEX", typ)
    case (S_IDENTIFIER); call write_atom ("IDENTIFIER", typ)
    case (S_KEYWORD);    call write_atom ("KEYWORD", typ)
    case (S_QUOTED)
       call write_quotes (typ, def, &
            del = [rule%delimiter_left, rule%delimiter_right])
    case (S_SEQUENCE)
       call write_sequence ("SEQUENCE", typ, def, size (rule%child))
    case (S_GROUP)
       call write_sequence ("GROUP", typ, def, size (rule%child), &
            del = [rule%delimiter_left, rule%delimiter_right])
    case (S_LIST)
       call write_sequence ("LIST", typ, def, size (rule%child), &
            sep = rule%separator)
    case (S_ARGS)
       call write_sequence ("ARGUMENTS", typ, def, size (rule%child), &
            del = [rule%delimiter_left, rule%delimiter_right], &
            sep = rule%separator)
```

```
  case (S_ALTERNATIVE)
     call write_sequence ("ALTERNATIVE", typ, def, size (rule%child), &
          sep = var_str ("|"))
  end select
  if (adv)  write (u, *)
contains
  subroutine write_type (type)
    character(*), intent(in) :: type
    character(11) :: str
    str = type
    write (u, "(1x,A)", advance="no")  str
  end subroutine write_type
  subroutine write_key
    write (u, "(1x,A)", advance="no")  char (wkey (rule))
  end subroutine write_key
  subroutine write_atom (type, typ)
    character(*), intent(in) :: type
    logical, intent(in) :: typ
    if (typ)  call write_type (type)
    call write_key
  end subroutine write_atom
  subroutine write_maybe_quoted (string)
    character(*), intent(in) :: string
    character, parameter :: q = "'"
    character, parameter :: qq = '"'
    if (verify (string, UNQUOTED) == 0) then
       write (u, "(1x,A)", advance = "no") trim (string)
    else if (verify (string, q) == 0) then
       write (u, "(1x,A)", advance = "no") qq // trim (string) // qq
    else
       write (u, "(1x,A)", advance = "no") q // trim (string) // q
    end if
  end subroutine write_maybe_quoted
  subroutine write_quotes (typ, def, del)
    logical, intent(in) :: typ, def
    type(string_t), dimension(2), intent(in) :: del
    if (typ)  call write_type ("QUOTED")
    call write_key
    if (def) then
       write (u, "(1x,'=')", advance="no")
       call write_maybe_quoted (char (del(1)))
       write (u, "(1x,A)", advance="no") "..."
       call write_maybe_quoted (char (del(2)))
    end if
  end subroutine write_quotes
  subroutine write_sequence (type, typ, def, n, del, sep)
    character(*), intent(in) :: type
    logical, intent(in) :: typ, def
    integer, intent(in) :: n
    type(string_t), dimension(2), intent(in), optional :: del
    type(string_t), intent(in), optional :: sep
    integer :: i
    if (typ)  call write_type (type)
    call write_key
```

```
      if (def) then
         write (u, "(1x,'=')", advance="no")
         if (present (del))  call write_maybe_quoted (char (del(1)))
         do i = 1, n
            if (i > 1 .and. present (sep)) &
                 call write_maybe_quoted (char (sep))
            write (u, "(1x,A)", advance="no")  &
                 char (wkey (syntax_rule_get_sub_ptr(rule, i)))
            if (i == n)  write (u, "(A)", advance="no")  trim (rule%modifier)
         end do
         if (present (del))  call write_maybe_quoted (char (del(2)))
      end if
    end subroutine write_sequence
  end subroutine syntax_rule_write
```

In the printed representation, the keyword strings are enclosed as `<...>`, unless they are bare keywords. Bare keywords are enclosed as `'..'` if they contain a character which is not a letter, digit, or underscore. If they contain a single-quote character, they are enclosed as `"..."`. (A keyword must not contain both single- and double-quotes.)

⟨*Syntax: subroutines*⟩+≡

```
  function wkey (rule) result (string)
    type(string_t) :: string
    type(syntax_rule_t), intent(in) :: rule
    select case (rule%type)
    case (S_KEYWORD)
       if (verify (rule%keyword, UNQUOTED) == 0) then
          string = rule%keyword
       else if (scan (rule%keyword, "'") == 0) then
          string = "'" // rule%keyword // "'"
       else
          string = '"' // rule%keyword // '"'
       end if
    case default
       string = "<" // rule%keyword // ">"
    end select
  end function wkey
```

7.3.3 Completing syntax rules

Set the separator and delimiter entries, using defaults:

⟨*Syntax: subroutines*⟩+≡

```
  subroutine syntax_rule_set_separator (rule, separator)
    type(syntax_rule_t), intent(inout) :: rule
    type(string_t), intent(in), optional :: separator
    if (present (separator)) then
       rule%separator = separator
    else
       rule%separator = ","
    end if
  end subroutine syntax_rule_set_separator
```

```
subroutine syntax_rule_set_delimiter (rule, delimiter)
  type(syntax_rule_t), intent(inout) :: rule
  type(string_t), dimension(2), intent(in), optional :: delimiter
  if (present (delimiter)) then
     rule%delimiter_left = delimiter(1)
     rule%delimiter_right = delimiter(2)
  else
     rule%delimiter_left = "("
     rule%delimiter_right = ")"
  end if
end subroutine syntax_rule_set_delimiter
```

Set the modifier entry and corresponding flags:

⟨*Syntax: subroutines*⟩+≡

```
function is_modifier (string) result (ok)
  logical :: ok
  type(string_t), intent(in) :: string
  select case (char (string))
  case (" ", "?", "*", "+");  ok = .true.
  case default;               ok = .false.
  end select
end function is_modifier

subroutine syntax_rule_set_modifier (rule, modifier)
  type(syntax_rule_t), intent(inout) :: rule
  type(string_t), intent(in) :: modifier
  rule%modifier = char (modifier)
  select case (rule%modifier)
  case (" ")
  case ("?");  rule%opt = .true.
  case ("*");  rule%opt = .true.;  rule%rep = .true.
  case ("+");  rule%rep = .true.
  case default
     call msg_bug (" Syntax: sequence modifier '" // rule%modifier &
          // "' is not one of '+' '*' '?'")
  end select
end subroutine syntax_rule_set_modifier
```

Check a finalized rule for completeness

⟨*Syntax: subroutines*⟩+≡

```
subroutine syntax_rule_check (rule)
  type(syntax_rule_t), intent(in) :: rule
  if (rule%keyword == "")  call msg_bug ("Rule key not set")
  select case (rule%type)
  case (S_UNKNOWN);  call bug (" Undefined rule")
  case (S_IGNORE, S_LOGICAL, S_INTEGER, S_REAL, S_COMPLEX, &
           S_IDENTIFIER, S_KEYWORD)
  case (S_QUOTED)
     if (rule%delimiter_left == "" .or. rule%delimiter_right == "") &
          call bug (" Missing quote character(s)")
  case (S_SEQUENCE)
  case (S_GROUP)
```

```
      if (rule%delimiter_left == "" .or. rule%delimiter_right == "") &
           call bug (" Missing delimiter(s)")
   case (S_LIST)
      if (rule%separator == "") call bug (" Missing separator")
   case (S_ARGS)
      if (rule%delimiter_left == "" .or. rule%delimiter_right == "") &
           call bug (" Missing delimiter(s)")
      if (rule%separator == "") call bug (" Missing separator")
   case (S_ALTERNATIVE)
   case default
      call bug (" Undefined syntax code")
   end select
   select case (rule%type)
   case (S_SEQUENCE, S_GROUP, S_LIST, S_ARGS, S_ALTERNATIVE)
      if (allocated (rule%child)) then
         if (.not.all (rule_is_associated (rule%child))) &
              call bug (" Child rules not all associated")
      else
         call bug (" Parent rule without children")
      end if
   case default
      if (allocated (rule%child))  call bug (" Non-parent rule with children")
   end select
contains
   subroutine bug (string)
     character(*), intent(in) :: string
     call msg_bug (" Syntax table: Rule " // char (rule%keyword) // ": " &
          // string)
   end subroutine bug
end subroutine syntax_rule_check
```

7.3.4 Accessing rules

This is the API for syntax rules:

⟨*Syntax: public*⟩+≡

```
public :: syntax_rule_get_type
```

⟨*Syntax: subroutines*⟩+≡

```
function syntax_rule_get_type (rule) result (type)
  integer :: type
  type(syntax_rule_t), intent(in) :: rule
  type = rule%type
end function syntax_rule_get_type
```

⟨*Syntax: public*⟩+≡

```
public :: syntax_rule_get_key
```

⟨*Syntax: syntax rule: TBP*⟩+≡

```
procedure :: get_key => syntax_rule_get_key
```

⟨*Syntax: subroutines*⟩+≡

```
function syntax_rule_get_key (rule) result (key)
  class(syntax_rule_t), intent(in) :: rule
  type(string_t) :: key
```

```
    key = rule%keyword
  end function syntax_rule_get_key
```

⟨*Syntax: public*⟩+≡

```
  public :: syntax_rule_get_separator
  public :: syntax_rule_get_delimiter
```

⟨*Syntax: subroutines*⟩+≡

```
  function syntax_rule_get_separator (rule) result (separator)
    type(string_t) :: separator
    type(syntax_rule_t), intent(in) :: rule
    separator = rule%separator
  end function syntax_rule_get_separator

  function syntax_rule_get_delimiter (rule) result (delimiter)
    type(string_t), dimension(2) :: delimiter
    type(syntax_rule_t), intent(in) :: rule
    delimiter = [rule%delimiter_left, rule%delimiter_right]
  end function syntax_rule_get_delimiter
```

Accessing child rules. If we use `syntax_rule_get_n_sub` for determining loop bounds, we do not need a check in the second routine.

⟨*Syntax: public*⟩+≡

```
  public :: syntax_rule_get_n_sub
  public :: syntax_rule_get_sub_ptr
```

⟨*Syntax: subroutines*⟩+≡

```
  function syntax_rule_get_n_sub (rule) result (n)
    integer :: n
    type(syntax_rule_t), intent(in) :: rule
    if (allocated (rule%child)) then
       n = size (rule%child)
    else
       n = 0
    end if
  end function syntax_rule_get_n_sub

  function syntax_rule_get_sub_ptr (rule, i) result (sub)
    type(syntax_rule_t), pointer :: sub
    type(syntax_rule_t), intent(in), target :: rule
    integer, intent(in) :: i
    sub => rule%child(i)%p
  end function syntax_rule_get_sub_ptr

  subroutine syntax_rule_set_sub (rule, i, sub)
    type(syntax_rule_t), intent(inout) :: rule
    integer, intent(in) :: i
    type(syntax_rule_t), intent(in), target :: sub
    rule%child(i)%p => sub
  end subroutine syntax_rule_set_sub
```

Return the modifier flags:

⟨*Syntax: public*⟩+≡

```
public :: syntax_rule_last_optional
public :: syntax_rule_last_repetitive
```

⟨*Syntax: subroutines*⟩+≡

```
function syntax_rule_last_optional (rule) result (opt)
  logical :: opt
  type(syntax_rule_t), intent(in) :: rule
  opt = rule%opt
end function syntax_rule_last_optional
function syntax_rule_last_repetitive (rule) result (rep)
  logical :: rep
  type(syntax_rule_t), intent(in) :: rule
  rep = rule%rep
end function syntax_rule_last_repetitive
```

Return true if the rule is atomic, i.e., logical, real, keyword etc.

⟨*Syntax: public*⟩+≡

```
public :: syntax_rule_is_atomic
```

⟨*Syntax: subroutines*⟩+≡

```
function syntax_rule_is_atomic (rule) result (atomic)
  logical :: atomic
  type(syntax_rule_t), intent(in) :: rule
  select case (rule%type)
  case (S_LOGICAL, S_INTEGER, S_REAL, S_COMPLEX, S_IDENTIFIER, &
           S_KEYWORD, S_QUOTED)
     atomic = .true.
  case default
     atomic = .false.
  end select
end function syntax_rule_is_atomic
```

7.3.5 Syntax tables

A syntax table contains the tree of syntax rules and, for direct parser access, the list of valid keywords.

Types

The syntax contains an array of rules and a list of keywords. The array is actually used as a tree, where the top rule is the first array element, and the other rules are recursively pointed to by this first rule. (No rule should be used twice or be unused.) The keyword list is derived from the rule tree.

Objects of this type need the target attribute if they are associated with a lexer. The keyword list will be pointed to by this lexer.

⟨*Syntax: public*⟩+≡

```
public :: syntax_t
```

⟨*Syntax: types*⟩+≡

```
type :: syntax_t
   private
   type(syntax_rule_t), dimension(:), allocatable :: rule
   type(keyword_list_t) :: keyword_list
```

```
end type syntax_t
```

Constructor/destructor

Initialize and finalize syntax tables

⟨*Syntax: public*⟩+≡

```
public :: syntax_init
public :: syntax_final
```

There are two ways to create a syntax: hard-coded from rules or dynamically from file.

⟨*Syntax: interfaces*⟩≡

```
interface syntax_init
   module procedure syntax_init_from_ifile
end interface
```

The syntax definition is read from an `ifile` object which contains the syntax definitions in textual form, one rule per line. This interface allows for determining the number of rules beforehand.

To parse the rule definitions, we make up a temporary lexer. Obviously, we cannot use a generic parser yet, so we have to hardcode the parsing process.

⟨*Syntax: subroutines*⟩+≡

```
subroutine syntax_init_from_ifile (syntax, ifile)
  type(syntax_t), intent(out), target :: syntax
  type(ifile_t), intent(in) :: ifile
  type(lexer_t) :: lexer
  type(line_p) :: line
  type(string_t) :: string
  integer :: n_token
  integer :: i
  call lexer_init (lexer, &
     comment_chars = "", &
     quote_chars = "<'""", &
     quote_match = ">'""", &
     single_chars = "?*+|=,()", &
     special_class = ["."], &
     keyword_list = null ())
  allocate (syntax%rule (ifile_get_length (ifile)))
  call line_init (line, ifile)
  do i = 1, size (syntax%rule)
     string = line_get_string_advance (line)
     call set_rule_type_and_key (syntax%rule(i), string, lexer)
  end do
  call line_init (line, ifile)
  do i = 1, size (syntax%rule)
     string = line_get_string_advance (line)
     select case (syntax%rule(i)%type)
     case (S_QUOTED, S_SEQUENCE, S_GROUP, S_LIST, S_ARGS, S_ALTERNATIVE)
        n_token = get_n_token (string, lexer)
        call set_rule_contents &
             (syntax%rule(i), syntax, n_token, string, lexer)
     end select
```

```
    end do
    call line_final (line)
    call lexer_final (lexer)
    call syntax_make_keyword_list (syntax)
    if (.not. all (syntax%rule%used)) then
       do i = 1, size (syntax%rule)
          if (.not. syntax%rule(i)%used) then
             call syntax_rule_write (syntax%rule(i), 6)
          end if
       end do
       call msg_bug (" Syntax table: unused rules")
    end if
  end subroutine syntax_init_from_ifile
```

For a given rule defined in the input, the first task is to determine its type and key. With these, we can initialize the rule in the table, postponing the association of children.

⟨*Syntax: subroutines*⟩+≡

```
  subroutine set_rule_type_and_key (rule, string, lexer)
    type(syntax_rule_t), intent(inout) :: rule
    type(string_t), intent(in) :: string
    type(lexer_t), intent(inout) :: lexer
    type(stream_t), target :: stream
    type(lexeme_t) :: lexeme
    type(string_t) :: key
    character(2) :: type
    call lexer_clear (lexer)
    call stream_init (stream, string)
    call lexer_assign_stream (lexer, stream)
    call lex (lexeme, lexer)
    type = lexeme_get_string (lexeme)
    call lex (lexeme, lexer)
    key = lexeme_get_contents (lexeme)
    call stream_final (stream)
    if (trim (key) /= "") then
       select case (type)
       case ("IG");  call syntax_rule_init (rule, key, S_IGNORE)
       case ("LO");  call syntax_rule_init (rule, key, S_LOGICAL)
       case ("IN");  call syntax_rule_init (rule, key, S_INTEGER)
       case ("RE");  call syntax_rule_init (rule, key, S_REAL)
       case ("CO");  call syntax_rule_init (rule, key, S_COMPLEX)
       case ("ID");  call syntax_rule_init (rule, key, S_IDENTIFIER)
       case ("KE");  call syntax_rule_init (rule, key, S_KEYWORD)
       case ("QU");  call syntax_rule_init (rule, key, S_QUOTED)
       case ("SE");  call syntax_rule_init (rule, key, S_SEQUENCE)
       case ("GR");  call syntax_rule_init (rule, key, S_GROUP)
       case ("LI");  call syntax_rule_init (rule, key, S_LIST)
       case ("AR");  call syntax_rule_init (rule, key, S_ARGS)
       case ("AL");  call syntax_rule_init (rule, key, S_ALTERNATIVE)
       case default
          call lexer_show_location (lexer)
          call msg_bug (" Syntax definition: unknown type '" // type // "'")
       end select
```

```
    else
       print *, char (string)
       call msg_bug (" Syntax definition: empty rule key")
    end if
  end subroutine set_rule_type_and_key
```

This function returns the number of tokens in an input line.

⟨*Syntax: subroutines*⟩+≡

```
  function get_n_token (string, lexer) result (n)
    integer :: n
    type(string_t), intent(in) :: string
    type(lexer_t), intent(inout) :: lexer
    type(stream_t), target :: stream
    type(lexeme_t) :: lexeme
    integer :: i
    call lexer_clear (lexer)
    call stream_init (stream, string)
    call lexer_assign_stream (lexer, stream)
    i = 0
    do
       call lex (lexeme, lexer)
       if (lexeme_is_break (lexeme))  exit
       i = i + 1
    end do
    n = i
    call stream_final (stream)
end function get_n_token
```

This subroutine extracts the rule contents for an input line. There are three tasks: (1) determine the number of children, depending on the rule type; (2) find and set the separator and delimiter strings, if required; (3) scan the child rules, find them in the syntax table and associate the parent rule with them.

⟨*Syntax: subroutines*⟩+≡

```
  subroutine set_rule_contents (rule, syntax, n_token, string, lexer)
    type(syntax_rule_t), intent(inout) :: rule
    type(syntax_t), intent(in), target :: syntax
    integer, intent(in) :: n_token
    type(string_t), intent(in) :: string
    type(lexer_t), intent(inout) :: lexer
    type(stream_t), target :: stream
    type(lexeme_t), dimension(n_token) :: lexeme
    integer :: i, n_children
    call lexer_clear (lexer)
    call stream_init (stream, string)
    call lexer_assign_stream (lexer, stream)
    do i = 1, n_token
       call lex (lexeme(i), lexer)
    end do
    call stream_final (stream)
    n_children = get_n_children ()
    call set_delimiters
    if (n_children > 1)  call set_separator
    if (n_children > 0)  call set_children
```

```
contains
  function get_n_children () result (n_children)
    integer :: n_children
    select case (rule%type)
    case (S_QUOTED)
       if (n_token /= 6)  call broken_rule (rule)
       n_children = 0
    case (S_GROUP)
       if (n_token /= 6)  call broken_rule (rule)
       n_children = 1
    case (S_SEQUENCE)
       if (is_modifier (lexeme_get_string (lexeme(n_token)))) then
          if (n_token <= 4)  call broken_rule (rule)
          call syntax_rule_set_modifier &
               (rule, lexeme_get_string (lexeme(n_token)))
          n_children = n_token - 4
       else
          if (n_token <= 3) call broken_rule (rule)
          n_children = n_token - 3
       end if
    case (S_LIST)
       if (is_modifier (lexeme_get_string (lexeme(n_token)))) then
          if (n_token <= 4 .or. mod (n_token, 2) /= 1) &
               call broken_rule (rule)
          call syntax_rule_set_modifier &
               (rule, lexeme_get_string (lexeme(n_token)))
       else if (n_token <= 3 .or. mod (n_token, 2) /= 0) then
          call broken_rule (rule)
       end if
       n_children = (n_token - 2) / 2
    case (S_ARGS)
       if (is_modifier (lexeme_get_string (lexeme(n_token-1)))) then
          if (n_token <= 6 .or. mod (n_token, 2) /= 1) &
               call broken_rule (rule)
          call syntax_rule_set_modifier &
               (rule, lexeme_get_string (lexeme(n_token-1)))
       else if (n_token <= 5 .or. mod (n_token, 2) /= 0) then
          call broken_rule (rule)
       end if
       n_children = (n_token - 4) / 2
    case (S_ALTERNATIVE)
       if (n_token <= 3 .or. mod (n_token, 2) /= 0)  call broken_rule (rule)
       n_children = (n_token - 2) / 2
    end select
  end function get_n_children
  subroutine set_delimiters
    type(string_t), dimension(2) :: delimiter
    select case (rule%type)
    case (S_QUOTED, S_GROUP, S_ARGS)
       delimiter(1) = lexeme_get_contents (lexeme(4))
       delimiter(2) = lexeme_get_contents (lexeme(n_token))
       call syntax_rule_set_delimiter (rule, delimiter)
    end select
  end subroutine set_delimiters
```

```
    subroutine set_separator
      type(string_t) :: separator
      select case (rule%type)
      case (S_LIST)
         separator = lexeme_get_contents (lexeme(5))
         call syntax_rule_set_separator (rule, separator)
      case (S_ARGS)
         separator = lexeme_get_contents (lexeme(6))
         call syntax_rule_set_separator (rule, separator)
      end select
    end subroutine set_separator
    subroutine set_children
      allocate (rule%child(n_children))
      select case (rule%type)
      case (S_GROUP)
         call syntax_rule_set_sub (rule, 1, syntax_get_rule_ptr (syntax, &
              lexeme_get_contents (lexeme(5))))
      case (S_SEQUENCE)
         do i = 1, n_children
            call syntax_rule_set_sub (rule, i, syntax_get_rule_ptr (syntax, &
                 lexeme_get_contents (lexeme(i+3))))
         end do
      case (S_LIST, S_ALTERNATIVE)
         do i = 1, n_children
            call syntax_rule_set_sub (rule, i, syntax_get_rule_ptr (syntax, &
                 lexeme_get_contents (lexeme(2*i+2))))
         end do
      case (S_ARGS)
         do i = 1, n_children
            call syntax_rule_set_sub (rule, i, syntax_get_rule_ptr (syntax, &
                 lexeme_get_contents (lexeme(2*i+3))))
         end do
      end select
    end subroutine set_children
    subroutine broken_rule (rule)
      type(syntax_rule_t), intent(in) :: rule
      call lexer_show_location (lexer)
      call msg_bug (" Syntax definition: broken rule '" &
           // char (wkey (rule)) // "'")
    end subroutine broken_rule
  end subroutine set_rule_contents
```

This routine completes the syntax table object. We assume that the rule array is set up. We associate the top rule with the first entry in the rule array and build up the keyword list.

The keyword list includes delimiters and separators. Filling it can only be done after all rules are set. We scan the rule tree. For each keyword that we find, we try to add it to the keyword list; the pointer to the last element is carried along with the recursive scanning. Before appending a keyword, we check whether it is already in the list.

⟨*Syntax: subroutines*⟩+≡

```
  subroutine syntax_make_keyword_list (syntax)
    type(syntax_t), intent(inout), target :: syntax
```

```
  type(syntax_rule_t), pointer :: rule
  rule => syntax%rule(1)
  call rule_scan_rec (rule, syntax%keyword_list)
contains
  recursive subroutine rule_scan_rec (rule, keyword_list)
    type(syntax_rule_t), pointer :: rule
    type(keyword_list_t), intent(inout) :: keyword_list
    integer :: i
    if (rule%used)  return
    rule%used = .true.
    select case (rule%type)
    case (S_UNKNOWN)
       call msg_bug (" Syntax: rule tree contains undefined rule")
    case (S_KEYWORD)
       call keyword_list_add (keyword_list, rule%keyword)
    end select
    select case (rule%type)
    case (S_LIST, S_ARGS)
       call keyword_list_add (keyword_list, rule%separator)
    end select
    select case (rule%type)
    case (S_GROUP, S_ARGS)
       call keyword_list_add (keyword_list, rule%delimiter_left)
       call keyword_list_add (keyword_list, rule%delimiter_right)
    end select
    select case (rule%type)
    case (S_SEQUENCE, S_GROUP, S_LIST, S_ARGS, S_ALTERNATIVE)
       if (.not. allocated (rule%child)) &
            call msg_bug (" Syntax: Non-terminal rule without children")
    case default
       if (allocated (rule%child)) &
            call msg_bug (" Syntax: Terminal rule with children")
    end select
    if (allocated (rule%child)) then
       do i = 1, size (rule%child)
          call rule_scan_rec (rule%child(i)%p, keyword_list)
       end do
    end if
  end subroutine rule_scan_rec
end subroutine syntax_make_keyword_list
```

The finalizer deallocates the rule pointer array and deletes the keyword list.

⟨*Syntax: subroutines*⟩+≡

```
subroutine syntax_final (syntax)
  type(syntax_t), intent(inout) :: syntax
  if (allocated (syntax%rule))  deallocate (syntax%rule)
  call keyword_list_final (syntax%keyword_list)
end subroutine syntax_final
```

7.3.6 Accessing the syntax table

Return a pointer to the top rule

⟨*Syntax: public*⟩+≡
```
public :: syntax_get_top_rule_ptr
```

⟨*Syntax: subroutines*⟩+≡
```
function syntax_get_top_rule_ptr (syntax) result (rule)
  type(syntax_rule_t), pointer :: rule
  type(syntax_t), intent(in), target :: syntax
  if (allocated (syntax%rule)) then
     rule => syntax%rule(1)
  else
     rule => null ()
  end if
end function syntax_get_top_rule_ptr
```

Assign the pointer to the rule associated with a given key (assumes that the rule array is allocated)

⟨*Syntax: public*⟩+≡
```
public :: syntax_get_rule_ptr
```

⟨*Syntax: subroutines*⟩+≡
```
function syntax_get_rule_ptr (syntax, key) result (rule)
  type(syntax_rule_t), pointer :: rule
  type(syntax_t), intent(in), target :: syntax
  type(string_t), intent(in) :: key
  integer :: i
  do i = 1, size (syntax%rule)
     if (syntax%rule(i)%keyword == key) then
        rule => syntax%rule(i)
        return
     end if
  end do
  call msg_bug (" Syntax table: Rule " // char (key) // " not found")
end function syntax_get_rule_ptr
```

Return a pointer to the keyword list

⟨*Syntax: public*⟩+≡
```
public :: syntax_get_keyword_list_ptr
```

⟨*Syntax: subroutines*⟩+≡
```
function syntax_get_keyword_list_ptr (syntax) result (keyword_list)
  type(keyword_list_t), pointer :: keyword_list
  type(syntax_t), intent(in), target :: syntax
  keyword_list => syntax%keyword_list
end function syntax_get_keyword_list_ptr
```

7.3.7 I/O

Write a readable representation of the syntax table

⟨*Syntax: public*⟩+≡
```
public :: syntax_write
```

⟨*Syntax: subroutines*⟩+≡

```
subroutine syntax_write (syntax, unit)
  type(syntax_t), intent(in) :: syntax
  integer, intent(in), optional :: unit
  integer :: u
  integer :: i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(A)") "Syntax table:"
  if (allocated (syntax%rule)) then
     do i = 1, size (syntax%rule)
        call syntax_rule_write (syntax%rule(i), u)
     end do
  else
     write (u, "(1x,A)") "[not allocated]"
  end if
  call keyword_list_write (syntax%keyword_list, u)
end subroutine syntax_write
```

7.4 The parser

On a small scale, the parser interprets the string tokens returned by the lexer; they are interpreted as numbers, keywords and such and stored as a typed object. On a large scale, a text is read, parsed, and a syntax rule set is applied such that the tokens are stored as a parse tree. Syntax errors are spotted in this process, so the resulting parse tree is syntactically correct by definition.

⟨`parser.f90`⟩≡

```
⟨File header⟩

module parser

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_defs, only: FMT_19
  use system_defs, only: DIGITS
  use diagnostics
  use md5
  use lexers
  use syntax_rules

⟨Standard module head⟩

⟨Parser: public⟩

⟨Parser: types⟩

⟨Parser: interfaces⟩

contains

⟨Parser: procedures⟩

end module parser
```

7.4.1 The token type

Tokens are elements of the parsed input that carry a value: logical, integer, real, quoted string, (unquoted) identifier, or known keyword. Note that non-keyword tokens also have an abstract key attached to them.

This is an obvious candidate for polymorphism.

⟨*Parser: types*⟩≡

```
type :: token_t
   private
   integer :: type = S_UNKNOWN
   logical, pointer :: lval => null ()
   integer, pointer :: ival => null ()
   real(default), pointer :: rval => null ()
   complex(default), pointer :: cval => null ()
   type(string_t), pointer :: sval => null ()
   type(string_t), pointer :: kval => null ()
```

```
    type(string_t), dimension(:), pointer :: quote => null ()
end type token_t
```

Create a token from the lexeme returned by the lexer: Allocate storage and try to interpret the lexeme according to the type that is requested by the parser. For a keyword token, match the lexeme against the requested key. If successful, set the token type, value, and key. Otherwise, set the type to S_UNKNOWN.

⟨*Parser: procedures*⟩≡

```
subroutine token_init (token, lexeme, requested_type, key)
  type(token_t), intent(out) :: token
  type(lexeme_t), intent(in) :: lexeme
  integer, intent(in) :: requested_type
  type(string_t), intent(in) :: key
  integer :: type
  type = lexeme_get_type (lexeme)
  token%type = S_UNKNOWN
  select case (requested_type)
  case (S_LOGICAL)
     if (type == T_IDENTIFIER)  call read_logical &
          (char (lexeme_get_string (lexeme)))
  case (S_INTEGER)
     if (type == T_NUMERIC)  call read_integer &
          (char (lexeme_get_string (lexeme)))
  case (S_REAL)
     if (type == T_NUMERIC)  call read_real &
          (char (lexeme_get_string (lexeme)))
  case (S_COMPLEX)
     if (type == T_NUMERIC)  call read_complex &
          (char (lexeme_get_string (lexeme)))
  case (S_IDENTIFIER)
     if (type == T_IDENTIFIER)  call read_identifier &
          (lexeme_get_string (lexeme))
  case (S_KEYWORD)
     if (type == T_KEYWORD)  call check_keyword &
          (lexeme_get_string (lexeme), key)
  case (S_QUOTED)
     if (type == T_QUOTED)   call read_quoted &
          (lexeme_get_contents (lexeme), lexeme_get_delimiters (lexeme))
  case default
     print *, requested_type
     call msg_bug (" Invalid token type code requested by the parser")
  end select
  if (token%type /= S_UNKNOWN) then
     allocate (token%kval)
     token%kval = key
  end if
contains
  subroutine read_logical (s)
    character(*), intent(in) :: s
    select case (s)
    case ("t", "T", "true", "TRUE", "y", "Y", "yes", "YES")
       allocate (token%lval)
       token%lval = .true.
```

```
      token%type = S_LOGICAL
   case ("f", "F", "false", "FALSE", "n", "N", "no", "NO")
      allocate (token%lval)
      token%lval = .false.
      token%type = S_LOGICAL
   end select
end subroutine read_logical
subroutine read_integer (s)
   character(*), intent(in) :: s
   integer :: tmp, iostat
   if (verify (s, DIGITS) == 0) then
      read (s, *, iostat=iostat) tmp
      if (iostat == 0) then
         allocate (token%ival)
         token%ival = tmp
         token%type = S_INTEGER
      end if
   end if
end subroutine read_integer
subroutine read_real (s)
   character(*), intent(in) :: s
   real(default) :: tmp
   integer :: iostat
   read (s, *, iostat=iostat) tmp
   if (iostat == 0) then
      allocate (token%rval)
      token%rval = tmp
      token%type = S_REAL
   end if
end subroutine read_real
subroutine read_complex (s)
   character(*), intent(in) :: s
   complex(default) :: tmp
   integer :: iostat
   read (s, *, iostat=iostat) tmp
   if (iostat == 0) then
      allocate (token%cval)
      token%cval = tmp
      token%type = S_COMPLEX
   end if
end subroutine read_complex
subroutine read_identifier (s)
   type(string_t), intent(in) :: s
   allocate (token%sval)
   token%sval = s
   token%type = S_IDENTIFIER
end subroutine read_identifier
subroutine check_keyword (s, key)
   type(string_t), intent(in) :: s
   type(string_t), intent(in) :: key
   if (key == s)  token%type = S_KEYWORD
end subroutine check_keyword
subroutine read_quoted (s, del)
   type(string_t), intent(in) :: s
```

```
      type(string_t), dimension(2), intent(in) :: del
      allocate (token%sval, token%quote(2))
      token%sval = s
      token%quote(1) = del(1)
      token%quote(2) = del(2)
      token%type = S_QUOTED
    end subroutine read_quoted
  end subroutine token_init
```

Manually set a token to a keyword.

⟨*Parser: procedures*⟩+≡

```
  subroutine token_init_key (token, key)
    type(token_t), intent(out) :: token
    type(string_t), intent(in) :: key
    token%type = S_KEYWORD
    allocate (token%kval)
    token%kval = key
  end subroutine token_init_key
```

Reset a token to an empty state, freeing allocated memory, and deallocate the token itself.

⟨*Parser: procedures*⟩+≡

```
  subroutine token_final (token)
    type(token_t), intent(inout) :: token
    token%type = S_UNKNOWN
    if (associated (token%lval))  deallocate (token%lval)
    if (associated (token%ival))  deallocate (token%ival)
    if (associated (token%rval))  deallocate (token%rval)
    if (associated (token%sval))  deallocate (token%sval)
    if (associated (token%kval))  deallocate (token%kval)
    if (associated (token%quote))  deallocate (token%quote)
  end subroutine token_final
```

Check for empty=valid token:

⟨*Parser: procedures*⟩+≡

```
  function token_is_valid (token) result (valid)
    logical :: valid
    type(token_t), intent(in) :: token
    valid = token%type /= S_UNKNOWN
  end function token_is_valid
```

Write the contents of a token.

⟨*Parser: procedures*⟩+≡

```
  subroutine token_write (token, unit)
    type(token_t), intent(in) :: token
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    select case (token%type)
    case (S_LOGICAL)
       write (u, "(L1)") token%lval
    case (S_INTEGER)
```

```
      write (u, "(I0)") token%ival
    case (S_REAL)
      write (u, "(" // FMT_19 // ")") token%rval
    case (S_COMPLEX)
      write (u, "(’(’," // FMT_19 // ",’,’," // FMT_19 // ",’)’)") token%cval
    case (S_IDENTIFIER)
      write (u, "(A)") char (token%sval)
    case (S_KEYWORD)
      write (u, "(A,A)") ’[keyword] ’ // char (token%kval)
    case (S_QUOTED)
      write (u, "(A)") &
           char (token%quote(1)) // char (token%sval) // char (token%quote(2))
    case default
      write (u, "(A)") ’[empty]’
    end select
  end subroutine token_write
```

Token assignment via deep copy. This is useful to avoid confusion when the token is transferred to some parse-tree node.

⟨*Parser: interfaces*⟩≡

```
  interface assignment(=)
     module procedure token_assign
     module procedure token_assign_integer
     module procedure token_assign_real
     module procedure token_assign_complex
     module procedure token_assign_logical
     module procedure token_assign_string
  end interface
```

We need to copy only the contents that are actually assigned, the other pointers remain disassociated.

⟨*Parser: procedures*⟩+≡

```
  subroutine token_assign (token, token_in)
    type(token_t), intent(out) :: token
    type(token_t), intent(in) :: token_in
    token%type = token_in%type
    select case (token%type)
    case (S_LOGICAL);    allocate (token%lval);  token%lval = token_in%lval
    case (S_INTEGER);    allocate (token%ival);  token%ival = token_in%ival
    case (S_REAL);       allocate (token%rval);  token%rval = token_in%rval
    case (S_COMPLEX); allocate (token%cval);  token%cval = token_in%cval
    case (S_IDENTIFIER); allocate (token%sval);  token%sval = token_in%sval
    case (S_QUOTED);     allocate (token%sval);  token%sval = token_in%sval
       allocate (token%quote(2));  token%quote = token_in%quote
    end select
    if (token%type /= S_UNKNOWN) then
       allocate (token%kval);  token%kval = token_in%kval
    end if
  end subroutine token_assign
```

We need to copy only the contents that are actually assigned, the other pointers remain disassociated.

⟨*Parser: procedures*⟩+≡

```
subroutine token_assign_integer (token, ival)
  type(token_t), intent(out) :: token
  integer, intent(in) :: ival
  token%type = S_INTEGER
  allocate (token%ival)
  token%ival = ival
end subroutine token_assign_integer

subroutine token_assign_real (token, rval)
  type(token_t), intent(out) :: token
  real(default), intent(in) :: rval
  token%type = S_REAL
  allocate (token%rval)
  token%rval = rval
end subroutine token_assign_real

subroutine token_assign_complex (token, cval)
  type(token_t), intent(out) :: token
  complex(default), intent(in) :: cval
  token%type = S_COMPLEX
  allocate (token%cval)
  token%cval = cval
end subroutine token_assign_complex

subroutine token_assign_logical (token, lval)
  type(token_t), intent(out) :: token
  logical, intent(in) :: lval
  token%type = S_LOGICAL
  allocate (token%lval)
  token%lval = lval
end subroutine token_assign_logical

subroutine token_assign_string (token, sval)
  type(token_t), intent(out) :: token
  type(string_t), intent(in) :: sval
  token%type = S_QUOTED
  allocate (token%sval)
  token%sval = sval
  allocate (token%quote(2));  token%quote = '"'
end subroutine token_assign_string
```

7.4.2 Retrieve token contents

These functions all do a trivial sanity check that should avoid crashes.

⟨*Parser: procedures*⟩+≡

```
function token_get_logical (token) result (lval)
  logical :: lval
  type(token_t), intent(in) :: token
  if (associated (token%lval)) then
     lval = token%lval
  else
```

```
     call token_mismatch (token, "logical")
  end if
end function token_get_logical

function token_get_integer (token) result (ival)
  integer :: ival
  type(token_t), intent(in) :: token
  if (associated (token%ival)) then
     ival = token%ival
  else
     call token_mismatch (token, "integer")
  end if
end function token_get_integer

function token_get_real (token) result (rval)
  real(default) :: rval
  type(token_t), intent(in) :: token
  if (associated (token%rval)) then
     rval = token%rval
  else
     call token_mismatch (token, "real")
  end if
end function token_get_real

function token_get_cmplx (token) result (cval)
  complex(default) :: cval
  type(token_t), intent(in) :: token
  if (associated (token%cval)) then
     cval = token%cval
  else
     call token_mismatch (token, "complex")
  end if
end function token_get_cmplx

function token_get_string (token) result (sval)
  type(string_t) :: sval
  type(token_t), intent(in) :: token
  if (associated (token%sval)) then
     sval = token%sval
  else
     call token_mismatch (token, "string")
  end if
end function token_get_string

function token_get_key (token) result (kval)
  type(string_t) :: kval
  type(token_t), intent(in) :: token
  if (associated (token%kval)) then
     kval = token%kval
  else
     call token_mismatch (token, "keyword")
  end if
end function token_get_key
```

```
function token_get_quote (token) result (quote)
  type(string_t), dimension(2) :: quote
  type(token_t), intent(in) :: token
  if (associated (token%quote)) then
     quote = token%quote
  else
     call token_mismatch (token, "quote")
  end if
end function token_get_quote
```

⟨*Parser: procedures*⟩+≡

```
subroutine token_mismatch (token, type)
  type(token_t), intent(in) :: token
  character(*), intent(in) :: type
  write (6, "(A)", advance="no")  "Token: "
  call token_write (token)
  call msg_bug (" Token type mismatch; value required as " // type)
end subroutine token_mismatch
```

7.4.3 The parse tree: nodes

The parser will generate a parse tree from the input stream. Each node in this parse tree points to the syntax rule that was applied. (Since syntax rules are stored in a pointer-type array within the syntax table, they qualify as targets.) A leaf node contains a token. A branch node has subnodes. The subnodes are stored as a list, so each node also has a `next` pointer.

⟨*Parser: public*⟩≡

```
public :: parse_node_t
```

⟨*Parser: types*⟩+≡

```
type :: parse_node_t
   private
   type(syntax_rule_t), pointer :: rule => null ()
   type(token_t) :: token
   integer :: n_sub = 0
   type(parse_node_t), pointer :: sub_first => null ()
   type(parse_node_t), pointer :: sub_last => null ()
   type(parse_node_t), pointer :: next => null ()
 contains
 ⟨Parser: parse node: TBP⟩
end type parse_node_t
```

Container for parse node pointers, useful for creating pointer arrays:

⟨*Parser: public*⟩+≡

```
public :: parse_node_p
```

⟨*Parser: types*⟩+≡

```
type :: parse_node_p
  type(parse_node_t), pointer :: ptr => null ()
end type parse_node_p
```

Output. The first version writes a node together with its sub-node tree, organized by indentation.

⟨*Parser: parse node: TBP*⟩≡
```
procedure :: write => parse_node_write_rec
```

⟨*Parser: public*⟩+≡
```
public :: parse_node_write_rec
```

⟨*Parser: procedures*⟩+≡
```
recursive subroutine parse_node_write_rec (node, unit, short, depth)
  class(parse_node_t), intent(in), target :: node
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: short
  integer, intent(in), optional :: depth
  integer :: u, d
  type(parse_node_t), pointer :: current
  u = given_output_unit (unit);  if (u < 0)  return
  d = 0;  if (present (depth))  d = depth
  call parse_node_write (node, u, short=short)
  current => node%sub_first
  do while (associated (current))
     write (u, "(A)", advance = "no")  repeat ("|  ", d)
     call parse_node_write_rec (current, unit, short, d+1)
     current => current%next
  end do
end subroutine parse_node_write_rec
```

This does the actual output for a single node, without recursion.

⟨*Parser: public*⟩+≡
```
public :: parse_node_write
```

⟨*Parser: procedures*⟩+≡
```
subroutine parse_node_write (node, unit, short)
  class(parse_node_t), intent(in) :: node
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: short
  integer :: u
  type(parse_node_t), pointer :: current
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "('+ ')", advance = "no")
  if (associated (node%rule)) then
     call syntax_rule_write (node%rule, u, &
          short=short, key_only=.true., advance=.false.)
     if (token_is_valid (node%token)) then
        write (u, "('  = ')", advance="no")
        call token_write (node%token, u)
     else if (associated (node%sub_first)) then
        write (u, "('  = ')", advance="no")
        current => node%sub_first
        do while (associated (current))
           call syntax_rule_write (current%rule, u, &
                short=.true., key_only=.true., advance=.false.)
           current => current%next
        end do
        write (u, *)
```

```
      else
         write (u, *)
      end if
   else
      write (u, *) "[empty]"
   end if
end subroutine parse_node_write
```

Finalize the token and recursively finalize and deallocate all sub-nodes.

⟨*Parser: public*⟩+≡

```
public :: parse_node_final
```

⟨*Parser: procedures*⟩+≡

```
recursive subroutine parse_node_final (node, recursive)
  type(parse_node_t), intent(inout) :: node
  type(parse_node_t), pointer :: current
  logical, intent(in), optional :: recursive
  logical :: rec
  rec = .true.;  if (present (recursive))  rec = recursive
  call token_final (node%token)
  if (rec) then
     do while (associated (node%sub_first))
        current => node%sub_first
        node%sub_first => node%sub_first%next
        call parse_node_final (current)
        deallocate (current)
     end do
  end if
end subroutine parse_node_final
```

7.4.4 Filling nodes

The constructors allocate and initialize the node. There are two possible initializers (in a later version, should correspond to different type extensions).

First, leaf (terminal) nodes. The token constructor does the actual work, looking at the requested type and key for the given rule and matching against the lexeme contents. If it fails, the token will keep the type `S_UNKNOWN` and remain empty. Otherwise, we have a valid node which contains the new token.

If the lexeme argument is absent, the token is left empty.

⟨*Parser: procedures*⟩+≡

```
subroutine parse_node_create_leaf (node, rule, lexeme)
  type(parse_node_t), pointer :: node
  type(syntax_rule_t), intent(in), target :: rule
  type(lexeme_t), intent(in) :: lexeme
  allocate (node)
  node%rule => rule
  call token_init (node%token, lexeme, &
       syntax_rule_get_type (rule), syntax_rule_get_key (rule))
  if (.not. token_is_valid (node%token))  deallocate (node)
end subroutine parse_node_create_leaf
```

This version allows us to manually create a leaf node that holds a keyword.

⟨*Parser: public*⟩+≡

```
public :: parse_node_create_key
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_node_create_key (node, rule)
  type(parse_node_t), intent(out) :: node
  type(syntax_rule_t), intent(in), target :: rule
  node%rule => rule
  call token_init_key (node%token, syntax_rule_get_key (rule))
end subroutine parse_node_create_key
```

This version allows us to manually create a leaf node that holds a fixed value. Only one of the optional values should be provided.

⟨*Parser: public*⟩+≡

```
public :: parse_node_create_value
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_node_create_value (node, rule, ival, rval, cval, sval, lval)
  type(parse_node_t), intent(out) :: node
  type(syntax_rule_t), intent(in), target :: rule
  integer, intent(in), optional :: ival
  real(default), intent(in), optional :: rval
  complex(default), intent(in), optional :: cval
  type(string_t), intent(in), optional :: sval
  logical, intent(in), optional :: lval
  node%rule => rule
  call parse_node_set_value (node, ival, rval, cval, sval, lval)
end subroutine parse_node_create_value
```

Directly set the value without changing anything else.

⟨*Parser: public*⟩+≡

```
public :: parse_node_set_value
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_node_set_value (node, ival, rval, cval, sval, lval)
  type(parse_node_t), intent(inout) :: node
  integer, intent(in), optional :: ival
  real(default), intent(in), optional :: rval
  complex(default), intent(in), optional :: cval
  type(string_t), intent(in), optional :: sval
  logical, intent(in), optional :: lval
  if (present (ival)) then
     node%token = ival
  else if (present (rval)) then
     node%token = rval
  else if (present (cval)) then
     node%token = cval
  else if (present (lval)) then
     node%token = lval
  else if (present (sval)) then
     node%token = sval
  end if
end subroutine parse_node_set_value
```

Second, branch nodes. We first assign the rule:

⟨*Parser: public*⟩+≡
```
public :: parse_node_create_branch
```

⟨*Parser: procedures*⟩+≡
```
subroutine parse_node_create_branch (node, rule)
  type(parse_node_t), pointer :: node
  type(syntax_rule_t), intent(in), target :: rule
  allocate (node)
  node%rule => rule
end subroutine parse_node_create_branch
```

Copy a node. This is a shallow copy. Note that we have to nullify the `next` pointer if we don't want to inherit the context of the original node.

⟨*Parser: parse node: TBP*⟩+≡
```
procedure :: copy => parse_node_copy
```

⟨*Parser: procedures*⟩+≡
```
subroutine parse_node_copy (node, copy)
  class(parse_node_t), intent(in) :: node
  type(parse_node_t), pointer, intent(out) :: copy
  allocate (copy)
  select type (node)
  type is (parse_node_t)
     copy = node
  end select
  copy%next => null ()
end subroutine parse_node_copy
```

Append a sub-node. The sub-node must exist and be a valid target, otherwise nothing is done.

⟨*Parser: public*⟩+≡
```
public :: parse_node_append_sub
```

⟨*Parser: parse node: TBP*⟩+≡
```
procedure :: append_sub => parse_node_append_sub
```

⟨*Parser: procedures*⟩+≡
```
subroutine parse_node_append_sub (node, sub)
  class(parse_node_t), intent(inout) :: node
  type(parse_node_t), pointer :: sub
  if (associated (sub)) then
     if (associated (node%sub_last)) then
        node%sub_last%next => sub
     else
        node%sub_first => sub
     end if
     node%sub_last => sub
  end if
end subroutine parse_node_append_sub
```

For easy access, once the list is complete we count the number of sub-nodes. If there are no subnodes, the whole node is deleted.

⟨*Parser: public*⟩+≡
```
public :: parse_node_freeze_branch
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_node_freeze_branch (node)
  type(parse_node_t), pointer :: node
  type(parse_node_t), pointer :: current
  node%n_sub = 0
  current => node%sub_first
  do while (associated (current))
     node%n_sub = node%n_sub + 1
     current => current%next
  end do
  if (node%n_sub == 0)  deallocate (node)
end subroutine parse_node_freeze_branch
```

Replace the syntax rule. This makes sense only if the parse node adheres to the syntax of the new rule.

⟨*Parser: public*⟩+≡

```
public :: parse_node_replace_rule
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_node_replace_rule (node, rule)
  type(parse_node_t), pointer :: node
  type(syntax_rule_t), intent(in), target :: rule
  node%rule => rule
end subroutine parse_node_replace_rule
```

Replace the last subnode by the target node. Since the subnodes are stored as a linked list, we can do this only if we copy the subnodes. Furthermore, the target node must also be copied, and the `next` pointer of the copy is nullified. This ensures that we cannot modify the originals at the subnode level.

All copies are shallow copies. This implies that further modifications at the sub-subnode level will affect the original nodes and must therefore be forbidden.

Use with care, this invites to memory mismanagement. The copy nodes can be deallocated, but not finalized, since its subnodes are the same objects as the subnodes of the target node.

⟨*Parser: public*⟩+≡

```
public :: parse_node_replace_last_sub
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_node_replace_last_sub (node, pn_target)
  type(parse_node_t), intent(inout), target :: node
  type(parse_node_t), intent(in), target :: pn_target
  type(parse_node_t), pointer :: current, current_copy, previous
  integer :: i
  select case (node%n_sub)
  case (1)
     allocate (current_copy)
     current_copy = pn_target
     node%sub_first => current_copy
  case (2:)
     current => node%sub_first
     allocate (current_copy)
     current_copy = current
     node%sub_first => current_copy
```

```
      previous => current_copy
      do i = 1, node%n_sub - 2
         current => current%next
         allocate (current_copy)
         current_copy = current
         previous%next => current_copy
         previous => current_copy
      end do
      allocate (current_copy)
      current_copy = pn_target
      previous%next => current_copy
   case default
      call parse_node_write (node)
      call msg_bug ("'replace_last_sub' called for non-branch parse node")
   end select
   current_copy%next => null ()
   node%sub_last => current_copy
end subroutine parse_node_replace_last_sub
```

7.4.5 Accessing nodes

Return the node contents. Check if pointers are associated. No check when accessing a sub-node; assume that `parse_node_n_sub` is always used for the upper bound.

The token extractor returns a pointer.

⟨*Parser: public*⟩+≡

```
public :: parse_node_get_rule_ptr
public :: parse_node_get_n_sub
public :: parse_node_get_sub_ptr
public :: parse_node_get_next_ptr
public :: parse_node_get_last_sub_ptr
```

⟨*Parser: parse node: TBP*⟩+≡

```
procedure :: get_rule_ptr => parse_node_get_rule_ptr
procedure :: get_n_sub => parse_node_get_n_sub
procedure :: get_sub_ptr => parse_node_get_sub_ptr
procedure :: get_next_ptr => parse_node_get_next_ptr
```

⟨*Parser: procedures*⟩+≡

```
function parse_node_get_rule_ptr (node) result (rule)
  class(parse_node_t), intent(in) :: node
  type(syntax_rule_t), pointer :: rule
  if (associated (node%rule)) then
     rule => node%rule
  else
     rule => null ()
     call parse_node_undefined (node, "rule")
  end if
end function parse_node_get_rule_ptr

function parse_node_get_n_sub (node) result (n)
  class(parse_node_t), intent(in) :: node
  integer :: n
```

```
  n = node%n_sub
end function parse_node_get_n_sub

function parse_node_get_sub_ptr (node, n, tag, required) result (sub)
  class(parse_node_t), intent(in), target :: node
  type(parse_node_t), pointer :: sub
  integer, intent(in), optional :: n
  character(*), intent(in), optional :: tag
  logical, intent(in), optional :: required
  integer :: i
  sub => node%sub_first
  if (present (n)) then
     do i = 2, n
        if (associated (sub)) then
           sub => sub%next
        else
           return
        end if
     end do
  end if
  call parse_node_check (sub, tag, required)
end function parse_node_get_sub_ptr

function parse_node_get_next_ptr (sub, n, tag, required) result (next)
  class(parse_node_t), intent(in), target :: sub
  type(parse_node_t), pointer :: next
  integer, intent(in), optional :: n
  character(*), intent(in), optional :: tag
  logical, intent(in), optional :: required
  integer :: i
  next => sub%next
  if (present (n)) then
     do i = 2, n
        if (associated (next)) then
           next => next%next
        else
           exit
        end if
     end do
  end if
  call parse_node_check (next, tag, required)
end function parse_node_get_next_ptr

function parse_node_get_last_sub_ptr (node, tag, required) result (sub)
  type(parse_node_t), pointer :: sub
  type(parse_node_t), intent(in), target :: node
  character(*), intent(in), optional :: tag
  logical, intent(in), optional :: required
  sub => node%sub_last
  call parse_node_check (sub, tag, required)
end function parse_node_get_last_sub_ptr
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_node_undefined (node, obj)
```

```
    type(parse_node_t), intent(in) :: node
    character(*), intent(in) :: obj
    call parse_node_write (node, 6)
    call msg_bug (" Parse-tree node: " // obj // " requested, but undefined")
  end subroutine parse_node_undefined
```

Check if a parse node has a particular tag, and if it is associated:

⟨*Parser: public*⟩+≡

```
  public :: parse_node_check
```

⟨*Parser: procedures*⟩+≡

```
  subroutine parse_node_check (node, tag, required)
    type(parse_node_t), pointer :: node
    character(*), intent(in), optional :: tag
    logical, intent(in), optional :: required
    if (associated (node)) then
       if (present (tag)) then
          if (parse_node_get_rule_key (node) /= tag) &
               call parse_node_mismatch (tag, node)
       end if
    else
       if (present (required)) then
          if (required) &
               call msg_bug (" Missing node, expected <" // tag // ">")
       end if
    end if
  end subroutine parse_node_check
```

This is called by a parse-tree scanner if the expected and the actual nodes do not match

⟨*Parser: public*⟩+≡

```
  public :: parse_node_mismatch
```

⟨*Parser: procedures*⟩+≡

```
  subroutine parse_node_mismatch (string, parse_node)
    character(*), intent(in) :: string
    type(parse_node_t), intent(in) :: parse_node
    call parse_node_write (parse_node)
    call msg_bug (" Syntax mismatch, expected <" // string // ">.")
  end subroutine parse_node_mismatch
```

The following functions are wrappers for extracting the token contents.

⟨*Parser: public*⟩+≡

```
  public :: parse_node_get_logical
  public :: parse_node_get_integer
  public :: parse_node_get_real
  public :: parse_node_get_cmplx
  public :: parse_node_get_string
  public :: parse_node_get_key
  public :: parse_node_get_rule_key
```

⟨*Parser: parse node: TBP*⟩+≡

```
  procedure :: get_logical => parse_node_get_logical
```

```
procedure :: get_integer => parse_node_get_integer
procedure :: get_real => parse_node_get_real
procedure :: get_cmplx => parse_node_get_cmplx
procedure :: get_string => parse_node_get_string
procedure :: get_key => parse_node_get_key
procedure :: get_rule_key => parse_node_get_rule_key
```

⟨*Parser: procedures*⟩+≡

```
function parse_node_get_logical (node) result (lval)
  class(parse_node_t), intent(in), target :: node
  logical :: lval
  lval = token_get_logical (parse_node_get_token_ptr (node))
end function parse_node_get_logical

function parse_node_get_integer (node) result (ival)
  class(parse_node_t), intent(in), target :: node
  integer :: ival
  ival = token_get_integer (parse_node_get_token_ptr (node))
end function parse_node_get_integer

function parse_node_get_real (node) result (rval)
  class(parse_node_t), intent(in), target :: node
  real(default) :: rval
  rval = token_get_real (parse_node_get_token_ptr (node))
end function parse_node_get_real

function parse_node_get_cmplx (node) result (cval)
  class(parse_node_t), intent(in), target :: node
  complex(default) :: cval
  cval = token_get_cmplx (parse_node_get_token_ptr (node))
end function parse_node_get_cmplx

function parse_node_get_string (node) result (sval)
  class(parse_node_t), intent(in), target :: node
  type(string_t) :: sval
  sval = token_get_string (parse_node_get_token_ptr (node))
end function parse_node_get_string

function parse_node_get_key (node) result (kval)
  class(parse_node_t), intent(in), target :: node
  type(string_t) :: kval
  kval = token_get_key (parse_node_get_token_ptr (node))
end function parse_node_get_key

function parse_node_get_rule_key (node) result (kval)
  class(parse_node_t), intent(in), target :: node
  type(string_t) :: kval
  kval = syntax_rule_get_key (parse_node_get_rule_ptr (node))
end function parse_node_get_rule_key

function parse_node_get_token_ptr (node) result (token)
  type(token_t), pointer :: token
  type(parse_node_t), intent(in), target :: node
  if (token_is_valid (node%token)) then
     token => node%token
```

```
    else
       call parse_node_undefined (node, "token")
    end if
  end function parse_node_get_token_ptr
```

Return a MD5 sum for a parse node. The method is to write the node to a scratch file and to evaluate the MD5 sum of that file.

⟨*Parser: public*⟩+≡

```
  public :: parse_node_get_md5sum
```

⟨*Parser: procedures*⟩+≡

```
  function parse_node_get_md5sum (pn) result (md5sum_pn)
    character(32) :: md5sum_pn
    type(parse_node_t), intent(in) :: pn
    integer :: u
    u = free_unit ()
    open (unit = u, status = "scratch", action = "readwrite")
    call parse_node_write_rec (pn, unit=u)
    rewind (u)
    md5sum_pn = md5sum (u)
    close (u)
  end function parse_node_get_md5sum
```

7.4.6 The parse tree

The parse tree is a tree of nodes, where leaf nodes hold a valid token, while branch nodes point to a list of sub-nodes.

⟨*Parser: public*⟩+≡

```
  public :: parse_tree_t
```

⟨*Parser: types*⟩+≡

```
  type :: parse_tree_t
     private
     type(parse_node_t), pointer :: root_node => null ()
   contains
   ⟨Parser: parse tree: TBP⟩
  end type parse_tree_t
```

The parser. Its arguments are the parse tree (which should be empty initially), the lexer (which should be already set up), the syntax table (which should be valid), and the input stream. The input stream is completely parsed, using the lexer setup and the syntax rules as given, and the parse tree is built accordingly.

If `check_eof` is absent or true, the parser will complain about trailing garbage. Otherwise, it will ignore it.

By default, the input stream is matched against the top rule in the specified syntax. If `key` is given, it is matched against the rule with the specified key instead.

Failure at the top level means that no rule could match at all; in this case the error message will show the top rule.

⟨*Parser: public*⟩+≡

```
  public :: parse_tree_init
```

⟨*Parser: parse tree: TBP*⟩≡

```
procedure :: parse => parse_tree_init
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_tree_init &
     (parse_tree, syntax, lexer, key, check_eof)
  class(parse_tree_t), intent(inout) :: parse_tree
  type(lexer_t), intent(inout) :: lexer
  type(syntax_t), intent(in), target :: syntax
  type(string_t), intent(in), optional :: key
  logical, intent(in), optional :: check_eof
  type(syntax_rule_t), pointer :: rule
  type(lexeme_t) :: lexeme
  type(parse_node_t), pointer :: node
  logical :: ok, check
  check = .true.;  if (present (check_eof)) check = check_eof
  call lexer_clear (lexer)
  if (present (key)) then
     rule => syntax_get_rule_ptr (syntax, key)
  else
     rule => syntax_get_top_rule_ptr (syntax)
  end if
  if (associated (rule)) then
     call parse_node_match_rule (node, rule, ok)
     if (ok) then
        parse_tree%root_node => node
     else
        call parse_error (rule, lexeme)
     end if
     if (check) then
        call lex (lexeme, lexer)
        if (.not. lexeme_is_eof (lexeme)) then
           call lexer_show_location (lexer)
           call msg_fatal (" Syntax error " &
                // "(at or before the location indicated above)")
        end if
     end if
  else
     call msg_bug (" Parser failed because syntax is empty")
  end if
contains
⟨Parser: internal subroutines of parse_tree_init⟩
end subroutine parse_tree_init
```

The parser works recursively, following the rule tree, building the tree of nodes on the fly. If the given rule matches, the node is filled on return. If not, the node remains empty.

⟨*Parser: internal subroutines of* parse_tree_init⟩≡

```
recursive subroutine parse_node_match_rule (node, rule, ok)
  type(parse_node_t), pointer :: node
  type(syntax_rule_t), intent(in), target :: rule
  logical, intent(out) :: ok
  logical, parameter :: debug = .false.
  integer :: type
```

```
    if (debug)  write (6, "(A)", advance="no") "Parsing rule: "
    if (debug)  call syntax_rule_write (rule, 6)
    node => null ()
    type = syntax_rule_get_type (rule)
    if (syntax_rule_is_atomic (rule)) then
       call lex (lexeme, lexer)
       if (debug)  write (6, "(A)", advance="no") "Token: "
       if (debug)  call lexeme_write (lexeme, 6)
       call parse_node_create_leaf (node, rule, lexeme)
       ok = associated (node)
       if (.not. ok)  call lexer_put_back (lexer, lexeme)
    else
       select case (type)
       case (S_ALTERNATIVE);  call parse_alternative (node, rule, ok)
       case (S_GROUP);        call parse_group (node, rule, ok)
       case (S_SEQUENCE);     call parse_sequence (node, rule, .false., ok)
       case (S_LIST);         call parse_sequence (node, rule, .true., ok)
       case (S_ARGS);         call parse_args (node, rule, ok)
       case (S_IGNORE);       call parse_ignore (node, ok)
       end select
    end if
    if (debug) then
       if (ok) then
          write (6, "(A)", advance="no") "Matched rule: "
       else
          write (6, "(A)", advance="no") "Failed rule: "
       end if
       call syntax_rule_write (rule)
       if (associated (node)) call parse_node_write (node)
    end if
  end subroutine parse_node_match_rule
```

Parse an alternative: try each case. If the match succeeds, the node has been filled, so return. If nothing works, return failure.

⟨*Parser: internal subroutines of* parse_tree_init⟩+≡

```
  recursive subroutine parse_alternative (node, rule, ok)
    type(parse_node_t), pointer :: node
    type(syntax_rule_t), intent(in), target :: rule
    logical, intent(out) :: ok
    integer :: i
    do i = 1, syntax_rule_get_n_sub (rule)
       call parse_node_match_rule (node, syntax_rule_get_sub_ptr (rule, i), ok)
       if (ok)  return
    end do
    ok = .false.
  end subroutine parse_alternative
```

Parse a group: the first and third lexemes have to be the delimiters, the second one is parsed as the actual node, using now the child rule. If the first match fails, return with failure. If the other matches fail, issue an error, since we cannot lex back more than one item.

⟨*Parser: internal subroutines of* parse_tree_init⟩+≡

```
  recursive subroutine parse_group (node, rule, ok)
    type(parse_node_t), pointer :: node
    type(syntax_rule_t), intent(in), target :: rule
```

```
    logical, intent(out) :: ok
    type(string_t), dimension(2) :: delimiter
    delimiter = syntax_rule_get_delimiter (rule)
    call lex (lexeme, lexer)
    if (lexeme_get_string (lexeme) == delimiter(1)) then
       call parse_node_match_rule (node, syntax_rule_get_sub_ptr (rule, 1), ok)
       if (ok) then
          call lex (lexeme, lexer)
          if (lexeme_get_string (lexeme) == delimiter(2)) then
             ok = .true.
          else
             call parse_error (rule, lexeme)
          end if
       else
          call parse_error (rule, lexeme)
       end if
    else
       call lexer_put_back (lexer, lexeme)
       ok = .false.
    end if
  end subroutine parse_group
```

Parsing a sequence. The last rule element may be special: optional and/or repetitive. Each sub-node that matches is appended to the sub-node list of the parent node.

If `sep` is true, we look for a separator after each element.

⟨*Parser: internal subroutines of* `parse_tree_init`⟩+≡

```
  recursive subroutine parse_sequence (node, rule, sep, ok)
    type(parse_node_t), pointer :: node
    type(syntax_rule_t), intent(in), target :: rule
    logical, intent(in) :: sep
    logical, intent(out) :: ok
    type(parse_node_t), pointer :: current
    integer :: i, n
    logical :: opt, rep, cont
    type(string_t) :: separator
    call parse_node_create_branch (node, rule)
    if (sep)  separator = syntax_rule_get_separator (rule)
    n = syntax_rule_get_n_sub (rule)
    opt = syntax_rule_last_optional (rule)
    rep = syntax_rule_last_repetitive (rule)
    ok = .true.
    cont = .true.
    SCAN_RULE: do i = 1, n
       call parse_node_match_rule &
            (current, syntax_rule_get_sub_ptr (rule, i), cont)
       if (cont) then
          call parse_node_append_sub (node, current)
          if (sep .and. (i<n .or. rep)) then
             call lex (lexeme, lexer)
             if (lexeme_get_string (lexeme) /= separator) then
                call lexer_put_back (lexer, lexeme)
                cont = .false.
                exit SCAN_RULE
```

```
            end if
         end if
      else
         if (i == n .and. opt) then
            exit SCAN_RULE
         else if (i == 1) then
            ok = .false.
            exit SCAN_RULE
         else
            call parse_error (rule, lexeme)
         end if
      end if
   end do SCAN_RULE
   if (rep) then
      do while (cont)
         call parse_node_match_rule &
              (current, syntax_rule_get_sub_ptr (rule, n), cont)
         if (cont) then
            call parse_node_append_sub (node, current)
            if (sep) then
               call lex (lexeme, lexer)
               if (lexeme_get_string (lexeme) /= separator) then
                  call lexer_put_back (lexer, lexeme)
                  cont = .false.
               end if
            end if
         else
            if (sep)  call parse_error (rule, lexeme)
         end if
      end do
   end if
   call parse_node_freeze_branch (node)
end subroutine parse_sequence
```

Argument list: We use the parse_group code, but call parse_sequence inside.

⟨*Parser: internal subroutines of* parse_tree_init⟩+≡

```
recursive subroutine parse_args (node, rule, ok)
   type(parse_node_t), pointer :: node
   type(syntax_rule_t), intent(in), target :: rule
   logical, intent(out) :: ok
   type(string_t), dimension(2) :: delimiter
   delimiter = syntax_rule_get_delimiter (rule)
   call lex (lexeme, lexer)
   if (lexeme_get_string (lexeme) == delimiter(1)) then
      call parse_sequence (node, rule, .true., ok)
      if (ok) then
         call lex (lexeme, lexer)
         if (lexeme_get_string (lexeme) == delimiter(2)) then
            ok = .true.
         else
            call parse_error (rule, lexeme)
         end if
      else
         call parse_error (rule, lexeme)
      end if
```

```
    else
       call lexer_put_back (lexer, lexeme)
       ok = .false.
    end if
  end subroutine parse_args
```

The IGNORE syntax reads one lexeme and discards it if it is numeric, logical or string/identifier (but not a keyword). This is a successful match. Otherwise, the match fails. The node pointer is returned disassociated in any case.

⟨*Parser: internal subroutines of* parse_tree_init⟩+≡

```
  subroutine parse_ignore (node, ok)
    type(parse_node_t), pointer :: node
    logical, intent(out) :: ok
    call lex (lexeme, lexer)
    select case (lexeme_get_type (lexeme))
    case (T_NUMERIC, T_IDENTIFIER, T_QUOTED)
       ok = .true.
    case default
       ok = .false.
    end select
    node => null ()
  end subroutine parse_ignore
```

If the match fails and we cannot step back:

⟨*Parser: internal subroutines of* parse_tree_init⟩+≡

```
  subroutine parse_error (rule, lexeme)
    type(syntax_rule_t), intent(in) :: rule
    type(lexeme_t), intent(in) :: lexeme
    character(80) :: buffer
    integer :: u, iostat
    call lexer_show_location (lexer)
    u = free_unit ()
    open (u, status = "scratch")
    write (u, "(A)", advance="no")  "Expected syntax:"
    call syntax_rule_write (rule, u)
    write (u, "(A)", advance="no")  "Found token:"
    call lexeme_write (lexeme, u)
    rewind (u)
    do
       read (u, "(A)", iostat=iostat)  buffer
       if (iostat /= 0)  exit
       call msg_message (trim (buffer))
    end do
    call msg_fatal (" Syntax error " &
         // "(at or before the location indicated above)")
  end subroutine parse_error
```

The finalizer recursively deallocates all nodes and their contents. For each node, parse_node_final is called on the sub-nodes, which in turn deallocates the token or sub-node array contained within each of them. At the end, only the top node remains to be deallocated.

⟨*Parser: public*⟩+≡

```
  public :: parse_tree_final
```

⟨*Parser: parse tree: TBP*⟩+≡

```
  procedure :: final => parse_tree_final
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_tree_final (parse_tree)
  class(parse_tree_t), intent(inout) :: parse_tree
  if (associated (parse_tree%root_node)) then
     call parse_node_final (parse_tree%root_node)
     deallocate (parse_tree%root_node)
  end if
end subroutine parse_tree_final
```

Print the parse tree. Print one token per line, indented according to the depth of the node.

The `verbose` version includes type identifiers for the nodes.

⟨*Parser: public*⟩+≡

```
public :: parse_tree_write
```

⟨*Parser: parse tree: TBP*⟩+≡

```
procedure :: write => parse_tree_write
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_tree_write (parse_tree, unit, verbose)
  class(parse_tree_t), intent(in) :: parse_tree
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u
  logical :: short
  u = given_output_unit (unit);  if (u < 0)  return
  short = .true.;  if (present (verbose))  short = .not. verbose
  write (u, "(A)") "Parse tree:"
  if (associated (parse_tree%root_node)) then
     call parse_node_write_rec (parse_tree%root_node, unit, short, 1)
  else
     write (u, *) "[empty]"
  end if
end subroutine parse_tree_write
```

This is a generic error that can be issued if the parse tree does not meet the expectaions of the parser. This most certainly indicates a bug.

⟨*Parser: public*⟩+≡

```
public :: parse_tree_bug
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_tree_bug (node, keys)
  type(parse_node_t), intent(in) :: node
  character(*), intent(in) :: keys
  call parse_node_write (node)
  call msg_bug (" Inconsistency in parse tree: expected " // keys)
end subroutine parse_tree_bug
```

7.4.7 Access the parser

For scanning the parse tree we give access to the top node, as a node pointer. Of course, there should be no write access.

⟨*Parser: parse tree: TBP*⟩+≡

```
procedure :: get_root_ptr => parse_tree_get_root_ptr
```

⟨*Parser: procedures*⟩+≡

```
function parse_tree_get_root_ptr (parse_tree) result (node)
  class(parse_tree_t), intent(in) :: parse_tree
  type(parse_node_t), pointer :: node
  node => parse_tree%root_node
end function parse_tree_get_root_ptr
```

7.4.8 Tools

This operation traverses the parse tree and simplifies any occurences of a set of syntax rules. If such a parse node has only one sub-node, it is replaced by that subnode. (This makes sense only of the rules to eliminate have no meaningful token.)

⟨*Parser: public*⟩+≡

```
public :: parse_tree_reduce
```

⟨*Parser: procedures*⟩+≡

```
subroutine parse_tree_reduce (parse_tree, rule_key)
  type(parse_tree_t), intent(inout) :: parse_tree
  type(string_t), dimension(:), intent(in) :: rule_key
  type(parse_node_t), pointer :: pn
  pn => parse_tree%root_node
  if (associated (pn)) then
     call parse_node_reduce (pn, null(), null())
  end if
contains
  recursive subroutine parse_node_reduce (pn, pn_prev, pn_parent)
    type(parse_node_t), intent(inout), pointer :: pn
    type(parse_node_t), intent(in), pointer :: pn_prev, pn_parent
    type(parse_node_t), pointer :: pn_sub, pn_sub_prev, pn_tmp
    pn_sub_prev => null ()
    pn_sub => pn%sub_first
    do while (associated (pn_sub))
       call parse_node_reduce (pn_sub, pn_sub_prev, pn)
       pn_sub_prev => pn_sub
       pn_sub => pn_sub%next
    end do
    if (parse_node_get_n_sub (pn) == 1) then
       if (matches (parse_node_get_rule_key (pn), rule_key)) then
          pn_tmp => pn
          pn => pn%sub_first
          if (associated (pn_prev)) then
             pn_prev%next => pn
          else if (associated (pn_parent)) then
             pn_parent%sub_first => pn
          else
             parse_tree%root_node => pn
          end if
          if (associated (pn_tmp%next)) then
             pn%next => pn_tmp%next
          else if (associated (pn_parent)) then
```

```
            pn_parent%sub_last => pn
          end if
          call parse_node_final (pn_tmp, recursive=.false.)
          deallocate (pn_tmp)
        end if
      end if
    end subroutine parse_node_reduce
    function matches (key, key_list) result (flag)
      logical :: flag
      type(string_t), intent(in) :: key
      type(string_t), dimension(:), intent(in) :: key_list
      integer :: i
      flag = .true.
      do i = 1, size (key_list)
         if (key == key_list(i))  return
      end do
      flag = .false.
    end function matches
  end subroutine parse_tree_reduce
```

7.4.9 Applications

For a file of the form

```
process foo, bar
  <something>
process xyz
  <something>
```

get the `<something>` entry node for the first matching process tag. If no matching entry is found, the node pointer remains unassociated.

⟨*Parser: public*⟩+≡

```
  public :: parse_tree_get_process_ptr
```

⟨*Parser: procedures*⟩+≡

```
  function parse_tree_get_process_ptr (parse_tree, process) result (node)
    type(parse_node_t), pointer :: node
    type(parse_tree_t), intent(in), target :: parse_tree
    type(string_t), intent(in) :: process
    type(parse_node_t), pointer :: node_root, node_process_def
    type(parse_node_t), pointer :: node_process_phs, node_process_list
    integer :: j
    node_root => parse_tree%get_root_ptr ()
    if (associated (node_root)) then
       node_process_phs => parse_node_get_sub_ptr (node_root)
       SCAN_FILE: do while (associated (node_process_phs))
          node_process_def => parse_node_get_sub_ptr (node_process_phs)
          node_process_list => parse_node_get_sub_ptr (node_process_def, 2)
          do j = 1, parse_node_get_n_sub (node_process_list)
             if (parse_node_get_string &
                  (parse_node_get_sub_ptr (node_process_list, j)) &
                  == process) then
                node => parse_node_get_next_ptr (node_process_def)
```

```
                return
             end if
          end do
          node_process_phs => parse_node_get_next_ptr (node_process_phs)
       end do SCAN_FILE
       node => null ()
    else
       node => null ()
    end if
  end function parse_tree_get_process_ptr
```

7.4.10 Unit tests

Test module, followed by the corresponding implementation module.

⟨parser_ut.f90⟩≡

```
⟨File header⟩

module parser_ut
  use unit_tests
  use parser_uti

⟨Standard module head⟩

⟨Parser: public test⟩

contains

⟨Parser: test driver⟩

end module parser_ut
```

⟨parser_uti.f90⟩≡

```
⟨File header⟩

module parser_uti

  use syntax_rules

  use parser

⟨Standard module head⟩

⟨Parser: test declarations⟩

contains

⟨Parser: tests⟩

end module parser_uti
```

API: driver for the unit tests below.

⟨*Parser: public test*⟩≡

```
  public :: parse_test
```

⟨*Parser: test driver*⟩≡

```
subroutine parse_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨Parser: execute tests⟩
end subroutine parse_test
```

This checks the parser.

⟨*Parser: execute tests*⟩≡

```
call test (parse_1, "parse_1", &
     "check the parser", &
     u, results)
```

⟨*Parser: test declarations*⟩≡

```
public :: parse_1
```

⟨*Parser: tests*⟩≡

```
subroutine parse_1 (u)
  use ifiles
  use lexers
  integer, intent(in) :: u

  type(ifile_t) :: ifile
  type(syntax_t), target :: syntax
  type(lexer_t) :: lexer
  type(stream_t), target :: stream
  type(parse_tree_t), target :: parse_tree

  write (u, "(A)")  "* Test output: Parsing"
  write (u, "(A)")  "*   Purpose: test parse routines"
  write (u, "(A)")

  call ifile_append (ifile, "SEQ expr = term addition*")
  call ifile_append (ifile, "SEQ addition = plus_or_minus term")
  call ifile_append (ifile, "SEQ term = factor multiplication*")
  call ifile_append (ifile, "SEQ multiplication = times_or_over factor")
  call ifile_append (ifile, "SEQ factor = atom exponentiation*")
  call ifile_append (ifile, "SEQ exponentiation = '^' atom")
  call ifile_append (ifile, "ALT atom = real | delimited_expr")
  call ifile_append (ifile, "GRO delimited_expr = ( expr )")
  call ifile_append (ifile, "ALT plus_or_minus = '+' | '-'")
  call ifile_append (ifile, "ALT times_or_over = '*' | '/'")
  call ifile_append (ifile, "KEY '+'")
  call ifile_append (ifile, "KEY '-'")
  call ifile_append (ifile, "KEY '*'")
  call ifile_append (ifile, "KEY '/'")
  call ifile_append (ifile, "KEY '^'")
  call ifile_append (ifile, "REA real")

  write (u, "(A)")  "* File contents (syntax definition):"
  call ifile_write (ifile, u)
  write (u, "(A)")  "EOF"
  write (u, "(A)")

  call syntax_init (syntax, ifile)
```

```
  call ifile_final (ifile)
  call syntax_write (syntax, u)
  write (u, "(A)")

  call lexer_init (lexer, &
       comment_chars = "", &
       quote_chars = "'", &
       quote_match = "'", &
       single_chars = "+-*/^()", &
       special_class = [""] , &
       keyword_list = syntax_get_keyword_list_ptr (syntax))
  call lexer_write_setup (lexer, u)
  write (u, "(A)")

  call ifile_append (ifile, "(27+8^3-2/3)*(4+7)^2*99")
  write (u, "(A)")  "* File contents (input file):"
  call ifile_write (ifile, u)
  write (u, "(A)")  "EOF"
  print *

  call stream_init (stream, ifile)
  call lexer_assign_stream (lexer, stream)
  call parse_tree_init (parse_tree, syntax, lexer)
  call stream_final (stream)
  call parse_tree_write (parse_tree, u, .true.)
  print *

  write (u, "(A)")  "* Cleanup, everything should now be empty:"
  write (u, "(A)")

  call parse_tree_final (parse_tree)
  call parse_tree_write (parse_tree, u, .true.)
  write (u, "(A)")

  call lexer_final (lexer)
  call lexer_write_setup (lexer, u)
  write (u, "(A)")

  call ifile_final (ifile)
  write (u, "(A)")  "* File contents:"
  call ifile_write (ifile, u)
  write (u, "(A)")  "EOF"
  write (u, "(A)")

  call syntax_final (syntax)
  call syntax_write (syntax, u)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: parser_1"

end subroutine parse_1
```

7.5 XML Parser

The XML parser is actually independent from the previous modules of lexer and parser. However, for a generic I/O interface we make use of the `stream_t` facility.

We need the XML parser for reading and writing LHEF data files. Only a subset of XML is actually relevant. The parser is of the "pull" type, i.e., the program steers the reading of XML data in a context-sensitive manner.

⟨`xml.f90`⟩≡

```
⟨File header⟩

module xml

⟨Use strings⟩
  use io_units
  use system_defs, only: BLANK, TAB
  use diagnostics
  use ifiles
  use lexers

⟨Standard module head⟩

⟨XML: public⟩

⟨XML: types⟩

contains

⟨XML: procedures⟩

end module xml
```

7.5.1 Cached Stream

The stream type as defined in the `lexer` module is versatile regarding the choice of input channel, but it does not allow reading a section more than once. Here, we define an extension where we can return a string to the stream, which is stored in a cache variable, and presented to the caller for the next read.

⟨*XML: public*⟩≡

```
public :: cstream_t
```

⟨*XML: types*⟩≡

```
type, extends (stream_t) :: cstream_t
   logical :: cache_is_empty = .true.
   type(string_t) :: cache
 contains
 ⟨XML: cstream: TBP⟩
end type cstream_t
```

The initializers are simply inherited.

Finalizer: also inherited, in essence:

⟨*XML: cstream: TBP*⟩≡

```
procedure :: final => cstream_final
```

⟨*XML: procedures*⟩≡
```
subroutine cstream_final (stream)
  class(cstream_t), intent(inout) :: stream
  stream%cache_is_empty = .true.
  call stream%stream_t%final ()
end subroutine cstream_final
```

Get record: now, if there is a cache string, return this instead of the record from the stream.

⟨*XML: cstream: TBP*⟩+≡
```
procedure :: get_record => cstream_get_record
```

⟨*XML: procedures*⟩+≡
```
subroutine cstream_get_record (cstream, string, iostat)
  class(cstream_t), intent(inout) :: cstream
  type(string_t), intent(out) :: string
  integer, intent(out) :: iostat
  if (cstream%cache_is_empty) then
     call stream_get_record (cstream%stream_t, string, iostat)
  else
     string = cstream%cache
     cstream%cache_is_empty = .true.
     iostat = 0
  end if
end subroutine cstream_get_record
```

Revert: return the (partially read) record to the stream, putting it in the cache.

⟨*XML: cstream: TBP*⟩+≡
```
procedure :: revert_record => cstream_revert_record
```

⟨*XML: procedures*⟩+≡
```
subroutine cstream_revert_record (cstream, string)
  class(cstream_t), intent(inout) :: cstream
  type(string_t), intent(in) :: string
  if (cstream%cache_is_empty) then
     cstream%cache = string
     cstream%cache_is_empty = .false.
  else
     call msg_bug ("CStream: attempt to revert twice")
  end if
end subroutine cstream_revert_record
```

7.5.2 Attributes

A tag attribute has a name and a value; both are strings. When the attribute is defined, the `known` flag indicates this.

⟨*XML: types*⟩+≡
```
type :: attribute_t
   type(string_t) :: name
   type(string_t) :: value
   logical :: known = .false.
 contains
```

```
  ⟨XML: attribute: TBP⟩
end type attribute_t
```

Output in standard format, non-advancing. (If the value is unknown, we indicate this by a question mark, which is non-standard.)

⟨*XML: attribute: TBP*⟩≡

```
procedure :: write => attribute_write
```

⟨*XML: procedures*⟩+≡

```
subroutine attribute_write (object, unit)
  class(attribute_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(A,'=')", advance = "no")  char (object%name)
  if (object%known) then
     write (u, "(A,A,A)", advance = "no")  '"', char (object%value), '"'
  else
     write (u, "('?')", advance = "no")
  end if
end subroutine attribute_write
```

This is a genuine constructor. The value is optional.

⟨*XML: public*⟩+≡

```
public :: xml_attribute
```

⟨*XML: procedures*⟩+≡

```
function xml_attribute (name, value) result (attribute)
  type(string_t), intent(in) :: name
  type(string_t), intent(in), optional :: value
  type(attribute_t) :: attribute
  attribute%name = name
  if (present (value)) then
     attribute%value = value
     attribute%known = .true.
  else
     attribute%known = .false.
  end if
end function xml_attribute
```

Set a value explicitly.

⟨*XML: attribute: TBP*⟩+≡

```
procedure :: set_value => attribute_set_value
```

⟨*XML: procedures*⟩+≡

```
subroutine attribute_set_value (attribute, value)
  class(attribute_t), intent(inout) :: attribute
  type(string_t), intent(in) :: value
  attribute%value = value
  attribute%known = .true.
end subroutine attribute_set_value
```

Extract a value. If unset, return "?"

⟨*XML: attribute: TBP*⟩+≡
```
procedure :: get_value => attribute_get_value
```

⟨*XML: procedures*⟩+≡
```
function attribute_get_value (attribute) result (value)
  class(attribute_t), intent(in) :: attribute
  type(string_t) :: value
  if (attribute%known) then
     value = attribute%value
  else
     value = "?"
  end if
end function attribute_get_value
```

7.5.3 The Tag Type

The basic entity is the internal representation of an XML tag. The tag has a name, a well-defined set of attributes which may be mandatory or optional, and a flag that determines whether there is content or not. The content itself is not stored in the data structure.

⟨*XML: public*⟩+≡
```
public :: xml_tag_t
```

⟨*XML: types*⟩+≡
```
type :: xml_tag_t
   type(string_t) :: name
   type(attribute_t), dimension(:), allocatable :: attribute
   logical :: has_content = .false.
 contains
 ⟨XML: tag: TBP⟩
end type xml_tag_t
```

Initialization. There are different versions, depending on content.

⟨*XML: tag: TBP*⟩≡
```
generic :: init => init_no_attributes
procedure :: init_no_attributes => xml_tag_init_no_attributes
```

⟨*XML: procedures*⟩+≡
```
subroutine xml_tag_init_no_attributes (tag, name, has_content)
  class(xml_tag_t), intent(out) :: tag
  type(string_t), intent(in) :: name
  logical, intent(in), optional :: has_content
  tag%name = name
  allocate (tag%attribute (0))
  if (present (has_content))  tag%has_content = has_content
end subroutine xml_tag_init_no_attributes
```

This version sets attributes.

⟨*XML: tag: TBP*⟩+≡
```
generic :: init => init_with_attributes
procedure :: init_with_attributes => xml_tag_init_with_attributes
```

⟨*XML: procedures*⟩+≡

```
subroutine xml_tag_init_with_attributes (tag, name, attribute, has_content)
  class(xml_tag_t), intent(out) :: tag
  type(string_t), intent(in) :: name
  type(attribute_t), dimension(:), intent(in) :: attribute
  logical, intent(in), optional :: has_content
  tag%name = name
  allocate (tag%attribute (size (attribute)))
  tag%attribute = attribute
  if (present (has_content))  tag%has_content = has_content
end subroutine xml_tag_init_with_attributes
```

Set an attribute value explicitly.

⟨*XML: tag: TBP*⟩+≡

```
procedure :: set_attribute => xml_tag_set_attribute
```

⟨*XML: procedures*⟩+≡

```
subroutine xml_tag_set_attribute (tag, i, value)
  class(xml_tag_t), intent(inout) :: tag
  integer, intent(in) :: i
  type(string_t), intent(in) :: value
  call tag%attribute(i)%set_value (value)
end subroutine xml_tag_set_attribute
```

Get an attribute value.

⟨*XML: tag: TBP*⟩+≡

```
procedure :: get_attribute => xml_tag_get_attribute
```

⟨*XML: procedures*⟩+≡

```
function xml_tag_get_attribute (tag, i) result (value)
  class(xml_tag_t), intent(in) :: tag
  integer, intent(in) :: i
  type(string_t) :: value
  value = tag%attribute(i)%get_value ()
end function xml_tag_get_attribute
```

Output to an I/O unit, default STDOUT. We use non-advancing output.

⟨*XML: tag: TBP*⟩+≡

```
generic :: write => write_without_content
procedure :: write_without_content => xml_tag_write
```

⟨*XML: procedures*⟩+≡

```
subroutine xml_tag_write (tag, unit)
  class(xml_tag_t), intent(in) :: tag
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "('<',A)", advance = "no")  char (tag%name)
  do i = 1, size (tag%attribute)
     write (u, "(1x)", advance = "no")
     call tag%attribute(i)%write (u)
  end do
  if (tag%has_content) then
     write (u, "('>')", advance = "no")
```

```
    else
       write (u, "(' />')", advance = "no")
    end if
  end subroutine xml_tag_write
```

If there is content, we should write the context next (arbitrary format), the write the corresponding closing tag. Again, non-advancing.

⟨*XML: tag: TBP*⟩+≡

```
  procedure :: close => xml_tag_close
```

⟨*XML: procedures*⟩+≡

```
  subroutine xml_tag_close (tag, unit)
    class(xml_tag_t), intent(in) :: tag
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "('</',A,'>')", advance = "no")  char (tag%name)
  end subroutine xml_tag_close
```

Given content as a single string, we can write tag, content, and closing at once

⟨*XML: tag: TBP*⟩+≡

```
  generic :: write => write_with_content
  procedure :: write_with_content => xml_tag_write_with_content
```

⟨*XML: procedures*⟩+≡

```
  subroutine xml_tag_write_with_content (tag, content, unit)
    class(xml_tag_t), intent(in) :: tag
    type(string_t), intent(in) :: content
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    call tag%write (u)
    write (u, "(A)", advance = "no")  char (content)
    call tag%close (u)
  end subroutine xml_tag_write_with_content
```

Input from stream. We know what we are looking for, so we check if the name matches, then fill attributes. We report an error if (a) an I/O error occurs, (b) we reach EOF before encountering the tag, (c) if the tag is incomplete.

Trailing text after reading a tag is put back to the input stream.

We assume that the tag is not broken across records, and that there is only one tag within the record. This is more restricted than standard XML.

⟨*XML: tag: TBP*⟩+≡

```
  procedure :: read => xml_tag_read
```

⟨*XML: procedures*⟩+≡

```
  subroutine xml_tag_read (tag, cstream, success)
    class(xml_tag_t), intent(inout) :: tag
    type(cstream_t), intent(inout) :: cstream
    logical, intent(out) :: success
    type(string_t) :: string
    integer :: iostat, p1, p2
    character(2), parameter :: WS = BLANK // TAB
```

```
logical :: done

! Skip comments and blank lines
FIND_NON_COMMENT: do
   FIND_NONEMPTY_RECORD: do
      call cstream%get_record (string, iostat)
      if (iostat /= 0)  call err_io ()
      p1 = verify (string, WS)
      if (p1 > 0)  exit FIND_NONEMPTY_RECORD
   end do FIND_NONEMPTY_RECORD

   ! Look for comment beginning
   p2 = p1 + 3
   if (extract (string, p1, p2) /= "<!--")  exit FIND_NON_COMMENT

   ! Look for comment end, then restart
   string = extract (string, p2 + 1)
   FIND_COMMENT_END: do
      do p1 = 1, len (string) - 2
         p2 = p1 + 2
         if (extract (string, p1, p2) == "-->") then

            ! Return trailing text to the stream
            string = extract (string, p2 + 1)
            if (string /= "")  call cstream%revert_record (string)
            exit FIND_COMMENT_END

         end if
      end do
      call cstream%get_record (string, iostat)
      if (iostat /= 0)  call err_io ()
   end do FIND_COMMENT_END
end do FIND_NON_COMMENT

! Look for opening <
p2 = p1
if (extract (string, p1, p2) /= "<") then
   call cstream%revert_record (string)
   success = .false.;  return
else

   ! Look for tag name
   string = extract (string, p2 + 1)
   p1 = verify (string, WS);  if (p1 == 0)  call err_incomplete ()
   p2 = p1 + len (tag%name) - 1
   if (extract (string, p1, p2) /= tag%name) then
      call cstream%revert_record ("<" // string)
      success = .false.;  return
   else

      ! Look for attributes
      string = extract (string, p2 + 1)
      READ_ATTRIBUTES: do
         call tag%read_attribute (string, done)
```

```
            if (done)  exit READ_ATTRIBUTES
         end do READ_ATTRIBUTES

         ! Look for closing >
         p1 = verify (string, WS);  if (p1 == 0)  call err_incomplete ()
         p2 = p1
         if (extract (string, p1, p1) == ">") then
            tag%has_content = .true.
         else

            ! Look for closing />
            p2 = p1 + 1
            if (extract (string, p1, p2) /= "/>")  call err_incomplete ()
         end if

         ! Return trailing text to the stream
         string = extract (string, p2 + 1)
         if (string /= "")  call cstream%revert_record (string)
         success = .true.

      end if
   end if

contains

   subroutine err_io ()
     select case (iostat)
     case (:-1)
        call msg_fatal ("XML: Error reading tag '" // char (tag%name) &
             // "': end of file")
     case (1:)
        call msg_fatal ("XML: Error reading tag '" // char (tag%name) &
             // "': I/O error")
     end select
     success = .false.
   end subroutine err_io

   subroutine err_incomplete ()
     call msg_fatal ("XML: Error reading tag '" // char (tag%name) &
          // "': tag incomplete")
     success = .false.
   end subroutine err_incomplete

end subroutine xml_tag_read
```

Read a single attribute. If the attribute is valid, assign the value. Setting a value twice should be an error, but is not detected. If the attribute is unknown, ignore it. If we reach the closing sign, report this.

⟨*XML: tag: TBP*⟩+≡

```
procedure :: read_attribute => xml_tag_read_attribute
```

⟨*XML: procedures*⟩+≡

```
subroutine xml_tag_read_attribute (tag, string, done)
  class(xml_tag_t), intent(inout) :: tag
```

```
  type(string_t), intent(inout) :: string
  logical, intent(out) :: done
  character(2), parameter :: WS = BLANK // TAB
  type(string_t) :: name, value
  integer :: p1, p2, i

  p1 = verify (string, WS);  if (p1 == 0)  call err ()
  p2 = p1

  ! Look for first terminating '>' or '/>'
  if (extract (string, p1, p2) == ">") then
     done = .true.
  else
     p2 = p1 + 1
     if (extract (string, p1, p2) == "/>") then
        done = .true.
     else

        ! Look for '='
        p2 = scan (string, '=')
        if (p2 == 0)  call err ()
        name = trim (extract (string, p1, p2 - 1))

        ! Look for '"'
        string = extract (string, p2 + 1)
        p1 = verify (string, WS);  if (p1 == 0)  call err ()
        p2 = p1
        if (extract (string, p1, p2) /= '"')  call err ()

        ! Look for matching '"' and get value
        string = extract (string, p2 + 1)
        p1 = 1
        p2 = scan (string, '"')
        if (p2 == 0)  call err ()
        value = extract (string, p1, p2 - 1)

        SCAN_KNOWN_ATTRIBUTES: do i = 1, size (tag%attribute)
           if (name == tag%attribute(i)%name) then
              call tag%attribute(i)%set_value (value)
              exit SCAN_KNOWN_ATTRIBUTES
           end if
        end do SCAN_KNOWN_ATTRIBUTES

        string = extract (string, p2 + 1)
        done = .false.
     end if
  end if

contains

  subroutine err ()
    call msg_fatal ("XML: Error reading attributes of '" // char (tag%name) &
         // "': syntax error")
  end subroutine err
```

```
end subroutine xml_tag_read_attribute
```

Read the content string of a tag. We check for the appropriate closing tag and report it. If a closing tag does not match in name, ignore it.

Note: this assumes that no tag with the same name is embedded in the current content. Also, we do not check for XML validity inside the content.

⟨*XML: tag: TBP*⟩+≡

```
procedure :: read_content => xml_tag_read_content
```

⟨*XML: procedures*⟩+≡

```
subroutine xml_tag_read_content (tag, cstream, content, closing)
  class(xml_tag_t), intent(in) :: tag
  type(cstream_t), intent(inout) :: cstream
  type(string_t), intent(out) :: content
  type(string_t) :: string
  logical, intent(out) :: closing
  integer :: iostat
  integer :: p0, p1, p2
  character(2), parameter :: WS = BLANK // TAB
  call cstream%get_record (content, iostat)
  if (iostat /= 0)  call err_io ()
  closing = .false.
  FIND_CLOSING: do p0 = 1, len (content) - 1

     ! Look for terminating </
     p1 = p0
     p2 = p1 + 1
     if (extract (content, p1, p2) == "</") then

        ! Look for closing tag name
        string = extract (content, p2 + 1)
        p1 = verify (string, WS);  if (p1 == 0)  call err_incomplete ()
        p2 = p1 + len (tag%name) - 1
        if (extract (string, p1, p2) == tag%name) then

           ! Tag name matches: look for final >
           string = extract (string, p2 + 1)
           p1 = verify (string, WS);  if (p1 == 0)  call err_incomplete ()
           p2 = p1
           if (extract (string, p1, p2) /= ">")  call err_incomplete ()

           ! Return trailing text to the stream
           string = extract (string, p2 + 1)
           if (string /= "")  call cstream%revert_record (string)
           content = extract (content, 1, p0 -1)
           closing = .true.
           exit FIND_CLOSING

        end if
     end if
  end do FIND_CLOSING

contains
```

```
    subroutine err_io ()
      select case (iostat)
      case (:-1)
         call msg_fatal ("XML: Error reading content of '" // char (tag%name) &
              // "': end of file")
      case (1:)
         call msg_fatal ("XML: Error reading content of '" // char (tag%name) &
              // "': I/O error")
      end select
      closing = .false.
    end subroutine err_io

    subroutine err_incomplete ()
      call msg_fatal ("XML: Error reading content '" // char (tag%name) &
           // "': closing tag incomplete")
      closing = .false.
    end subroutine err_incomplete

  end subroutine xml_tag_read_content
```

7.5.4 Unit tests

Test module, followed by the corresponding implementation module.

⟨xml_ut.f90⟩≡

```
  ⟨File header⟩

  module xml_ut
    use unit_tests
    use xml_uti

  ⟨Standard module head⟩

  ⟨XML: public test⟩

  contains

  ⟨XML: test driver⟩

  end module xml_ut
```

⟨xml_uti.f90⟩≡

```
  ⟨File header⟩

  module xml_uti

  ⟨Use strings⟩
    use io_units

    use xml

  ⟨Standard module head⟩
```

```
⟨XML: test declarations⟩

contains

⟨XML: tests⟩

⟨XML: test auxiliary⟩

end module xml_uti
```

API: driver for the unit tests below.

⟨*XML: public test*⟩≡

```
    public :: xml_test
```

⟨*XML: test driver*⟩≡

```
    subroutine xml_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨XML: execute tests⟩
  end subroutine xml_test
```

7.5.5 Auxiliary Routines

Show the contents of a temporary file, i.e., open unit.

⟨*XML: test auxiliary*⟩≡

```
    subroutine show (u_tmp, u)
      integer, intent(in) :: u_tmp, u
      character (80) :: buffer
      integer :: iostat
      write (u, "(A)")  "File content:"
      rewind (u_tmp)
      do
         read (u_tmp, "(A)", iostat = iostat)  buffer
         if (iostat /= 0)  exit
         write (u, "(A)")  trim (buffer)
      end do
      rewind (u_tmp)
    end subroutine show
```

7.5.6 Basic Tag I/O

Write a tag and read it again, using a temporary file.

⟨*XML: execute tests*⟩≡

```
    call test (xml_1, "xml_1", &
         "basic I/O", &
         u, results)
```

⟨*XML: test declarations*⟩≡

```
    public :: xml_1
```

⟨*XML: tests*⟩≡

```
subroutine xml_1 (u)
  integer, intent(in) :: u
  type(xml_tag_t), allocatable :: tag
  integer :: u_tmp
  type(cstream_t) :: cstream
  logical :: success

  write (u, "(A)")  "* Test output: xml_1"
  write (u, "(A)")  "*   Purpose: write and read tag"
  write (u, "(A)")

  write (u, "(A)")  "* Empty tag"
  write (u, *)

  u_tmp = free_unit ()
  open (u_tmp, status = "scratch", action = "readwrite")

  allocate (tag)
  call tag%init (var_str ("tagname"))
  call tag%write (u_tmp)
  write (u_tmp, *)
  deallocate (tag)

  call show (u_tmp, u)

  write (u, *)
  write (u, "(A)")  "Result from read:"
  call cstream%init (u_tmp)
  allocate (tag)
  call tag%init (var_str ("tagname"))
  call tag%read (cstream, success)
  call tag%write (u)
  write (u, *)
  write (u, "(A,L1)")  "success = ", success
  deallocate (tag)
  close (u_tmp)
  call cstream%final ()

  write (u, *)
  write (u, "(A)")  "* Tag with preceding blank lines"
  write (u, *)

  u_tmp = free_unit ()
  open (u_tmp, status = "scratch", action = "readwrite")

  allocate (tag)
  call tag%init (var_str ("tagname"))
  write (u_tmp, *)
  write (u_tmp, "(A)")  "    "
  write (u_tmp, "(2x)", advance = "no")
  call tag%write (u_tmp)
  write (u_tmp, *)
  deallocate (tag)
```

```
call show (u_tmp, u)

write (u, *)
write (u, "(A)")  "Result from read:"
call cstream%init (u_tmp)
allocate (tag)
call tag%init (var_str ("tagname"))
call tag%read (cstream, success)
call tag%write (u)
write (u, *)
write (u, "(A,L1)")  "success = ", success
deallocate (tag)
close (u_tmp)
call cstream%final ()

write (u, *)
write (u, "(A)")  "* Tag with preceding comments"
write (u, *)

u_tmp = free_unit ()
open (u_tmp, status = "scratch", action = "readwrite")

allocate (tag)
call tag%init (var_str ("tagname"))
write (u_tmp, "(A)")  "<!-- comment -->"
write (u_tmp, *)
write (u_tmp, "(A)")  "<!-- multiline"
write (u_tmp, "(A)")  "     comment -->"
call tag%write (u_tmp)
write (u_tmp, *)
deallocate (tag)

call show (u_tmp, u)

write (u, *)
write (u, "(A)")  "Result from read:"
call cstream%init (u_tmp)
allocate (tag)
call tag%init (var_str ("tagname"))
call tag%read (cstream, success)
call tag%write (u)
write (u, *)
write (u, "(A,L1)")  "success = ", success
close (u_tmp)
deallocate (tag)

call cstream%final ()

write (u, *)
write (u, "(A)")  "* Tag with name mismatch"
write (u, *)

u_tmp = free_unit ()
```

```
    open (u_tmp, status = "scratch", action = "readwrite")

    allocate (tag)
    call tag%init (var_str ("wrongname"))
    call tag%write (u_tmp)
    write (u_tmp, *)
    deallocate (tag)

    call show (u_tmp, u)

    write (u, *)
    write (u, "(A)")  "Result from read:"
    call cstream%init (u_tmp)
    allocate (tag)
    call tag%init (var_str ("tagname"))
    call tag%read (cstream, success)
    call tag%write (u)
    write (u, *)
    write (u, "(A,L1)")  "success = ", success
    deallocate (tag)
    close (u_tmp)
    call cstream%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: xml_1"

  end subroutine xml_1
```

7.5.7 Optional Tag

Write and read two tags, one of them optional.

⟨*XML: execute tests*⟩+≡

```
  call test (xml_2, "xml_2", &
       "optional tag", &
       u, results)
```

⟨*XML: test declarations*⟩+≡

```
  public :: xml_2
```

⟨*XML: tests*⟩+≡

```
  subroutine xml_2 (u)
    integer, intent(in) :: u
    type(xml_tag_t), allocatable :: tag1, tag2
    integer :: u_tmp
    type(cstream_t) :: cstream
    logical :: success

    write (u, "(A)")  "* Test output: xml_2"
    write (u, "(A)")  "*   Purpose: handle optional tag"
    write (u, "(A)")

    write (u, "(A)")  "* Optional tag present"
    write (u, *)
```

```
u_tmp = free_unit ()
open (u_tmp, status = "scratch", action = "readwrite")

allocate (tag1)
call tag1%init (var_str ("option"))
call tag1%write (u_tmp)
write (u_tmp, *)
allocate (tag2)
call tag2%init (var_str ("tagname"))
call tag2%write (u_tmp)
write (u_tmp, *)
deallocate (tag1, tag2)

call show (u_tmp, u)

write (u, *)
write (u, "(A)")  "Result from read:"
call cstream%init (u_tmp)
allocate (tag1)
call tag1%init (var_str ("option"))
call tag1%read (cstream, success)
call tag1%write (u)
write (u, *)
write (u, "(A,L1)")  "success = ", success
write (u, *)
allocate (tag2)
call tag2%init (var_str ("tagname"))
call tag2%read (cstream, success)
call tag2%write (u)
write (u, *)
write (u, "(A,L1)")  "success = ", success
deallocate (tag1, tag2)
close (u_tmp)
call cstream%final ()

write (u, *)
write (u, "(A)")  "* Optional tag absent"
write (u, *)

u_tmp = free_unit ()
open (u_tmp, status = "scratch", action = "readwrite")

allocate (tag2)
call tag2%init (var_str ("tagname"))
call tag2%write (u_tmp)
write (u_tmp, *)
deallocate (tag2)

call show (u_tmp, u)

write (u, *)
write (u, "(A)")  "Result from read:"
call cstream%init (u_tmp)
```

```
    allocate (tag1)
    call tag1%init (var_str ("option"))
    call tag1%read (cstream, success)
    call tag1%write (u)
    write (u, *)
    write (u, "(A,L1)")  "success = ", success
    write (u, *)
    allocate (tag2)
    call tag2%init (var_str ("tagname"))
    call tag2%read (cstream, success)
    call tag2%write (u)
    write (u, *)
    write (u, "(A,L1)")  "success = ", success
    deallocate (tag1, tag2)
    close (u_tmp)
    call cstream%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: xml_2"

  end subroutine xml_2
```

7.5.8 Optional Tag

Write and read a tag with single-line content.

⟨*XML: execute tests*⟩+≡

```
  call test (xml_3, "xml_3", &
       "content", &
       u, results)
```

⟨*XML: test declarations*⟩+≡

```
  public :: xml_3
```

⟨*XML: tests*⟩+≡

```
  subroutine xml_3 (u)
    integer, intent(in) :: u
    type(xml_tag_t), allocatable :: tag
    integer :: u_tmp
    type(cstream_t) :: cstream
    logical :: success, closing
    type(string_t) :: content

    write (u, "(A)")  "* Test output: xml_3"
    write (u, "(A)")  "*   Purpose: handle tag with content"
    write (u, "(A)")

    write (u, "(A)")  "* Tag without content"
    write (u, *)

    u_tmp = free_unit ()
    open (u_tmp, status = "scratch", action = "readwrite")

    allocate (tag)
```

```
call tag%init (var_str ("tagname"))
call tag%write (u_tmp)
write (u_tmp, *)
deallocate (tag)

call show (u_tmp, u)

write (u, *)
write (u, "(A)")  "Result from read:"
call cstream%init (u_tmp)
allocate (tag)
call tag%init (var_str ("tagname"))
call tag%read (cstream, success)
call tag%write (u)
write (u, *)
write (u, "(A,L1)")  "success = ", success
write (u, "(A,L1)")  "content = ", tag%has_content
write (u, *)
deallocate (tag)
close (u_tmp)
call cstream%final ()

write (u, "(A)")  "* Tag with content"
write (u, *)

u_tmp = free_unit ()
open (u_tmp, status = "scratch", action = "readwrite")

allocate (tag)
call tag%init (var_str ("tagname"), has_content = .true.)
call tag%write (var_str ("Content text"), u_tmp)
write (u_tmp, *)
deallocate (tag)

call show (u_tmp, u)

write (u, *)
write (u, "(A)")  "Result from read:"
call cstream%init (u_tmp)
allocate (tag)
call tag%init (var_str ("tagname"))
call tag%read (cstream, success)
call tag%read_content (cstream, content, closing)
call tag%write (u)
write (u, "(A)", advance = "no")  char (content)
call tag%close (u)
write (u, *)
write (u, "(A,L1)")  "success = ", success
write (u, "(A,L1)")  "content = ", tag%has_content
write (u, "(A,L1)")  "closing = ", closing
deallocate (tag)
close (u_tmp)
call cstream%final ()
```

```
    write (u, *)
    write (u, "(A)")  "* Tag with multiline content"
    write (u, *)

    u_tmp = free_unit ()
    open (u_tmp, status = "scratch", action = "readwrite")

    allocate (tag)
    call tag%init (var_str ("tagname"), has_content = .true.)
    call tag%write (u_tmp)
    write (u_tmp, *)
    write (u_tmp, "(A)")  "Line 1"
    write (u_tmp, "(A)")  "Line 2"
    call tag%close (u_tmp)
    write (u_tmp, *)
    deallocate (tag)

    call show (u_tmp, u)

    write (u, *)
    write (u, "(A)")  "Result from read:"
    call cstream%init (u_tmp)
    allocate (tag)
    call tag%init (var_str ("tagname"))
    call tag%read (cstream, success)
    call tag%write (u)
    write (u, *)
    do
       call tag%read_content (cstream, content, closing)
       if (closing)  exit
       write (u, "(A)")  char (content)
    end do
    call tag%close (u)
    write (u, *)
    write (u, "(A,L1)")  "success = ", success
    write (u, "(A,L1)")  "content = ", tag%has_content
    deallocate (tag)
    close (u_tmp)
    call cstream%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: xml_3"

  end subroutine xml_3
```

7.5.9 Basic Tag I/O

Write a tag and read it again, using a temporary file.

⟨*XML: execute tests*⟩+≡

```
  call test (xml_4, "xml_4", &
       "attributes", &
       u, results)
```

⟨*XML: test declarations*⟩+≡

```
public :: xml_4
```

⟨*XML: tests*⟩+≡

```
subroutine xml_4 (u)
  integer, intent(in) :: u
  type(xml_tag_t), allocatable :: tag
  integer :: u_tmp
  type(cstream_t) :: cstream
  logical :: success

  write (u, "(A)")  "* Test output: xml_4"
  write (u, "(A)")  "*   Purpose: handle tag with attributes"
  write (u, "(A)")

  write (u, "(A)")  "* Tag with one mandatory and one optional attribute,"
  write (u, "(A)")  "* unknown attribute ignored"
  write (u, *)

  u_tmp = free_unit ()
  open (u_tmp, status = "scratch", action = "readwrite")

  allocate (tag)
  call tag%init (var_str ("tagname"), &
       [xml_attribute (var_str ("a1"), var_str ("foo")), &
        xml_attribute (var_str ("a3"), var_str ("gee"))])
  call tag%write (u_tmp)
  deallocate (tag)

  call show (u_tmp, u)

  write (u, *)
  write (u, "(A)")  "Result from read:"
  call cstream%init (u_tmp)
  allocate (tag)
  call tag%init (var_str ("tagname"), &
       [xml_attribute (var_str ("a1")), &
        xml_attribute (var_str ("a2"), var_str ("bar"))])
  call tag%read (cstream, success)
  call tag%write (u)
  write (u, *)
  deallocate (tag)
  close (u_tmp)
  call cstream%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: xml_4"

end subroutine xml_4
```

Chapter 8

Random-Number Generator

These modules implement abstract types and tools for random-number generation.

rng_base Abstract random-number generator and factory

selectors Selection depending on weights and random numbers

Implementation of the RNG abstract types:

Module `rng_tao`: Interface to the TAO random number generator which the VAMP package provides. Note that VAMP explicitly requests this generator.

Module `rng{stream`: Implementation of RNGstream proposed by L'Ecuyer. The RNGStream provides access to different streams and/or substream of random numbers.

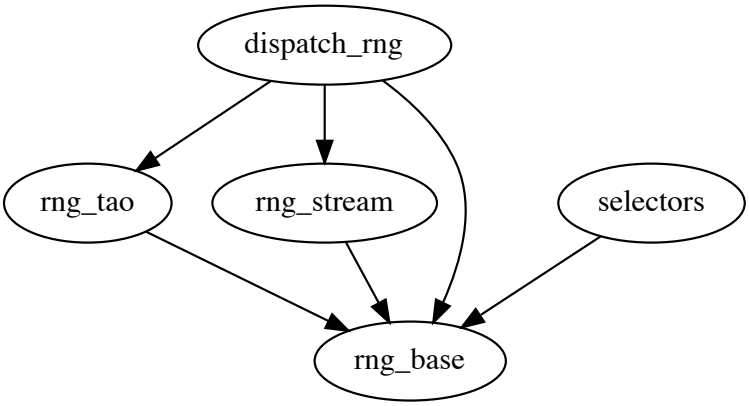

Figure 8.1: Module dependencies in `src/rng`.

8.1 Generic Random-Number Generator

For all generator implementations, we define a `rng` type which represents the state of a random-number generator with the associated methods that produce a random number. Furthermore, we define a `rng_factory` type. An object of this type is capable of allocating a sequence of `rng` objects. These generator states should be, if possible, statistically independent, so they can be used in parallel in different places of the event-generation chain.

⟨`rng_base.f90`⟩≡

```
⟨File header⟩

module rng_base

⟨Use kinds⟩
  use kinds, only: i16
  use constants, only: TWOPI

⟨Standard module head⟩

⟨RNG base: public⟩

⟨RNG base: types⟩

⟨RNG base: interfaces⟩

contains

⟨RNG base: procedures⟩

end module rng_base
```

8.1.1 Generator type

The rng object is actually the state of the random-number generator. The methods initialize/reset and call the generator for this state.

⟨*RNG base: public*⟩≡

```
public :: rng_t
```

⟨*RNG base: types*⟩≡

```
type, abstract :: rng_t
 contains
 ⟨RNG base: rng: TBP⟩
end type rng_t
```

The `init` method initializes the generator and sets a seed. We should implement the interface such that a single integer is sufficient for a seed.

The seed may be omitted. The behavior without seed is not defined, however.

⟨*RNG base: rng: TBP*⟩≡

```
procedure (rng_init), deferred :: init
```

⟨*RNG base: interfaces*⟩≡

```
abstract interface
   subroutine rng_init (rng, seed)
```

```
      import
      class(rng_t), intent(out) :: rng
      integer, intent(in), optional :: seed
   end subroutine rng_init
end interface
```

The `final` method deallocates memory where necessary and allows for another call of `init` to reset the generator.

⟨*RNG base: rng: TBP*⟩+≡

```
procedure (rng_final), deferred :: final
```

⟨*RNG base: interfaces*⟩+≡

```
abstract interface
   subroutine rng_final (rng)
      import
      class(rng_t), intent(inout) :: rng
   end subroutine rng_final
end interface
```

Output. We should, at least, identify the generator.

⟨*RNG base: rng: TBP*⟩+≡

```
procedure (rng_write), deferred :: write
```

⟨*RNG base: interfaces*⟩+≡

```
abstract interface
   subroutine rng_write (rng, unit, indent)
      import
      class(rng_t), intent(in) :: rng
      integer, intent(in), optional :: unit, indent
   end subroutine rng_write
end interface
```

These routines generate a single and an array of uniformly distributed default-precision random numbers, respectively.

⟨*RNG base: rng: TBP*⟩+≡

```
generic :: generate => generate_single, generate_array
procedure (rng_generate_single), deferred :: generate_single
procedure (rng_generate_array), deferred :: generate_array
```

⟨*RNG base: interfaces*⟩+≡

```
abstract interface
   subroutine rng_generate_single (rng, x)
      import
      class(rng_t), intent(inout) :: rng
      real(default), intent(out) :: x
   end subroutine rng_generate_single
end interface

abstract interface
   subroutine rng_generate_array (rng, x)
      import
      class(rng_t), intent(inout) :: rng
      real(default), dimension(:), intent(out) :: x
```

```
    end subroutine rng_generate_array
end interface
```

These routines generate a single and an array of Gaussian (normal) distributed default-precision random numbers, respectively. Mean is 0 and $\sigma = 1$. Note that $z = \mu + \sigma x$ then distributes with mean μ and variance σ^2.

The algorithm uses twice as much uniformly distributed random numbers, taken from the PDG review.

⟨*RNG base: rng: TBP*⟩+≡

```
generic :: generate_gaussian => &
     rng_generate_gaussian_single, rng_generate_gaussian_array
procedure, private :: rng_generate_gaussian_single
procedure, private :: rng_generate_gaussian_array
```

⟨*RNG base: procedures*⟩≡

```
subroutine rng_generate_gaussian_single (rng, x)
  class(rng_t), intent(inout) :: rng
  real(default), intent(out) :: x
  real(default), dimension(2) :: u
  call rng%generate (u)
  x = sin (twopi * u(1)) * sqrt (- 2 * log (u(2)))
end subroutine rng_generate_gaussian_single

subroutine rng_generate_gaussian_array (rng, x)
  class(rng_t), intent(inout) :: rng
  real(default), dimension(:), intent(out) :: x
  integer :: i
  do i = 1, size (x)
     call rng%generate_gaussian (x(i))
  end do
end subroutine rng_generate_gaussian_array
```

8.1.2 RNG Factory

A factory object has a `make` method that allocates and initializes a new generator of appropriate type. It uses a 16-bit integer for initialization. For a real-life implementation, the factory should return a sequence of statistically independent generators, and for different seeds, the sequences should also be independent.

⟨*RNG base: public*⟩+≡

```
public :: rng_factory_t
```

⟨*RNG base: types*⟩+≡

```
type, abstract :: rng_factory_t
 contains
 ⟨RNG base: rng factory: TBP⟩
end type rng_factory_t
```

Output. Should be short, just report the seed and current state of the factory.

⟨*RNG base: rng factory: TBP*⟩≡

```
procedure (rng_factory_write), deferred :: write
```

⟨*RNG base: interfaces*⟩+≡

```
abstract interface
   subroutine rng_factory_write (object, unit)
     import
     class(rng_factory_t), intent(in) :: object
     integer, intent(in), optional :: unit
   end subroutine rng_factory_write
end interface
```

Initialize. It should be possible to do this repeatedly, resetting the state. The default seed should be 0.

⟨*RNG base: rng factory: TBP*⟩+≡

```
procedure (rng_factory_init), deferred :: init
```

⟨*RNG base: interfaces*⟩+≡

```
abstract interface
   subroutine rng_factory_init (factory, seed)
     import
     class(rng_factory_t), intent(out) :: factory
     integer(i16), intent(in), optional :: seed
   end subroutine rng_factory_init
end interface
```

Spawn a new generator.

⟨*RNG base: rng factory: TBP*⟩+≡

```
procedure (rng_factory_make), deferred :: make
```

⟨*RNG base: interfaces*⟩+≡

```
abstract interface
   subroutine rng_factory_make (factory, rng)
     import
     class(rng_factory_t), intent(inout) :: factory
     class(rng_t), intent(out), allocatable :: rng
   end subroutine rng_factory_make
end interface
```

8.1.3 Unit tests

Test module, followed by the corresponding implementation module.

⟨`rng_base_ut.f90`⟩≡

```
⟨File header⟩

module rng_base_ut
  use unit_tests
  use rng_base_uti

⟨Standard module head⟩

⟨RNG base: public test⟩

⟨RNG base: public test auxiliary⟩
```

```
contains

⟨RNG base: test driver⟩

end module rng_base_ut
```

⟨rng_base_uti.f90⟩≡

```
⟨File header⟩

module rng_base_uti

⟨Use kinds⟩
  use kinds, only: i16
  use format_utils, only: write_indent
  use io_units

  use rng_base

⟨Standard module head⟩

⟨RNG base: public test auxiliary⟩

⟨RNG base: test declarations⟩

⟨RNG base: test types⟩

contains

⟨RNG base: tests⟩

⟨RNG base: test auxiliary⟩

end module rng_base_uti
```

API: driver for the unit tests below.

⟨RNG base: public test⟩≡

```
public :: rng_base_test
```

⟨RNG base: test driver⟩≡

```
subroutine rng_base_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨RNG base: execute tests⟩
end subroutine rng_base_test
```

Test generator

The test 'mock' random generator generates a repeating series with the numbers $0.1, 0.3, 0.5, 0.7, 0.9$. It has an integer stored as state. The integer must be one of $1, 3, 5, 7, 9$.

⟨RNG base: public test auxiliary⟩≡

```
public :: rng_test_t
```

⟨*RNG base: test types*⟩≡

```
type, extends (rng_t) :: rng_test_t
   integer :: state = 1
 contains
   procedure :: write => rng_test_write
   procedure :: init => rng_test_init
   procedure :: final => rng_test_final
   procedure :: generate_single => rng_test_generate_single
   procedure :: generate_array => rng_test_generate_array
end type rng_test_t
```

Output. The state is a single number, so print it.

⟨*RNG base: test auxiliary*⟩≡

```
subroutine rng_test_write (rng, unit, indent)
  class(rng_test_t), intent(in) :: rng
  integer, intent(in), optional :: unit, indent
  integer :: u, ind
  u = given_output_unit (unit)
  ind = 0;  if (present (indent))  ind = indent
  call write_indent (u, ind)
  write (u, "(A,I0,A)")  "Random-number generator: &
       &test (state = ", rng%state, ")"
end subroutine rng_test_write
```

The default seed is 1.

⟨*RNG base: test auxiliary*⟩+≡

```
subroutine rng_test_init (rng, seed)
  class(rng_test_t), intent(out) :: rng
  integer, intent(in), optional :: seed
  if (present (seed))  rng%state = seed
end subroutine rng_test_init
```

Nothing to finalize:

⟨*RNG base: test auxiliary*⟩+≡

```
subroutine rng_test_final (rng)
  class(rng_test_t), intent(inout) :: rng
end subroutine rng_test_final
```

Generate a single number and advance the state.

⟨*RNG base: test auxiliary*⟩+≡

```
subroutine rng_test_generate_single (rng, x)
  class(rng_test_t), intent(inout) :: rng
  real(default), intent(out) :: x
  x = rng%state / 10._default
  rng%state = mod (rng%state + 2, 10)
end subroutine rng_test_generate_single
```

The array generator calls the single-item generator multiple times.

⟨*RNG base: test auxiliary*⟩+≡

```
subroutine rng_test_generate_array (rng, x)
  class(rng_test_t), intent(inout) :: rng
```

```
    real(default), dimension(:), intent(out) :: x
    integer :: i
    do i = 1, size (x)
       call rng%generate (x(i))
    end do
  end subroutine rng_test_generate_array
```

Test Factory

This factory makes rng_test_t generators, initialized with integers 1, 3, 5, 7, 9 if given the input 0, 1, 2, 3, 4. The generators within one sequence are all identical, however.

⟨*RNG base: public test auxiliary*⟩+≡

```
  public :: rng_test_factory_t
```

⟨*RNG base: test types*⟩+≡

```
  type, extends (rng_factory_t) :: rng_test_factory_t
     integer :: seed = 1
   contains
   ⟨RNG base: rng test factory: TBP⟩
  end type rng_test_factory_t
```

Output.

⟨*RNG base: rng test factory: TBP*⟩≡

```
  procedure :: write => rng_test_factory_write
```

⟨*RNG base: test auxiliary*⟩+≡

```
  subroutine rng_test_factory_write (object, unit)
    class(rng_test_factory_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(1x,A,I0,A)")  "RNG factory: test (", object%seed, ")"
  end subroutine rng_test_factory_write
```

Initialize, translating the given seed.

⟨*RNG base: rng test factory: TBP*⟩+≡

```
  procedure :: init => rng_test_factory_init
```

⟨*RNG base: test auxiliary*⟩+≡

```
  subroutine rng_test_factory_init (factory, seed)
    class(rng_test_factory_t), intent(out) :: factory
    integer(i16), intent(in), optional :: seed
    if (present (seed))  factory%seed = mod (seed * 2 + 1, 10)
  end subroutine rng_test_factory_init
```

⟨*RNG base: rng test factory: TBP*⟩+≡

```
  procedure :: make => rng_test_factory_make
```

⟨*RNG base: test auxiliary*⟩+≡

```
subroutine rng_test_factory_make (factory, rng)
  class(rng_test_factory_t), intent(inout) :: factory
  class(rng_t), intent(out), allocatable :: rng
  allocate (rng_test_t :: rng)
  select type (rng)
  type is (rng_test_t)
     call rng%init (int (factory%seed))
  end select
end subroutine rng_test_factory_make
```

Generator check

Initialize the generator and draw random numbers.

⟨*RNG base: execute tests*⟩≡

```
call test (rng_base_1, "rng_base_1", &
     "rng initialization and call", &
     u, results)
```

⟨*RNG base: test declarations*⟩≡

```
public :: rng_base_1
```

⟨*RNG base: tests*⟩≡

```
subroutine rng_base_1 (u)
  integer, intent(in) :: u
  class(rng_t), allocatable :: rng

  real(default) :: x
  real(default), dimension(2) :: x2

  write (u, "(A)")  "* Test output: rng_base_1"
  write (u, "(A)")  "*   Purpose: initialize and call a test random-number &
       &generator"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize generator"
  write (u, "(A)")

  allocate (rng_test_t :: rng)
  call rng%init (3)

  call rng%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Get random number"
  write (u, "(A)")

  call rng%generate (x)
  write (u, "(A,2(1x,F9.7))")  "x =", x

  write (u, "(A)")
  write (u, "(A)")  "* Get random number pair"
  write (u, "(A)")
```

```
    call rng%generate (x2)
    write (u, "(A,2(1x,F9.7))")  "x =", x2

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call rng%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rng_base_1"

  end subroutine rng_base_1
```

Factory check

Set up a factory and spawn generators.

⟨*RNG base: execute tests*⟩+≡

```
  call test (rng_base_2, "rng_base_2", &
       "rng factory", &
       u, results)
```

⟨*RNG base: test declarations*⟩+≡

```
  public :: rng_base_2
```

⟨*RNG base: tests*⟩+≡

```
  subroutine rng_base_2 (u)
    integer, intent(in) :: u
    type(rng_test_factory_t) :: rng_factory
    class(rng_t), allocatable :: rng

    write (u, "(A)")  "* Test output: rng_base_2"
    write (u, "(A)")  "*   Purpose: initialize and use a rng factory"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize factory"
    write (u, "(A)")

    call rng_factory%init ()
    call rng_factory%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Make a generator"
    write (u, "(A)")

    call rng_factory%make (rng)
    call rng%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call rng%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rng_base_2"
```

```
end subroutine rng_base_2
```

8.2 Select from a weighted sample

⟨selectors.f90⟩≡
```
⟨File header⟩

module selectors

⟨Use kinds⟩
  use io_units
  use diagnostics
  use rng_base

⟨Standard module head⟩

⟨Selectors: public⟩

⟨Selectors: types⟩

contains

⟨Selectors: procedures⟩

end module selectors
```

8.2.1 Selector type

The rng object is actually the state of the random-number generator. The methods initialize/reset and call the generator for this state.

⟨*Selectors: public*⟩≡
```
public :: selector_t
```

⟨*Selectors: types*⟩≡
```
type :: selector_t
   integer, dimension(:), allocatable :: map
   real(default), dimension(:), allocatable :: weight
   real(default), dimension(:), allocatable :: acc
 contains
 ⟨Selectors: selector: TBP⟩
end type selector_t
```

Display contents.

⟨*Selectors: selector: TBP*⟩≡
```
procedure :: write => selector_write
```

⟨*Selectors: procedures*⟩≡
```
subroutine selector_write (object, unit)
  class(selector_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Selector: i, weight, acc. weight"
  if (allocated (object%weight)) then
     do i = 1, size (object%weight)
        write (u, "(3x,I0,1x,ES19.12,1x,ES19.12)") &
```

```
            object%map(i), object%weight(i), object%acc(i)
      end do
   else
      write (u, "(3x,A)")  "[undefined]"
   end if
end subroutine selector_write
```

We pack the input weight array such that zero-weight entries are removed. We also normalize it. This makes a `map` array for mapping the selected weight to the actual entry necessary.

We may encounter a case where all weights are zero. We do not reject this, but set up the selector so that it always returns the first entry.

⟨*Selectors: selector: TBP*⟩+≡

```
procedure :: init => selector_init
```

⟨*Selectors: procedures*⟩+≡

```
subroutine selector_init (selector, weight, negative_weights)
  class(selector_t), intent(out) :: selector
  real(default), dimension(:), intent(in) :: weight
  logical, intent(in), optional :: negative_weights
  real(default) :: s
  integer :: n, i
  logical :: neg_wgt
  logical, dimension(:), allocatable :: mask
  if (size (weight) == 0) &
       call msg_bug ("Selector init: zero-size weight array")
  neg_wgt = .false.
  if (present (negative_weights))  neg_wgt = negative_weights
  if (.not. neg_wgt .and. any (weight < 0)) &
       call msg_fatal ("Selector init: negative weight encountered")
  s = sum (weight)
  allocate (mask (size (weight)), &
       source = weight /= 0)
  n = count (mask)
  if (n > 0) then
     allocate (selector%map (n), &
          source = pack ([(i, i = 1, size (weight))], mask))
     allocate (selector%weight (n), &
          source = pack (abs (weight) / s, mask))
     allocate (selector%acc (n))
     selector%acc(1) = selector%weight(1)
     do i = 2, n - 1
        selector%acc(i) = selector%acc(i-1) + selector%weight(i)
     end do
     selector%acc(n) = 1
  else
     allocate (selector%map (1), source = 1)
     allocate (selector%weight (1), source = 0._default)
     allocate (selector%acc (1), source = 1._default)
  end if
end subroutine selector_init
```

Select an entry based upon the number `x`, which should be a uniformly dis-

tributed random number between 0 and 1.

⟨*Selectors: selector: TBP*⟩+≡

```
procedure :: select => selector_select
```

⟨*Selectors: procedures*⟩+≡

```
function selector_select (selector, x) result (n)
  class(selector_t), intent(in) :: selector
  real(default), intent(in) :: x
  integer :: n
  integer :: i
  if (x < 0 .or. x > 1) &
       call msg_bug ("Selector: random number out of range")
  do i = 1, size (selector%acc)
     if (x <= selector%acc(i))  exit
  end do
  n = selector%map(i)
end function selector_select
```

Use the provided random-number generator to select an entry. (Unless there is only one entry.)

⟨*Selectors: selector: TBP*⟩+≡

```
procedure :: generate => selector_generate
```

⟨*Selectors: procedures*⟩+≡

```
subroutine selector_generate (selector, rng, n)
  class(selector_t), intent(in) :: selector
  class(rng_t), intent(inout) :: rng
  integer, intent(out) :: n
  real(default) :: x
  select case (size (selector%acc))
  case (1);  n = 1
  case default
     call rng%generate (x)
     n = selector%select (x)
  end select
end subroutine selector_generate
```

Determine the normalized weight for a selected entry. We use a linear search for the inverse lookup, assuming that efficiency is not an issue for this function.

⟨*Selectors: selector: TBP*⟩+≡

```
procedure :: get_weight => selector_get_weight
```

⟨*Selectors: procedures*⟩+≡

```
function selector_get_weight (selector, n) result (weight)
  class(selector_t), intent(in) :: selector
  integer, intent(in) :: n
  real(default) :: weight
  integer :: i
  do i = 1, size (selector%weight)
     if (selector%map(i) == n) then
        weight = selector%weight(i)
        return
     end if
  end do
```

```
    weight = 0
  end function selector_get_weight
```

8.2.2 Unit tests

Test module, followed by the corresponding implementation module.

⟨selectors_ut.f90⟩≡

```
  ⟨File header⟩

  module selectors_ut
    use unit_tests
    use selectors_uti

  ⟨Standard module head⟩

  ⟨Selectors: public test⟩

  contains

  ⟨Selectors: test driver⟩

  end module selectors_ut
```

⟨selectors_uti.f90⟩≡

```
  ⟨File header⟩

  module selectors_uti

  ⟨Use kinds⟩
    use rng_base

    use selectors

    use rng_base_ut, only: rng_test_t

  ⟨Standard module head⟩

  ⟨Selectors: test declarations⟩

  contains

  ⟨Selectors: tests⟩

  end module selectors_uti
```

API: driver for the unit tests below.

⟨*Selectors: public test*⟩≡

```
    public :: selectors_test
```

⟨*Selectors: test driver*⟩≡

```
    subroutine selectors_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Selectors: execute tests⟩
```

```
end subroutine selectors_test
```

Basic check

Initialize the selector and draw random numbers.

⟨*Selectors: execute tests*⟩≡

```
call test (selectors_1, "selectors_1", &
     "rng initialization and call", &
     u, results)
```

⟨*Selectors: test declarations*⟩≡

```
public :: selectors_1
```

⟨*Selectors: tests*⟩≡

```
subroutine selectors_1 (u)
  integer, intent(in) :: u
  type(selector_t) :: selector
  class(rng_t), allocatable, target :: rng
  integer :: i, n

  write (u, "(A)")  "* Test output: selectors_1"
  write (u, "(A)")  "*   Purpose: initialize a selector and test it"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize selector"
  write (u, "(A)")

  call selector%init &
       ([2._default, 3.5_default, 0._default, &
       2._default, 0.5_default, 2._default])
  call selector%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Select numbers using predictable test generator"
  write (u, "(A)")

  allocate (rng_test_t :: rng)
  call rng%init (1)

  do i = 1, 5
     call selector%generate (rng, n)
     write (u, "(1x,I0)")  n
  end do

  write (u, "(A)")
  write (u, "(A)")  "* Select numbers using real input number"
  write (u, "(A)")

  write (u, "(1x,A,I0)")  "select(0.00) = ", selector%select (0._default)
  write (u, "(1x,A,I0)")  "select(0.77) = ", selector%select (0.77_default)
  write (u, "(1x,A,I0)")  "select(1.00) = ", selector%select (1._default)

  write (u, "(A)")
  write (u, "(A)")  "* Get weight"
```

```
    write (u, "(A)")

    write (u, "(1x,A,ES19.12)")  "weight(2) =", selector%get_weight(2)
    write (u, "(1x,A,ES19.12)")  "weight(3) =", selector%get_weight(3)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call rng%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: selectors_1"

  end subroutine selectors_1
```

8.3 TAO Random-Number Generator

This module provides an implementation for the generic random-number generator. Actually, we interface the TAO random-number generator which is available via the VAMP package.

⟨rng_tao.f90⟩≡

```
⟨File header⟩

module rng_tao

⟨Use kinds⟩
  use io_units
  use format_utils, only: write_indent
  use tao_random_numbers !NODEP!

  use rng_base

⟨Standard module head⟩

⟨RNG tao: public⟩

⟨RNG tao: types⟩

contains

⟨RNG tao: procedures⟩

end module rng_tao
```

8.3.1 Generator type

The rng object is actually the state of the random-number generator. The methods initialize/reset and call the generator for this state.

We keep the seed, in case we want to recover it later, and count the number of calls since seeding.

⟨*RNG tao: public*⟩≡

```
    public :: rng_tao_t
```

⟨*RNG tao: types*⟩≡

```
    type, extends (rng_t) :: rng_tao_t
       integer :: seed = 0
       integer :: n_calls = 0
       type(tao_random_state) :: state
     contains
     ⟨RNG tao: rng tao: TBP⟩
    end type rng_tao_t
```

Output: Display seed and number of calls.

⟨*RNG tao: rng tao: TBP*⟩≡

```
    procedure :: write => rng_tao_write
```

⟨*RNG tao: procedures*⟩≡

```
    subroutine rng_tao_write (rng, unit, indent)
      class(rng_tao_t), intent(in) :: rng
      integer, intent(in), optional :: unit, indent
      integer :: u, ind
      u = given_output_unit (unit)
      ind = 0;  if (present (indent))  ind = indent
      call write_indent (u, ind)
      write (u, "(A)")  "TAO random-number generator:"
      call write_indent (u, ind)
      write (u, "(2x,A,I0)")  "seed  = ", rng%seed
      call write_indent (u, ind)
      write (u, "(2x,A,I0)")  "calls = ", rng%n_calls
    end subroutine rng_tao_write
```

The `init` method initializes the generator and sets a seed. We should implement the interface such that a single integer is sufficient for a seed.

The seed may be omitted. The default seed is 0.

⟨*RNG tao: rng tao: TBP*⟩+≡

```
    procedure :: init => rng_tao_init
```

⟨*RNG tao: procedures*⟩+≡

```
    subroutine rng_tao_init (rng, seed)
      class(rng_tao_t), intent(out) :: rng
      integer, intent(in), optional :: seed
      if (present (seed))  rng%seed = seed
      call tao_random_create (rng%state, rng%seed)
    end subroutine rng_tao_init
```

The `final` method deallocates memory where necessary and allows for another call of `init` to reset the generator.

⟨*RNG tao: rng tao: TBP*⟩+≡

```
    procedure :: final => rng_tao_final
```

⟨*RNG tao: procedures*⟩+≡

```
    subroutine rng_tao_final (rng)
      class(rng_tao_t), intent(inout) :: rng
      call tao_random_destroy (rng%state)
    end subroutine rng_tao_final
```

These routines generate a single and an array of default-precision random numbers, respectively.

We have to convert from explicit double to abstract default precision. Under normal conditions, both are equivalent, however. Unless, someone decides to do single precision, there is always an interface for `tao_random_numbers`.

⟨*RNG tao: rng tao: TBP*⟩+≡
```
procedure :: generate_single => rng_tao_generate_single
procedure :: generate_array => rng_tao_generate_array
```

⟨*RNG tao: procedures*⟩+≡
```
subroutine rng_tao_generate_single (rng, x)
  class(rng_tao_t), intent(inout) :: rng
  real(default), intent(out) :: x
  real(default) :: r
  call tao_random_number (rng%state, r)
  x = r
  rng%n_calls = rng%n_calls + 1
end subroutine rng_tao_generate_single

subroutine rng_tao_generate_array (rng, x)
  class(rng_tao_t), intent(inout) :: rng
  real(default), dimension(:), intent(out) :: x
  real(default) :: r
  integer :: i
  do i = 1, size (x)
     call tao_random_number (rng%state, r)
     x(i) = r
  end do
  rng%n_calls = rng%n_calls + size (x)
end subroutine rng_tao_generate_array
```

Factory

This factory makes `rng_tao_t` generators, initialized with the seeds

$$s_i = s_0 * 2^{16} + i \tag{8.1}$$

where s_0 is the seed (a 16-bit integer) given to the factory object, and i is the index in the generated sequence of generators, starting with zero.

⟨*RNG tao: public*⟩+≡
```
public :: rng_tao_factory_t
```

⟨*RNG tao: types*⟩+≡
```
type, extends (rng_factory_t) :: rng_tao_factory_t
   integer(i16) :: s = 0
   integer(i16) :: i = 0
 contains
 ⟨RNG tao: rng tao factory: TBP⟩
end type rng_tao_factory_t
```

Output.

⟨*RNG tao: rng tao factory: TBP*⟩≡
```
procedure :: write => rng_tao_factory_write
```

⟨*RNG tao: procedures*⟩+≡
```
subroutine rng_tao_factory_write (object, unit)
  class(rng_tao_factory_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A,2(I0,A))") &
       "RNG factory: tao (", object%s, ",", object%i, ")"
end subroutine rng_tao_factory_write
```

Initialize, translating the given seed.

⟨*RNG tao: rng tao factory: TBP*⟩+≡
```
procedure :: init => rng_tao_factory_init
```

⟨*RNG tao: procedures*⟩+≡
```
subroutine rng_tao_factory_init (factory, seed)
  class(rng_tao_factory_t), intent(out) :: factory
  integer(i16), intent(in), optional :: seed
  if (present (seed))  factory%s = seed
end subroutine rng_tao_factory_init
```

⟨*RNG tao: rng tao factory: TBP*⟩+≡
```
procedure :: make => rng_tao_factory_make
```

⟨*RNG tao: procedures*⟩+≡
```
subroutine rng_tao_factory_make (factory, rng)
  class(rng_tao_factory_t), intent(inout) :: factory
  class(rng_t), intent(out), allocatable :: rng
  allocate (rng_tao_t :: rng)
  select type (rng)
  type is (rng_tao_t)
     call rng%init (factory%s * 65536 + factory%i)
     factory%i = int (factory%i + 1, kind = i16)
  end select
end subroutine rng_tao_factory_make
```

8.3.2 Unit tests

Test module, followed by the corresponding implementation module.

⟨rng_tao_ut.f90⟩≡
```
⟨File header⟩

module rng_tao_ut
  use unit_tests
  use rng_tao_uti

⟨Standard module head⟩

⟨RNG tao: public test⟩

contains
```

```
  ⟨RNG tao: test driver⟩

  end module rng_tao_ut
```

⟨rng_tao_uti.f90⟩≡

```
  ⟨File header⟩

  module rng_tao_uti

  ⟨Use kinds⟩
    use kinds, only: i16
    use rng_base

    use rng_tao

  ⟨Standard module head⟩

  ⟨RNG tao: test declarations⟩

  contains

  ⟨RNG tao: tests⟩

  end module rng_tao_uti
```

API: driver for the unit tests below.

⟨*RNG tao: public test*⟩≡

```
    public :: rng_tao_test
```

⟨*RNG tao: test driver*⟩≡

```
    subroutine rng_tao_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨RNG tao: execute tests⟩
    end subroutine rng_tao_test
```

Generator check

Initialize the generator and draw random numbers.

⟨*RNG tao: execute tests*⟩≡

```
    call test (rng_tao_1, "rng_tao_1", &
         "rng initialization and call", &
         u, results)
```

⟨*RNG tao: test declarations*⟩≡

```
    public :: rng_tao_1
```

⟨*RNG tao: tests*⟩≡

```
    subroutine rng_tao_1 (u)
      integer, intent(in) :: u
      class(rng_t), allocatable, target :: rng

      real(default) :: x
      real(default), dimension(2) :: x2
```

```
    write (u, "(A)")  "* Test output: rng_tao_1"
    write (u, "(A)")  "*   Purpose: initialize and call the TAO random-number &
         &generator"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize generator (default seed)"
    write (u, "(A)")

    allocate (rng_tao_t :: rng)
    call rng%init ()

    call rng%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Get random number"
    write (u, "(A)")

    call rng%generate (x)
    write (u, "(A,2(1x,F9.7))")  "x =", x

    write (u, "(A)")
    write (u, "(A)")  "* Get random number pair"
    write (u, "(A)")

    call rng%generate (x2)
    write (u, "(A,2(1x,F9.7))")  "x =", x2

    write (u, "(A)")
    call rng%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call rng%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rng_tao_1"

  end subroutine rng_tao_1
```

Factory check

Set up a factory and spawn generators.

⟨*RNG tao: execute tests*⟩+≡

```
  call test (rng_tao_2, "rng_tao_2", &
       "rng factory", &
       u, results)
```

⟨*RNG tao: test declarations*⟩+≡

```
  public :: rng_tao_2
```

⟨*RNG tao: tests*⟩+≡

```
  subroutine rng_tao_2 (u)
    integer, intent(in) :: u
```

```
type(rng_tao_factory_t) :: rng_factory
class(rng_t), allocatable :: rng
real(default) :: x

write (u, "(A)")  "* Test output: rng_tao_2"
write (u, "(A)")  "*   Purpose: initialize and use a rng factory"
write (u, "(A)")

write (u, "(A)")  "* Initialize factory"
write (u, "(A)")

call rng_factory%init ()
call rng_factory%write (u)

write (u, "(A)")
write (u, "(A)")  "* Make a generator"
write (u, "(A)")

call rng_factory%make (rng)
call rng%write (u)
call rng%generate (x)
write (u, *)
write (u, "(1x,A,F7.5)")  "x = ", x
call rng%final ()
deallocate (rng)

write (u, "(A)")
write (u, "(A)")  "* Repeat"
write (u, "(A)")

call rng_factory%make (rng)
call rng%write (u)
call rng%generate (x)
write (u, *)
write (u, "(1x,A,F7.5)")  "x = ", x
call rng%final ()
deallocate (rng)

write (u, *)
call rng_factory%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize factory with different seed"
write (u, "(A)")

call rng_factory%init (1_i16)
call rng_factory%write (u)

write (u, "(A)")
write (u, "(A)")  "* Make a generator"
write (u, "(A)")

call rng_factory%make (rng)
call rng%write (u)
```

```
    call rng%generate (x)
    write (u, *)
    write (u, "(1x,A,F7.5)")  "x = ", x
    call rng%final ()
    deallocate (rng)

    write (u, *)
    call rng_factory%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rng_tao_2"

  end subroutine rng_tao_2
```

8.4 RNG Stream

We implement the RNGstream proposed and implemented in C++, C and Java by L'Ecuyer et al. (LEcuyer, Pierre, Richard Simard, E. Jack Chen, and W. David Kelton. An Object-Oriented Random-Number Package with Many Long Streams and Substreams. Operations Research 50, no. 6 (December 2002): 10731075. doi:10.1287/opre.50.6.1073.358.). RNGstream allows us to access multiple independent streams of random numbers.

⟨rng_stream.f90⟩≡

```
⟨File header⟩

module rng_stream

⟨Use kinds⟩
  use kinds, only: i16
  use io_units
  use format_utils, only: write_indent
  use diagnostics

  use rng_base

⟨Standard module head⟩

⟨RNG stream: parameters⟩

⟨RNG stream: public⟩

⟨RNG stream: types⟩

contains

⟨RNG stream: procedures⟩

end module rng_stream
```

We store the current position in the stream of random numbers. Furthermore, we store the beginning of the (latest) substream and the initial starting point

of the stream, which corresponds to the "seed". The default seed is 1234.

⟨*RNG stream: public*⟩≡
```
public :: rng_stream_t
```

⟨*RNG stream: types*⟩≡
```
type, extends(rng_t) :: rng_stream_t
   private
   logical :: antithetic = .false.
   logical :: increased_precision = .false.
   real(default), dimension(3, 2) :: current_position = 1234._default
   real(default), dimension(3, 2) :: beginning_substream = 1234._default
   real(default), dimension(3, 2) :: initial_stream = 1234._default
 contains
 ⟨RNG stream: rng stream: TBP⟩
end type rng_stream_t
```

Output. Display current position and the (sub-)stream positions.

⟨*RNG stream: rng stream: TBP*⟩≡
```
procedure, public :: write => rng_stream_write
```

⟨*RNG stream: procedures*⟩≡
```
subroutine rng_stream_write (rng, unit, indent)
  class(rng_stream_t), intent(in) :: rng
  integer, intent(in), optional :: unit
  integer, intent(in), optional :: indent
  integer :: j, u, ind
  u = given_output_unit (unit)
  ind = 0; if (present (indent)) ind = indent
  call write_indent (u, ind)
  write (u, "(A)") "RNG Stream generator"
  call write_indent (u, ind)
  write (u, "(A)") "Current position = [ "
  call write_indent (u, ind)
  write (u, "(3(F20.1,',',1X))") rng%current_position(:, 1)
  call write_indent (u, ind)
  write (u, "(3(F20.1,',',1X))") rng%current_position(:, 2)
  call write_indent (u, ind)
  write (u, "(A)") "]"
  call write_indent (u, ind)
  write (u, "(A)") "Beginning substream = [ "
  call write_indent (u, ind)
  write (u, "(3(F20.1,',',1X))") rng%beginning_substream(:, 1)
  call write_indent (u, ind)
  write (u, "(3(F20.1,',',1X))") rng%beginning_substream(:, 2)
  call write_indent (u, ind)
  write (u, "(A)") "]"
  call write_indent (u, ind)
  write (u, "(A)") "Initial stream = [ "
  call write_indent (u, ind)
  write (u, "(3(F20.1,',',1X))") rng%initial_stream(:, 1)
  call write_indent (u, ind)
  write (u, "(3(F20.1,',',1X))") rng%initial_stream(:, 2)
  call write_indent (u, ind)
  write (u, "(A)") "]"
```

```
end subroutine rng_stream_write
```

Initialise the generator. Set the states to the (optional) seed.

⟨*RNG stream: rng stream: TBP*⟩+≡

```
procedure, public :: init => rng_stream_init
```

⟨*RNG stream: procedures*⟩+≡

```
subroutine rng_stream_init (rng, seed)
  class(rng_stream_t), intent(out) :: rng
  integer, intent(in), optional :: seed
  real(default), dimension(3, 2) :: real_seed
  if (present (seed)) then
     real_seed = real(seed, default)
     call rng%set_initial_stream (real_seed)
  end if
end subroutine rng_stream_init
```

Finalize. Nothing to do.

⟨*RNG stream: rng stream: TBP*⟩+≡

```
procedure, public :: final => rng_stream_final
```

⟨*RNG stream: procedures*⟩+≡

```
subroutine rng_stream_final (rng)
  class(rng_stream_t), intent(inout) :: rng
end subroutine rng_stream_final
```

Set the initial stream, but test the seed before. They are not allowed to be 0 or larger than m_1 or m_2.

⟨*RNG stream: rng stream: TBP*⟩+≡

```
procedure, public :: set_initial_stream => rng_stream_set_initial_stream
```

⟨*RNG stream: procedures*⟩+≡

```
subroutine rng_stream_set_initial_stream (rng, seed)
  class(rng_stream_t), intent(inout) :: rng
  real(default), dimension(3, 2), intent(in) :: seed
  if (any (seed(:, 1) > m1)) then
     write (msg_buffer, "(A, F20.1, 1X, A)") "ERROR: seed(:, 1) >= ", m1, ",&
          & Seed is not set."
     call msg_fatal ()
  end if
  if (any (seed(:, 2) > m2)) then
     write (msg_buffer, "(A, F20.1, 1X, A)") "ERROR: seed(:, 2) >= ", m2, ",&
          & Seed is not set."
     call msg_fatal ()
  end if
  if (any (seed(:, 1) == 0)) then
     write (msg_buffer, "(A, F20.1, 1X, A)") "ERROR: First 3 seed = 0."
     call msg_fatal ()
  end if
  if (any (seed(:, 2) == 0)) then
     write (msg_buffer, "(A, F20.1, 1X, A)") "ERROR: Last 3 seed = 0."
     call msg_fatal ()
  end if
```

```
    rng%initial_stream = seed
    call rng%reset_stream ()
  end subroutine rng_stream_set_initial_stream
```

Generate a single real random number from the current state of (sub-)stream.

The machine representation of the result can lead to negative numbers. We add m_1 or m_2 to the respective number to ensure they are all positive and are still congruent to $\mod m_1$ or $\mod m_2$.

The final formula $x = y(3,1) - y(3,2) \pmod{m_1}$ can lead to negative random numbers. Instead of directly calculating the modulo, we check whether $y(3,1)$ is larger than $y(3,2)$, or not. If $x(3,1) > y(3,2)$, then $y(3,1) - y(3,2)$ is congruent to $y(3,1) - y(3,2) \mod m_1$. If $y(3,1) < y(3,2)$, then $y(3,1) - y(3,2) + m_1$ is congruent to $y(3,1) - y(3,2) \mod m_1$.

⟨*RNG stream: parameters*⟩≡

```
  real(default), parameter :: &
       m1 = 4294967087.0_default, &
       m2 = 4294944443.0_default, &
       norm = 1.0_default / (m1 + 1.0_default)
  real(default), dimension(3, 3), parameter :: A1p0 = reshape (&
       [0.0_default, 1.0_default, 0.0_default, &
       0.0_default, 0.0_default, 1.0_default, &
       -810728.0_default, 1403580.0_default, 0.0_default], &
       shape (A1p0), order = [2, 1])
  real(default), dimension(3, 3), parameter :: A2p0 = reshape (&
       [0.0_default, 1.0_default, 0.0_default, &
       0.0_default, 0.0_default, 1.0_default, &
       -1370589.0_default, 0.0_default, 527612.0_default], &
       shape (A2p0), order = [2, 1])
```

⟨*RNG stream: rng stream: TBP*⟩+≡

```
  procedure, public :: generate_single => rng_stream_generate_single
```

⟨*RNG stream: procedures*⟩+≡

```
  subroutine rng_stream_generate_single (rng, x)
    class(rng_stream_t), intent(inout) :: rng
    real(default), intent(out) :: x
    associate (y => rng%current_position)
      y(:, 1) = mod (matmul (A1p0, y(:, 1)), m1)
      if (y(3, 1) < 0.) y(3, 1) = y(3, 1) + m1
      y(:, 2) = mod (matmul (A2p0, y(:, 2)), m2)
      if (y(3, 2) < 0.) y(3, 2) = y(3, 2) + m2
      x = y(3, 1) - y(3, 2)
      if (x < 0.) x = x + m1
      x = x * norm
    end associate
    if (rng%antithetic) x = 1. - x
  end subroutine rng_stream_generate_single
```

Generate an array of real random numbers.

⟨*RNG stream: rng stream: TBP*⟩+≡

```
  procedure, public :: generate_array => rng_stream_generate_array
```

⟨*RNG stream: procedures*⟩+≡

```
subroutine rng_stream_generate_array (rng, x)
  class(rng_stream_t), intent(inout) :: rng
  real(default), dimension(:), intent(out) :: x
  integer :: j
  do j = 1, size (x)
     call rng%generate_single(x(j))
  end do
end subroutine rng_stream_generate_array
```

Reset to initial stream.

⟨*RNG stream: rng stream: TBP*⟩+≡

```
procedure, public :: reset_stream => rng_stream_reset_stream
```

⟨*RNG stream: procedures*⟩+≡

```
subroutine rng_stream_reset_stream (rng)
  class(rng_stream_t), intent(inout) :: rng
  rng%current_position = rng%initial_stream
  rng%beginning_substream = rng%initial_stream
end subroutine rng_stream_reset_stream
```

Reset to beginning of substream.

⟨*RNG stream: rng stream: TBP*⟩+≡

```
procedure, public :: reset_substream => rng_stream_reset_substream
```

⟨*RNG stream: procedures*⟩+≡

```
subroutine rng_stream_reset_substream (rng)
  class(rng_stream_t), intent(inout) :: rng
  rng%current_position = rng%beginning_substream
end subroutine rng_stream_reset_substream
```

Advance to next substream.

⟨*RNG stream: parameters*⟩+≡

```
real(default), dimension(3, 3), parameter :: A1p76 = reshape (&
     [82758667.0_default, 1871391091.0_default, 4127413238.0_default, &
     3672831523.0_default, 69195019.0_default, 1871391091.0_default, &
     3672091415.0_default, 352874325.0_default, 69195019.0_default], &
     shape(A1p76), order = [2, 1])
real(default), dimension(3, 3), parameter :: A2p76 = reshape (&
     [1511326704.0_default, 3759209742.0_default, 1610795712.0_default, &
     4292754251.0_default, 1511326704.0_default, 3889917532.0_default, &
     3859662829.0_default, 4292754251.0_default, 3708466080.0_default], &
     shape(A2p76), order = [2, 1])
```

⟨*RNG stream: rng stream: TBP*⟩+≡

```
procedure, public :: next_substream => rng_stream_next_substream
```

⟨*RNG stream: procedures*⟩+≡

```
subroutine rng_stream_next_substream (rng)
  class(rng_stream_t), intent(inout) :: rng
  associate (x => rng%beginning_substream)
    x(:,1) = matmul_mod (A1p76, x(:,1), m1)
    x(:,2) = matmul_mod (A2p76, x(:,2), m2)
  end associate
```

```
    call rng%reset_substream ()
  end subroutine rng_stream_next_substream
```

Advance state of the current stream by n steps. We do not touch the beginning of the substream nor the initial stream. The purpose of the procedure is to reproduce the same random number (streams) among multiple workers without additional communication cost. E.g. worker 1 needs the first 5000 random numbers and the second one needs the next 5000 random numbers. In such a way we can exactly reproduce results (upon numerical noise) in single or parallel runs with the RngStream. The procedure must only be used with care! Misuse can lead to wrong results!

⟨*RNG stream: parameter*⟩≡

```
  real(default), dimension(3, 3), parameter :: A1p0 = reshape (&
       [0.0_default, 1.0_default, 0.0_default, &
       0.0_default, 0.0_default, 1.0_default, &
       -810728.0_default, 1403580.0_default, 0.0_default], &
       shape(A1p0), order = [2, 1])
  real(default), dimension(3, 3), parameter :: A2p0 = reshape (&
       [0.0_default, 1.0_default, 0.0_default, &
       0.0_default, 0.0_default, 1.0_default, &
       -1370589.0_default, 0.0_default, 527612.0_default], &
       shape(A2p0), order = [2, 1])
```

⟨*RNG stream: rng stream: TBP*⟩+≡

```
  procedure, public :: advance_state => rng_stream_advance_state
```

⟨*RNG stream: procedures*⟩+≡

```
  subroutine rng_stream_advance_state (rng, n)
    class(rng_stream_t), intent(inout) :: rng
    integer, intent(in) :: n
    real(default), dimension(3, 3) :: A1pn, A2pn
    associate (x => rng%current_position)
      A1pn = matpow_mod (A1p0, m1, n)
      A2pn = matpow_mod (A2p0, m2, n)
      x(:,1) = matmul_mod (A1pn, x(:,1), m1)
      x(:,2) = matmul_mod (A2pn, x(:,2), m2)
    end associate
  end subroutine rng_stream_advance_state
```

A divide and conquer algorithm is applied to calculate $A^n \mod m$. We refer to the original paper from L'Ecuyer for further reading.

⟨*RNG stream: procedures*⟩+≡

```
  function matpow_mod (a, m, n) result (c)
    real(default), dimension(3, 3), intent(in) :: a
    real(default), intent(in) :: m
    integer, intent(in) :: n
    real(default), dimension(3, 3) :: c
    real(default), dimension(3, 3) :: w
    integer :: i
    w = a
    c = 0.; forall (i = 1:3) c(i, i) = 1.
    i = n
    do while (i > 0)
```

```
      if (mod(i, 2) /= 0) c = matmat_mod (w, c, m)
      w = matmat_mod (w, w, m)
      i = i / 2
    end do
  end function matpow_mod

  function matmat_mod (a, b, m) result (c)
    real(default), dimension(3, 3), intent(in) :: a
    real(default), dimension(3, 3), intent(in) :: b
    real(default), dimension(3, 3) :: c
    real(default), intent(in) :: m
    integer :: i
    do i = 1, 3
      c(:, i) = matmul_mod (a, b(:, i), m)
    end do
  end function matmat_mod
```

8.4.1 Factory

We utilize the streams proposed by L'Ecuyer where we use precalulated transition matrices to jump between different (sub-)streams.

The random generator has the period ρ and the transition function $T : s_n \to s_{n+1}$ whereas s_n is the generator's state after n steps. Furthermore, it is $T^\rho(s) = s$. The stream of random number then is disjoint into streams and substreams. The length of the streams, which divide the overall stream, should be $Z = 2^z$ and partition those streams again in $V = 2^v$ substreams of length $W = 2^w$.

We store the seed and the initial stream, also, the beginning of the substream.

⟨*RNG stream: public*⟩+≡

```
public :: rng_stream_factory_t
```

⟨*RNG stream: types*⟩+≡

```
type, extends(rng_factory_t) :: rng_stream_factory_t
   real(default), dimension(3, 2) :: seed = 1234._default
 contains
 ⟨RNG stream: rng stream factory: TBP⟩
end type rng_stream_factory_t
```

Output.

⟨*RNG stream: rng stream factory: TBP*⟩≡

```
procedure, public :: write => rng_stream_factory_write
```

⟨*RNG stream: procedures*⟩+≡

```
subroutine rng_stream_factory_write (object, unit)
  class(rng_stream_factory_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)") "RNG Stream factory"
  write (u, "(1x,A,6(F20.1,',',1X),A)") "Next seed = [ "
  write (u, "(1x,3(F20.1,',',1X))") object%seed(:, 1)
  write (u, "(1x,3(F20.1,',',1X))") object%seed(:, 2)
```

```
      write (u, "(1x,A)") "]"
    end subroutine rng_stream_factory_write
```

Initialize. Default seed is 1234. We do not check whether the seed is valid, this is later done in `rng_stream_init`.

Additionally, we have to explicitly convert from `integer` to `real` because we are using floating-point arithmetic and not integer arithemtic due to a larger set of number in the representation of real numbers.

⟨*RNG stream: rng stream factory: TBP*⟩+≡

```
    procedure, public :: init => rng_stream_factory_init
```

⟨*RNG stream: procedures*⟩+≡

```
    subroutine rng_stream_factory_init (factory, seed)
      class(rng_stream_factory_t), intent(out) :: factory
      integer(i16), intent(in), optional :: seed
      real(default), dimension(3, 2) :: real_seed
      if (present (seed)) then
         factory%seed = real(seed, default)
      end if
    end subroutine rng_stream_factory_init
```

Allocate a new RNG and different stream states, accordingly.

⟨*RNG stream: rng stream factory: TBP*⟩+≡

```
    procedure, public :: make => rng_stream_factory_make
```

⟨*RNG stream: procedures*⟩+≡

```
    subroutine rng_stream_factory_make (factory, rng)
      class(rng_stream_factory_t), intent(inout) :: factory
      class(rng_t), intent(out), allocatable :: rng
      allocate (rng_stream_t :: rng)
      select type (rng)
      type is (rng_stream_t)
         call rng%init ()
         call rng%set_initial_stream (factory%seed)
      end select
      call factory%advance_seed ()
    end subroutine rng_stream_factory_make
```

Advance seed. Use the precalculated transistion matrices for the streams.

⟨*RNG stream: parameters*⟩+≡

```
  real(default), dimension(3,3), parameter :: A1p127 = reshape (&
       [2427906178.0_default, 3580155704.0_default, 949770784.0_default, &
       226153695.0_default, 1230515664.0_default, 3580155704.0_default, &
       1988835001.0_default, 986791581.0_default, 1230515664.0_default], &
       shape(A1p127), order = [2, 1])
  real(default), dimension(3,3), parameter :: A2p127 = reshape (&
       [1464411153.0_default, 277697599.0_default, 1610723613.0_default, &
       32183930.0_default, 1464411153.0_default, 1022607788.0_default, &
       2824425944.0_default, 32183930.0_default, 2093834863.0_default], &
       shape(A2p127), order = [2, 1])
```

⟨*RNG stream: rng stream factory: TBP*⟩+≡

```
    procedure, private :: advance_seed => rng_stream_factory_advance_seed
```

⟨*RNG stream: procedures*⟩+≡

```
subroutine rng_stream_factory_advance_seed (factory)
  class(rng_stream_factory_t), intent(inout) :: factory
  factory%seed(:,1) = matmul_mod (A1p127, factory%seed(:,1), m1)
  factory%seed(:,2) = matmul_mod (A2p127, factory%seed(:,2), m2)
end subroutine rng_stream_factory_advance_seed
```

Matrix modulo calculation optimized for 2^{53} representation of floating numbers. Speed efficiency is not a requirement, because the streams should not be advanced that often.

⟨*RNG stream: public*⟩+≡

```
private :: matmul_mod
```

⟨*RNG stream: procedures*⟩+≡

```
function matmul_mod (a, u, m) result (v)
  real(default), dimension(:, :), intent(in) :: a
  real(default), dimension(:), intent(in) :: u
  real(default), intent(in) :: m
  real(default), dimension(size (u)) :: v
  integer :: i
  do i = 1, 3
     v(i) = mult_mod (a(i, 1), u(1), 0.0_default, m)
     v(i) = mult_mod (a(i, 2), u(2), v(i), m)
     v(i) = mult_mod (a(i, 3), u(3), v(i), m)
  end do
end function matmul_mod
```

Decompose.

Decomposition to ensure exact accuracy for the case v exceeds 2^{53}.

$$a \cdot smodm = ((a1 \cdot s \mod m) * 2^{17} + a2 \cdot s) \mod m.$$

⟨*RNG stream: parameters*⟩+≡

```
real(default), parameter :: &
     a12 = 1403580.0_default, &
     a13n = 810728.0_default, &
     a21 = 527612.0_default, &
     a23n = 1370589.0_default, &
     two53 = 9007199254740992.0_default, &
     two17 = 131072.0_default
```

⟨*RNG stream: public*⟩+≡

```
private :: mult_mod
```

⟨*RNG stream: procedures*⟩+≡

```
function mult_mod (a, b, c, m) result (v)
  real(default), intent(in) :: a
  real(default), intent(in) :: b
  real(default), intent(in) :: c
  real(default), intent(in) :: m
  real(default) :: v
  integer :: a1
  real(default) :: a2
  v = a * b + c
  if (v >= two53 .or. v <= -two53) then
```

```
        a1 = int (a / two17)
        a2 = a - a1 * two17
        v = mod (a1 * b, m)
        v = v * two17 + a2 * b + c
      end if
      v = mod (v, m)
      if (v < 0.0) v = v + m
    end function mult_mod
```

8.4.2 Unit tests

Test module, followed by the corresponding implementation module.

⟨rng_stream_ut.f90⟩≡

```
  ⟨File header⟩

  module rng_stream_ut
    use unit_tests
    use rng_stream_uti

  ⟨Standard module head⟩

  ⟨RNG stream: public test⟩

  contains

  ⟨RNG stream: test driver⟩

  end module rng_stream_ut
```

⟨rng_stream_uti.f90⟩≡

```
  ⟨File header⟩

  module rng_stream_uti

  ⟨Use kinds⟩
    use kinds, only: i16
    use rng_base

    use rng_stream

  ⟨Standard module head⟩

  ⟨RNG stream: test declarations⟩

  contains

  ⟨RNG stream: tests⟩

  end module rng_stream_uti
```

API: driver for the unit tests below.

⟨*RNG stream: public test*⟩≡

```
    public :: rng_stream_test
```

⟨*RNG stream: test driver*⟩≡

```
subroutine rng_stream_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨RNG stream: execute tests⟩
end subroutine rng_stream_test
```

Generator check

Initialize the generator and draw random numbers.

⟨*RNG stream: execute tests*⟩≡

```
call test (rng_stream_1, "rng_stream_1", &
     "rng initialization and call", &
     u, results)
```

⟨*RNG stream: test declarations*⟩≡

```
public :: rng_stream_1
```

⟨*RNG stream: tests*⟩≡

```
subroutine rng_stream_1 (u)
  integer, intent(in) :: u
  class(rng_t), allocatable, target :: rng

  real(default) :: x
  real(default), dimension(2) :: x2

  write (u, "(A)")  "* Test output: rng_stream_1"
  write (u, "(A)")  "*   Purpose: initialize and call the RNGstream random-number &
       &generator"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize generator (default seed)"
  write (u, "(A)")

  allocate (rng_stream_t :: rng)
  call rng%init ()

  call rng%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Get random number"
  write (u, "(A)")

  call rng%generate (x)
  write (u, "(A,2(1x,F9.7))")  "x =", x

  write (u, "(A)")
  write (u, "(A)")  "* Get random number pair"
  write (u, "(A)")

  call rng%generate (x2)
  write (u, "(A,2(1x,F9.7))")  "x =", x2

  write (u, "(A)")
```

```
    call rng%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Jump to next substream and get random number"
    write (u, "(A)")

    select type (rng)
    type is (rng_stream_t)
       call rng%next_substream ()
    end select
    call rng%generate (x)
    write (u, "(A,2(1x,F9.7))")  "x =", x

    write (u, "(A)")
    call rng%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call rng%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rng_stream_1"

  end subroutine rng_stream_1
```

Factory check

Set up a factory and spawn generators.

⟨*RNG stream: execute tests*⟩+≡

```
  call test (rng_stream_2, "rng_stream_2", &
       "rng factory", &
       u, results)
```

⟨*RNG stream: test declarations*⟩+≡

```
  public :: rng_stream_2
```

⟨*RNG stream: tests*⟩+≡

```
  subroutine rng_stream_2 (u)
    integer, intent(in) :: u
    type(rng_stream_factory_t) :: rng_factory
    class(rng_t), allocatable :: rng
    real(default) :: x

    write (u, "(A)")  "* Test output: rng_stream_2"
    write (u, "(A)")  "*   Purpose: initialize and use a rng factory"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize factory"
    write (u, "(A)")

    call rng_factory%init ()
    call rng_factory%write (u)
```

```
  write (u, "(A)")
  write (u, "(A)")  "* Make a generator"
  write (u, "(A)")

  call rng_factory%make (rng)
  call rng%write (u)
  call rng%generate (x)
  write (u, *)
  write (u, "(1x,A,F7.5)")  "x = ", x
  call rng%final ()
  deallocate (rng)

  write (u, "(A)")
  write (u, "(A)")  "* Repeat"
  write (u, "(A)")

  call rng_factory%make (rng)
  call rng%write (u)
  call rng%generate (x)
  write (u, *)
  write (u, "(1x,A,F7.5)")  "x = ", x
  call rng%final ()
  deallocate (rng)

  write (u, *)
  call rng_factory%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize factory with different seed"
  write (u, "(A)")

  call rng_factory%init (1_i16)
  call rng_factory%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Make a generator"
  write (u, "(A)")

  call rng_factory%make (rng)
  call rng%write (u)
  call rng%generate (x)
  write (u, *)
  write (u, "(1x,A,F7.5)")  "x = ", x
  call rng%final ()
  deallocate (rng)

  write (u, *)
  call rng_factory%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: rng_stream_2"

end subroutine rng_stream_2
```

RNG Stream extension

Initialize the generator and advance state by 15389 steps and compare with regular advanced stream position.

⟨*RNG stream: execute tests*⟩+≡

```
call test (rng_stream_3, "rng_stream_3", &
     "rng initialization and advance state", &
     u, results)
```

⟨*RNG stream: test declarations*⟩+≡

```
public :: rng_stream_3
```

⟨*RNG stream: tests*⟩+≡

```
subroutine rng_stream_3 (u)
  integer, intent(in) :: u
  class(rng_t), allocatable, target :: rng

  integer :: i
  real(default) :: x
  real(default), dimension(2) :: x2

  write (u, "(A)")  "* Test output: rng_stream_3"
  write (u, "(A)")  "*   Purpose: initialize and advance the state of the random-number &
       &generator"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize generator (default seed)"
  write (u, "(A)")

  allocate (rng_stream_t :: rng)
  call rng%init ()

  call rng%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Advance state by 15389 by iteration"
  write (u, "(A)")

  do i = 1, 15389
     call rng%generate (x)
  end do

  call rng%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Get random number"
  write (u, "(A)")

  call rng%generate (x)
  write (u, "(A,2(1x,F9.7))")  "x =", x

  write (u, "(A)")
  write (u, "(A)")  "* Advance state by 15389 by procedure"
  write (u, "(A)")
```

```
  select type (rng)
  type is (rng_stream_t)
     call rng%reset_substream ()
     call rng%advance_state (15389)
  end select

  call rng%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Get random number"
  write (u, "(A)")

  call rng%generate (x)
  write (u, "(A,2(1x,F9.7))")  "x =", x


  write (u, "(A)")
  write (u, "(A)")  "* Get random number pair"
  write (u, "(A)")

  call rng%generate (x2)
  write (u, "(A,2(1x,F9.7))")  "x =", x2

  write (u, "(A)")
  call rng%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Jump to next substream and get random number"
  write (u, "(A)")

  select type (rng)
  type is (rng_stream_t)
     call rng%next_substream ()
  end select
  call rng%generate (x)
  write (u, "(A,2(1x,F9.7))")  "x =", x

  write (u, "(A)")
  call rng%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call rng%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: rng_stream_3"

end subroutine rng_stream_3
```

8.5 Dispatch

⟨dispatch_rng.f90⟩≡

```
⟨File header⟩

module dispatch_rng

  use kinds, only: i16
⟨Use strings⟩
  use diagnostics
  use variables

  use rng_base
  use rng_tao
  use rng_stream

⟨Standard module head⟩

⟨Dispatch rng: public⟩

⟨Dispatch rng: variables⟩

contains

⟨Dispatch rng: procedures⟩

end module dispatch_rng
```

Allocate a random-number generator factory according to the variable `$rng_method`, using the current seed in the global record. We take only the lower 15 bits of the seed, so the actual value fits into a positive 16-bit signed integer.

Since we want to guarantee that all random-number generators in a run are independent, we increment the global seed by one after creating the rng factory. If the user wants to have identical sequences, he can always set the seed manually, before it is used.

⟨*Dispatch rng: public*⟩≡

```
public :: dispatch_rng_factory
```

⟨*Dispatch rng: procedures*⟩≡

```
subroutine dispatch_rng_factory (rng_factory, var_list_global, var_list_local)
  class(rng_factory_t), allocatable, intent(inout) :: rng_factory
  type(var_list_t), intent(inout) :: var_list_global
  type(var_list_t), intent(in), optional :: var_list_local
  type(var_list_t) :: local
  type(string_t) :: rng_method
  integer :: seed
  character(30) :: buffer
  integer(i16) :: s
  if (present (var_list_local)) then
     local = var_list_local
  else
     local = var_list_global
  end if
  rng_method = local%get_sval (var_str ("$rng_method"))
  seed = local%get_ival (var_str ("seed"))
  s = int (mod (seed, 32768), i16)
  select case (char (rng_method))
```

```
      case ("tao")
         allocate (rng_tao_factory_t :: rng_factory)
         call msg_message ("RNG: Initializing TAO random-number generator")
      case ("rng_stream")
         allocate (rng_stream_factory_t :: rng_factory)
         call msg_message ("RNG: Initializing RNG Stream random-number generator")
      case default
         if (associated (dispatch_rng_factory_extra)) then
            call dispatch_rng_factory_extra (rng_factory, var_list_global, var_list_local)
         end if
         if (.not. allocated (rng_factory)) then
            call msg_fatal ("Random-number generator '" &
                 // char (rng_method) // "' not implemented")
         end if
      end select
      write (buffer, "(I0)")  s
      call msg_message ("RNG: Setting seed for random-number generator to " &
              // trim (buffer))
      call rng_factory%init (s)
      call var_list_global%set_int (var_str ("seed"), seed + 1, &
           is_known = .true.)
   end subroutine dispatch_rng_factory
```

This is a hook that allows us to inject further handlers for RNG factory objects, in particular a test RNG.

⟨*Dispatch rng: public*⟩+≡

```
   public :: dispatch_rng_factory_extra
```

⟨*Dispatch rng: variables*⟩≡

```
   procedure (dispatch_rng_factory), pointer :: &
        dispatch_rng_factory_extra => null ()
```

8.5.1 Unit tests

Test module, followed by the corresponding implementation module.

⟨`dispatch_rng_ut.f90`⟩≡

```
  ⟨File header⟩

  module dispatch_rng_ut
    use unit_tests
    use dispatch_rng_uti

  ⟨Standard module head⟩

  ⟨Dispatch rng: public test⟩

  ⟨Dispatch rng: public test auxiliary⟩

  contains

  ⟨Dispatch rng: test driver⟩

  end module dispatch_rng_ut
```

⟨dispatch_rng_uti.f90⟩≡

```
⟨File header⟩

module dispatch_rng_uti

⟨Use strings⟩
  use variables
  use diagnostics
  use rng_base
  use dispatch_rng

⟨Standard module head⟩

⟨Dispatch rng: public test auxiliary⟩

⟨Dispatch rng: test declarations⟩

contains

⟨Dispatch rng: tests⟩

⟨Dispatch rng: test auxiliary⟩

end module dispatch_rng_uti
```

API: driver for the unit tests below.

⟨*Dispatch rng: public test*⟩≡

```
public ::dispatch_rng_test
```

⟨*Dispatch rng: test driver*⟩≡

```
subroutine dispatch_rng_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨Dispatch rng: execute tests⟩
end subroutine dispatch_rng_test
```

Select type: random number generator

This is an extra dispatcher that enables the test RNG. This procedure should be assigned to the `dispatch_rng_factory_extra` hook before any tests are executed.

⟨*Dispatch rng: public test auxiliary*⟩≡

```
public :: dispatch_rng_factory_test
```

⟨*Dispatch rng: test auxiliary*⟩≡

```
subroutine dispatch_rng_factory_test (rng_factory, var_list_global, var_list_local)
  use rng_base
  use rng_base_ut, only: rng_test_factory_t
  class(rng_factory_t), allocatable, intent(inout) :: rng_factory
  type(var_list_t), intent(inout) :: var_list_global
  type(var_list_t), intent(in), optional :: var_list_local
  type(var_list_t) :: local
  type(string_t) :: rng_method
```

```
    if (present (var_list_local)) then
       local = var_list_local
    else
       local = var_list_global
    end if
    rng_method = &
         local%get_sval (var_str ("$rng_method"))
    select case (char (rng_method))
    case ("unit_test")
       allocate (rng_test_factory_t :: rng_factory)
       call msg_message ("RNG: Initializing Test random-number generator")
    end select
  end subroutine dispatch_rng_factory_test
```

⟨*Dispatch rng: execute tests*⟩≡

```
  call test (dispatch_rng_1, "dispatch_rng_1", &
       "random-number generator", &
       u, results)
```

⟨*Dispatch rng: test declarations*⟩≡

```
  public :: dispatch_rng_1
```

⟨*Dispatch rng: tests*⟩≡

```
  subroutine dispatch_rng_1 (u)
    integer, intent(in) :: u
    type(var_list_t) :: var_list
    class(rng_factory_t), allocatable :: rng_factory

    write (u, "(A)")  "* Test output: dispatch_rng_1"
    write (u, "(A)")  "*   Purpose: select random-number generator"
    write (u, "(A)")

    call var_list%init_defaults (0)

    write (u, "(A)")  "* Allocate RNG factory as rng_test_factory_t"
    write (u, "(A)")

    call var_list%set_string (&
         var_str ("$rng_method"), &
         var_str ("unit_test"), is_known = .true.)
    call var_list%set_int (&
         var_str ("seed"), 1, is_known = .true.)
    call dispatch_rng_factory (rng_factory, var_list)
    call rng_factory%write (u)
    deallocate (rng_factory)

    write (u, "(A)")
    write (u, "(A)")  "* Allocate RNG factory as rng_tao_factory_t"
    write (u, "(A)")

    call var_list%set_string (&
         var_str ("$rng_method"), &
         var_str ("tao"), is_known = .true.)
    call dispatch_rng_factory (rng_factory, var_list)
    call rng_factory%write (u)
```

```
  deallocate (rng_factory)

  write (u, "(A)")
  write (u, "(A)")  "* Allocate RNG factory as rng_stream_factory_t"
  write (u, "(A)")

  call var_list%set_string (&
       var_str ("$rng_method"), &
       var_str ("rng_stream"), is_known = .true.)
  call dispatch_rng_factory (rng_factory, var_list)
  call rng_factory%write (u)
  deallocate (rng_factory)

  call var_list%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: dispatch_rng_1"

end subroutine dispatch_rng_1
```

Chapter 9

Physics

Here we collect definitions and functions that we need for (particle) physics in general, to make them available for the more specific needs of WHIZARD.

physics_defs Physical constants.

c_particles A simple data type for particles which is C compatible.

lorentz Define three-vectors, four-vectors and Lorentz transformations and common operations for them.

sm_physics Here, running functions are stored for special kinematical setup like running coupling constants, Catani-Seymour dipoles, or Sudakov factors.

sm_qcd Definitions and methods for dealing with the running QCD coupling.

shower_algorithms Algorithms typically used in Parton Showers as well as in their matching to NLO computations, e.g. with the POWHEG method.

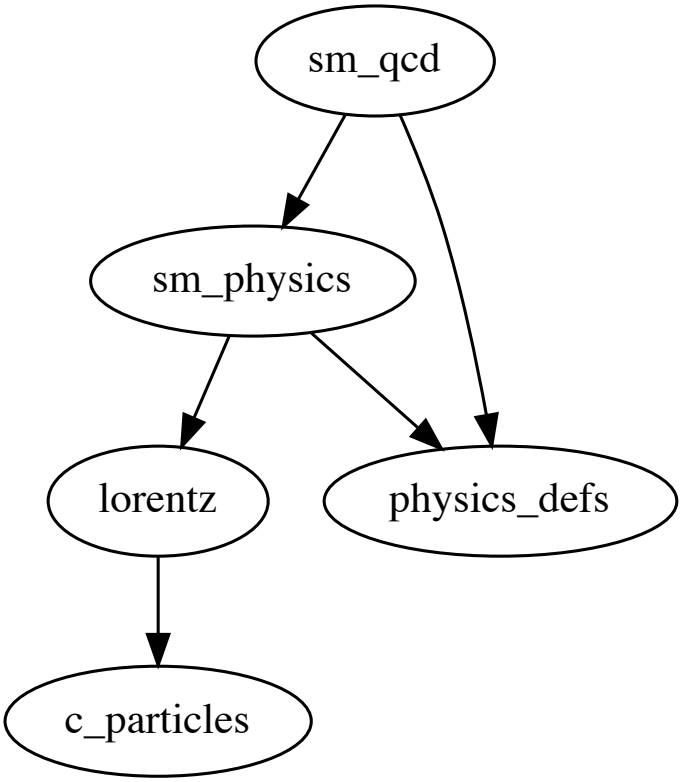

Figure 9.1: Module dependencies in `src/physics`.

9.1 Physics Constants

There is also the generic `constants` module. The constants listed here are more specific for particle physics.

⟨`physics_defs.f90`⟩≡

```
⟨File header⟩

module physics_defs

⟨Use kinds⟩
⟨Use strings⟩
  use constants, only: one, two, three

⟨Standard module head⟩

⟨Physics defs: public parameters⟩

⟨Physics defs: public⟩

⟨Physics defs: interfaces⟩

contains

⟨Physics defs: procedures⟩

end module physics_defs
```

9.1.1 Units

Conversion from energy units to cross-section units.

⟨*Physics defs: public parameters*⟩≡

```
    real(default), parameter, public :: &
         conv = 0.38937966e12_default
```

Rescaling factor.

⟨*Physics defs: public parameters*⟩+≡

```
    real(default), parameter, public :: &
         pb_per_fb = 1.e-3_default
```

String for the default energy and cross-section units.

⟨*Physics defs: public parameters*⟩+≡

```
    character(*), parameter, public :: &
         energy_unit = "GeV"
    character(*), parameter, public :: &
         cross_section_unit = "fb"
```

9.1.2 SM and QCD constants

⟨*Physics defs: public parameters*⟩+≡

```
    real(default), parameter, public :: &
         NC = three, &
         CF = (NC**2 - one) / two / NC, &
         CA = NC, &
         TR = one / two
```

9.1.3 Parameter Reference values

These are used exclusively in the context of running QCD parameters. In other contexts, we rely on the uniform parameter set as provided by the model definition, modifiable by the user.

⟨*Physics defs: public parameters*⟩+≡

```
real(default), public, parameter :: MZ_REF = 91.188_default
real(default), public, parameter :: ALPHA_QCD_MZ_REF = 0.1178_default
real(default), public, parameter :: LAMBDA_QCD_REF = 200.e-3_default
```

9.1.4 Particle codes

Let us define a few particle codes independent of the model.

We need an UNDEFINED value:

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: UNDEFINED = 0
```

SM fermions:

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: ELECTRON = 11
integer, parameter, public :: ELECTRON_NEUTRINO = 12
integer, parameter, public :: MUON = 13
integer, parameter, public :: MUON_NEUTRINO = 14
integer, parameter, public :: TAU = 15
integer, parameter, public :: TAU_NEUTRINO = 16
```

Gauge bosons:

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: GLUON = 21
integer, parameter, public :: PHOTON = 22
integer, parameter, public :: Z_BOSON = 23
integer, parameter, public :: W_BOSON = 24
```

Light mesons:

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: PION = 111
integer, parameter, public :: PIPLUS = 211
integer, parameter, public :: PIMINUS = - PIPLUS
```

Di-Quarks:

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: UD0 = 2101
integer, parameter, public :: UD1 = 2103
integer, parameter, public :: UU1 = 2203
```

Mesons:

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: K0L = 130
integer, parameter, public :: K0S = 310
integer, parameter, public :: K0 = 311
```

```
integer, parameter, public :: KPLUS = 321
integer, parameter, public :: DPLUS = 411
integer, parameter, public :: D0 = 421
integer, parameter, public :: B0 = 511
integer, parameter, public :: BPLUS = 521
```

Light baryons:

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: PROTON = 2212
integer, parameter, public :: NEUTRON = 2112
integer, parameter, public :: DELTAPLUSPLUS = 2224
integer, parameter, public :: DELTAPLUS = 2214
integer, parameter, public :: DELTA0 = 2114
integer, parameter, public :: DELTAMINUS = 1114
```

Strange baryons:

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: SIGMAPLUS = 3222
integer, parameter, public :: SIGMA0 = 3212
integer, parameter, public :: SIGMAMINUS = 3112
```

Charmed baryons:

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: SIGMACPLUSPLUS = 4222
integer, parameter, public :: SIGMACPLUS = 4212
integer, parameter, public :: SIGMAC0 = 4112
```

Bottom baryons:

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: SIGMAB0 = 5212
integer, parameter, public :: SIGMABPLUS = 5222
```

81-100 are reserved for internal codes. Hadron and beam remnants:

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: BEAM_REMNANT = 9999
integer, parameter, public :: HADRON_REMNANT = 90
integer, parameter, public :: HADRON_REMNANT_SINGLET = 91
integer, parameter, public :: HADRON_REMNANT_TRIPLET = 92
integer, parameter, public :: HADRON_REMNANT_OCTET = 93
```

Further particle codes for internal use:

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: INTERNAL = 94
integer, parameter, public :: INVALID = 97

integer, parameter, public :: COMPOSITE = 99
```

9.1.5 Spin codes

Somewhat redundant, but for better readability we define named constants for spin types. If the mass is nonzero, this is equal to the number of degrees of freedom.

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public:: UNKNOWN = 0
integer, parameter, public :: SCALAR = 1, SPINOR = 2, VECTOR = 3, &
                              VECTORSPINOR = 4, TENSOR = 5
```

Isospin types and charge types are counted in an analogous way, where charge type 1 is charge 0, 2 is charge 1/3, and so on. Zero always means unknown. Note that charge and isospin types have an explicit sign.

Color types are defined as the dimension of the representation.

9.1.6 NLO status codes

Used to specify whether a `term_instance_t` of a `process_instance_t` is associated with a Born, real-subtracted, virtual-subtracted or subtraction-dummy matrix element.

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: BORN = 0
integer, parameter, public :: NLO_REAL = 1
integer, parameter, public :: NLO_VIRTUAL = 2
integer, parameter, public :: NLO_MISMATCH = 3
integer, parameter, public :: NLO_DGLAP = 4
integer, parameter, public :: NLO_SUBTRACTION = 5
integer, parameter, public :: NLO_FULL = 6
integer, parameter, public :: GKS = 7
integer, parameter, public :: COMPONENT_UNDEFINED = 99
```

`NLO_FULL` is not strictly a component status code but having it is convenient.

⟨*Physics defs: public*⟩≡

```
public :: component_status
```

⟨*Physics defs: interfaces*⟩≡

```
interface component_status
   module procedure component_status_of_string
   module procedure component_status_to_string
end interface
```

⟨*Physics defs: procedures*⟩≡

```
elemental function component_status_of_string (string) result (i)
  integer :: i
  type(string_t), intent(in) :: string
  select case (char(string))
  case ("Born")
     i = BORN
  case ("Real")
     i = NLO_REAL
  case ("Virtual")
     i = NLO_VIRTUAL
  case ("Mismatch")
```

```
      i = NLO_MISMATCH
    case ("Dglap")
      i = NLO_DGLAP
    case ("Subtraction")
      i = NLO_SUBTRACTION
    case ("Full")
      i = NLO_FULL
    case ("GKS")
      i = GKS
    case default
      i = COMPONENT_UNDEFINED
    end select
  end function component_status_of_string

  elemental function component_status_to_string (i) result (string)
    type(string_t) :: string
    integer, intent(in) :: i
    select case (i)
    case (BORN)
      string = "Born"
    case (NLO_REAL)
      string = "Real"
    case (NLO_VIRTUAL)
      string = "Virtual"
    case (NLO_MISMATCH)
      string = "Mismatch"
    case (NLO_DGLAP)
      string = "Dglap"
    case (NLO_SUBTRACTION)
      string = "Subtraction"
    case (NLO_FULL)
      string = "Full"
    case (GKS)
      string = "GKS"
    case default
      string = "Undefined"
    end select
  end function component_status_to_string
```

⟨*Physics defs: public*⟩+≡

```
  public :: is_nlo_component
```

⟨*Physics defs: procedures*⟩+≡

```
  elemental function is_nlo_component (comp) result (is_nlo)
    logical :: is_nlo
    integer, intent(in) :: comp
    select case (comp)
    case (BORN : GKS)
      is_nlo = .true.
    case default
      is_nlo = .false.
    end select
  end function is_nlo_component
```

⟨*Physics defs: public*⟩+≡

```
public :: is_subtraction_component
```

⟨*Physics defs: procedures*⟩+≡

```
function is_subtraction_component (emitter, nlo_type) result (is_subtraction)
  logical :: is_subtraction
  integer, intent(in) :: emitter, nlo_type
  is_subtraction = nlo_type == NLO_REAL .and. emitter < 0
end function is_subtraction_component
```

9.1.7 Threshold

Some commonly used variables for the threshold computation

⟨*Physics defs: public parameters*⟩+≡

```
integer, parameter, public :: THR_POS_WP = 3
integer, parameter, public :: THR_POS_WM = 4
integer, parameter, public :: THR_POS_B = 5
integer, parameter, public :: THR_POS_BBAR = 6
integer, parameter, public :: THR_POS_GLUON = 7

integer, parameter, public :: THR_EMITTER_OFFSET = 4

integer, parameter, public :: NO_FACTORIZATION = 0
integer, parameter, public :: FACTORIZATION_THRESHOLD = 1

integer, dimension(2), parameter, public :: ass_quark = [5, 6]
integer, dimension(2), parameter, public :: ass_boson = [3, 4]

integer, parameter, public :: PROC_MODE_UNDEFINED = 0
integer, parameter, public :: PROC_MODE_TT = 1
integer, parameter, public :: PROC_MODE_WBWB = 2
```

⟨*Physics defs: public*⟩+≡

```
public :: thr_leg
```

⟨*Physics defs: procedures*⟩+≡

```
function thr_leg (emitter) result (leg)
  integer :: leg
  integer, intent(in) :: emitter
  leg = emitter - THR_EMITTER_OFFSET
end function thr_leg
```

9.2 C-compatible Particle Type

For easy communication with C code, we introduce a simple C-compatible type for particles. The components are either default C integers or default C doubles.

The `c_prt` type is transparent, and its contents should be regarded as part of the interface.

⟨`c_particles.f90`⟩≡

```
⟨File header⟩

module c_particles

  use, intrinsic :: iso_c_binding !NODEP!

  use io_units
  use format_defs, only: FMT_14, FMT_19

⟨Standard module head⟩

⟨C Particles: public⟩

⟨C Particles: types⟩

contains

⟨C Particles: procedures⟩
end module c_particles
```

⟨*C Particles: public*⟩≡

```
  public :: c_prt_t
```

⟨*C Particles: types*⟩≡

```
  type, bind(C) :: c_prt_t
     integer(c_int) :: type = 0
     integer(c_int) :: pdg = 0
     integer(c_int) :: polarized = 0
     integer(c_int) :: h = 0
     real(c_double) :: pe = 0
     real(c_double) :: px = 0
     real(c_double) :: py = 0
     real(c_double) :: pz = 0
     real(c_double) :: p2 = 0
  end type c_prt_t
```

This is for debugging only, there is no C binding. It is a simplified version of `prt_write`.

⟨*C Particles: public*⟩+≡

```
  public :: c_prt_write
```

⟨*C Particles: procedures*⟩≡

```
  subroutine c_prt_write (prt, unit)
    type(c_prt_t), intent(in) :: prt
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
```

```
  write (u, "(1x,A)", advance="no")  "prt("
  write (u, "(I0,':')", advance="no")  prt%type
  if (prt%polarized /= 0) then
     write (u, "(I0,'/',I0,'|')", advance="no")  prt%pdg, prt%h
  else
     write (u, "(I0,'|')", advance="no") prt%pdg
  end if
  write (u, "(" // FMT_14 // ",';'," // FMT_14 // ",',',"  // &
       FMT_14 // ",',',"  // FMT_14 // ")", advance="no") &
       prt%pe, prt%px, prt%py, prt%pz
  write (u, "('|'," // FMT_19 // ")", advance="no")  prt%p2
  write (u, "(A)")  ")"
end subroutine c_prt_write
```

9.3 Lorentz algebra

Define Lorentz vectors, three-vectors, boosts, and some functions to manipulate them.

To make maximum use of this, all functions, if possible, are declared elemental (or pure, if this is not possible).

⟨lorentz.f90⟩≡

```
⟨File header⟩

module lorentz

⟨Use kinds with double⟩
  use numeric_utils
  use io_units
  use constants, only: pi, twopi, degree, zero, one, two, eps0, tiny_07
  use format_defs, only: FMT_11, FMT_13, FMT_15, FMT_19
  use format_utils, only: pac_fmt
  use diagnostics
  use c_particles

⟨Standard module head⟩

⟨Lorentz: public⟩

⟨Lorentz: public operators⟩

⟨Lorentz: public functions⟩

⟨Lorentz: types⟩

⟨Lorentz: parameters⟩

⟨Lorentz: interfaces⟩

contains

⟨Lorentz: procedures⟩
end module lorentz
```

9.3.1 Three-vectors

First of all, let us introduce three-vectors in a trivial way. The functions and overloaded elementary operations clearly are too much overhead, but we like to keep the interface for three-vectors and four-vectors exactly parallel. By the way, we might attach a label to a vector by extending the type definition later.

⟨Lorentz: public⟩≡

```
public :: vector3_t
```

⟨Lorentz: types⟩≡

```
type :: vector3_t
   real(default), dimension(3) :: p
end type vector3_t
```

Output a vector

⟨*Lorentz: public*⟩+≡
```
public :: vector3_write
```

⟨*Lorentz: procedures*⟩≡
```
subroutine vector3_write (p, unit, testflag)
  type(vector3_t), intent(in) :: p
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  character(len=7) :: fmt
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call pac_fmt (fmt, FMT_19, FMT_15, testflag)
  write(u, "(1x,A,3(1x," // fmt // "))") 'P = ', p%p
end subroutine vector3_write
```

This is a three-vector with zero components

⟨*Lorentz: public*⟩+≡
```
public :: vector3_null
```

⟨*Lorentz: parameters*⟩≡
```
type(vector3_t), parameter :: vector3_null = &
     vector3_t ([ zero, zero, zero ])
```

Canonical three-vector:

⟨*Lorentz: public*⟩+≡
```
public :: vector3_canonical
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function vector3_canonical (k) result (p)
  type(vector3_t) :: p
  integer, intent(in) :: k
  p = vector3_null
  p%p(k) = 1
end function vector3_canonical
```

A moving particle (k-axis, or arbitrary axis). Note that the function for the generic momentum cannot be elemental.

⟨*Lorentz: public*⟩+≡
```
public :: vector3_moving
```

⟨*Lorentz: interfaces*⟩≡
```
interface vector3_moving
   module procedure vector3_moving_canonical
   module procedure vector3_moving_generic
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function vector3_moving_canonical (p, k) result(q)
  type(vector3_t) :: q
  real(default), intent(in) :: p
  integer, intent(in) :: k
  q = vector3_null
  q%p(k) = p
```

```
end function vector3_moving_canonical
pure function vector3_moving_generic (p) result(q)
  real(default), dimension(3), intent(in) :: p
  type(vector3_t) :: q
  q%p = p
end function vector3_moving_generic
```

Equality and inequality

⟨*Lorentz: public operators*⟩≡

```
public :: operator(==), operator(/=)
```

⟨*Lorentz: interfaces*⟩+≡

```
interface operator(==)
   module procedure vector3_eq
end interface
interface operator(/=)
   module procedure vector3_neq
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function vector3_eq (p, q) result (r)
  logical :: r
  type(vector3_t), intent(in) :: p,q
  r = all (abs (p%p - q%p) < eps0)
end function vector3_eq
elemental function vector3_neq (p, q) result (r)
  logical :: r
  type(vector3_t), intent(in) :: p,q
  r = any (abs(p%p - q%p) > eps0)
end function vector3_neq
```

Define addition and subtraction

⟨*Lorentz: public operators*⟩+≡

```
public :: operator(+), operator(-)
```

⟨*Lorentz: interfaces*⟩+≡

```
interface operator(+)
   module procedure add_vector3
end interface
interface operator(-)
   module procedure sub_vector3
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function add_vector3 (p, q) result (r)
  type(vector3_t) :: r
  type(vector3_t), intent(in) :: p,q
  r%p = p%p + q%p
end function add_vector3
elemental function sub_vector3 (p, q) result (r)
  type(vector3_t) :: r
  type(vector3_t), intent(in) :: p,q
  r%p = p%p - q%p
end function sub_vector3
```

The multiplication sign is overloaded with scalar multiplication; similarly division:

⟨*Lorentz: public operators*⟩+≡
```
public :: operator(*), operator(/)
```

⟨*Lorentz: interfaces*⟩+≡
```
interface operator(*)
   module procedure prod_integer_vector3, prod_vector3_integer
   module procedure prod_real_vector3, prod_vector3_real
end interface
interface operator(/)
   module procedure div_vector3_real, div_vector3_integer
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function prod_real_vector3 (s, p) result (q)
  type(vector3_t) :: q
  real(default), intent(in) :: s
  type(vector3_t), intent(in) :: p
  q%p = s * p%p
end function prod_real_vector3
elemental function prod_vector3_real (p, s) result (q)
  type(vector3_t) :: q
  real(default), intent(in) :: s
  type(vector3_t), intent(in) :: p
  q%p = s * p%p
end function prod_vector3_real
elemental function div_vector3_real (p, s) result (q)
  type(vector3_t) :: q
  real(default), intent(in) :: s
  type(vector3_t), intent(in) :: p
  q%p = p%p/s
end function div_vector3_real
elemental function prod_integer_vector3 (s, p) result (q)
  type(vector3_t) :: q
  integer, intent(in) :: s
  type(vector3_t), intent(in) :: p
  q%p = s * p%p
end function prod_integer_vector3
elemental function prod_vector3_integer (p, s) result (q)
  type(vector3_t) :: q
  integer, intent(in) :: s
  type(vector3_t), intent(in) :: p
  q%p = s * p%p
end function prod_vector3_integer
elemental function div_vector3_integer (p, s) result (q)
  type(vector3_t) :: q
  integer, intent(in) :: s
  type(vector3_t), intent(in) :: p
  q%p = p%p/s
end function div_vector3_integer
```

The multiplication sign can also indicate scalar products:

⟨*Lorentz: interfaces*⟩+≡

```
interface operator(*)
   module procedure prod_vector3
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function prod_vector3 (p, q) result (s)
  real(default) :: s
  type(vector3_t), intent(in) :: p,q
  s = dot_product (p%p, q%p)
end function prod_vector3
```

⟨*Lorentz: public functions*⟩≡

```
public :: cross_product
```

⟨*Lorentz: interfaces*⟩+≡

```
interface cross_product
   module procedure vector3_cross_product
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function vector3_cross_product (p, q) result (r)
  type(vector3_t) :: r
  type(vector3_t), intent(in) :: p,q
  integer :: i
  do i=1,3
     r%p(i) = dot_product (p%p, matmul(epsilon_three(i,:,:), q%p))
  end do
end function vector3_cross_product
```

Exponentiation is defined only for integer powers. Odd powers mean take the square root; so `p**1` is the length of `p`.

⟨*Lorentz: public operators*⟩+≡

```
public :: operator(**)
```

⟨*Lorentz: interfaces*⟩+≡

```
interface operator(**)
   module procedure power_vector3
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function power_vector3 (p, e) result (s)
  real(default) :: s
  type(vector3_t), intent(in) :: p
  integer, intent(in) :: e
  s = dot_product (p%p, p%p)
  if (e/=2) then
     if (mod(e,2)==0) then
        s = s**(e/2)
     else
        s = sqrt(s)**e
     end if
  end if
end function power_vector3
```

Finally, we need a negation.

⟨*Lorentz: interfaces*⟩+≡
```
interface operator(-)
   module procedure negate_vector3
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function negate_vector3 (p) result (q)
  type(vector3_t) :: q
  type(vector3_t), intent(in) :: p
  integer :: i
  do i = 1, 3
     if (abs (p%p(i)) < eps0) then
        q%p(i) = 0
     else
        q%p(i) = -p%p(i)
     end if
  end do
end function negate_vector3
```

The sum function can be useful:

⟨*Lorentz: public functions*⟩+≡
```
public :: sum
```

⟨*Lorentz: interfaces*⟩+≡
```
interface sum
   module procedure sum_vector3
end interface
```

⟨*Lorentz: public*⟩+≡
```
public :: vector3_set_component
```

⟨*Lorentz: procedures*⟩+≡
```
subroutine vector3_set_component (p, i, value)
  type(vector3_t), intent(inout) :: p
  integer, intent(in) :: i
  real(default), intent(in) :: value
  p%p(i) = value
end subroutine vector3_set_component
```

⟨*Lorentz: procedures*⟩+≡
```
pure function sum_vector3 (p) result (q)
  type(vector3_t) :: q
  type(vector3_t), dimension(:), intent(in) :: p
  integer :: i
  do i=1, 3
     q%p(i) = sum (p%p(i))
  end do
end function sum_vector3
```

Any component:

⟨*Lorentz: public*⟩+≡
```
public :: vector3_get_component
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function vector3_get_component (p, k) result (c)
  type(vector3_t), intent(in) :: p
  integer, intent(in) :: k
  real(default) :: c
  c = p%p(k)
end function vector3_get_component
```

Extract all components. This is not elemental.

⟨*Lorentz: public*⟩+≡

```
public :: vector3_get_components
```

⟨*Lorentz: procedures*⟩+≡

```
pure function vector3_get_components (p) result (a)
  type(vector3_t), intent(in) :: p
  real(default), dimension(3) :: a
  a = p%p
end function vector3_get_components
```

This function returns the direction of a three-vector, i.e., a normalized three-vector. If the vector is null, we return a null vector.

⟨*Lorentz: public functions*⟩+≡

```
public :: direction
```

⟨*Lorentz: interfaces*⟩+≡

```
interface direction
   module procedure vector3_get_direction
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function vector3_get_direction (p) result (q)
  type(vector3_t) :: q
  type(vector3_t), intent(in) :: p
  real(default) :: pp
  pp = p**1
  if (pp > eps0) then
     q%p = p%p / pp
  else
     q%p = 0
  end if
end function vector3_get_direction
```

9.3.2 Four-vectors

In four-vectors the zero-component needs special treatment, therefore we do not use the standard operations. Sure, we pay for the extra layer of abstraction by losing efficiency; so we have to assume that the time-critical applications do not involve four-vector operations.

⟨*Lorentz: public*⟩+≡

```
public :: vector4_t
```

⟨*Lorentz: types*⟩+≡
```
type :: vector4_t
   real(default), dimension(0:3) :: p = &
      [zero, zero, zero, zero]
contains
⟨Lorentz: vector4: TBP⟩
end type vector4_t
```

Output a vector

⟨*Lorentz: public*⟩+≡
```
public :: vector4_write
```

⟨*Lorentz: vector4: TBP*⟩≡
```
procedure :: write => vector4_write
```

⟨*Lorentz: procedures*⟩+≡
```
subroutine vector4_write &
       (p, unit, show_mass, testflag, compressed, ultra)
  class(vector4_t), intent(in) :: p
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: show_mass, testflag, compressed, ultra
  logical :: comp, sm, tf, extreme
  integer :: u
  character(len=7) :: fmt
  real(default) :: m
  comp = .false.; if (present (compressed))  comp = compressed
  sm = .false.;  if (present (show_mass))  sm = show_mass
  tf = .false.;  if (present (testflag))  tf = testflag
  extreme = .false.; if (present (ultra))  extreme = ultra
  if (extreme) then
     call pac_fmt (fmt, FMT_19, FMT_11, testflag)
  else
     call pac_fmt (fmt, FMT_19, FMT_13, testflag)
  end if
  u = given_output_unit (unit);  if (u < 0)  return
  if (comp) then
     write (u, "(4(F12.3,1X))", advance="no")  p%p(0:3)
  else
     write (u, "(1x,A,1x," // fmt // ")") 'E = ', p%p(0)
     write (u, "(1x,A,3(1x," // fmt // "))") 'P = ', p%p(1:)
     if (sm) then
        m = p**1
        if (tf)  call pacify (m, tolerance = 1E-6_default)
        write (u, "(1x,A,1x," // fmt // ")") 'M = ', m
     end if
  end if
end subroutine vector4_write
```

Binary I/O

⟨*Lorentz: public*⟩+≡
```
public :: vector4_write_raw
public :: vector4_read_raw
```

⟨*Lorentz: procedures*⟩+≡
```
subroutine vector4_write_raw (p, u)
```

```
    type(vector4_t), intent(in) :: p
    integer, intent(in) :: u
    write (u) p%p
  end subroutine vector4_write_raw

  subroutine vector4_read_raw (p, u, iostat)
    type(vector4_t), intent(out) :: p
    integer, intent(in) :: u
    integer, intent(out), optional :: iostat
    read (u, iostat=iostat) p%p
  end subroutine vector4_read_raw
```

This is a four-vector with zero components

⟨*Lorentz: public*⟩+≡

```
  public :: vector4_null
```

⟨*Lorentz: parameters*⟩+≡

```
  type(vector4_t), parameter :: vector4_null = &
       vector4_t ([ zero, zero, zero, zero ])
```

Canonical four-vector:

⟨*Lorentz: public*⟩+≡

```
  public :: vector4_canonical
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function vector4_canonical (k) result (p)
    type(vector4_t) :: p
    integer, intent(in) :: k
    p = vector4_null
    p%p(k) = 1
  end function vector4_canonical
```

A particle at rest:

⟨*Lorentz: public*⟩+≡

```
  public :: vector4_at_rest
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function vector4_at_rest (m) result (p)
    type(vector4_t) :: p
    real(default), intent(in) :: m
    p = vector4_t ([ m, zero, zero, zero ])
  end function vector4_at_rest
```

A moving particle (k-axis, or arbitrary axis)

⟨*Lorentz: public*⟩+≡

```
  public :: vector4_moving
```

⟨*Lorentz: interfaces*⟩+≡

```
  interface vector4_moving
     module procedure vector4_moving_canonical
     module procedure vector4_moving_generic
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function vector4_moving_canonical (E, p, k) result (q)
  type(vector4_t) :: q
  real(default), intent(in) :: E, p
  integer, intent(in) :: k
  q = vector4_at_rest(E)
  q%p(k) = p
end function vector4_moving_canonical
elemental function vector4_moving_generic (E, p) result (q)
  type(vector4_t) :: q
  real(default), intent(in) :: E
  type(vector3_t), intent(in) :: p
  q%p(0) = E
  q%p(1:) = p%p
end function vector4_moving_generic
```

Equality and inequality

⟨*Lorentz: interfaces*⟩+≡

```
interface operator(==)
   module procedure vector4_eq
end interface
interface operator(/=)
   module procedure vector4_neq
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function vector4_eq (p, q) result (r)
  logical :: r
  type(vector4_t), intent(in) :: p,q
  r = all (abs (p%p - q%p) < eps0)
end function vector4_eq
elemental function vector4_neq (p, q) result (r)
  logical :: r
  type(vector4_t), intent(in) :: p,q
  r = any (abs (p%p - q%p) > eps0)
end function vector4_neq
```

Addition and subtraction:

⟨*Lorentz: interfaces*⟩+≡

```
interface operator(+)
   module procedure add_vector4
end interface
interface operator(-)
   module procedure sub_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function add_vector4 (p,q) result (r)
  type(vector4_t) :: r
  type(vector4_t), intent(in) :: p,q
  r%p = p%p + q%p
end function add_vector4
elemental function sub_vector4 (p,q) result (r)
  type(vector4_t) :: r
```

```
    type(vector4_t), intent(in) :: p,q
    r%p = p%p - q%p
  end function sub_vector4
```

We also need scalar multiplication and division:

⟨*Lorentz: interfaces*⟩+≡

```
  interface operator(*)
     module procedure prod_real_vector4, prod_vector4_real
     module procedure prod_integer_vector4, prod_vector4_integer
  end interface
  interface operator(/)
     module procedure div_vector4_real
     module procedure div_vector4_integer
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function prod_real_vector4 (s, p) result (q)
    type(vector4_t) :: q
    real(default), intent(in) :: s
    type(vector4_t), intent(in) :: p
    q%p = s * p%p
  end function prod_real_vector4
  elemental function prod_vector4_real (p, s) result (q)
    type(vector4_t) :: q
    real(default), intent(in) :: s
    type(vector4_t), intent(in) :: p
    q%p = s * p%p
  end function prod_vector4_real
  elemental function div_vector4_real (p, s) result (q)
    type(vector4_t) :: q
    real(default), intent(in) :: s
    type(vector4_t), intent(in) :: p
    q%p = p%p/s
  end function div_vector4_real
  elemental function prod_integer_vector4 (s, p) result (q)
    type(vector4_t) :: q
    integer, intent(in) :: s
    type(vector4_t), intent(in) :: p
    q%p = s * p%p
  end function prod_integer_vector4
  elemental function prod_vector4_integer (p, s) result (q)
    type(vector4_t) :: q
    integer, intent(in) :: s
    type(vector4_t), intent(in) :: p
    q%p = s * p%p
  end function prod_vector4_integer
  elemental function div_vector4_integer (p, s) result (q)
    type(vector4_t) :: q
    integer, intent(in) :: s
    type(vector4_t), intent(in) :: p
    q%p = p%p/s
  end function div_vector4_integer
```

Scalar products and squares in the Minkowski sense:

⟨*Lorentz: interfaces*⟩+≡

```
interface operator(*)
   module procedure prod_vector4
end interface
interface operator(**)
   module procedure power_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function prod_vector4 (p, q) result (s)
  real(default) :: s
  type(vector4_t), intent(in) :: p,q
  s = p%p(0)*q%p(0) - dot_product(p%p(1:), q%p(1:))
end function prod_vector4
```

The power operation for four-vectors is signed, i.e., `p**1` is positive for timelike and negative for spacelike vectors. Note that `(p**1)**2` is not necessarily equal to `p**2`.

⟨*Lorentz: procedures*⟩+≡

```
elemental function power_vector4 (p, e) result (s)
  real(default) :: s
  type(vector4_t), intent(in) :: p
  integer, intent(in) :: e
  s = p * p
  if (e /= 2) then
     if (mod(e, 2) == 0) then
        s = s**(e / 2)
     else if (s >= 0) then
        s = sqrt(s)**e
     else
        s = -(sqrt(abs(s))**e)
     end if
  end if
end function power_vector4
```

Finally, we introduce a negation

⟨*Lorentz: interfaces*⟩+≡

```
interface operator(-)
   module procedure negate_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function negate_vector4 (p) result (q)
  type(vector4_t) :: q
  type(vector4_t), intent(in) :: p
  integer :: i
  do i = 0, 3
     if (abs (p%p(i)) < eps0) then
        q%p(i) = 0
     else
        q%p(i) = -p%p(i)
     end if
```

```
    end do
  end function negate_vector4
```

The sum function can be useful:

⟨*Lorentz: interfaces*⟩+≡

```
  interface sum
     module procedure sum_vector4, sum_vector4_mask
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
  pure function sum_vector4 (p) result (q)
    type(vector4_t) :: q
    type(vector4_t), dimension(:), intent(in) :: p
    integer :: i
    do i = 0, 3
       q%p(i) = sum (p%p(i))
    end do
  end function sum_vector4

  pure function sum_vector4_mask (p, mask) result (q)
    type(vector4_t) :: q
    type(vector4_t), dimension(:), intent(in) :: p
    logical, dimension(:), intent(in) :: mask
    integer :: i
    do i = 0, 3
       q%p(i) = sum (p%p(i), mask=mask)
    end do
  end function sum_vector4_mask
```

9.3.3 Conversions

Manually set a component of the four-vector:

⟨*Lorentz: public*⟩+≡

```
  public :: vector4_set_component
```

⟨*Lorentz: procedures*⟩+≡

```
  subroutine vector4_set_component (p, k, c)
    type(vector4_t), intent(inout) :: p
    integer, intent(in) :: k
    real(default), intent(in) :: c
    p%p(k) = c
  end subroutine vector4_set_component
```

Any component:

⟨*Lorentz: public*⟩+≡

```
  public :: vector4_get_component
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function vector4_get_component (p, k) result (c)
    real(default) :: c
    type(vector4_t), intent(in) :: p
    integer, intent(in) :: k
```

```
    c = p%p(k)
  end function vector4_get_component
```

Extract all components. This is not elemental.

⟨*Lorentz: public*⟩+≡

```
  public :: vector4_get_components
```

⟨*Lorentz: procedures*⟩+≡

```
  pure function vector4_get_components (p) result (a)
    real(default), dimension(0:3) :: a
    type(vector4_t), intent(in) :: p
    a = p%p
  end function vector4_get_components
```

This function returns the space part of a four-vector, such that we can apply three-vector operations on it:

⟨*Lorentz: public functions*⟩+≡

```
  public :: space_part
```

⟨*Lorentz: interfaces*⟩+≡

```
  interface space_part
     module procedure vector4_get_space_part
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function vector4_get_space_part (p) result (q)
    type(vector3_t) :: q
    type(vector4_t), intent(in) :: p
    q%p = p%p(1:)
  end function vector4_get_space_part
```

This function returns the direction of a four-vector, i.e., a normalized three-vector. If the four-vector has zero space part, we return a null vector.

⟨*Lorentz: interfaces*⟩+≡

```
  interface direction
     module procedure vector4_get_direction
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function vector4_get_direction (p) result (q)
    type(vector3_t) :: q
    type(vector4_t), intent(in) :: p
    real(default) :: qq
    q%p = p%p(1:)
    qq = q**1
    if (abs(qq) > eps0) then
       q%p = q%p / qq
    else
       q%p = 0
    end if
  end function vector4_get_direction
```

Change the sign of the spatial part of a four-vector

⟨*Lorentz: public*⟩+≡
```
public :: vector4_invert_direction
```

⟨*Lorentz: procedures*⟩+≡
```
elemental subroutine vector4_invert_direction (p)
  type(vector4_t), intent(inout) :: p
  p%p(1:3) = -p%p(1:3)
end subroutine vector4_invert_direction
```

This function returns the four-vector as an ordinary array. A second version for an array of four-vectors.

⟨*Lorentz: public*⟩+≡
```
public :: assignment (=)
```

⟨*Lorentz: interfaces*⟩+≡
```
interface assignment (=)
   module procedure array_from_vector4_1, array_from_vector4_2, &
          array_from_vector3_1, array_from_vector3_2, &
          vector4_from_array, vector3_from_array
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
pure subroutine array_from_vector4_1 (a, p)
  real(default), dimension(:), intent(out) :: a
  type(vector4_t), intent(in) :: p
  a = p%p
end subroutine array_from_vector4_1

pure subroutine array_from_vector4_2 (a, p)
  type(vector4_t), dimension(:), intent(in) :: p
  real(default), dimension(:,:), intent(out) :: a
  integer :: i
  forall (i=1:size(p))
     a(:,i) = p(i)%p
  end forall
end subroutine array_from_vector4_2

pure subroutine array_from_vector3_1 (a, p)
  real(default), dimension(:), intent(out) :: a
  type(vector3_t), intent(in) :: p
  a = p%p
end subroutine array_from_vector3_1

pure subroutine array_from_vector3_2 (a, p)
  type(vector3_t), dimension(:), intent(in) :: p
  real(default), dimension(:,:), intent(out) :: a
  integer :: i
  forall (i=1:size(p))
     a(:,i) = p(i)%p
  end forall
end subroutine array_from_vector3_2

pure subroutine vector4_from_array (p, a)
  type(vector4_t), intent(out) :: p
```

```
    real(default), dimension(:), intent(in) :: a
    p%p(0:3) = a
  end subroutine vector4_from_array

  pure subroutine vector3_from_array (p, a)
    type(vector3_t), intent(out) :: p
    real(default), dimension(:), intent(in) :: a
    p%p(1:3) = a
  end subroutine vector3_from_array
```

⟨*Lorentz: public*⟩+≡

```
  public :: vector4
```

⟨*Lorentz: procedures*⟩+≡

```
  pure function vector4 (a) result (p)
    type(vector4_t) :: p
    real(default), intent(in), dimension(4) :: a
    p%p = a
  end function vector4
```

⟨*Lorentz: vector4: TBP*⟩+≡

```
  procedure :: to_pythia6 => vector4_to_pythia6
```

⟨*Lorentz: procedures*⟩+≡

```
  pure function vector4_to_pythia6 (vector4, m) result (p)
    real(double), dimension(1:5) :: p
    class(vector4_t), intent(in) :: vector4
    real(default), intent(in), optional :: m
    p(1:3) = vector4%p(1:3)
    p(4) = vector4%p(0)
    if (present (m)) then
       p(5) = m
    else
       p(5) = vector4 ** 1
    end if
  end function vector4_to_pythia6
```

Transform the momentum of a `c_prt` object into a four-vector and vice versa:

⟨*Lorentz: interfaces*⟩+≡

```
  interface assignment (=)
     module procedure vector4_from_c_prt, c_prt_from_vector4
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
  pure subroutine vector4_from_c_prt (p, c_prt)
    type(vector4_t), intent(out) :: p
    type(c_prt_t), intent(in) :: c_prt
    p%p(0) = c_prt%pe
    p%p(1) = c_prt%px
    p%p(2) = c_prt%py
    p%p(3) = c_prt%pz
  end subroutine vector4_from_c_prt

  pure subroutine c_prt_from_vector4 (c_prt, p)
```

```
    type(c_prt_t), intent(out) :: c_prt
    type(vector4_t), intent(in) :: p
    c_prt%pe = p%p(0)
    c_prt%px = p%p(1)
    c_prt%py = p%p(2)
    c_prt%pz = p%p(3)
    c_prt%p2 = p ** 2
  end subroutine c_prt_from_vector4
```

Initialize a `c_prt_t` object with the components of a four-vector as its kinematical entries. Compute the invariant mass, or use the optional mass-squared value instead.

⟨*Lorentz: public*⟩+≡

```
  public :: vector4_to_c_prt
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function vector4_to_c_prt (p, p2) result (c_prt)
    type(c_prt_t) :: c_prt
    type(vector4_t), intent(in) :: p
    real(default), intent(in), optional :: p2
    c_prt%pe = p%p(0)
    c_prt%px = p%p(1)
    c_prt%py = p%p(2)
    c_prt%pz = p%p(3)
    if (present (p2)) then
       c_prt%p2 = p2
    else
       c_prt%p2 = p ** 2
    end if
  end function vector4_to_c_prt
```

⟨*Lorentz: public*⟩+≡

```
  public :: phs_point_t
```

⟨*Lorentz: types*⟩+≡

```
  type :: phs_point_t
     type(vector4_t), dimension(:), allocatable :: p
     integer :: n_momenta = 0
  contains
  ⟨Lorentz: phs point: TBP⟩
  end type phs_point_t
```

⟨*Lorentz: interfaces*⟩+≡

```
  interface operator(==)
     module procedure phs_point_eq
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function phs_point_eq (phs_point_1, phs_point_2) result (eq)
    logical :: eq
    type(phs_point_t), intent(in) :: phs_point_1, phs_point_2
    eq = all (phs_point_1%p == phs_point_2%p)
  end function phs_point_eq
```

⟨*Lorentz: interfaces*⟩+≡

```
interface operator(*)
   module procedure prod_LT_phs_point
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function prod_LT_phs_point (L, phs_point) result (phs_point_LT)
  type(phs_point_t) :: phs_point_LT
  type(lorentz_transformation_t), intent(in) :: L
  type(phs_point_t), intent(in) :: phs_point
  phs_point_LT = size (phs_point%p)
  phs_point_LT%p = L * phs_point%p
end function prod_LT_phs_point
```

⟨*Lorentz: interfaces*⟩+≡

```
interface assignment(=)
   module procedure phs_point_from_n, phs_point_from_vector4, &
                    phs_point_from_phs_point
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
pure subroutine phs_point_from_n (phs_point, n_particles)
  type(phs_point_t), intent(out) :: phs_point
  integer, intent(in) :: n_particles
  allocate (phs_point%p (n_particles))
  phs_point%n_momenta = n_particles
  phs_point%p = vector4_null
end subroutine phs_point_from_n
```

⟨*Lorentz: phs point: TBP*⟩≡

⟨*Lorentz: procedures*⟩+≡

```
pure subroutine phs_point_from_vector4 (phs_point, p)
  type(phs_point_t), intent(out) :: phs_point
  type(vector4_t), intent(in), dimension(:) :: p
  phs_point%n_momenta = size (p)
  allocate (phs_point%p (phs_point%n_momenta), source = p)
end subroutine phs_point_from_vector4
```

⟨*Lorentz: procedures*⟩+≡

```
pure subroutine phs_point_from_phs_point (phs_point, phs_point_in)
  type(phs_point_t), intent(out) :: phs_point
  type(phs_point_t), intent(in) :: phs_point_in
  phs_point%n_momenta = phs_point_in%n_momenta
  allocate (phs_point%p (phs_point%n_momenta))
  phs_point%p = phs_point_in%p
end subroutine phs_point_from_phs_point
```

⟨*Lorentz: phs point: TBP*⟩+≡

```
procedure :: final => phs_point_final
```

⟨*Lorentz: procedures*⟩+≡
```
subroutine phs_point_final (phs_point)
  class(phs_point_t), intent(inout) :: phs_point
  deallocate (phs_point%p)
  phs_point%n_momenta = 0
end subroutine phs_point_final
```

⟨*Lorentz: phs point: TBP*⟩+≡
```
procedure :: write => phs_point_write
```

⟨*Lorentz: procedures*⟩+≡
```
subroutine phs_point_write (phs_point, unit, show_mass, testflag, &
    check_conservation, ultra, n_in)
  class(phs_point_t), intent(in) :: phs_point
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: show_mass
  logical, intent(in), optional :: testflag, ultra
  logical, intent(in), optional :: check_conservation
  integer, intent(in), optional :: n_in
  call vector4_write_set (phs_point%p, unit = unit, show_mass = show_mass, &
       testflag = testflag, check_conservation = check_conservation, &
       ultra = ultra, n_in = n_in)
end subroutine phs_point_write
```

⟨*Lorentz: phs point: TBP*⟩+≡
```
procedure :: get_x => phs_point_get_x
```

⟨*Lorentz: procedures*⟩+≡
```
function phs_point_get_x (phs_point, E_beam) result (x)
  real(default), dimension(2) :: x
  class(phs_point_t), intent(in) :: phs_point
  real(default), intent(in) :: E_beam
  x = phs_point%p(1:2)%p(0) / E_beam
end function phs_point_get_x
```

9.3.4 Angles

Return the angles in a canonical system. The angle ϕ is defined between $0 \leq \phi < 2\pi$. In degenerate cases, return zero.

⟨*Lorentz: public functions*⟩+≡
```
public :: azimuthal_angle
```

⟨*Lorentz: interfaces*⟩+≡
```
interface azimuthal_angle
   module procedure vector3_azimuthal_angle
   module procedure vector4_azimuthal_angle
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function vector3_azimuthal_angle (p) result (phi)
  real(default) :: phi
  type(vector3_t), intent(in) :: p
  if (any (abs (p%p(1:2)) > 0)) then
```

```
      phi = atan2(p%p(2), p%p(1))
      if (phi < 0) phi = phi + twopi
    else
      phi = 0
    end if
  end function vector3_azimuthal_angle
  elemental function vector4_azimuthal_angle (p) result (phi)
    real(default) :: phi
    type(vector4_t), intent(in) :: p
    phi = vector3_azimuthal_angle (space_part (p))
  end function vector4_azimuthal_angle
```

Azimuthal angle in degrees

⟨*Lorentz: public functions*⟩+≡

```
  public :: azimuthal_angle_deg
```

⟨*Lorentz: interfaces*⟩+≡

```
  interface azimuthal_angle_deg
     module procedure vector3_azimuthal_angle_deg
     module procedure vector4_azimuthal_angle_deg
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function vector3_azimuthal_angle_deg (p) result (phi)
    real(default) :: phi
    type(vector3_t), intent(in) :: p
    phi = vector3_azimuthal_angle (p) / degree
  end function vector3_azimuthal_angle_deg
  elemental function vector4_azimuthal_angle_deg (p) result (phi)
    real(default) :: phi
    type(vector4_t), intent(in) :: p
    phi = vector4_azimuthal_angle (p) / degree
  end function vector4_azimuthal_angle_deg
```

The azimuthal distance of two vectors. This is the difference of the azimuthal angles, but cannot be larger than π: The result is between $-\pi < \Delta\phi \leq \pi$.

⟨*Lorentz: public functions*⟩+≡

```
  public :: azimuthal_distance
```

⟨*Lorentz: interfaces*⟩+≡

```
  interface azimuthal_distance
     module procedure vector3_azimuthal_distance
     module procedure vector4_azimuthal_distance
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function vector3_azimuthal_distance (p, q) result (dphi)
    real(default) :: dphi
    type(vector3_t), intent(in) :: p,q
    dphi = vector3_azimuthal_angle (q) - vector3_azimuthal_angle (p)
    if (dphi <= -pi) then
       dphi = dphi + twopi
    else if (dphi > pi) then
       dphi = dphi - twopi
    end if
```

```
end function vector3_azimuthal_distance
elemental function vector4_azimuthal_distance (p, q) result (dphi)
  real(default) :: dphi
  type(vector4_t), intent(in) :: p,q
  dphi = vector3_azimuthal_distance &
       (space_part (p), space_part (q))
end function vector4_azimuthal_distance
```

The same in degrees:

⟨*Lorentz: public functions*⟩+≡

```
public :: azimuthal_distance_deg
```

⟨*Lorentz: interfaces*⟩+≡

```
interface azimuthal_distance_deg
   module procedure vector3_azimuthal_distance_deg
   module procedure vector4_azimuthal_distance_deg
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function vector3_azimuthal_distance_deg (p, q) result (dphi)
  real(default) :: dphi
  type(vector3_t), intent(in) :: p,q
  dphi = vector3_azimuthal_distance (p, q) / degree
end function vector3_azimuthal_distance_deg
elemental function vector4_azimuthal_distance_deg (p, q) result (dphi)
  real(default) :: dphi
  type(vector4_t), intent(in) :: p,q
  dphi = vector4_azimuthal_distance (p, q) / degree
end function vector4_azimuthal_distance_deg
```

The polar angle is defined $0 \leq \theta \leq \pi$. Note that `ATAN2` has the reversed order of arguments: `ATAN2(Y,X)`. Here, x is the 3-component while y is the transverse momentum which is always nonnegative. Therefore, the result is nonnegative as well.

⟨*Lorentz: public functions*⟩+≡

```
public :: polar_angle
```

⟨*Lorentz: interfaces*⟩+≡

```
interface polar_angle
   module procedure polar_angle_vector3
   module procedure polar_angle_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function polar_angle_vector3 (p) result (theta)
  real(default) :: theta
  type(vector3_t), intent(in) :: p
  if (any (abs (p%p) > 0)) then
     theta = atan2 (sqrt(p%p(1)**2 + p%p(2)**2), p%p(3))
  else
     theta = 0
  end if
end function polar_angle_vector3
elemental function polar_angle_vector4 (p) result (theta)
  real(default) :: theta
```

```
    type(vector4_t), intent(in) :: p
    theta = polar_angle (space_part (p))
  end function polar_angle_vector4
```

This is the cosine of the polar angle: $-1 \leq \cos\theta \leq 1$.

⟨*Lorentz: public functions*⟩+≡

```
  public :: polar_angle_ct
```

⟨*Lorentz: interfaces*⟩+≡

```
  interface polar_angle_ct
     module procedure polar_angle_ct_vector3
     module procedure polar_angle_ct_vector4
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function polar_angle_ct_vector3 (p) result (ct)
    real(default) :: ct
    type(vector3_t), intent(in) :: p
    if (any (abs (p%p) > 0)) then
       ct = p%p(3) / p**1
    else
       ct = 1
    end if
  end function polar_angle_ct_vector3
  elemental function polar_angle_ct_vector4 (p) result (ct)
    real(default) :: ct
    type(vector4_t), intent(in) :: p
    ct = polar_angle_ct (space_part (p))
  end function polar_angle_ct_vector4
```

The polar angle in degrees.

⟨*Lorentz: public functions*⟩+≡

```
  public :: polar_angle_deg
```

⟨*Lorentz: interfaces*⟩+≡

```
  interface polar_angle_deg
     module procedure polar_angle_deg_vector3
     module procedure polar_angle_deg_vector4
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function polar_angle_deg_vector3 (p) result (theta)
    real(default) :: theta
    type(vector3_t), intent(in) :: p
    theta = polar_angle (p) / degree
  end function polar_angle_deg_vector3
  elemental function polar_angle_deg_vector4 (p) result (theta)
    real(default) :: theta
    type(vector4_t), intent(in) :: p
    theta = polar_angle (p) / degree
  end function polar_angle_deg_vector4
```

This is the angle enclosed between two three-momenta. If one of the momenta is zero, we return an angle of zero. The range of the result is $0 \leq \theta \leq \pi$. If there is only one argument, take the positive z axis as reference.

⟨*Lorentz: public functions*⟩+≡
```
public :: enclosed_angle
```

⟨*Lorentz: interfaces*⟩+≡
```
interface enclosed_angle
   module procedure enclosed_angle_vector3
   module procedure enclosed_angle_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function enclosed_angle_vector3 (p, q) result (theta)
  real(default) :: theta
  type(vector3_t), intent(in) :: p, q
  theta = acos (enclosed_angle_ct (p, q))
end function enclosed_angle_vector3
elemental function enclosed_angle_vector4 (p, q) result (theta)
  real(default) :: theta
  type(vector4_t), intent(in) :: p, q
  theta = enclosed_angle (space_part (p), space_part (q))
end function enclosed_angle_vector4
```

The cosine of the enclosed angle.

⟨*Lorentz: public functions*⟩+≡
```
public :: enclosed_angle_ct
```

⟨*Lorentz: interfaces*⟩+≡
```
interface enclosed_angle_ct
   module procedure enclosed_angle_ct_vector3
   module procedure enclosed_angle_ct_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function enclosed_angle_ct_vector3 (p, q) result (ct)
  real(default) :: ct
  type(vector3_t), intent(in) :: p, q
  if (any (abs (p%p) > 0) .and. any (abs (q%p) > 0)) then
     ct = p*q / (p**1 * q**1)
     if (ct>1) then
        ct = 1
     else if (ct<-1) then
        ct = -1
     end if
  else
     ct = 1
  end if
end function enclosed_angle_ct_vector3
elemental function enclosed_angle_ct_vector4 (p, q) result (ct)
  real(default) :: ct
  type(vector4_t), intent(in) :: p, q
  ct = enclosed_angle_ct (space_part (p), space_part (q))
end function enclosed_angle_ct_vector4
```

The enclosed angle in degrees.

⟨*Lorentz: public functions*⟩+≡
```
public :: enclosed_angle_deg
```

⟨*Lorentz: interfaces*⟩+≡
```
interface enclosed_angle_deg
   module procedure enclosed_angle_deg_vector3
   module procedure enclosed_angle_deg_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function enclosed_angle_deg_vector3 (p, q) result (theta)
  real(default) :: theta
  type(vector3_t), intent(in) :: p, q
  theta = enclosed_angle (p, q) / degree
end function enclosed_angle_deg_vector3
elemental function enclosed_angle_deg_vector4 (p, q) result (theta)
  real(default) :: theta
  type(vector4_t), intent(in) :: p, q
  theta = enclosed_angle (p, q) / degree
end function enclosed_angle_deg_vector4
```

The polar angle of the first momentum w.r.t. the second momentum, evaluated in the rest frame of the second momentum. If the second four-momentum is not timelike, return zero.

⟨*Lorentz: public functions*⟩+≡
```
public :: enclosed_angle_rest_frame
public :: enclosed_angle_ct_rest_frame
public :: enclosed_angle_deg_rest_frame
```

⟨*Lorentz: interfaces*⟩+≡
```
interface enclosed_angle_rest_frame
   module procedure enclosed_angle_rest_frame_vector4
end interface
interface enclosed_angle_ct_rest_frame
   module procedure enclosed_angle_ct_rest_frame_vector4
end interface
interface enclosed_angle_deg_rest_frame
   module procedure enclosed_angle_deg_rest_frame_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function enclosed_angle_rest_frame_vector4 (p, q) result (theta)
  type(vector4_t), intent(in) :: p, q
  real(default) :: theta
  theta = acos (enclosed_angle_ct_rest_frame (p, q))
end function enclosed_angle_rest_frame_vector4
elemental function enclosed_angle_ct_rest_frame_vector4 (p, q) result (ct)
  type(vector4_t), intent(in) :: p, q
  real(default) :: ct
  if (invariant_mass(q) > 0) then
     ct = enclosed_angle_ct ( &
          space_part (boost(-q, invariant_mass (q)) * p), &
          space_part (q))
  else
```

```
      ct = 1
    end if
  end function enclosed_angle_ct_rest_frame_vector4
  elemental function enclosed_angle_deg_rest_frame_vector4 (p, q) &
      result (theta)
    type(vector4_t), intent(in) :: p, q
    real(default) :: theta
    theta = enclosed_angle_rest_frame (p, q) / degree
  end function enclosed_angle_deg_rest_frame_vector4
```

9.3.5 More kinematical functions (some redundant)

The scalar transverse momentum (assuming the z axis is longitudinal)

⟨*Lorentz: public functions*⟩+≡

```
  public :: transverse_part
```

⟨*Lorentz: interfaces*⟩+≡

```
  interface transverse_part
     module procedure transverse_part_vector4_beam_axis
     module procedure transverse_part_vector4_vector4
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function transverse_part_vector4_beam_axis (p) result (pT)
    real(default) :: pT
    type(vector4_t), intent(in) :: p
    pT = sqrt(p%p(1)**2 + p%p(2)**2)
  end function transverse_part_vector4_beam_axis

  elemental function transverse_part_vector4_vector4 (p1, p2) result (pT)
    real(default) :: pT
    type(vector4_t), intent(in) :: p1, p2
    real(default) :: p1_norm, p2_norm, p1p2, pT2
    p1_norm = space_part_norm(p1)**2
    p2_norm = space_part_norm(p2)**2
!    p1p2 = p1%p(1:3)*p2%p(1:3)
    p1p2 = vector4_get_space_part(p1) * vector4_get_space_part(p2)
    pT2 = (p1_norm*p2_norm - p1p2)/p1_norm
    pT = sqrt (pT2)
  end function transverse_part_vector4_vector4
```

The scalar longitudinal momentum (assuming the z axis is longitudinal). Identical to `momentum_z_component`.

⟨*Lorentz: public functions*⟩+≡

```
  public :: longitudinal_part
```

⟨*Lorentz: interfaces*⟩+≡

```
  interface longitudinal_part
     module procedure longitudinal_part_vector4
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function longitudinal_part_vector4 (p) result (pL)
  real(default) :: pL
  type(vector4_t), intent(in) :: p
  pL = p%p(3)
end function longitudinal_part_vector4
```

Absolute value of three-momentum

⟨*Lorentz: public functions*⟩+≡

```
public :: space_part_norm
```

⟨*Lorentz: interfaces*⟩+≡

```
interface space_part_norm
   module procedure space_part_norm_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function space_part_norm_vector4 (p) result (p3)
  real(default) :: p3
  type(vector4_t), intent(in) :: p
  p3 = sqrt (p%p(1)**2 + p%p(2)**2 + p%p(3)**2)
end function space_part_norm_vector4
```

The energy (the zeroth component)

⟨*Lorentz: public functions*⟩+≡

```
public :: energy
```

⟨*Lorentz: interfaces*⟩+≡

```
interface energy
   module procedure energy_vector4
   module procedure energy_vector3
   module procedure energy_real
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function energy_vector4 (p) result (E)
  real(default) :: E
  type(vector4_t), intent(in) :: p
  E = p%p(0)
end function energy_vector4
```

Alternative: The energy corresponding to a given momentum and mass. If the mass is omitted, it is zero

⟨*Lorentz: procedures*⟩+≡

```
elemental function energy_vector3 (p, mass) result (E)
  real(default) :: E
  type(vector3_t), intent(in) :: p
  real(default), intent(in), optional :: mass
  if (present (mass)) then
     E = sqrt (p**2 + mass**2)
  else
     E = p**1
  end if
end function energy_vector3
```

```
elemental function energy_real (p, mass) result (E)
  real(default) :: E
  real(default), intent(in) :: p
  real(default), intent(in), optional :: mass
  if (present (mass)) then
     E = sqrt (p**2 + mass**2)
  else
     E = abs (p)
  end if
end function energy_real
```

The invariant mass of four-momenta. Zero for lightlike, negative for spacelike momenta.

⟨*Lorentz: public functions*⟩+≡

```
public :: invariant_mass
```

⟨*Lorentz: interfaces*⟩+≡

```
interface invariant_mass
   module procedure invariant_mass_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function invariant_mass_vector4 (p) result (m)
  real(default) :: m
  type(vector4_t), intent(in) :: p
  real(default) :: msq
  msq = p*p
  if (msq >= 0) then
     m = sqrt (msq)
  else
     m = - sqrt (abs (msq))
  end if
end function invariant_mass_vector4
```

The invariant mass squared. Zero for lightlike, negative for spacelike momenta.

⟨*Lorentz: public functions*⟩+≡

```
public :: invariant_mass_squared
```

⟨*Lorentz: interfaces*⟩+≡

```
interface invariant_mass_squared
   module procedure invariant_mass_squared_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function invariant_mass_squared_vector4 (p) result (msq)
  real(default) :: msq
  type(vector4_t), intent(in) :: p
  msq = p*p
end function invariant_mass_squared_vector4
```

The transverse mass. If the mass squared is negative, this value also is negative.

⟨*Lorentz: public functions*⟩+≡

```
public :: transverse_mass
```

⟨*Lorentz: interfaces*⟩+≡

```
interface transverse_mass
   module procedure transverse_mass_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function transverse_mass_vector4 (p) result (m)
  real(default) :: m
  type(vector4_t), intent(in) :: p
  real(default) :: msq
  msq = p%p(0)**2 - p%p(1)**2 - p%p(2)**2
  if (msq >= 0) then
     m = sqrt (msq)
  else
     m = - sqrt (abs (msq))
  end if
end function transverse_mass_vector4
```

The rapidity (defined if particle is massive or $p_\perp > 0$)

⟨*Lorentz: public functions*⟩+≡

```
public :: rapidity
```

⟨*Lorentz: interfaces*⟩+≡

```
interface rapidity
   module procedure rapidity_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function rapidity_vector4 (p) result (y)
  real(default) :: y
  type(vector4_t), intent(in) :: p
  y = .5 * log( (energy (p) + longitudinal_part (p)) &
       &       /(energy (p) - longitudinal_part (p)))
end function rapidity_vector4
```

The pseudorapidity (defined if $p_\perp > 0$)

⟨*Lorentz: public functions*⟩+≡

```
public :: pseudorapidity
```

⟨*Lorentz: interfaces*⟩+≡

```
interface pseudorapidity
   module procedure pseudorapidity_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function pseudorapidity_vector4 (p) result (eta)
  real(default) :: eta
  type(vector4_t), intent(in) :: p
  eta = -log( tan (.5 * polar_angle (p)))
end function pseudorapidity_vector4
```

The rapidity distance (defined if both $p_\perp > 0$)

⟨*Lorentz: public functions*⟩+≡

```
public :: rapidity_distance
```

⟨*Lorentz: interfaces*⟩+≡
```
interface rapidity_distance
   module procedure rapidity_distance_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function rapidity_distance_vector4 (p, q) result (dy)
  type(vector4_t), intent(in) :: p, q
  real(default) :: dy
  dy = rapidity (q) - rapidity (p)
end function rapidity_distance_vector4
```

The pseudorapidity distance (defined if both $p_\perp > 0$)

⟨*Lorentz: public functions*⟩+≡
```
public :: pseudorapidity_distance
```

⟨*Lorentz: interfaces*⟩+≡
```
interface pseudorapidity_distance
   module procedure pseudorapidity_distance_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function pseudorapidity_distance_vector4 (p, q) result (deta)
  real(default) :: deta
  type(vector4_t), intent(in) :: p, q
  deta = pseudorapidity (q) - pseudorapidity (p)
end function pseudorapidity_distance_vector4
```

The distance on the $\eta - \phi$ cylinder:

⟨*Lorentz: public functions*⟩+≡
```
public :: eta_phi_distance
```

⟨*Lorentz: interfaces*⟩+≡
```
interface eta_phi_distance
   module procedure eta_phi_distance_vector4
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function eta_phi_distance_vector4 (p, q) result (dr)
  type(vector4_t), intent(in) :: p, q
  real(default) :: dr
  dr = sqrt ( &
       pseudorapidity_distance (p, q)**2 &
       + azimuthal_distance (p, q)**2)
end function eta_phi_distance_vector4
```

9.3.6 Lorentz transformations

⟨*Lorentz: public*⟩+≡
```
public :: lorentz_transformation_t
```

⟨*Lorentz: types*⟩+≡

```
type :: lorentz_transformation_t
   private
   real(default), dimension(0:3, 0:3) :: L
 contains
 ⟨Lorentz: lorentz transformation: TBP⟩
end type lorentz_transformation_t
```

Output:

⟨*Lorentz: public*⟩+≡

```
public :: lorentz_transformation_write
```

⟨*Lorentz: lorentz transformation: TBP*⟩≡

```
procedure :: write => lorentz_transformation_write
```

⟨*Lorentz: procedures*⟩+≡

```
subroutine lorentz_transformation_write (L, unit, testflag, ultra)
  class(lorentz_transformation_t), intent(in) :: L
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag, ultra
  integer :: u, i
  logical :: ult
  character(len=7) :: fmt
  ult = .false.; if (present (ultra)) ult = ultra
  if (ult) then
     call pac_fmt (fmt, FMT_19, FMT_11, ultra)
  else
     call pac_fmt (fmt, FMT_19, FMT_13, testflag)
  end if
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A,3(1x," // fmt // "))")  "L00 = ", L%L(0,0)
  write (u, "(1x,A,3(1x," // fmt // "))")  "L0j = ", L%L(0,1:3)
  do i = 1, 3
     write (u, "(1x,A,I0,A,3(1x," // fmt // "))")  &
          "L", i, "0 = ", L%L(i,0)
     write (u, "(1x,A,I0,A,3(1x," // fmt // "))")  &
          "L", i, "j = ", L%L(i,1:3)
  end do
end subroutine lorentz_transformation_write
```

Extract all components:

⟨*Lorentz: public*⟩+≡

```
public :: lorentz_transformation_get_components
```

⟨*Lorentz: procedures*⟩+≡

```
pure function lorentz_transformation_get_components (L) result (a)
  type(lorentz_transformation_t), intent(in) :: L
  real(default), dimension(0:3,0:3) :: a
  a = L%L
end function lorentz_transformation_get_components
```

9.3.7 Functions of Lorentz transformations

For the inverse, we make use of the fact that $\Lambda^{\mu\nu}\Lambda_{\mu\rho} = \delta^{\nu}_{\rho}$. So, lowering the indices and transposing is sufficient.

⟨*Lorentz: public functions*⟩+≡
```
public :: inverse
```

⟨*Lorentz: interfaces*⟩+≡
```
interface inverse
   module procedure lorentz_transformation_inverse
end interface
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function lorentz_transformation_inverse (L) result (IL)
  type(lorentz_transformation_t) :: IL
  type(lorentz_transformation_t), intent(in) :: L
  IL%L(0,0) = L%L(0,0)
  IL%L(0,1:) = -L%L(1:,0)
  IL%L(1:,0) = -L%L(0,1:)
  IL%L(1:,1:) = transpose(L%L(1:,1:))
end function lorentz_transformation_inverse
```

9.3.8 Invariants

These are used below. The first array index is varying fastest in `FORTRAN`; therefore the extra minus in the odd-rank tensor epsilon.

⟨*Lorentz: parameters*⟩+≡
```
integer, dimension(3,3), parameter :: delta_three = &
     & reshape( source = [ 1,0,0, 0,1,0, 0,0,1 ], &
     &          shape  = [3,3] )
integer, dimension(3,3,3), parameter :: epsilon_three = &
     & reshape( source = [ 0, 0,0,  0,0,-1,   0,1,0, &
     &                     0, 0,1,  0,0, 0,  -1,0,0, &
     &                     0,-1,0,  1,0, 0,   0,0,0 ],&
     &          shape = [3,3,3] )
```

This could be of some use:

⟨*Lorentz: public*⟩+≡
```
public :: identity
```

⟨*Lorentz: parameters*⟩+≡
```
type(lorentz_transformation_t), parameter :: &
     & identity = &
     & lorentz_transformation_t ( &
     & reshape( source = [ one, zero, zero, zero, &
     &                     zero, one, zero, zero, &
     &                     zero, zero, one, zero, &
     &                     zero, zero, zero, one ],&
     &          shape = [4,4] ) )
```

⟨*Lorentz: public*⟩+≡
```
public :: space_reflection
```

⟨*Lorentz: parameters*⟩+≡

```
type(lorentz_transformation_t), parameter :: &
     & space_reflection = &
     & lorentz_transformation_t ( &
     & reshape( source = [ one, zero, zero, zero, &
     &                     zero,-one, zero, zero, &
     &                     zero, zero,-one, zero, &
     &                     zero, zero, zero,-one ],&
     &          shape = [4,4] ) )
```

Builds a unit vector orthogal to the input vector in the xy-plane.

⟨*Lorentz: public functions*⟩+≡

```
public :: create_orthogonal
```

⟨*Lorentz: procedures*⟩+≡

```
function create_orthogonal (p_in) result (p_out)
  type(vector3_t), intent(in) :: p_in
  type(vector3_t) :: p_out
  real(default) :: ab
  ab = sqrt (p_in%p(1)**2 + p_in%p(2)**2)
  if (abs (ab) < eps0) then
    p_out%p(1) = 1
    p_out%p(2) = 0
    p_out%p(3) = 0
  else
    p_out%p(1) = p_in%p(2)
    p_out%p(2) = -p_in%p(1)
    p_out%p(3) = 0
    p_out = p_out / ab
  end if
end function create_orthogonal
```

⟨*Lorentz: public functions*⟩+≡

```
public :: create_unit_vector
```

⟨*Lorentz: procedures*⟩+≡

```
function create_unit_vector (p_in) result (p_out)
  type(vector4_t), intent(in) :: p_in
  type(vector3_t) :: p_out
  p_out%p = p_in%p(1:3) / space_part_norm (p_in)
end function create_unit_vector
```

⟨*Lorentz: public functions*⟩+≡

```
public :: normalize
```

⟨*Lorentz: procedures*⟩+≡

```
function normalize(p) result (p_norm)
  type(vector3_t) :: p_norm
  type(vector3_t), intent(in) :: p
  real(default) :: abs
  abs = sqrt (p%p(1)**2 + p%p(2)**2 + p%p(3)**2)
  p_norm = p / abs
end function normalize
```

Computes the invariant mass of the momenta sum given by the indices in `i_res_born` and the optional argument `i_emitter`.

⟨*Lorentz: public*⟩+≡
```
public :: compute_resonance_mass
```

⟨*Lorentz: procedures*⟩+≡
```
pure function compute_resonance_mass (p, i_res_born, i_gluon) result (m)
  real(default) :: m
  type(vector4_t), intent(in), dimension(:) :: p
  integer, intent(in), dimension(:) :: i_res_born
  integer, intent(in), optional :: i_gluon
  type(vector4_t) :: p_res
  p_res = get_resonance_momentum (p, i_res_born, i_gluon)
  m = p_res**1
end function compute_resonance_mass
```

⟨*Lorentz: public*⟩+≡
```
public :: get_resonance_momentum
```

⟨*Lorentz: procedures*⟩+≡
```
pure function get_resonance_momentum (p, i_res_born, i_gluon) result (p_res)
  type(vector4_t) :: p_res
  type(vector4_t), intent(in), dimension(:) :: p
  integer, intent(in), dimension(:) :: i_res_born
  integer, intent(in), optional :: i_gluon
  integer :: i
  p_res = vector4_null
  do i = 1, size (i_res_born)
     p_res = p_res + p (i_res_born(i))
  end do
  if (present (i_gluon)) p_res = p_res + p (i_gluon)
end function get_resonance_momentum
```

⟨*Lorentz: public*⟩+≡
```
public :: create_two_particle_decay
```

⟨*Lorentz: procedures*⟩+≡
```
function create_two_particle_decay (s, p1, p2) result (p_rest)
  type(vector4_t), dimension(3) :: p_rest
  real(default), intent(in) :: s
  type(vector4_t), intent(in) :: p1, p2
  real(default) :: m1_sq, m2_sq
  real(default) :: E1, E2, p
  m1_sq = p1**2; m2_sq = p2**2
  p = sqrt (lambda (s, m1_sq, m2_sq)) / (two * sqrt (s))
  E1 = sqrt (m1_sq + p**2); E2 = sqrt (m2_sq + p**2)
  p_rest(1)%p = [sqrt (s), zero, zero, zero]
  p_rest(2)%p(0) = E1
  p_rest(2)%p(1:3) = p * p1%p(1:3) / space_part_norm (p1)
  p_rest(3)%p(0) = E2; p_rest(3)%p(1:3) = -p_rest(2)%p(1:3)
end function create_two_particle_decay
```

This function creates a phase-space point for a $1 \to 3$ decay in the decaying particle's rest frame. There are three rest frames for this system, corresponding to s-, t,- and u-channel momentum exchange, also referred to as Gottfried-Jackson frames. Below, we choose the momentum with index 1 to be aligned along the z-axis. We then have

$$\begin{aligned} s_1 &= (p_1 + p_2)^2 , \\ s_2 &= (p_2 + p_3)^2 , \\ s_3 &= (p_1 + p_3)^2 , \\ s_1 + s_2 + s_3 &= s + m_1^2 + m_2^2 + m_3^2. \end{aligned}$$

From these we can construct

$$\begin{aligned} E_1^{R23} &= \frac{s - s_2 - m_1^2}{2\sqrt{s_2}} \quad & P_1^{R23} &= \frac{\lambda^{1/2}(s, s_2, m_1^2)}{2\sqrt{s_2}}, \\ E_2^{R23} &= \frac{s_2 + m_2^2 - m_3^2}{2\sqrt{s_2}} \quad & P_2^{R23} &= \frac{\lambda^{1/2}(s_2, m_2^2, m_3^2)}{2\sqrt{s_2}}, \\ E_3^{R23} &= \frac{s_2 + m_3^2 - m_2^2}{2\sqrt{s_2}} \quad & P_3^{R23} &= P_2^{R23}, \end{aligned}$$

where $R23$ denotes the Gottfried-Jackson frame of our choice. Finally, the scattering angle θ_{12}^{R23} between momentum 1 and 2 can be determined to be

$$\cos\theta_{12}^{R23} = \frac{(s - s_2 - m_1^2)(s_2 + m_2^2 - m_3^2) + 2s_2(m_1^2 + m_2^2 - s_1)}{\lambda^{1/2}(s, s_2, m_1^2)\lambda^{1/2}(s_2, m_2^2, m_3^2)}$$

⟨*Lorentz: public*⟩+≡

```
public :: create_three_particle_decay
```

⟨*Lorentz: procedures*⟩+≡

```
function create_three_particle_decay (p1, p2, p3) result (p_rest)
  type(vector4_t), dimension(4) :: p_rest
  type(vector4_t), intent(in) :: p1, p2, p3
  real(default) :: E1, E2, E3
  real(default) :: pr1, pr2, pr3
  real(default) :: s, s1, s2, s3
  real(default) :: m1_sq, m2_sq, m3_sq
  real(default) :: cos_theta_12
  type(vector3_t) :: v3_unit
  type(lorentz_transformation_t) :: rot
  m1_sq = p1**2
  m2_sq = p2**2
  m3_sq = p3**2
  s1 = (p1 + p2)**2
  s2 = (p2 + p3)**2
  s3 = (p3 + p1)**2
  s = s1 + s2 + s3 - m1_sq - m2_sq - m3_sq
  E1 = (s - s2 - m1_sq) / (two * sqrt (s2))
  E2 = (s2 + m2_sq - m3_sq) / (two * sqrt (s2))
  E3 = (s2 + m3_sq - m2_sq) / (two * sqrt (s2))
  pr1 = sqrt (lambda (s, s2, m1_sq)) / (two * sqrt (s2))
  pr2 = sqrt (lambda (s2, m2_sq, m3_sq)) / (two * sqrt(s2))
```

```
    pr3 = pr2
    cos_theta_12 = ((s - s2 - m1_sq) * (s2 + m2_sq - m3_sq) + two * s2 * (m1_sq + m2_sq - s1)) / &
         sqrt (lambda (s, s2, m1_sq) * lambda (s2, m2_sq, m3_sq))
    v3_unit%p = [zero, zero, one]
    p_rest(1)%p(0) = E1
    p_rest(1)%p(1:3) = v3_unit%p * pr1
    p_rest(2)%p(0) = E2
    p_rest(2)%p(1:3) = v3_unit%p * pr2
    p_rest(3)%p(0) = E3
    p_rest(3)%p(1:3) = v3_unit%p * pr3
    p_rest(4)%p(0) = (s + s2 - m1_sq) / (2 * sqrt (s2))
    p_rest(4)%p(1:3) = - p_rest(1)%p(1:3)
    rot = rotation (cos_theta_12, sqrt (one - cos_theta_12**2), 2)
    p_rest(2) = rot * p_rest(2)
    p_rest(3)%p(1:3) = - p_rest(2)%p(1:3)
  end function create_three_particle_decay
```

⟨*Lorentz: public*⟩+≡

```
  public :: evaluate_one_to_two_splitting_special
```

⟨*Lorentz: interfaces*⟩+≡

```
  abstract interface
     subroutine evaluate_one_to_two_splitting_special (p_origin, &
          p1_in, p2_in, p1_out, p2_out, msq_in, jac)
       import
       type(vector4_t), intent(in) :: p_origin
       type(vector4_t), intent(in) :: p1_in, p2_in
       type(vector4_t), intent(inout) :: p1_out, p2_out
       real(default), intent(in), optional :: msq_in
       real(default), intent(inout), optional :: jac
     end subroutine evaluate_one_to_two_splitting_special
  end interface
```

⟨*Lorentz: public*⟩+≡

```
  public :: generate_on_shell_decay
```

⟨*Lorentz: procedures*⟩+≡

```
  recursive subroutine generate_on_shell_decay (p_dec, &
      p_in, p_out, i_real, msq_in, jac, evaluate_special)
    type(vector4_t), intent(in) :: p_dec
    type(vector4_t), intent(in), dimension(:) :: p_in
    type(vector4_t), intent(inout), dimension(:) :: p_out
    integer, intent(in) :: i_real
    real(default), intent(in), optional :: msq_in
    real(default), intent(inout), optional :: jac
    procedure(evaluate_one_to_two_splitting_special), intent(in), &
          pointer, optional :: evaluate_special
    type(vector4_t) :: p_dec_new
    integer :: n_recoil
    n_recoil = size (p_in) - 1
    if (n_recoil > 1) then
       if (present (evaluate_special)) then
          call evaluate_special (p_dec, p_in(1), sum (p_in (2 : n_recoil + 1)), &
               p_out(i_real), p_dec_new)
```

```
         call generate_on_shell_decay (p_dec_new, p_in (2 : ), p_out, &
              i_real + 1, msq_in, jac, evaluate_special)
      else
         call evaluate_one_to_two_splitting (p_dec, p_in(1), &
              sum (p_in (2 : n_recoil + 1)), p_out(i_real), p_dec_new, msq_in, jac)
         call generate_on_shell_decay (p_dec_new, p_in (2 : ), p_out, &
              i_real + 1, msq_in, jac)
      end if
   else
      call evaluate_one_to_two_splitting (p_dec, p_in(1), p_in(2), &
           p_out(i_real), p_out(i_real + 1), msq_in, jac)
   end if

end subroutine generate_on_shell_decay

subroutine evaluate_one_to_two_splitting (p_origin, &
    p1_in, p2_in, p1_out, p2_out, msq_in, jac)
   type(vector4_t), intent(in) :: p_origin
   type(vector4_t), intent(in) :: p1_in, p2_in
   type(vector4_t), intent(inout) :: p1_out, p2_out
   real(default), intent(in), optional :: msq_in
   real(default), intent(inout), optional :: jac
   type(lorentz_transformation_t) :: L
   type(vector4_t) :: p1_rest, p2_rest
   real(default) :: m, msq, msq1, msq2
   real(default) :: E1, E2, p
   real(default) :: lda, rlda_soft

   call get_rest_frame (p1_in, p2_in, p1_rest, p2_rest)

   msq = p_origin**2; m = sqrt(msq)
   msq1 = p1_in**2; msq2 = p2_in**2

   lda = lambda (msq, msq1, msq2)
   if (lda < zero) then
      print *, 'Encountered lambda < 0 in 1 -> 2 splitting! '
      print *, 'lda: ', lda
      print *, 'm: ', m, 'msq: ', msq
      print *, 'm1: ', sqrt (msq1), 'msq1: ', msq1
      print *, 'm2: ', sqrt (msq2), 'msq2: ', msq2
      stop
   end if
   p = sqrt (lda) / (two * m)

   E1 = sqrt (msq1 + p**2)
   E2 = sqrt (msq2 + p**2)

   p1_out = shift_momentum (p1_rest, E1, p)
   p2_out = shift_momentum (p2_rest, E2, p)

   L = boost (p_origin, p_origin**1)
   p1_out = L  * p1_out
   p2_out = L  * p2_out
```

```
  if (present (jac) .and. present (msq_in)) then
     jac = jac * sqrt(lda) / msq
     rlda_soft = sqrt (lambda (msq_in, msq1, msq2))
     !!! We have to undo the Jacobian which has already been
     !!! supplied by the Born phase space.
     jac = jac * msq_in / rlda_soft
  end if

contains

 subroutine get_rest_frame (p1_in, p2_in, p1_out, p2_out)
   type(vector4_t), intent(in) :: p1_in, p2_in
   type(vector4_t), intent(out) :: p1_out, p2_out
   type(lorentz_transformation_t) :: L
   L = inverse (boost (p1_in + p2_in, (p1_in + p2_in)**1))
   p1_out = L * p1_in; p2_out = L * p2_in
 end subroutine get_rest_frame

 function shift_momentum (p_in, E, p) result (p_out)
   type(vector4_t) :: p_out
   type(vector4_t), intent(in) :: p_in
   real(default), intent(in) :: E, p
   type(vector3_t) :: vec
   vec = p_in%p(1:3) / space_part_norm (p_in)
   p_out = vector4_moving (E, p * vec)
 end function shift_momentum

end subroutine evaluate_one_to_two_splitting
```

9.3.9 Boosts

We build Lorentz transformations from boosts and rotations. In both cases we can supply a three-vector which defines the axis and (hyperbolic) angle. For a boost, this is the vector $\vec{\beta} = \vec{p}/E$, such that a particle at rest with mass m is boosted to a particle with three-vector $\vec{p}$. Here, we have

$$\beta = \tanh\chi = p/E, \qquad \gamma = \cosh\chi = E/m, \qquad \beta\gamma = \sinh\chi = p/m \tag{9.1}$$

⟨*Lorentz: public functions*⟩+≡

```
public :: boost
```

⟨*Lorentz: interfaces*⟩+≡

```
interface boost
   module procedure boost_from_rest_frame
   module procedure boost_from_rest_frame_vector3
   module procedure boost_generic
   module procedure boost_canonical
end interface
```

In the first form, the argument is some four-momentum, the space part of which determines a direction, and the associated mass (which is not checked against the four-momentum). The boost vector $\gamma\vec{\beta}$ is then given by $\vec{p}/m$. This boosts from the rest frame of a particle to the current frame. To be explicit, if $\vec{p}$ is

the momentum of a particle and m its mass, $L(\vec{p}/m)$ is the transformation that turns $(m;\vec{0})$ into $(E;\vec{p})$. Conversely, the inverse transformation boosts a vector *into* the rest frame of a particle, in particular $(E;\vec{p})$ into $(m;\vec{0})$.

⟨*Lorentz: procedures*⟩+≡

```
elemental function boost_from_rest_frame (p, m) result (L)
  type(lorentz_transformation_t) :: L
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: m
  L = boost_from_rest_frame_vector3 (space_part (p), m)
end function boost_from_rest_frame
elemental function boost_from_rest_frame_vector3 (p, m) result (L)
  type(lorentz_transformation_t) :: L
  type(vector3_t), intent(in) :: p
  real(default), intent(in) :: m
  type(vector3_t) :: beta_gamma
  real(default) :: bg2, g, c
  integer :: i,j
  if (m > eps0) then
     beta_gamma = p / m
     bg2 = beta_gamma**2
  else
     bg2 = 0
     L = identity
     return
  end if
  if (bg2 > eps0) then
     g = sqrt(1 + bg2);  c = (g-1)/bg2
  else
     g = one + bg2 / two
     c = one / two
  end if
  L%L(0,0)  = g
  L%L(0,1:) = beta_gamma%p
  L%L(1:,0) = L%L(0,1:)
  do i=1,3
     do j=1,3
        L%L(i,j) = delta_three(i,j) + c*beta_gamma%p(i)*beta_gamma%p(j)
     end do
  end do
end function boost_from_rest_frame_vector3
```

A canonical boost is a boost along one of the coordinate axes, which we may supply as an integer argument. Here, $\gamma\beta$ is scalar.

⟨*Lorentz: procedures*⟩+≡

```
elemental function boost_canonical (beta_gamma, k) result (L)
  type(lorentz_transformation_t) :: L
  real(default), intent(in) :: beta_gamma
  integer, intent(in) :: k
  real(default) :: g
  g = sqrt(1 + beta_gamma**2)
  L = identity
  L%L(0,0) = g
  L%L(0,k) = beta_gamma
  L%L(k,0) = L%L(0,k)
```

```
    L%L(k,k) = L%L(0,0)
  end function boost_canonical
```

Instead of a canonical axis, we can supply an arbitrary axis which need not be normalized. If it is zero, return the unit matrix.

⟨*Lorentz: procedures*⟩+≡

```
  elemental function boost_generic (beta_gamma, axis) result (L)
    type(lorentz_transformation_t) :: L
    real(default), intent(in) :: beta_gamma
    type(vector3_t), intent(in) :: axis
    if (any (abs (axis%p) > 0)) then
       L = boost_from_rest_frame_vector3 (beta_gamma * axis, axis**1)
    else
       L = identity
    end if
  end function boost_generic
```

9.3.10 Rotations

For a rotation, the vector defines the rotation axis, and its length the rotation angle.

⟨*Lorentz: public functions*⟩+≡

```
  public :: rotation
```

⟨*Lorentz: interfaces*⟩+≡

```
  interface rotation
     module procedure rotation_generic
     module procedure rotation_canonical
     module procedure rotation_generic_cs
     module procedure rotation_canonical_cs
  end interface
```

If $\cos\phi$ and $\sin\phi$ is already known, we do not have to calculate them. Of course, the user has to ensure that $\cos^2\phi + \sin^2\phi = 1$, and that the given axis `n` is normalized to one. In the second form, the length of `axis` is the rotation angle.

⟨*Lorentz: procedures*⟩+≡

```
  elemental function rotation_generic_cs (cp, sp, axis) result (R)
    type(lorentz_transformation_t) :: R
    real(default), intent(in) :: cp, sp
    type(vector3_t), intent(in) :: axis
    integer :: i,j
    R = identity
    do i=1,3
       do j=1,3
          R%L(i,j) = cp*delta_three(i,j) + (1-cp)*axis%p(i)*axis%p(j)  &
               &   - sp*dot_product(epsilon_three(i,j,:), axis%p)
       end do
    end do
  end function rotation_generic_cs
  elemental function rotation_generic (axis) result (R)
    type(lorentz_transformation_t) :: R
    type(vector3_t), intent(in) :: axis
```

```
    real(default) :: phi
    if (any (abs(axis%p) > 0)) then
       phi = abs(axis**1)
       R = rotation_generic_cs (cos(phi), sin(phi), axis/phi)
    else
       R = identity
    end if
  end function rotation_generic
```

Alternatively, give just the angle and label the coordinate axis by an integer.

⟨*Lorentz: procedures*⟩+≡

```
  elemental function rotation_canonical_cs (cp, sp, k) result (R)
    type(lorentz_transformation_t) :: R
    real(default), intent(in) :: cp, sp
    integer, intent(in) :: k
    integer :: i,j
    R = identity
    do i=1,3
       do j=1,3
          R%L(i,j) = -sp*epsilon_three(i,j,k)
       end do
       R%L(i,i) = cp
    end do
    R%L(k,k) = 1
  end function rotation_canonical_cs
  elemental function rotation_canonical (phi, k) result (R)
    type(lorentz_transformation_t) :: R
    real(default), intent(in) :: phi
    integer, intent(in) :: k
    R = rotation_canonical_cs(cos(phi), sin(phi), k)
  end function rotation_canonical
```

This is viewed as a method for the first argument (three-vector): Reconstruct the rotation that rotates it into the second three-vector.

⟨*Lorentz: public functions*⟩+≡

```
  public :: rotation_to_2nd
```

⟨*Lorentz: interfaces*⟩+≡

```
  interface rotation_to_2nd
     module procedure rotation_to_2nd_generic
     module procedure rotation_to_2nd_canonical
  end interface
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function rotation_to_2nd_generic (p, q) result (R)
    type(lorentz_transformation_t) :: R
    type(vector3_t), intent(in) :: p, q
    type(vector3_t) :: a, b, ab
    real(default) :: ct, st
    if (any (abs (p%p) > 0) .and. any (abs (q%p) > 0)) then
       a = direction (p)
       b = direction (q)
       ab = cross_product(a,b)
       ct = a * b;  st = ab**1
       if (abs(st) > eps0) then
          R = rotation_generic_cs (ct, st, ab / st)
```

```
      else if (ct < 0) then
         R = space_reflection
      else
         R = identity
      end if
   else
      R = identity
   end if
end function rotation_to_2nd_generic
```

The same for a canonical axis: The function returns the transformation that rotates the k-axis into the direction of p.

⟨*Lorentz: procedures*⟩+≡

```
elemental function rotation_to_2nd_canonical (k, p) result (R)
  type(lorentz_transformation_t) :: R
  integer, intent(in) :: k
  type(vector3_t), intent(in) :: p
  type(vector3_t) :: b, ab
  real(default) :: ct, st
  integer :: i, j
  if (any (abs (p%p) > 0)) then
     b = direction (p)
     ab%p = 0
     do i = 1, 3
        do j = 1, 3
           ab%p(j) = ab%p(j) + b%p(i) * epsilon_three(i,j,k)
        end do
     end do
     ct = b%p(k);  st = ab**1
     if (abs(st) > eps0) then
        R = rotation_generic_cs (ct, st, ab / st)
     else if (ct < 0) then
        R = space_reflection
     else
        R = identity
     end if
  else
     R = identity
  end if
end function rotation_to_2nd_canonical
```

9.3.11 Composite Lorentz transformations

This function returns the transformation that, given a pair of vectors $p_{1,2}$, (a) boosts from the rest frame of the c.m. system (with invariant mass m) into the lab frame where p_i are defined, and (b) turns the given axis (or the canonical vectors $\pm e_k$) in the rest frame into the directions of $p_{1,2}$ in the lab frame. Note that the energy components are not used; for a consistent result one should have $(p_1 + p_2)^2 = m^2$.

⟨*Lorentz: public functions*⟩+≡

```
public :: transformation
```

⟨*Lorentz: interfaces*⟩+≡

```
interface transformation
   module procedure transformation_rec_generic
   module procedure transformation_rec_canonical
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function transformation_rec_generic (axis, p1, p2, m) result (L)
  type(vector3_t), intent(in) :: axis
  type(vector4_t), intent(in) :: p1, p2
  real(default), intent(in) :: m
  type(lorentz_transformation_t) :: L
  L = boost (p1 + p2, m)
  L = L * rotation_to_2nd (axis, space_part (inverse (L) * p1))
end function transformation_rec_generic
elemental function transformation_rec_canonical (k, p1, p2, m) result (L)
  integer, intent(in) :: k
  type(vector4_t), intent(in) :: p1, p2
  real(default), intent(in) :: m
  type(lorentz_transformation_t) :: L
  L = boost (p1 + p2, m)
  L = L * rotation_to_2nd (k, space_part (inverse (L) * p1))
end function transformation_rec_canonical
```

9.3.12 Applying Lorentz transformations

Multiplying vectors and Lorentz transformations is straightforward.

⟨*Lorentz: interfaces*⟩+≡

```
interface operator(*)
   module procedure prod_LT_vector4
   module procedure prod_LT_LT
   module procedure prod_vector4_LT
end interface
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function prod_LT_vector4 (L, p) result (np)
  type(vector4_t) :: np
  type(lorentz_transformation_t), intent(in) :: L
  type(vector4_t), intent(in) :: p
  np%p = matmul (L%L, p%p)
end function prod_LT_vector4
elemental function prod_LT_LT (L1, L2) result (NL)
  type(lorentz_transformation_t) :: NL
  type(lorentz_transformation_t), intent(in) :: L1,L2
  NL%L = matmul (L1%L, L2%L)
end function prod_LT_LT
elemental function prod_vector4_LT (p, L) result (np)
  type(vector4_t) :: np
  type(vector4_t), intent(in) :: p
  type(lorentz_transformation_t), intent(in) :: L
  np%p = matmul (p%p, L%L)
end function prod_vector4_LT
```

9.3.13 Special Lorentz transformations

These routines have their application in the generation and extraction of angles in the phase-space sampling routine. Since this part of the program is time-critical, we calculate the composition of transformations directly instead of multiplying rotations and boosts.

This Lorentz transformation is the composition of a rotation by ϕ around the 3 axis, a rotation by θ around the 2 axis, and a boost along the 3 axis:

$$L = B_3(\beta\gamma)\, R_2(\theta)\, R_3(\phi) \tag{9.2}$$

Instead of the angles we provide sine and cosine.

⟨*Lorentz: public functions*⟩+≡
```
public :: LT_compose_r3_r2_b3
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function LT_compose_r3_r2_b3 &
     (cp, sp, ct, st, beta_gamma) result (L)
  type(lorentz_transformation_t) :: L
  real(default), intent(in) :: cp, sp, ct, st, beta_gamma
  real(default) :: gamma
  if (abs(beta_gamma) < eps0) then
     L%L(0,0)  = 1
     L%L(1:,0) = 0
     L%L(0,1:) = 0
     L%L(1,1:) = [  ct*cp, -ct*sp, st ]
     L%L(2,1:) = [     sp,     cp,  zero ]
     L%L(3,1:) = [ -st*cp,  st*sp, ct ]
  else
     gamma = sqrt(1 + beta_gamma**2)
     L%L(0,0)  = gamma
     L%L(1,0)  = 0
     L%L(2,0)  = 0
     L%L(3,0)  = beta_gamma
     L%L(0,1:) = beta_gamma * [ -st*cp,  st*sp, ct ]
     L%L(1,1:) =              [  ct*cp, -ct*sp, st ]
     L%L(2,1:) =              [     sp,     cp, zero ]
     L%L(3,1:) = gamma      * [ -st*cp,  st*sp, ct ]
  end if
end function LT_compose_r3_r2_b3
```

Different ordering:

$$L = B_3(\beta\gamma)\, R_3(\phi)\, R_2(\theta) \tag{9.3}$$

⟨*Lorentz: public functions*⟩+≡
```
public :: LT_compose_r2_r3_b3
```

⟨*Lorentz: procedures*⟩+≡
```
elemental function LT_compose_r2_r3_b3 &
     (ct, st, cp, sp, beta_gamma) result (L)
  type(lorentz_transformation_t) :: L
  real(default), intent(in) :: ct, st, cp, sp, beta_gamma
  real(default) :: gamma
  if (abs(beta_gamma) < eps0) then
     L%L(0,0)  = 1
```

```
      L%L(1:,0) = 0
      L%L(0,1:) = 0
      L%L(1,1:) = [  ct*cp,    -sp,     st*cp ]
      L%L(2,1:) = [  ct*sp,     cp,     st*sp ]
      L%L(3,1:) = [ -st   ,   zero,     ct    ]
   else
      gamma = sqrt(1 + beta_gamma**2)
      L%L(0,0)  = gamma
      L%L(1,0)  = 0
      L%L(2,0)  = 0
      L%L(3,0)  = beta_gamma
      L%L(0,1:) = beta_gamma * [ -st   ,   zero,     ct    ]
      L%L(1,1:) =              [  ct*cp,    -sp,     st*cp ]
      L%L(2,1:) =              [  ct*sp,     cp,     st*sp ]
      L%L(3,1:) = gamma      * [ -st   ,   zero,     ct    ]
   end if
end function LT_compose_r2_r3_b3
```

This function returns the previous Lorentz transformation applied to an arbitrary four-momentum and extracts the space part of the result:

$$\vec{n} = [B_3(\beta\gamma)\, R_2(\theta)\, R_3(\phi)\, p]_{\text{space part}} \tag{9.4}$$

The second variant applies if there is no rotation

⟨*Lorentz: public functions*⟩+≡

```
public :: axis_from_p_r3_r2_b3, axis_from_p_b3
```

⟨*Lorentz: procedures*⟩+≡

```
elemental function axis_from_p_r3_r2_b3 &
     (p, cp, sp, ct, st, beta_gamma) result (n)
  type(vector3_t) :: n
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: cp, sp, ct, st, beta_gamma
  real(default) :: gamma, px, py
  px = cp * p%p(1) - sp * p%p(2)
  py = sp * p%p(1) + cp * p%p(2)
  n%p(1) =  ct * px + st * p%p(3)
  n%p(2) = py
  n%p(3) = -st * px + ct * p%p(3)
  if (abs(beta_gamma) > eps0) then
     gamma = sqrt(1 + beta_gamma**2)
     n%p(3) = n%p(3) * gamma + p%p(0) * beta_gamma
  end if
end function axis_from_p_r3_r2_b3

elemental function axis_from_p_b3 (p, beta_gamma) result (n)
  type(vector3_t) :: n
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: beta_gamma
  real(default) :: gamma
  n%p = p%p(1:3)
  if (abs(beta_gamma) > eps0) then
     gamma = sqrt(1 + beta_gamma**2)
     n%p(3) = n%p(3) * gamma + p%p(0) * beta_gamma
```

```
    end if
  end function axis_from_p_b3
```

9.3.14 Special functions

The Källén function, mostly used for the phase space. This is equivalent to $\lambda(x, y, z) = x^2 + y^2 + z^2 - 2xy - 2xz - 2yz$.

⟨*Lorentz: public functions*⟩+≡

```
  public :: lambda
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental function lambda (m1sq, m2sq, m3sq)
    real(default) :: lambda
    real(default), intent(in) :: m1sq, m2sq, m3sq
    lambda = (m1sq - m2sq - m3sq)**2 - 4*m2sq*m3sq
  end function lambda
```

Return a pair of head-to-head colliding momenta, given the collider energy, particle masses, and optionally the momentum of the c.m. system.

⟨*Lorentz: public functions*⟩+≡

```
  public :: colliding_momenta
```

⟨*Lorentz: procedures*⟩+≡

```
  function colliding_momenta (sqrts, m, p_cm) result (p)
    type(vector4_t), dimension(2) :: p
    real(default), intent(in) :: sqrts
    real(default), dimension(2), intent(in), optional :: m
    real(default), intent(in), optional :: p_cm
    real(default), dimension(2) :: dmsq
    real(default) :: ch, sh
    real(default), dimension(2) :: E0, p0
    integer, dimension(2), parameter :: sgn = [1, -1]
    if (abs(sqrts) < eps0) then
       call msg_fatal (" Colliding beams: sqrts is zero (please set sqrts)")
       p = vector4_null;  return
    else if (sqrts <= 0) then
       call msg_fatal (" Colliding beams: sqrts is negative")
       p = vector4_null;  return
    end if
    if (present (m)) then
       dmsq = sgn * (m(1)**2-m(2)**2)
       E0 = (sqrts + dmsq/sqrts) / 2
       if (any (E0 < m)) then
          call msg_fatal &
               (" Colliding beams: beam energy is less than particle mass")
          p = vector4_null;  return
       end if
       p0 = sgn * sqrt (E0**2 - m**2)
    else
       E0 = sqrts / 2
       p0 = sgn * E0
    end if
```

```
    if (present (p_cm)) then
       sh = p_cm / sqrts
       ch = sqrt (1 + sh**2)
       p = vector4_moving (E0 * ch + p0 * sh, E0 * sh + p0 * ch, 3)
    else
       p = vector4_moving (E0, p0, 3)
    end if
  end function colliding_momenta
```

This subroutine is for the purpose of numerical checks and comparisons. The idea is to set a number to zero if it is numerically equivalent with zero. The equivalence is established by comparing with a `tolerance` argument. We implement this for vectors and transformations.

⟨*Lorentz: public functions*⟩+≡

```
  public :: pacify
```

⟨*Lorentz: interfaces*⟩+≡

```
  interface pacify
     module procedure pacify_vector3
     module procedure pacify_vector4
     module procedure pacify_LT
  end interface pacify
```

⟨*Lorentz: procedures*⟩+≡

```
  elemental subroutine pacify_vector3 (p, tolerance)
    type(vector3_t), intent(inout) :: p
    real(default), intent(in) :: tolerance
    where (abs (p%p) < tolerance)  p%p = zero
  end subroutine pacify_vector3

  elemental subroutine pacify_vector4 (p, tolerance)
    type(vector4_t), intent(inout) :: p
    real(default), intent(in) :: tolerance
    where (abs (p%p) < tolerance)  p%p = zero
  end subroutine pacify_vector4

  elemental subroutine pacify_LT (LT, tolerance)
    type(lorentz_transformation_t), intent(inout) :: LT
    real(default), intent(in) :: tolerance
    where (abs (LT%L) < tolerance)  LT%L = zero
  end subroutine pacify_LT
```

⟨*Lorentz: public*⟩+≡

```
  public :: vector_set_reshuffle
```

⟨*Lorentz: procedures*⟩+≡

```
  subroutine vector_set_reshuffle (p1, list, p2)
    type(vector4_t), intent(in), dimension(:), allocatable :: p1
    integer, intent(in), dimension(:), allocatable :: list
    type(vector4_t), intent(out), dimension(:), allocatable :: p2
    integer :: n, n_p
    n_p = size (p1)
    if (size (list) /= n_p) return
```

```
    allocate (p2 (n_p))
    do n = 1, n_p
      p2(n) = p1(list(n))
    end do
  end subroutine vector_set_reshuffle
```

⟨*Lorentz: public*⟩+≡

```
  public :: vector_set_is_cms
```

⟨*Lorentz: procedures*⟩+≡

```
  function vector_set_is_cms (p) result (is_cms)
    logical :: is_cms
    type(vector4_t), dimension(:), intent(in) :: p
    integer :: i
    type(vector4_t) :: p_sum
    p_sum%p = 0._default
    do i = 1, size (p)
       p_sum = p_sum + p(i)
    end do
    is_cms = p_sum%p(0) > zero .and. all (abs (p_sum%p(1:3)) < tiny_07)
  end function vector_set_is_cms
```

⟨*Lorentz: public*⟩+≡

```
  public :: vector4_write_set
```

⟨*Lorentz: procedures*⟩+≡

```
  subroutine vector4_write_set (p, unit, show_mass, testflag, &
        check_conservation, ultra, n_in)
    type(vector4_t), intent(in), dimension(:) :: p
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: show_mass
    logical, intent(in), optional :: testflag, ultra
    logical, intent(in), optional :: check_conservation
    integer, intent(in), optional :: n_in
    logical :: extreme
    integer :: i, j
    real(default), dimension(0:3) :: p_tot
    character(len=7) :: fmt
    integer :: u
    logical :: yorn, is_test
    integer :: n
    extreme = .false.; if (present (ultra))  extreme = ultra
    is_test = .false.; if (present (testflag)) is_test = testflag
    u = given_output_unit (unit);  if (u < 0)  return
    n = 2; if (present (n_in)) n = n_in
    p_tot = 0
    yorn = .false.; if (present (check_conservation)) yorn = check_conservation
    do i = 1, size (p)
      if (yorn .and. i > n) then
         forall (j=0:3) p_tot(j) = p_tot(j) - p(i)%p(j)
      else
         forall (j=0:3) p_tot(j) = p_tot(j) + p(i)%p(j)
      end if
      call vector4_write (p(i), u, show_mass=show_mass, &
```

```
          testflag=testflag, ultra=ultra)
      end do
      if (extreme) then
         call pac_fmt (fmt, FMT_19, FMT_11, testflag)
      else
         call pac_fmt (fmt, FMT_19, FMT_15, testflag)
      end if
      if (is_test)  call pacify (p_tot, 1.E-9_default)
      if (.not. is_test) then
         write (u, "(A5)") 'Total: '
         write (u, "(1x,A,1x," // fmt // ")")    "E = ", p_tot(0)
         write (u, "(1x,A,3(1x," // fmt // "))") "P = ", p_tot(1:)
      end if
    end subroutine vector4_write_set
```

⟨*Lorentz: public*⟩+≡

```
    public :: vector4_check_momentum_conservation
```

⟨*Lorentz: procedures*⟩+≡

```
    subroutine vector4_check_momentum_conservation (p, n_in, unit, &
       abs_smallness, rel_smallness, verbose)
      type(vector4_t), dimension(:), intent(in) :: p
      integer, intent(in) :: n_in
      integer, intent(in), optional :: unit
      real(default), intent(in), optional :: abs_smallness, rel_smallness
      logical, intent(in), optional :: verbose
      integer :: u, i
      type(vector4_t) :: psum_in, psum_out
      logical, dimension(0:3) :: p_diff
      logical :: verb
      u = given_output_unit (unit);  if (u < 0)  return
      verb = .false.; if (present (verbose)) verb = verbose
      psum_in = vector4_null
      do i = 1, n_in
         psum_in = psum_in + p(i)
      end do
      psum_out = vector4_null
      do i = n_in + 1, size (p)
         psum_out = psum_out + p(i)
      end do
      !!! !!! !!! Workaround for gfortran-4.8.4 bug
      do i = 0, 3
         p_diff(i) = vanishes (psum_in%p(i) - psum_out%p(i), &
              abs_smallness = abs_smallness, rel_smallness = rel_smallness)
      end do
      if (.not. all (p_diff)) then
         call msg_warning ("Momentum conservation: FAIL", unit = u)
         if (verb) then
            write (u, "(A)") "Incoming:"
            call vector4_write (psum_in, u)
            write (u, "(A)") "Outgoing:"
            call vector4_write (psum_out, u)
         end if
      else
```

```
      write (u, "(A)") "Momentum conservation: CHECK"
    end if
  end subroutine vector4_check_momentum_conservation
```

This computes the quantities

$$\langle ij\rangle = \sqrt{|S_{ij}|}e^{i\phi_{ij}}, [ij] \qquad = \sqrt{|S_{ij}|}e^{1\tilde{\phi}_{ij}},$$

with $S_{ij} = (p_i + p_j)^2$. The phase space factor ϕ_{ij} is determined by

$$\cos\phi_{ij} = \frac{p_i^1 p_j^+ - p_j^1 p_i^+}{\sqrt{p_i^+ p_j^+ S_{ij}}}, \sin\phi_{ij} \qquad = \frac{p_i^2 p_j^+ - p_j^2 p_i^+}{\sqrt{p_i^+ p_j^+ S_{ij}}}.$$

After $\langle ij\rangle$ has been computed according to these formulae, $[ij]$ can be obtained by using the relation $S_{ij} = \langle ij\rangle[ji]$ and taking into account that $[ij] = -[ji]$. Thus, a minus-sign has to be applied.

⟨*Lorentz: public*⟩+≡

```
  public :: spinor_product
```

⟨*Lorentz: procedures*⟩+≡

```
  subroutine spinor_product (p1, p2, prod1, prod2)
    type(vector4_t), intent(in) :: p1, p2
    complex(default), intent(out) :: prod1, prod2
    real(default) :: sij
    complex(default) :: phase
    real(default) :: pp_1, pp_2
    pp_1 = p1%p(0) + p1%p(3)
    pp_2 = p2%p(0) + p2%p(3)
    sij = (p1+p2)**2
    phase = cmplx ((p1%p(1)*pp_2 - p2%p(1)*pp_1)/sqrt (sij*pp_1*pp_2), &
                   (p1%p(2)*pp_2 - p2%p(2)*pp_1)/sqrt (sij*pp_1*pp_2), &
                    default)
    !!! <ij>
    prod1 = sqrt (sij) * phase
    !!! [ij]
    if (abs(prod1) > 0) then
       prod2 = - sij / prod1
    else
       prod2 = 0
    end if
  end subroutine spinor_product
```

9.4 Special Physics functions

Here, we declare functions that are specific for the Standard Model, including QCD: fixed and running α_s, Catani-Seymour dipole terms, loop functions, etc.

To make maximum use of this, all functions, if possible, are declared elemental (or pure, if this is not possible).

⟨sm_physics.f90⟩≡

```
⟨File header⟩

module sm_physics

⟨Use kinds⟩
  use io_units
  use constants
  use numeric_utils
  use diagnostics
  use physics_defs
  use lorentz

⟨Standard module head⟩

⟨SM physics: public⟩

⟨SM physics: parameters⟩

contains

⟨SM physics: procedures⟩

end module sm_physics
```

9.4.1 Running α_s

Then we define the coefficients of the beta function of QCD (as a reference cf. the Particle Data Group), where n_f is the number of active flavors in two different schemes:

$$\beta_0 = 11 - \frac{2}{3} n_f \tag{9.5}$$

$$\beta_1 = 51 - \frac{19}{3} n_f \tag{9.6}$$

$$\beta_2 = 2857 - \frac{5033}{9} n_f + \frac{325}{27} n_f^2 \tag{9.7}$$

$$b_0 = \frac{1}{12\pi} \left(11 C_A - 2 n_f\right) \tag{9.8}$$

$$b_1 = \frac{1}{24\pi^2} \left(17 C_A^2 - 5 C_A n_f - 3 C_F n_f\right) \tag{9.9}$$

$$b_2 = \frac{1}{(4\pi)^3} \left(\frac{2857}{54} C_A^3 - \frac{1415}{54} * C_A^2 n_f - \frac{205}{18} C_A C_F n_f + C_F^2 n_f + \frac{79}{54} C_A n_f * *2 + \frac{11}{9} C_F n_f * *2 \right) \tag{9.10}$$

⟨*SM physics: public*⟩≡

```
public :: beta0, beta1, beta2, coeff_b0, coeff_b1, coeff_b2
```

⟨*SM physics: procedures*⟩≡

```
pure function beta0 (nf)
  real(default), intent(in) :: nf
  real(default) :: beta0
  beta0 = 11.0_default - two/three * nf
end function beta0

pure function beta1 (nf)
  real(default), intent(in) :: nf
  real(default) :: beta1
  beta1 = 51.0_default - 19.0_default/three * nf
end function beta1

pure function beta2 (nf)
  real(default), intent(in) :: nf
  real(default) :: beta2
  beta2 = 2857.0_default - 5033.0_default / 9.0_default * &
                  nf + 325.0_default/27.0_default * nf**2
end function beta2

pure function coeff_b0 (nf)
  real(default), intent(in) :: nf
  real(default) :: coeff_b0
  coeff_b0 = (11.0_default * CA - two * nf) / (12.0_default * pi)
end function coeff_b0

pure function coeff_b1 (nf)
  real(default), intent(in) :: nf
  real(default) :: coeff_b1
  coeff_b1 = (17.0_default * CA**2 - five * CA * nf - three * CF * nf) / &
             (24.0_default * pi**2)
end function coeff_b1

pure function coeff_b2 (nf)
  real(default), intent(in) :: nf
  real(default) :: coeff_b2
  coeff_b2 = (2857.0_default/54.0_default * CA**3 - &
                  1415.0_default/54.0_default * &
                  CA**2 * nf - 205.0_default/18.0_default * CA*CF*nf &
                  + 79.0_default/54.0_default * CA*nf**2 + &
                  11.0_default/9.0_default * CF * nf**2) / (four*pi)**3
end function coeff_b2
```

There should be two versions of running α_s, one which takes the scale and Λ_{QCD} as input, and one which takes the scale and e.g. $\alpha_s(m_Z)$ as input. Here, we take the one which takes the QCD scale and scale as inputs from the PDG book.

⟨*SM physics: public*⟩+≡

```
public :: running_as, running_as_lam
```

⟨*SM physics: procedures*⟩+≡

```
pure function running_as (scale, al_mz, mz, order, nf) result (ascale)
```

```
  real(default), intent(in) :: scale
  real(default), intent(in), optional :: al_mz, nf, mz
  integer, intent(in), optional :: order
  integer :: ord
  real(default) :: az, m_z, as_log, n_f, b0, b1, b2, ascale
  real(default) :: as0, as1
  if (present (mz)) then
     m_z = mz
  else
     m_z = MZ_REF
  end if
  if (present (order)) then
     ord = order
  else
     ord = 0
  end if
  if (present (al_mz)) then
     az = al_mz
  else
     az = ALPHA_QCD_MZ_REF
  end if
  if (present (nf)) then
     n_f = nf
  else
     n_f = 5
  end if
  b0 = coeff_b0 (n_f)
  b1 = coeff_b1 (n_f)
  b2 = coeff_b2 (n_f)
  as_log = one + b0 * az * log(scale**2/m_z**2)
  as0 = az / as_log
  as1 = as0 - as0**2 * b1/b0 * log(as_log)
  select case (ord)
  case (0)
     ascale = as0
  case (1)
     ascale = as1
  case (2)
     ascale = as1 + as0**3 * (b1**2/b0**2 * ((log(as_log))**2 - &
          log(as_log) + as_log - one) - b2/b0 * (as_log - one))
  case default
     ascale = as0
  end select
end function running_as

pure function running_as_lam (nf, scale, lambda, order) result (ascale)
  real(default), intent(in) :: nf, scale
  real(default), intent(in), optional :: lambda
  integer, intent(in), optional :: order
  real(default) :: lambda_qcd
  real(default) :: as0, as1, logmul, b0, b1, b2, ascale
  integer :: ord
  if (present (lambda)) then
     lambda_qcd = lambda
```

```
    else
       lambda_qcd = LAMBDA_QCD_REF
    end if
    if (present (order)) then
       ord = order
    else
       ord = 0
    end if
    b0 = beta0(nf)
    logmul = log(scale**2/lambda_qcd**2)
    as0 = four*pi / b0 / logmul
    if (ord > 0) then
       b1 = beta1(nf)
       as1 = as0 * (one - two* b1 / b0**2 * log(logmul) / logmul)
    end if
    select case (ord)
    case (0)
       ascale = as0
    case (1)
       ascale = as1
    case (2)
       b2 = beta2(nf)
       ascale = as1 + as0 * four * b1**2/b0**4/logmul**2 * &
            ((log(logmul) - 0.5_default)**2 + &
             b2*b0/8.0_default/b1**2 - five/four)
    case default
       ascale = as0
    end select
  end function running_as_lam
```

9.4.2 Catani-Seymour Parameters

These are fundamental constants of the Catani-Seymour dipole formalism. Since the corresponding parameters for the gluon case depend on the number of flavors which is treated as an argument, there we do have functions and not parameters.

$$\gamma_q = \gamma_{\bar{q}} = \frac{3}{2} C_F \qquad \gamma_g = \frac{11}{6} C_A - \frac{2}{3} T_R N_f \tag{9.11}$$

$$K_q = K_{\bar{q}} = \left(\frac{7}{2} - \frac{\pi^2}{6}\right) C_F \qquad K_g = \left(\frac{67}{18} - \frac{\pi^2}{6}\right) C_A - \frac{10}{9} T_R N_f \tag{9.12}$$

⟨*SM physics: parameters*⟩≡

```
  real(kind=default), parameter, public ::  gamma_q = three/two * CF, &
     k_q = (7.0_default/two - pi**2/6.0_default) * CF
```

⟨*SM physics: public*⟩+≡

```
  public :: gamma_g, k_g
```

⟨*SM physics: procedures*⟩+≡

```
  elemental function gamma_g (nf) result (gg)
     real(kind=default), intent(in) :: nf
     real(kind=default) :: gg
```

```
   gg = 11.0_default/6.0_default * CA - two/three * TR * nf
end function gamma_g

elemental function k_g (nf) result (kg)
   real(kind=default), intent(in) :: nf
   real(kind=default) :: kg
   kg = (67.0_default/18.0_default - pi**2/6.0_default) * CA - &
              10.0_default/9.0_default * TR * nf
end function k_g
```

9.4.3 Mathematical Functions

The dilogarithm. This simplified version is bound to double precision, and restricted to argument values less or equal to unity, so we do not need complex algebra. The wrapper converts it to default precision (which is, of course, a no-op if double=default).

The routine calculates the dilogarithm through mapping on the area where there is a quickly convergent series (adapted from an F77 routine by Hans Kuijf, 1988): Map x such that x is not in the neighbourhood of 1. Note that $|z| = -\ln(1-x)$ is always smaller than 1.10, but $\frac{1.10^{19}}{19!}\text{Bernoulli}_{19} = 2.7\times10^{-15}$.

⟨*SM physics: public*⟩+≡

```
public :: Li2
```

⟨*SM physics: procedures*⟩+≡

```
elemental function Li2 (x)
    use kinds, only: double
    real(default), intent(in) :: x
    real(default) :: Li2
    Li2 = real( Li2_double (real(x, kind=double)), kind=default)
end function Li2
```

⟨*SM physics: procedures*⟩+≡

```
elemental function Li2_double (x)  result (Li2)
  use kinds, only: double
  real(kind=double), intent(in) :: x
  real(kind=double) :: Li2
  real(kind=double), parameter :: pi2_6 = pi**2/6
  if (abs(1-x) < tiny_07) then
     Li2 = pi2_6
  else if (abs(1-x) <  0.5_double) then
     Li2 = pi2_6 - log(1-x) * log(x) - Li2_restricted (1-x)
  else if (abs(x) > 1.d0) then
     ! Li2 = 0
     ! call msg_bug (" Dilogarithm called outside of defined range.")
     !!! Reactivate Dilogarithm identity
      Li2 = -pi2_6 - 0.5_default * log(-x) * log(-x) - Li2_restricted (1/x)
  else
     Li2 = Li2_restricted (x)
  end if
contains
  elemental function Li2_restricted (x) result (Li2)
```

```
      real(kind=double), intent(in) :: x
      real(kind=double) :: Li2
      real(kind=double) :: tmp, z, z2
      z = - log (1-x)
      z2 = z**2
! Horner's rule for the powers z^3 through z^19
      tmp = 43867._double/798._double
      tmp = tmp * z2 /342._double - 3617._double/510._double
      tmp = tmp * z2 /272._double + 7._double/6._double
      tmp = tmp * z2 /210._double - 691._double/2730._double
      tmp = tmp * z2 /156._double + 5._double/66._double
      tmp = tmp * z2 /110._double - 1._double/30._double
      tmp = tmp * z2 / 72._double + 1._double/42._double
      tmp = tmp * z2 / 42._double - 1._double/30._double
      tmp = tmp * z2 / 20._double + 1._double/6._double
! The first three terms of the power series
      Li2 = z2 * z * tmp / 6._double - 0.25_double * z2 + z
    end function Li2_restricted
  end function Li2_double
```

9.4.4 Loop Integrals

These functions appear in the calculation of the effective one-loop coupling of a (pseudo)scalar to a vector boson pair.

⟨*SM physics: public*⟩+≡

```
  public :: faux
```

⟨*SM physics: procedures*⟩+≡

```
  elemental function faux (x) result (y)
    real(default), intent(in) :: x
    complex(default) :: y
    if (1 <= x) then
       y = asin(sqrt(1/x))**2
    else
       y = - 1/4.0_default * (log((1 + sqrt(1 - x))/ &
            (1 - sqrt(1 - x))) - cmplx (0.0_default, pi, kind=default))**2
    end if
  end function faux
```

⟨*SM physics: public*⟩+≡

```
  public :: fonehalf
```

⟨*SM physics: procedures*⟩+≡

```
  elemental function fonehalf (x) result (y)
    real(default), intent(in) :: x
    complex(default) :: y
    if (abs(x) < eps0) then
       y = 0
    else
       y = - 2.0_default * x * (1 + (1 - x) * faux(x))
    end if
  end function fonehalf
```

⟨*SM physics: public*⟩+≡
```
public :: fonehalf_pseudo
```

⟨*SM physics: procedures*⟩+≡
```
function fonehalf_pseudo (x) result (y)
  real(default), intent(in) :: x
  complex(default) :: y
  if (abs(x) < eps0) then
     y = 0
  else
     y = - 2.0_default * x * faux(x)
  end if
end function fonehalf_pseudo
```

⟨*SM physics: public*⟩+≡
```
public :: fone
```

⟨*SM physics: procedures*⟩+≡
```
elemental function fone (x) result  (y)
  real(default), intent(in) :: x
  complex(default) :: y
  if (abs(x) < eps0) then
     y = 2.0_default
  else
     y = 2.0_default + 3.0_default * x + &
          3.0_default * x * (2.0_default - x) * &
          faux(x)
  end if
end function fone
```

⟨*SM physics: public*⟩+≡
```
public :: gaux
```

⟨*SM physics: procedures*⟩+≡
```
elemental function gaux (x) result (y)
  real(default), intent(in) :: x
  complex(default) :: y
  if (1 <= x) then
     y = sqrt(x - 1) * asin(sqrt(1/x))
  else
     y = sqrt(1 - x) * (log((1 + sqrt(1 - x)) / &
          (1 - sqrt(1 - x))) - &
          cmplx (0.0_default, pi, kind=default)) / 2.0_default
  end if
end function gaux
```

⟨*SM physics: public*⟩+≡
```
public :: tri_i1
```

⟨*SM physics: procedures*⟩+≡
```
elemental function tri_i1 (a,b) result (y)
  real(default), intent(in) :: a,b
  complex(default) :: y
  if (a < eps0 .or. b < eps0) then
```

```
      y = 0
    else
      y = a*b/2.0_default/(a-b) + a**2 * b**2/2.0_default/(a-b)**2 * &
           (faux(a) - faux(b)) + &
           a**2 * b/(a-b)**2 * (gaux(a) - gaux(b))
    end if
  end function tri_i1
```

⟨*SM physics: public*⟩+≡

```
  public :: tri_i2
```

⟨*SM physics: procedures*⟩+≡

```
  elemental function tri_i2 (a,b) result (y)
    real(default), intent(in) :: a,b
    complex(default) :: y
    if (a < eps0 .or. b < eps0) then
      y = 0
    else
      y = - a * b / 2.0_default / (a-b) * (faux(a) - faux(b))
    end if
  end function tri_i2
```

9.4.5 More on α_s

These functions are for the running of the strong coupling constants, α_s.

⟨*SM physics: public*⟩+≡

```
  public :: run_b0
```

⟨*SM physics: procedures*⟩+≡

```
  elemental function run_b0 (nf) result (bnull)
    integer, intent(in) :: nf
    real(default) :: bnull
    bnull = 33.0_default - 2.0_default * nf
  end function run_b0
```

⟨*SM physics: public*⟩+≡

```
  public :: run_b1
```

⟨*SM physics: procedures*⟩+≡

```
  elemental function run_b1 (nf) result (bone)
      integer, intent(in) :: nf
      real(default) :: bone
      bone = 6.0_default * (153.0_default - 19.0_default * nf)/run_b0(nf)**2
  end function run_b1
```

⟨*SM physics: public*⟩+≡

```
  public :: run_aa
```

⟨*SM physics: procedures*⟩+≡
```
elemental function run_aa (nf) result (aaa)
    integer, intent(in) :: nf
    real(default) :: aaa
    aaa = 12.0_default * PI / run_b0(nf)
end function run_aa
```

⟨*SM physics: pubic functions*⟩≡
```
public :: run_bb
```

⟨*SM physics: procedures*⟩+≡
```
elemental function run_bb (nf) result (bbb)
  integer, intent(in) :: nf
  real(default) :: bbb
  bbb = run_b1(nf) / run_aa(nf)
end function run_bb
```

9.4.6 Functions for Catani-Seymour dipoles

For the automated Catani-Seymour dipole subtraction, we need the following functions.

⟨*SM physics: public*⟩+≡
```
public :: ff_dipole
```

⟨*SM physics: procedures*⟩+≡
```
pure subroutine ff_dipole (v_ijk,y_ijk,p_ij,pp_k,p_i,p_j,p_k)
    type(vector4_t), intent(in) :: p_i, p_j, p_k
    type(vector4_t), intent(out) :: p_ij, pp_k
    real(kind=default), intent(out) :: y_ijk
    real(kind=default) :: z_i
    real(kind=default), intent(out) :: v_ijk
    z_i   = (p_i*p_k) / ((p_k*p_j) + (p_k*p_i))
    y_ijk = (p_i*p_j) / ((p_i*p_j) + (p_i*p_k) + (p_j*p_k))
    p_ij  = p_i + p_j - y_ijk/(1.0_default - y_ijk) * p_k
    pp_k  = (1.0/(1.0_default - y_ijk)) * p_k
    !!! We don't multiply by alpha_s right here:
    v_ijk = 8.0_default * PI * CF * &
         (2.0 / (1.0 - z_i*(1.0 - y_ijk)) - (1.0 + z_i))
end subroutine ff_dipole
```

⟨*SM physics: public*⟩+≡
```
public :: fi_dipole
```

⟨*SM physics: procedures*⟩+≡
```
pure subroutine fi_dipole (v_ija,x_ija,p_ij,pp_a,p_i,p_j,p_a)
   type(vector4_t), intent(in) :: p_i, p_j, p_a
   type(vector4_t), intent(out) :: p_ij, pp_a
   real(kind=default), intent(out) :: x_ija
   real(kind=default) :: z_i
   real(kind=default), intent(out) :: v_ija
   z_i   = (p_i*p_a) / ((p_a*p_j) + (p_a*p_i))
   x_ija = ((p_i*p_a) + (p_j*p_a) - (p_i*p_j)) &
```

```
         / ((p_i*p_a) + (p_j*p_a))
      p_ij  = p_i + p_j - (1.0_default - x_ija) * p_a
      pp_a  = x_ija * p_a
      !!! We don't not multiply by alpha_s right here:
      v_ija = 8.0_default * PI * CF * &
           (2.0 / (1.0 - z_i + (1.0 - x_ija)) - (1.0 + z_i)) / x_ija
   end subroutine fi_dipole
```

⟨*SM physics: public*⟩+≡

```
   public :: if_dipole
```

⟨*SM physics: procedures*⟩+≡

```
   pure subroutine if_dipole (v_kja,u_j,p_aj,pp_k,p_k,p_j,p_a)
      type(vector4_t), intent(in) :: p_k, p_j, p_a
      type(vector4_t), intent(out) :: p_aj, pp_k
      real(kind=default), intent(out) :: u_j
      real(kind=default) :: x_kja
      real(kind=default), intent(out) :: v_kja
      u_j   = (p_a*p_j) / ((p_a*p_j) + (p_a*p_k))
      x_kja = ((p_a*p_k) + (p_a*p_j) - (p_j*p_k)) &
           / ((p_a*p_j) + (p_a*p_k))
      p_aj  = x_kja * p_a
      pp_k  = p_k + p_j - (1.0_default - x_kja) * p_a
      v_kja = 8.0_default * PI * CF * &
           (2.0 / (1.0 - x_kja + u_j) - (1.0 + x_kja)) / x_kja
   end subroutine if_dipole
```

This function depends on a variable number of final state particles whose kinematics all get changed by the initial-initial dipole insertion.

⟨*SM physics: public*⟩+≡

```
   public :: ii_dipole
```

⟨*SM physics: procedures*⟩+≡

```
   pure subroutine ii_dipole (v_jab,v_j,p_in,p_out,flag_1or2)
       type(vector4_t), dimension(:), intent(in) :: p_in
       type(vector4_t), dimension(size(p_in)-1), intent(out) :: p_out
       logical, intent(in) :: flag_1or2
       real(kind=default), intent(out) :: v_j
       real(kind=default), intent(out) :: v_jab
       type(vector4_t) :: p_a, p_b, p_j
       type(vector4_t) :: k, kk
       type(vector4_t) :: p_aj
       real(kind=default) :: x_jab
       integer :: i
       !!! flag_1or2 decides whether this a 12 or 21 dipole
       if (flag_1or2) then
          p_a = p_in(1)
          p_b = p_in(2)
       else
          p_b = p_in(1)
          p_a = p_in(2)
       end if
       !!! We assume that the unresolved particle has always the last
       !!! momentum
```

```
    p_j = p_in(size(p_in))
    x_jab = ((p_a*p_b) - (p_a*p_j) - (p_b*p_j)) / (p_a*p_b)
    v_j = (p_a*p_j) / (p_a * p_b)
    p_aj  = x_jab * p_a
    k     = p_a + p_b - p_j
    kk    = p_aj + p_b
    do i = 3, size(p_in)-1
       p_out(i) = p_in(i) - 2.0*((k+kk)*p_in(i))/((k+kk)*(k+kk)) * (k+kk) + &
            (2.0 * (k*p_in(i)) / (k*k)) * kk
    end do
    if (flag_1or2) then
       p_out(1) = p_aj
       p_out(2) = p_b
    else
       p_out(1) = p_b
       p_out(2) = p_aj
    end if
    v_jab = 8.0_default * PI * CF * &
         (2.0 / (1.0 - x_jab) - (1.0 + x_jab)) / x_jab
end subroutine ii_dipole
```

9.4.7 Distributions for integrated dipoles and such

Note that the following formulae are only meaningful for $0 \leq x \leq 1$.

The Dirac delta distribution, modified for Monte-Carlo sampling, centered at $x = 1 - \frac{\epsilon}{2}$:

⟨*SM physics: public*⟩+≡

```
public :: delta
```

⟨*SM physics: procedures*⟩+≡

```
elemental function delta (x,eps) result (z)
   real(kind=default), intent(in) :: x, eps
   real(kind=default) :: z
   if (x > one - eps) then
      z = one / eps
   else
      z = 0
   end if
end function delta
```

The +-distribution, $P_+(x) = \left(\frac{1}{1-x}\right)_+$, for the regularization of soft-collinear singularities. The constant part for the Monte-Carlo sampling is the integral over the splitting function divided by the weight for the WHIZARD numerical integration over the interval.

⟨*SM physics: public*⟩+≡

```
public :: plus_distr
```

⟨*SM physics: procedures*⟩+≡

```
elemental function plus_distr (x,eps) result (plusd)
   real(kind=default), intent(in) :: x, eps
   real(kind=default) :: plusd
   if (x > one - eps) then
      plusd = log(eps) / eps
```

```
      else
         plusd = one / (one - x)
      end if
   end function plus_distr
```

The splitting function in $D = 4$ dimensions, regularized as $+$-distributions if necessary:

$$P^{qq}(x) = P^{\bar{q}\bar{q}}(x) = C_F \cdot \left(\frac{1+x^2}{1-x}\right)_+ \tag{9.13}$$

$$P^{qg}(x) = P^{\bar{q}g}(x) = C_F \cdot \frac{1+(1-x)^2}{x} \tag{9.14}$$

$$P^{gq}(x) = P^{g\bar{q}}(x) = T_R \cdot \left[x^2 + (1-x)^2\right] \tag{9.15}$$

$$\begin{aligned} P^{gg}(x) = {} & 2C_A\left[\left(\frac{1}{1-x}\right)_+ + \frac{1-x}{x} - 1 + x(1-x)\right] \\ & + \delta(1-x)\left(\frac{11}{6}C_A - \frac{2}{3}N_f T_R\right) \end{aligned} \tag{9.16}$$

Since the number of flavors summed over in the gluon splitting function might depend on the physics case under consideration, it is implemented as an input variable.

⟨*SM physics: public*⟩+≡

```
   public :: pqq
```

⟨*SM physics: procedures*⟩+≡

```
   elemental function pqq (x,eps) result (pqqx)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: pqqx
      if (x > (1.0_default - eps)) then
         pqqx = (eps - one) / two + two * log(eps) / eps - &
              three * (eps - one) / eps / two
      else
         pqqx = (one + x**2) / (one - x)
      end if
      pqqx = CF * pqqx
   end function pqq
```

⟨*SM physics: public*⟩+≡

```
   public :: pgq
```

⟨*SM physics: procedures*⟩+≡

```
   elemental function pgq (x) result (pgqx)
      real(kind=default), intent(in) :: x
      real(kind=default) :: pgqx
      pgqx = TR * (x**2 + (one - x)**2)
   end function pgq
```

⟨*SM physics: public*⟩+≡

```
   public :: pqg
```

⟨*SM physics: procedures*⟩+≡

```
elemental function pqg (x) result (pqgx)
   real(kind=default), intent(in) :: x
   real(kind=default) :: pqgx
   pqgx = CF * (one + (one - x)**2) / x
end function pqg
```

⟨*SM physics: public*⟩+≡

```
public :: pgg
```

⟨*SM physics: procedures*⟩+≡

```
elemental function pgg (x, nf, eps) result (pggx)
  real(kind=default), intent(in) :: x, nf, eps
  real(kind=default) :: pggx
  pggx = two * CA * ( plus_distr (x, eps) + (one-x)/x - one + &
                 x*(one-x)) + delta (x, eps)  * gamma_g(nf)
end function pgg
```

For the qq and gg cases, there exist "regularized" versions of the splitting functions:

$$P^{qq}_{\rm reg}(x) = -C_F \cdot (1+x) \tag{9.17}$$

$$P^{gg}_{\rm reg}(x) = 2C_A \left[\frac{1-x}{x} - 1 + x(1-x)\right] \tag{9.18}$$

⟨*SM physics: public*⟩+≡

```
public :: pqq_reg
```

⟨*SM physics: procedures*⟩+≡

```
elemental function pqq_reg (x) result (pqqregx)
   real(kind=default), intent(in) :: x
   real(kind=default) :: pqqregx
   pqqregx = - CF * (one + x)
end function pqq_reg
```

⟨*SM physics: public*⟩+≡

```
public :: pgg_reg
```

⟨*SM physics: procedures*⟩+≡

```
elemental function pgg_reg (x) result (pggregx)
   real(kind=default), intent(in) :: x
   real(kind=default) :: pggregx
   pggregx = two * CA * ((one - x)/x - one + x*(one - x))
end function pgg_reg
```

Here, we collect the expressions needed for integrated Catani-Seymour dipoles, and the so-called flavor kernels. We always distinguish between the "ordinary" Catani-Seymour version, and the one including a phase-space slicing parameter, α.

The standard flavor kernels $\overline{K}^{ab}$ are:

$$\overline{K}^{qg}(x) = \overline{K}^{\bar{q}g}(x) = P^{qg}(x)\log((1-x)/x) + CF \times x \tag{9.19}$$

$$\overline{K}^{gq}(x) = \overline{K}^{g\bar{q}}(x) = P^{gq}(x)\log((1-x)/x) + TR \times 2x(1-x) \tag{9.20}$$

$$\begin{aligned}\overline{K}^{qq} &= C_F\left[\left(\frac{2}{1-x}\log\frac{1-x}{x}\right)_+ - (1+x)\log((1-x)/x) + (1-x)\right] \\ &\quad - (5-\pi^2)\cdot C_F\cdot\delta(1-x) \end{aligned}\tag{9.21}$$

$$\begin{aligned}\overline{K}^{gg} &= 2C_A\left[\left(\frac{1}{1-x}\log\frac{1-x}{x}\right)_+ + \left(\frac{1-x}{x} - 1 + x(1-x)\right)\log((1-x)/x)\right] \\ &\quad - \delta(1-x)\left[\left(\frac{50}{9} - \pi^2\right)C_A - \frac{16}{9}T_R N_f\right] \end{aligned}\tag{9.22}$$

⟨*SM physics: public*⟩+≡
```
public :: kbarqg
```

⟨*SM physics: procedures*⟩+≡
```
function kbarqg (x) result (kbarqgx)
  real(kind=default), intent(in) :: x
  real(kind=default) :: kbarqgx
  kbarqgx = pqg(x) * log((one-x)/x) + CF * x
end function kbarqg
```

⟨*SM physics: public*⟩+≡
```
public :: kbargq
```

⟨*SM physics: procedures*⟩+≡
```
function kbargq (x) result (kbargqx)
  real(kind=default), intent(in) :: x
  real(kind=default) :: kbargqx
  kbargqx = pgq(x) * log((one-x)/x) + two * TR * x * (one - x)
end function kbargq
```

⟨*SM physics: public*⟩+≡
```
public :: kbarqq
```

⟨*SM physics: procedures*⟩+≡
```
function kbarqq (x,eps) result (kbarqqx)
  real(kind=default), intent(in) :: x, eps
  real(kind=default) :: kbarqqx
  kbarqqx = CF*(log_plus_distr(x,eps) - (one+x) * log((one-x)/x) + (one - &
       x) - (five - pi**2) * delta(x,eps))
end function kbarqq
```

⟨*SM physics: public*⟩+≡
```
public :: kbargg
```

⟨*SM physics: procedures*⟩+≡
```
function kbargg (x,eps,nf) result (kbarggx)
  real(kind=default), intent(in) :: x, eps, nf
  real(kind=default) :: kbarggx
  kbarggx = CA * (log_plus_distr(x,eps) + two * ((one-x)/x - one + &
```

```
                                x*(one-x) * log((1-x)/x))) - delta(x,eps) * &
                                ((50.0_default/9.0_default - pi**2) * CA - &
                                16.0_default/9.0_default * TR * nf)
    end function kbargg
```

The $\tilde{K}$ are used when two identified hadrons participate:

$$\tilde{K}^{ab}(x) = P^{ab}_{\rm reg}(x)\cdot\log(1-x)+\delta^{ab}\mathbf{T}^2_a\left[\left(\frac{2}{1-x}\log(1-x)\right)_+ - \frac{\pi^2}{3}\delta(1-x)\right] \quad (9.23)$$

⟨*SM physics: public*⟩+≡
```
    public :: ktildeqq
```

⟨*SM physics: procedures*⟩+≡
```
    function ktildeqq (x,eps) result (ktildeqqx)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: ktildeqqx
      ktildeqqx = pqq_reg (x) * log(one-x) + CF * ( - log2_plus_distr (x,eps) &
                          - pi**2/three * delta(x,eps))
    end function ktildeqq
```

⟨*SM physics: public*⟩+≡
```
    public :: ktildeqg
```

⟨*SM physics: procedures*⟩+≡
```
    function ktildeqg (x,eps) result (ktildeqgx)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: ktildeqgx
      ktildeqgx = pqg (x) * log(one-x)
    end function ktildeqg
```

⟨*SM physics: public*⟩+≡
```
    public :: ktildegq
```

⟨*SM physics: procedures*⟩+≡
```
    function ktildegq (x,eps) result (ktildegqx)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: ktildegqx
      ktildegqx = pgq (x) * log(one-x)
    end function ktildegq
```

⟨*SM physics: public*⟩+≡
```
    public :: ktildegg
```

⟨*SM physics: procedures*⟩+≡
```
    function ktildegg (x,eps) result (ktildeggx)
      real(kind=default), intent(in) :: x, eps
      real(kind=default) :: ktildeggx
      ktildeggx = pgg_reg (x) * log(one-x) + CA * ( - &
         log2_plus_distr (x,eps) - pi**2/three * delta(x,eps))
    end function ktildegg
```

The insertion operator might not be necessary for a GOLEM interface but is demanded by the Les Houches NLO accord. It is a three-dimensional array, where the index always gives the inverse power of the DREG expansion parameter, ϵ.

⟨*SM physics: public*⟩+≡
```
public :: insert_q
```
⟨*SM physics: procedures*⟩+≡
```
pure function insert_q ()
  real(kind=default), dimension(0:2) :: insert_q
  insert_q(0) = gamma_q + k_q - pi**2/three * CF
  insert_q(1) = gamma_q
  insert_q(2) = CF
end function insert_q
```

⟨*SM physics: public*⟩+≡
```
public :: insert_g
```
⟨*SM physics: procedures*⟩+≡
```
pure function insert_g (nf)
  real(kind=default), intent(in) :: nf
  real(kind=default), dimension(0:2) :: insert_g
  insert_g(0) = gamma_g (nf) + k_g (nf) - pi**2/three * CA
  insert_g(1) = gamma_g (nf)
  insert_g(2) = CA
end function insert_g
```

For better convergence, one can exclude regions of phase space with a slicing parameter from the dipole subtraction procedure. First of all, the K functions get modified:

$$K_i(\alpha) = K_i - \mathbf{T}_i^2 \log^2 \alpha + \gamma_i(\alpha - 1 - \log \alpha) \tag{9.24}$$

⟨*SM physics: public*⟩+≡
```
public :: k_q_al, k_g_al
```
⟨*SM physics: procedures*⟩+≡
```
pure function k_q_al (alpha)
  real(kind=default), intent(in) :: alpha
  real(kind=default) :: k_q_al
  k_q_al = k_q - CF * (log(alpha))**2 + gamma_q * &
                    (alpha - one - log(alpha))
end function k_q_al

pure function k_g_al (alpha, nf)
  real(kind=default), intent(in) :: alpha, nf
  real(kind=default) :: k_g_al
  k_g_al = k_g (nf) - CA * (log(alpha))**2 + gamma_g (nf) * &
                   (alpha - one - log(alpha))
end function k_g_al
```

The +-distribution, but with a phase-space slicing parameter, α, $P_{1-\alpha}(x) = \left(\frac{1}{1-x}\right)_{1-x}$. Since we need the fatal error message here, this function cannot be elemental.

⟨*SM physics: public*⟩+≡
```
public :: plus_distr_al
```

⟨*SM physics: procedures*⟩+≡

```
function plus_distr_al (x,alpha,eps) result (plusd_al)
   real(kind=default), intent(in) :: x,  eps, alpha
   real(kind=default) :: plusd_al
   if ((one - alpha) >= (one - eps)) then
      plusd_al = zero
      call msg_fatal ('sm_physics, plus_distr_al: alpha and epsilon chosen wrongly')
   elseif (x < (1.0_default - alpha)) then
      plusd_al = 0
   else if (x > (1.0_default - eps)) then
      plusd_al = log(eps/alpha)/eps
   else
      plusd_al = one/(one-x)
   end if
 end function plus_distr_al
```

Introducing phase-space slicing parameters, these standard flavor kernels $\overline{K}^{ab}$ become:

$$\overline{K}^{qg}_{\alpha}(x) = \overline{K}^{\bar{q}g}_{\alpha}(x) = P^{qg}(x)\log(\alpha(1-x)/x) + C_F \times x \tag{9.25}$$

$$\overline{K}^{gq}_{\alpha}(x) = \overline{K}^{g\bar{q}}_{\alpha}(x) = P^{gq}(x)\log(\alpha(1-x)/x) + T_R \times 2x(1-x) \tag{9.26}$$

$$\begin{aligned}\overline{K}^{qq}_{\alpha} &= C_F(1-x) + P^{qq}_{\mathrm{reg}}(x)\log\frac{\alpha(1-x)}{x} \\ &\quad + C_F\delta(1-x)\log^2\alpha + C_F\left(\frac{2}{1-x}\log\frac{1-x}{x}\right)_+ \\ &\quad - \left(\gamma_q + K_q(\alpha) - \frac{5}{6}\pi^2 C_F\right)\cdot\delta(1-x)\; C_F\Big[+\frac{2}{1-x}\log\left(\frac{\alpha(2-x)}{1+\alpha-x}\right) - \theta(1-\alpha-x)\cdot\Big(\frac{2}{1-}\end{aligned} \tag{9.27}$$

$$\begin{aligned}\overline{K}^{gg}_{\alpha} &= P^{gg}_{\mathrm{reg}}(x)\log\frac{\alpha(1-x)}{x} + C_A\delta(1-x)\log^2\alpha \\ &\quad + C_A\left(\frac{2}{1-x}\log\frac{1-x}{x}\right)_+ - \left(\gamma_g + K_g(\alpha) - \frac{5}{6}\pi^2 C_A\right)\cdot\delta(1-x)\; C_A\Big[+\frac{2}{1-x}\log\Big(\frac{\alpha(2-}{1+\alpha}\end{aligned} \tag{9.28}$$

⟨*SM physics: public*⟩+≡

```
public :: kbarqg_al
```

⟨*SM physics: procedures*⟩+≡

```
function kbarqg_al (x,alpha,eps) result (kbarqgx)
  real(kind=default), intent(in) :: x, alpha, eps
  real(kind=default) :: kbarqgx
  kbarqgx = pqg (x) * log(alpha*(one-x)/x) + CF * x
end function kbarqg_al
```

⟨*SM physics: public*⟩+≡

```
public :: kbargq_al
```

⟨*SM physics: procedures*⟩+≡

```
function kbargq_al (x,alpha,eps) result (kbargqx)
  real(kind=default), intent(in) :: x, alpha, eps
  real(kind=default) :: kbargqx
  kbargqx = pgq (x) * log(alpha*(one-x)/x) + two * TR * x * (one-x)
end function kbargq_al
```

⟨*SM physics: public*⟩+≡
```
public :: kbarqq_al
```
⟨*SM physics: procedures*⟩+≡
```
function kbarqq_al (x,alpha,eps) result (kbarqqx)
   real(kind=default), intent(in) :: x, alpha, eps
   real(kind=default) :: kbarqqx
   kbarqqx = CF * (one - x) + pqq_reg(x) * log(alpha*(one-x)/x) &
            + CF * log_plus_distr(x,eps) &
           - (gamma_q + k_q_al(alpha) - CF * &
            five/6.0_default  * pi**2 - CF * (log(alpha))**2) * &
            delta(x,eps) + &
            CF * two/(one -x)*log(alpha*(two-x)/(one+alpha-x))
   if (x < (one-alpha)) then
      kbarqqx = kbarqqx - CF * two/(one-x) * log((two-x)/(one-x))
   end if
end function kbarqq_al
```

⟨*SM physics: public*⟩+≡
```
public :: kbargg_al
```
⟨*SM physics: procedures*⟩+≡
```
function kbargg_al (x,alpha,eps,nf) result (kbarggx)
   real(kind=default), intent(in) :: x, alpha, eps, nf
   real(kind=default) :: kbarggx
   kbarggx = pgg_reg(x) * log(alpha*(one-x)/x) &
            + CA * log_plus_distr(x,eps) &
           - (gamma_g(nf) + k_g_al(alpha,nf) - CA * &
            five/6.0_default  * pi**2 - CA * (log(alpha))**2) * &
            delta(x,eps) + &
            CA * two/(one -x)*log(alpha*(two-x)/(one+alpha-x))
   if (x < (one-alpha)) then
      kbarggx = kbarggx - CA * two/(one-x) * log((two-x)/(one-x))
   end if
end function kbargg_al
```

The $\tilde{K}$ flavor kernels in the presence of a phase-space slicing parameter, are:

$$\tilde{K}^{ab}(x,\alpha) = P^{qq,\mathrm{reg}}(x)\log\frac{1-x}{\alpha} + \ldots\ldots\ldots \tag{9.29}$$

⟨*SM physics: public*⟩+≡
```
public :: ktildeqq_al
```
⟨*SM physics: procedures*⟩+≡
```
function ktildeqq_al (x,alpha,eps) result (ktildeqqx)
  real(kind=default), intent(in) :: x, eps, alpha
  real(kind=default) :: ktildeqqx
  ktildeqqx = pqq_reg(x) * log((one-x)/alpha) + CF*( &
       - log2_plus_distr_al(x,alpha,eps) - Pi**2/three * delta(x,eps) &
       + (one+x**2)/(one-x) * log(min(one,(alpha/(one-x)))) &
       + two/(one-x) * log((one+alpha-x)/alpha))
  if (x > (one-alpha)) then
     ktildeqqx = ktildeqqx - CF*two/(one-x)*log(two-x)
  end if
end function ktildeqq_al
```

This is a logarithmic +-distribution, $\left(\frac{\log((1-x)/x)}{1-x}\right)_+$. For the sampling, we need the integral over this function over the incomplete sampling interval $[0, 1-\epsilon]$, which is $\log^2(x) + 2Li_2(x) - \frac{\pi^2}{3}$. As this function is negative definite for $\epsilon > 0.1816$, we take a hard upper limit for that sampling parameter, irrespective of the fact what the user chooses.

⟨*SM physics: public*⟩+≡
```
public :: log_plus_distr
```

⟨*SM physics: procedures*⟩+≡
```
function log_plus_distr (x,eps) result (lpd)
   real(kind=default), intent(in) :: x, eps
   real(kind=default) :: lpd, eps2
   eps2 = min (eps, 0.1816_default)
   if (x > (1.0_default - eps2)) then
      lpd = ((log(eps2))**2 + two*Li2(eps2) - pi**2/three)/eps2
   else
      lpd = two*log((one-x)/x)/(one-x)
   end if
end function log_plus_distr
```

Logarithmic +-distribution, $2\left(\frac{\log(1/(1-x))}{1-x}\right)_+$.

⟨*SM physics: public*⟩+≡
```
public :: log2_plus_distr
```

⟨*SM physics: procedures*⟩+≡
```
function log2_plus_distr (x,eps) result (lpd)
    real(kind=default), intent(in) :: x, eps
    real(kind=default) :: lpd
    if (x > (1.0_default - eps)) then
       lpd = - (log(eps))**2/eps
    else
       lpd = two*log(one/(one-x))/(one-x)
    end if
end function log2_plus_distr
```

Logarithmic +-distribution with phase-space slicing parameter, $2\left(\frac{\log(1/(1-x))}{1-x}\right)_{1-\alpha}$.

⟨*SM physics: public*⟩+≡
```
public :: log2_plus_distr_al
```

⟨*SM physics: procedures*⟩+≡
```
function log2_plus_distr_al (x,alpha,eps) result (lpd_al)
  real(kind=default), intent(in) :: x, eps, alpha
  real(kind=default) :: lpd_al
  if ((one - alpha) >= (one - eps)) then
     lpd_al = zero
     call msg_fatal ('alpha and epsilon chosen wrongly')
  elseif (x < (one - alpha)) then
     lpd_al = 0
  elseif (x > (1.0_default - eps)) then
     lpd_al = - ((log(eps))**2 - (log(alpha))**2)/eps
  else
     lpd_al = two*log(one/(one-x))/(one-x)
```

```
    end if
  end function log2_plus_distr_al
```

9.4.8 Splitting Functions

Analogue to the regularized distributions of the last subsection, we give here the unregularized splitting functions, relevant for the parton shower algorithm. We can use this unregularized version since there will be a cut-off ϵ that ensures that $\{z, 1-z\} > \epsilon(t)$. This cut-off seperates resolvable from unresolvable emissions.

p_xxx are the kernels that are summed over helicity:

⟨*SM physics: public*⟩+≡

```
  public :: p_qqg
  public :: p_gqq
  public :: p_ggg
```

$q \to qg$

⟨*SM physics: procedures*⟩+≡

```
  elemental function p_qqg (z) result (P)
    real(default), intent(in) :: z
    real(default) :: P
    P = CF * (one + z**2) / (one - z)
  end function p_qqg
```

$g \to q\bar{q}$

⟨*SM physics: procedures*⟩+≡

```
  elemental function p_gqq (z) result (P)
    real(default), intent(in) :: z
    real(default) :: P
    P = TR * (z**2 + (one - z)**2)
  end function p_gqq
```

$g \to gg$

⟨*SM physics: procedures*⟩+≡

```
  elemental function p_ggg (z) result (P)
    real(default), intent(in) :: z
    real(default) :: P
    P = NC * ((one - z) / z + z / (one - z) + z * (one - z))
  end function p_ggg
```

Analytically integrated splitting kernels:

⟨*SM physics: public*⟩+≡

```
  public :: integral_over_p_qqg
  public :: integral_over_p_gqq
  public :: integral_over_p_ggg
```

⟨*SM physics: procedures*⟩+≡

```
  pure function integral_over_p_qqg (zmin, zmax) result (integral)
    real(default), intent(in) :: zmin, zmax
    real(default) :: integral
    integral = (two / three) * (- zmax**2 + zmin**2 - &
         two * (zmax - zmin) + four * log((one - zmin) / (one - zmax)))
  end function integral_over_p_qqg
```

```
pure function integral_over_p_gqq (zmin, zmax) result (integral)
  real(default), intent(in) :: zmin, zmax
  real(default) :: integral
  integral = 0.5_default * ((two / three) * &
       (zmax**3 - zmin**3) - (zmax**2 - zmin**2) + (zmax - zmin))
end function integral_over_p_gqq

pure function integral_over_p_ggg (zmin, zmax) result (integral)
  real(default), intent(in) :: zmin, zmax
  real(default) :: integral
  integral = three * ((log(zmax) - two * zmax - &
       log(one - zmax) + zmax**2 / two - zmax**3 / three) - &
       (log(zmin) - zmin - zmin - log(one - zmin) + zmin**2 &
       / two - zmin**3 / three) )
end function integral_over_p_ggg
```

We can also use (massless) helicity dependent splitting functions:

⟨*SM physics: public*⟩+≡

```
public :: p_qqg_pol
```

$q_a \to q_b g_c$, the helicity of the quark is not changed by gluon emission and the gluon is preferably polarized in the branching plane ($l_c = 1$):

⟨*SM physics: procedures*⟩+≡

```
elemental function p_qqg_pol (z, l_a, l_b, l_c) result (P)
  real(default), intent(in) :: z
  integer, intent(in) :: l_a, l_b, l_c
  real(default) :: P
  if (l_a /= l_b) then
     P = zero
     return
  end if
  if (l_c == -1) then
     P = one - z
  else
     P = (one + z)**2 / (one - z)
  end if
  P = P * CF
end function p_qqg_pol
```

9.4.9 Top width

In order to produce sensible results, the widths have to be recomputed for each parameter and order. We start with the LO-expression for the top width given by the decay $t \to W^+, b$, cf. doi:10.1016/0550-3213(91)90530-B:
The analytic formula given there is

$$\Gamma = \frac{G_F m_t^2}{16\sqrt{2}\pi}\left[\mathcal{F}_0(\varepsilon, \xi^{-1/2}) - \frac{2\alpha_s}{3\pi}\mathcal{F}_1(\varepsilon, \xi^{-1/2})\right],$$

with

$$\mathcal{F}_0 = \frac{\sqrt{\lambda}}{2} f_0,$$
$$f_0 = 4\left[(1-\varepsilon^2)^2 + w^2(1+\varepsilon^2) - 2w^4\right],$$
$$\lambda = 1 + w^4 + \varepsilon^4 - 2(w^2 + \varepsilon^2 + w^2\varepsilon^2).$$

Defining

$$u_q = \frac{1+\varepsilon^2 - w^2 - \lambda^{1/2}}{1+\varepsilon^2 - w^2 + \lambda^{1/2}}$$

and

$$u_w = \frac{1-\varepsilon^2 + w^2 - \lambda^{1/2}}{1-\varepsilon^2 + w^2 + \lambda^{1/2}}$$

the factor $\mathcal{F}_1$ can be expressed as

$$\begin{aligned}
\mathcal{F}_1 = \frac{1}{2} f_0 (1+\varepsilon^2 - w^2) \Big[& \pi^2 + 2Li_2(u_w) - 2Li_2(1-u_w) - 4Li_2(u_q) \\
& - 4Li_2(u_q u_w) + \log\left(\frac{1-u_q}{w^2}\right)\log(1-u_q) - \log^2(1-u_q u_w) \\
& + \frac{1}{4}\log^2\left(\frac{w^2}{u_w}\right) - \log(u_w)\log\left[\frac{(1-u_q u_w)^2}{1-u_q}\right] - 2\log(u_q)\log\left[(1-u_q)(1-u_q u_w)\right]\Big] \\
& - \sqrt{\lambda} f_0 (2\log(w) + 3\log(\varepsilon) - 2\log\lambda) \\
& + 4(1-\varepsilon^2)\left[(1-\varepsilon^2)^2 + w^2(1+\varepsilon^2) - 4w^4\right]\log(u_w) \\
& \left[(3-\varepsilon^2 + 11\varepsilon^4 - \varepsilon^6) + w^2(6 - 12\varepsilon^2 + 2\varepsilon^4) - w^4(21 + 5\varepsilon^2) + 12w^6\right]\log(u_q) \\
& 6\sqrt{\lambda}(1-\varepsilon^2)(1+\varepsilon^2 - w^2)\log(\varepsilon) + \sqrt{\lambda}\left[-5 + 22\varepsilon^2 - 5\varepsilon^4 - 9w^2(1+\varepsilon^2) + 6w^4\right].
\end{aligned}$$

⟨*SM physics: public*⟩+≡

```
public :: top_width_sm_lo
```

⟨*SM physics: procedures*⟩+≡

```
elemental function top_width_sm_lo (alpha, sinthw, vtb, mtop, mw, mb) &
       result (gamma)
  real(default) :: gamma
  real(default), intent(in) :: alpha, sinthw, vtb, mtop, mw, mb
  real(default) :: kappa
  kappa = sqrt ((mtop**2 - (mw + mb)**2) * (mtop**2 - (mw - mb)**2))
  gamma = alpha / four * mtop / (two * sinthw**2) * &
       vtb**2 * kappa / mtop**2 * &
       ((mtop**2 + mb**2) / (two * mtop**2) + &
        (mtop**2 - mb**2)**2 / (two * mtop**2 * mw**2) - &
         mw**2 / mtop**2)
end function top_width_sm_lo
```

⟨*SM physics: public*⟩+≡

```
public :: g_mu_from_alpha
```

⟨*SM physics: procedures*⟩+≡

```
elemental function g_mu_from_alpha (alpha, mw, sinthw) result (g_mu)
  real(default) :: g_mu
  real(default), intent(in) :: alpha, mw, sinthw
  g_mu = pi * alpha / sqrt(two) / mw**2 / sinthw**2
end function g_mu_from_alpha
```

⟨*SM physics: public*⟩+≡
```
public :: alpha_from_g_mu
```

⟨*SM physics: procedures*⟩+≡
```
elemental function alpha_from_g_mu (g_mu, mw, sinthw) result (alpha)
  real(default) :: alpha
  real(default), intent(in) :: g_mu, mw, sinthw
  alpha = g_mu * sqrt(two) / pi * mw**2 * sinthw**2
end function alpha_from_g_mu
```

Cf. (3.3)-(3.7) in 1207.5018.

⟨*SM physics: public*⟩+≡
```
public :: top_width_sm_qcd_nlo_massless_b
```

⟨*SM physics: procedures*⟩+≡
```
elemental function top_width_sm_qcd_nlo_massless_b &
       (alpha, sinthw, mtop, mw, alphas) result (gamma)
  real(default) :: gamma
  real(default), intent(in) :: alpha, sinthw, mtop, mw, alphas
  real(default) :: prefac, g_mu, w2
  g_mu = g_mu_from_alpha (alpha, mw, sinthw)
  prefac = g_mu * mtop**3 / (16 * sqrt(two) * pi)
  w2 = mw**2 / mtop**2
  gamma = prefac * (f0 (w2) - (two * alphas) / (3 * Pi) * f1 (w2))
end function top_width_sm_qcd_nlo_massless_b
```

⟨*SM physics: public*⟩+≡
```
public :: f0
```

⟨*SM physics: procedures*⟩+≡
```
elemental function f0 (w2) result (f)
  real(default) :: f
  real(default), intent(in) :: w2
  f = two * (one - w2)**2 * (1 + 2 * w2)
end function f0
```

⟨*SM physics: public*⟩+≡
```
public :: f1
```

⟨*SM physics: procedures*⟩+≡
```
elemental function f1 (w2) result (f)
  real(default) :: f
  real(default), intent(in) :: w2
  f = f0 (w2) * (pi**2 + two * Li2 (w2) - two * Li2 (one - w2)) &
       + four * w2 * (one - w2 - two * w2**2) * log (w2) &
       + two * (one - w2)**2 * (five + four * w2) * log (one - w2) &
       - (one - w2) * (five + 9 * w2 - 6 * w2**2)
end function f1
```

Basically, the same as above but with m_b dependence, cf. Jezabek / Kuehn 1989.

⟨*SM physics: public*⟩+≡
```
public :: top_width_sm_qcd_nlo_jk
```

⟨*SM physics: procedures*⟩+≡

```
elemental function top_width_sm_qcd_nlo_jk &
       (alpha, sinthw, mtop, mw, mb, alphas) result (gamma)
  real(default) :: gamma
  real(default), intent(in) :: alpha, sinthw, mtop, mw, mb, alphas
  real(default) :: prefac, g_mu, eps2, i_xi
  g_mu = g_mu_from_alpha (alpha, mw, sinthw)
  prefac = g_mu * mtop**3 / (16 * sqrt(two) * pi)
  eps2 = (mb / mtop)**2
  i_xi = (mw / mtop)**2
  gamma = prefac * (ff0 (eps2, i_xi) - &
       (two * alphas) / (3 * Pi) * ff1 (eps2, i_xi))
end function top_width_sm_qcd_nlo_jk
```

Same as above, $m_b > 0$, with the slightly different implementation (2.6) of arXiv:1204.1513v1 by Campbell and Ellis.

⟨*SM physics: public*⟩+≡

```
public :: top_width_sm_qcd_nlo_ce
```

⟨*SM physics: procedures*⟩+≡

```
elemental function top_width_sm_qcd_nlo_ce (alpha, sinthw, mtop, mw, mb, alpha_s) result (gamma)
  real(default) :: gamma
  real(default), intent(in) :: alpha, sinthw, mtop, mw, mb, alpha_s
  real(default) :: pm, pp, p0, p3
  real(default) :: yw, yp
  real(default) :: W0, Wp, Wm, w2
  real(default) :: beta2
  real(default) :: f
  real(default) :: g_mu, gamma0
  beta2 = (mb / mtop)**2
  w2 = (mw / mtop)**2
  p0 = (one - w2 + beta2) / two
  p3 = sqrt (lambda (one, w2, beta2)) / two
  pp = p0 + p3
  pm = p0 - p3
  W0 = (one + w2 - beta2) / two
  Wp = W0 + p3
  Wm = W0 - p3
  yp = log (pp / pm) / two
  yw = log (Wp / Wm) / two
  f = (one - beta2)**2 + w2 * (one + beta2) - two * w2**2
  g_mu = g_mu_from_alpha (alpha, mw, sinthw)
  gamma0 = g_mu * mtop**3 / (8 * pi * sqrt(two))
  gamma = gamma0 * alpha_s / twopi * CF * &
        (8 * f * p0 * (Li2(one - pm) - Li2(one - pp) - two * Li2(one - pm / pp) &
       + yp * log((four * p3**2) / (pp**2 * Wp)) + yw * log (pp)) &
       + four * (one - beta2) * ((one - beta2)**2 + w2 * (one + beta2) - four * w2**2) * yw &
       + (3 - beta2 + 11 * beta2**2 - beta2**3 + w2 * (6 - 12 * beta2 + two * beta2**2) &
            - w2**2 * (21 + 5 * beta2) + 12 * w2**3) * yp &
       + 8 * f * p3 * log (sqrt(w2) / (four * p3**2)) &
       + 6 * (one - four * beta2 + 3 * beta2**2 + w2 * (3 + beta2) - four * w2**2) * p3 * log(sq
       + (5 - 22 * beta2 + 5 * beta2**2 + 9 * w2 * (one + beta2) - 6 * w2**2) * p3)
end function top_width_sm_qcd_nlo_ce
```

⟨*SM physics: public*⟩+≡
```
public :: ff0
```
⟨*SM physics: procedures*⟩+≡
```
elemental function ff0 (eps2, w2) result (f)
  real(default) :: f
  real(default), intent(in) :: eps2, w2
  f = one / two * sqrt(ff_lambda (eps2, w2)) * ff_f0 (eps2, w2)
end function ff0
```

⟨*SM physics: public*⟩+≡
```
public :: ff_f0
```
⟨*SM physics: procedures*⟩+≡
```
elemental function ff_f0 (eps2, w2) result (f)
  real(default) :: f
  real(default), intent(in) :: eps2, w2
  f = four * ((1 - eps2)**2 + w2 * (1 + eps2) - 2 * w2**2)
end function ff_f0
```

⟨*SM physics: public*⟩+≡
```
public :: ff_lambda
```
⟨*SM physics: procedures*⟩+≡
```
elemental function ff_lambda (eps2, w2) result (l)
  real(default) :: l
  real(default), intent(in) :: eps2, w2
  l = one + w2**2 + eps2**2 - two * (w2 + eps2 + w2 * eps2)
end function ff_lambda
```

⟨*SM physics: public*⟩+≡
```
public :: ff1
```
⟨*SM physics: procedures*⟩+≡
```
elemental function ff1 (eps2, w2) result (f)
  real(default) :: f
  real(default), intent(in) :: eps2, w2
  real(default) :: uq, uw, sq_lam, fff
  sq_lam = sqrt (ff_lambda (eps2, w2))
  fff = ff_f0 (eps2, w2)
  uw = (one - eps2 + w2 - sq_lam) / &
       (one - eps2 + w2 + sq_lam)
  uq = (one + eps2 - w2 - sq_lam) / &
       (one + eps2 - w2 + sq_lam)
  f = one / two * fff * (one + eps2 - w2) * &
       (pi**2 + two * Li2 (uw) - two * Li2 (one - uw) - four * Li2 (uq) &
        - four * Li2 (uq * uw) + log ((one - uq) / w2) * log (one - uq) &
        - log (one - uq * uw)**2 + one / four * log (w2 / uw)**2 &
        - log (uw) * log ((one - uq * uw)**2 / (one - uq)) &
        - two * log (uq) * log ((one - uq) * (one - uq * uw))) &
        - sq_lam * fff * (two * log (sqrt (w2)) &
        + three * log (sqrt (eps2)) - two * log (sq_lam**2)) &
        + four * (one - eps2) * ((one - eps2)**2 + w2 * (one + eps2) &
        - four * w2**2) * log (uw) &
```

```
          + (three - eps2 + 11 * eps2**2 - eps2**3 + w2 * &
            (6 - 12 * eps2 + 2 * eps2**2) - w2**2 * (21 + five * eps2) &
          + 12 * w2**3) * log (uq) &
          + 6 * sq_lam * (one - eps2) * &
            (one + eps2 - w2) * log (sqrt (eps2)) &
          + sq_lam * (- five + 22 * eps2 - five * eps2**2 - 9 * w2 * &
            (one + eps2) + 6 * w2**2)
    end function ff1
```

9.4.10 Unit tests

Test module, followed by the corresponding implementation module.

⟨sm_physics_ut.f90⟩≡

```
  ⟨File header⟩

  module sm_physics_ut
    use unit_tests
    use sm_physics_uti

  ⟨Standard module head⟩

  ⟨SM physics: public test⟩

  contains

  ⟨SM physics: test driver⟩

  end module sm_physics_ut
```

⟨sm_physics_uti.f90⟩≡

```
  ⟨File header⟩

  module sm_physics_uti

  ⟨Use kinds⟩
    use numeric_utils
    use format_defs, only: FMT_15
    use constants

    use sm_physics

  ⟨Standard module head⟩

  ⟨SM physics: test declarations⟩

  contains

  ⟨SM physics: tests⟩

  end module sm_physics_uti
```

API: driver for the unit tests below.

⟨*SM physics: public test*⟩≡

```
    public :: sm_physics_test
```

⟨*SM physics: test driver*⟩≡
```
subroutine sm_physics_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨SM physics: execute tests⟩
end subroutine sm_physics_test
```

Splitting functions

⟨*SM physics: execute tests*⟩≡
```
call test (sm_physics_1, "sm_physics_1", &
     "Splitting functions", &
     u, results)
```

⟨*SM physics: test declarations*⟩≡
```
public :: sm_physics_1
```

⟨*SM physics: tests*⟩≡
```
subroutine sm_physics_1 (u)
  integer, intent(in) :: u
  real(default) :: z = 0.75_default

  write (u, "(A)")  "* Test output: sm_physics_1"
  write (u, "(A)")  "*   Purpose: check analytic properties"
  write (u, "(A)")

  write (u, "(A)")  "* Splitting functions:"
  write (u, "(A)")

  call assert (u, vanishes (p_qqg_pol (z, +1, -1, +1)), "+-+")
  call assert (u, vanishes (p_qqg_pol (z, +1, -1, -1)), "+--")
  call assert (u, vanishes (p_qqg_pol (z, -1, +1, +1)), "-++")
  call assert (u, vanishes (p_qqg_pol (z, -1, +1, -1)), "-+-")

  !call assert (u, nearly_equal ( &
       !p_qqg_pol (z, +1, +1, -1) + p_qqg_pol (z, +1, +1, +1), &
       !p_qqg (z)), "pol sum")

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sm_physics_1"

end subroutine sm_physics_1
```

Top width

⟨*SM physics: execute tests*⟩+≡
```
call test(sm_physics_2, "sm_physics_2", &
          "Top width", u, results)
```

⟨*SM physics: test declarations*⟩+≡
```
public :: sm_physics_2
```

⟨*SM physics: tests*⟩+≡

```
subroutine sm_physics_2 (u)
  integer, intent(in) :: u
  real(default) :: mtop, mw, mz, mb, g_mu, sinthw, alpha, vtb, gamma0
  real(default) :: w2, alphas, alphas_mz, gamma1
  write (u, "(A)")  "* Test output: sm_physics_2"
  write (u, "(A)")  "*   Purpose: Check different top width computations"
  write (u, "(A)")

  write (u, "(A)")  "*   Values from [[1207.5018]] (massless b)"
  mtop = 172.0
  mw = 80.399
  mz = 91.1876
  mb = zero
  mb = 0.00001
  g_mu = 1.16637E-5
  sinthw = sqrt(one - mw**2 / mz**2)
  alpha = alpha_from_g_mu (g_mu, mw, sinthw)
  vtb = one
  w2 = mw**2 / mtop**2

  write (u, "(A)")  "*   Check Li2 implementation"
  call assert_equal (u, Li2(w2), 0.2317566263959552_default, &
       "Li2(w2)", rel_smallness=1.0E-6_default)
  call assert_equal (u, Li2(one - w2), 1.038200378935867_default, &
       "Li2(one - w2)", rel_smallness=1.0E-6_default)

  write (u, "(A)")  "*   Check LO Width"
  gamma0 = top_width_sm_lo (alpha, sinthw, vtb, mtop, mw, mb)
  call assert_equal (u, gamma0, 1.4655_default, &
       "top_width_sm_lo", rel_smallness=1.0E-5_default)
  alphas = zero
  gamma0 = top_width_sm_qcd_nlo_massless_b (alpha, sinthw, mtop, mw, alphas)
  call assert_equal (u, gamma0, 1.4655_default, &
       "top_width_sm_qcd_nlo_massless_b", rel_smallness=1.0E-5_default)
  gamma0 = top_width_sm_qcd_nlo_jk (alpha, sinthw, mtop, mw, mb, alphas)
  call assert_equal (u, gamma0, 1.4655_default, &
       "top_width_sm_qcd_nlo", rel_smallness=1.0E-5_default)

  write (u, "(A)")  "*   Check NLO Width"
  alphas_mz = 0.1202      ! MSTW2008 NLO fit
  alphas = running_as (mtop, alphas_mz, mz, 1, 5.0_default)
  gamma1 = top_width_sm_qcd_nlo_massless_b (alpha, sinthw, mtop, mw, alphas)
  call assert_equal (u, gamma1, 1.3376_default, rel_smallness=1.0E-4_default)
  gamma1 = top_width_sm_qcd_nlo_jk (alpha, sinthw, mtop, mw, mb, alphas)
  ! It would be nice to get one more significant digit but the
  ! expression is numerically rather unstable for mb -> 0
  call assert_equal (u, gamma1, 1.3376_default, rel_smallness=1.0E-3_default)

  write (u, "(A)")  "*   Values from threshold validation (massive b)"
  alpha = one / 125.924
  ! ee = 0.315901
  ! cw = 0.881903
  ! v = 240.024
```

```
  mtop = 172.0 ! This is the value for M1S !!!
  mb = 4.2
  sinthw = 0.47143
  mz = 91.188
  mw = 80.419
  call assert_equal (u, sqrt(one - mw**2 / mz**2), sinthw, "sinthw", rel_smallness=1.0E-6_defaul

  write (u, "(A)")  "*   Check LO Width"
  gamma0 = top_width_sm_lo (alpha, sinthw, vtb, mtop, mw, mb)
  call assert_equal (u, gamma0, 1.5386446_default, "gamma0", rel_smallness=1.0E-7_default)
  alphas = zero
  gamma0 = top_width_sm_qcd_nlo_jk (alpha, sinthw, mtop, mw, mb, alphas)
  call assert_equal (u, gamma0, 1.5386446_default, "gamma0", rel_smallness=1.0E-7_default)

  write (u, "(A)")  "*   Check NLO Width"
  alphas_mz = 0.118 !(Z pole, NLL running to mu_h)
  alphas = running_as (mtop, alphas_mz, mz, 1, 5.0_default)
  write (u, "(A," // FMT_15 // ")")  "*   alphas = ", alphas
  gamma1 = top_width_sm_qcd_nlo_jk (alpha, sinthw, mtop, mw, mb, alphas)
  write (u, "(A," // FMT_15 // ")")  "*   Gamma1 = ", gamma1

  mb = zero
  gamma1 = top_width_sm_qcd_nlo_massless_b (alpha, sinthw, mtop, mw, alphas)
  alphas = running_as (mtop, alphas_mz, mz, 1, 5.0_default)
  write (u, "(A," // FMT_15 // ")")  "*   Gamma1(mb=0) = ", gamma1

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sm_physics_2"
end subroutine sm_physics_2
```

9.5 QCD Coupling

We provide various distinct implementations of the QCD coupling. In this module, we define an abstract data type and three implementations: fixed, running with $\alpha_s(M_Z)$ as input, and running with Λ_{QCD} as input. We use the functions defined above in the module `sm_physics` but provide a common interface. Later modules may define additional implementations.

⟨`sm_qcd.f90`⟩≡

```
⟨File header⟩

module sm_qcd

⟨Use kinds⟩
  use io_units
  use format_defs, only: FMT_12
  use numeric_utils
  use diagnostics
  use md5
  use physics_defs
  use sm_physics

⟨Standard module head⟩

⟨SM qcd: public⟩

⟨SM qcd: types⟩

⟨SM qcd: interfaces⟩

contains

⟨SM qcd: procedures⟩

end module sm_qcd
```

9.5.1 Coupling: Abstract Data Type

This is the abstract version of the QCD coupling implementation.

⟨*SM qcd: public*⟩≡

```
public :: alpha_qcd_t
```

⟨*SM qcd: types*⟩≡

```
type, abstract :: alpha_qcd_t
 contains
 ⟨SM qcd: alpha qcd: TBP⟩
end type alpha_qcd_t
```

There must be an output routine.

⟨*SM qcd: alpha qcd: TBP*⟩≡

```
procedure (alpha_qcd_write), deferred :: write
```

⟨*SM qcd: interfaces*⟩≡
```
abstract interface
   subroutine alpha_qcd_write (object, unit)
     import
     class(alpha_qcd_t), intent(in) :: object
     integer, intent(in), optional :: unit
   end subroutine alpha_qcd_write
end interface
```

This method computes the running coupling, given a certain scale. All parameters (reference value, order of the approximation, etc.) must be set before calling this.

⟨*SM qcd: alpha qcd: TBP*⟩+≡
```
procedure (alpha_qcd_get), deferred :: get
```

⟨*SM qcd: interfaces*⟩+≡
```
abstract interface
   function alpha_qcd_get (alpha_qcd, scale) result (alpha)
     import
     class(alpha_qcd_t), intent(in) :: alpha_qcd
     real(default), intent(in) :: scale
     real(default) :: alpha
   end function alpha_qcd_get
end interface
```

9.5.2 Fixed Coupling

In this version, the α_s value is fixed, the `scale` argument of the `get` method is ignored. There is only one parameter, the value. By default, this is the value at M_Z.

⟨*SM qcd: public*⟩+≡
```
public :: alpha_qcd_fixed_t
```

⟨*SM qcd: types*⟩+≡
```
type, extends (alpha_qcd_t) :: alpha_qcd_fixed_t
   real(default) :: val = ALPHA_QCD_MZ_REF
 contains
 ⟨SM qcd: alpha qcd fixed: TBP⟩
end type alpha_qcd_fixed_t
```

Output.

⟨*SM qcd: alpha qcd fixed: TBP*⟩≡
```
procedure :: write => alpha_qcd_fixed_write
```

⟨*SM qcd: procedures*⟩≡
```
subroutine alpha_qcd_fixed_write (object, unit)
  class(alpha_qcd_fixed_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(3x,A)")  "QCD parameters (fixed coupling):"
  write (u, "(5x,A," // FMT_12 // ")")  "alpha = ", object%val
```

```
end subroutine alpha_qcd_fixed_write
```

Calculation: the scale is ignored in this case.

⟨*SM qcd: alpha qcd fixed: TBP*⟩+≡

```
procedure :: get => alpha_qcd_fixed_get
```

⟨*SM qcd: procedures*⟩+≡

```
function alpha_qcd_fixed_get (alpha_qcd, scale) result (alpha)
  class(alpha_qcd_fixed_t), intent(in) :: alpha_qcd
  real(default), intent(in) :: scale
  real(default) :: alpha
  alpha = alpha_qcd%val
end function alpha_qcd_fixed_get
```

9.5.3 Running Coupling

In this version, the α_s value runs relative to the value at a given reference scale. There are two parameters: the value of this scale (default: M_Z), the value of α_s at this scale, and the number of effective flavors. Furthermore, we have the order of the approximation.

⟨*SM qcd: public*⟩+≡

```
public :: alpha_qcd_from_scale_t
```

⟨*SM qcd: types*⟩+≡

```
type, extends (alpha_qcd_t) :: alpha_qcd_from_scale_t
   real(default) :: mu_ref = MZ_REF
   real(default) :: ref = ALPHA_QCD_MZ_REF
   integer :: order = 0
   integer :: nf = 5
 contains
 ⟨SM qcd: alpha qcd from scale: TBP⟩
end type alpha_qcd_from_scale_t
```

Output.

⟨*SM qcd: alpha qcd from scale: TBP*⟩≡

```
procedure :: write => alpha_qcd_from_scale_write
```

⟨*SM qcd: procedures*⟩+≡

```
subroutine alpha_qcd_from_scale_write (object, unit)
  class(alpha_qcd_from_scale_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(3x,A)")  "QCD parameters (running coupling):"
  write (u, "(5x,A," // FMT_12 // ")")  "Scale mu  = ", object%mu_ref
  write (u, "(5x,A," // FMT_12 // ")")  "alpha(mu) = ", object%ref
  write (u, "(5x,A,I0)")      "LL order  = ", object%order
  write (u, "(5x,A,I0)")      "N(flv)    = ", object%nf
end subroutine alpha_qcd_from_scale_write
```

Calculation: here, we call the function for running α_s that was defined in sm_physics above. The function does not take into account thresholds, so the number of flavors should be the correct one for the chosen scale. Normally, this should be the Z boson mass.

⟨*SM qcd: alpha qcd from scale: TBP*⟩+≡
```
procedure :: get => alpha_qcd_from_scale_get
```

⟨*SM qcd: procedures*⟩+≡
```
function alpha_qcd_from_scale_get (alpha_qcd, scale) result (alpha)
  class(alpha_qcd_from_scale_t), intent(in) :: alpha_qcd
  real(default), intent(in) :: scale
  real(default) :: alpha
  alpha = running_as (scale, &
       alpha_qcd%ref, alpha_qcd%mu_ref, alpha_qcd%order, &
       real (alpha_qcd%nf, kind=default))
end function alpha_qcd_from_scale_get
```

9.5.4 Running Coupling, determined by Λ_{QCD}

In this version, the input are the value Λ_{QCD} and the order of the approximation.

⟨*SM qcd: public*⟩+≡
```
public :: alpha_qcd_from_lambda_t
```

⟨*SM qcd: types*⟩+≡
```
type, extends (alpha_qcd_t) :: alpha_qcd_from_lambda_t
   real(default) :: lambda = LAMBDA_QCD_REF
   integer :: order = 0
   integer :: nf = 5
 contains
 ⟨SM qcd: alpha qcd from lambda: TBP⟩
end type alpha_qcd_from_lambda_t
```

Output.

⟨*SM qcd: alpha qcd from lambda: TBP*⟩≡
```
procedure :: write => alpha_qcd_from_lambda_write
```

⟨*SM qcd: procedures*⟩+≡
```
subroutine alpha_qcd_from_lambda_write (object, unit)
  class(alpha_qcd_from_lambda_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(3x,A)")  "QCD parameters (Lambda_QCD as input):"
  write (u, "(5x,A," // FMT_12 // ")")  "Lambda_QCD = ", object%lambda
  write (u, "(5x,A,I0)")      "LL order   = ", object%order
  write (u, "(5x,A,I0)")      "N(flv)     = ", object%nf
end subroutine alpha_qcd_from_lambda_write
```

Calculation: here, we call the second function for running α_s that was defined in sm_physics above. The Λ value should be the one that is appropriate for the chosen number of effective flavors. Again, thresholds are not incorporated.

⟨*SM qcd: alpha qcd from lambda: TBP*⟩+≡
```
procedure :: get => alpha_qcd_from_lambda_get
```

⟨*SM qcd: procedures*⟩+≡

```
function alpha_qcd_from_lambda_get (alpha_qcd, scale) result (alpha)
  class(alpha_qcd_from_lambda_t), intent(in) :: alpha_qcd
  real(default), intent(in) :: scale
  real(default) :: alpha
  alpha = running_as_lam (real (alpha_qcd%nf, kind=default), scale, &
       alpha_qcd%lambda, alpha_qcd%order)
end function alpha_qcd_from_lambda_get
```

9.5.5 Wrapper type

We could get along with a polymorphic QCD type, but a monomorphic wrapper type with a polymorphic component is easier to handle and probably safer (w.r.t. compiler bugs). However, we keep the object transparent, so we can set the type-specific parameters directly (by a `dispatch` routine).

⟨*SM qcd: public*⟩+≡

```
public :: qcd_t
```

⟨*SM qcd: types*⟩+≡

```
type :: qcd_t
   class(alpha_qcd_t), allocatable :: alpha
   character(32) :: md5sum = ""
 contains
 ⟨SM qcd: qcd: TBP⟩
end type qcd_t
```

Output. We first print the polymorphic `alpha` which contains a headline, then any extra components.

⟨*SM qcd: qcd: TBP*⟩≡

```
procedure :: write => qcd_write
```

⟨*SM qcd: procedures*⟩+≡

```
subroutine qcd_write (qcd, unit, show_md5sum)
  class(qcd_t), intent(in) :: qcd
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: show_md5sum
  logical :: show_md5
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  show_md5 = .true.;  if (present (show_md5sum))  show_md5 = show_md5sum
  if (allocated (qcd%alpha)) then
     call qcd%alpha%write (u)
  else
     write (u, "(3x,A)")  "QCD parameters (coupling undefined)"
  end if
  if (show_md5 .and. qcd%md5sum /= "") &
       write (u, "(5x,A,A,A)") "md5sum = '", qcd%md5sum, "'"
end subroutine qcd_write
```

Compute a MD5 sum for the `alpha_s` setup. This is done by writing them to a temporary file, using a standard format.

⟨*SM qcd: qcd: TBP*⟩+≡

```
    procedure :: compute_alphas_md5sum => qcd_compute_alphas_md5sum
```

⟨*SM qcd: procedures*⟩+≡

```
    subroutine qcd_compute_alphas_md5sum (qcd)
      class(qcd_t), intent(inout) :: qcd
      integer :: unit
      if (allocated (qcd%alpha)) then
         unit = free_unit ()
         open (unit, status="scratch", action="readwrite")
         call qcd%alpha%write (unit)
         rewind (unit)
         qcd%md5sum = md5sum (unit)
         close (unit)
      end if
    end subroutine qcd_compute_alphas_md5sum
```

Retrieve the MD5 sum of the qcd setup.

⟨*SM qcd: qcd: TBP*⟩+≡

```
    procedure :: get_md5sum => qcd_get_md5sum
```

⟨*SM qcd: procedures*⟩+≡

```
    function qcd_get_md5sum (qcd) result (md5sum)
      character(32) :: md5sum
      class(qcd_t), intent(inout) :: qcd
      md5sum = qcd%md5sum
    end function qcd_get_md5sum
```

9.5.6 Unit tests

Test module, followed by the corresponding implementation module.

⟨sm_qcd_ut.f90⟩≡

```
  ⟨File header⟩

  module sm_qcd_ut
    use unit_tests
    use sm_qcd_uti

  ⟨Standard module head⟩

  ⟨SM qcd: public test⟩

  contains

  ⟨SM qcd: test driver⟩

  end module sm_qcd_ut
```

⟨sm_qcd_uti.f90⟩≡

```
  ⟨File header⟩

  module sm_qcd_uti

  ⟨Use kinds⟩
```

```
  use physics_defs, only: MZ_REF

  use sm_qcd
```

⟨*Standard module head*⟩

⟨*SM qcd: test declarations*⟩

```
contains
```

⟨*SM qcd: tests*⟩

```
end module sm_qcd_uti
```

API: driver for the unit tests below.

⟨*SM qcd: public test*⟩≡

```
  public :: sm_qcd_test
```

⟨*SM qcd: test driver*⟩≡

```
  subroutine sm_qcd_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```

⟨*SM qcd: execute tests*⟩

```
  end subroutine sm_qcd_test
```

QCD Coupling

We check two different implementations of the abstract QCD coupling.

⟨*SM qcd: execute tests*⟩≡

```
  call test (sm_qcd_1, "sm_qcd_1", &
       "running alpha_s", &
       u, results)
```

⟨*SM qcd: test declarations*⟩≡

```
  public :: sm_qcd_1
```

⟨*SM qcd: tests*⟩≡

```
  subroutine sm_qcd_1 (u)
    integer, intent(in) :: u
    type(qcd_t) :: qcd

    write (u, "(A)")  "* Test output: sm_qcd_1"
    write (u, "(A)")  "*   Purpose: compute running alpha_s"
    write (u, "(A)")

    write (u, "(A)")  "* Fixed:"
    write (u, "(A)")

    allocate (alpha_qcd_fixed_t :: qcd%alpha)
    call qcd%compute_alphas_md5sum ()

    call qcd%write (u)
    write (u, *)
    write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
         qcd%alpha%get (MZ_REF)
```

```
write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
     qcd%alpha%get (1000._default)
write (u, *)
deallocate (qcd%alpha)

write (u, "(A)")  "* Running from MZ (LO):"
write (u, "(A)")

allocate (alpha_qcd_from_scale_t :: qcd%alpha)
call qcd%compute_alphas_md5sum ()

call qcd%write (u)
write (u, *)
write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
     qcd%alpha%get (MZ_REF)
write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
     qcd%alpha%get (1000._default)
write (u, *)

write (u, "(A)")  "* Running from MZ (NLO):"
write (u, "(A)")

select type (alpha => qcd%alpha)
type is (alpha_qcd_from_scale_t)
   alpha%order = 1
end select
call qcd%compute_alphas_md5sum ()

call qcd%write (u)
write (u, *)
write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
     qcd%alpha%get (MZ_REF)
write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
     qcd%alpha%get (1000._default)
write (u, *)

write (u, "(A)")  "* Running from MZ (NNLO):"
write (u, "(A)")

select type (alpha => qcd%alpha)
type is (alpha_qcd_from_scale_t)
   alpha%order = 2
end select
call qcd%compute_alphas_md5sum ()

call qcd%write (u)
write (u, *)
write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
     qcd%alpha%get (MZ_REF)
write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
     qcd%alpha%get (1000._default)
write (u, *)

deallocate (qcd%alpha)
```

```
  write (u, "(A)")  "* Running from Lambda_QCD (LO):"
  write (u, "(A)")

  allocate (alpha_qcd_from_lambda_t :: qcd%alpha)
  call qcd%compute_alphas_md5sum ()

  call qcd%write (u)
  write (u, *)
  write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
       qcd%alpha%get (MZ_REF)
  write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
       qcd%alpha%get (1000._default)
  write (u, *)

  write (u, "(A)")  "* Running from Lambda_QCD (NLO):"
  write (u, "(A)")

  select type (alpha => qcd%alpha)
  type is (alpha_qcd_from_lambda_t)
     alpha%order = 1
  end select
  call qcd%compute_alphas_md5sum ()

  call qcd%write (u)
  write (u, *)
  write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
       qcd%alpha%get (MZ_REF)
  write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
       qcd%alpha%get (1000._default)
  write (u, *)

  write (u, "(A)")  "* Running from Lambda_QCD (NNLO):"
  write (u, "(A)")

  select type (alpha => qcd%alpha)
  type is (alpha_qcd_from_lambda_t)
     alpha%order = 2
  end select
  call qcd%compute_alphas_md5sum ()

  call qcd%write (u)
  write (u, *)
  write (u, "(1x,A,F10.7)")  "alpha_s (mz)    =", &
       qcd%alpha%get (MZ_REF)
  write (u, "(1x,A,F10.7)")  "alpha_s (1 TeV) =", &
       qcd%alpha%get (1000._default)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sm_qcd_1"

end subroutine sm_qcd_1
```

9.6 Shower algorithms

⟨shower_algorithms.f90⟩≡

```
⟨File header⟩

module shower_algorithms

⟨Use kinds⟩
  use diagnostics
  use constants

⟨Standard module head⟩

⟨shower algorithms: public⟩

⟨shower algorithms: interfaces⟩

contains

⟨shower algorithms: procedures⟩

⟨shower algorithms: tests⟩

end module shower_algorithms
```

We want to generate emission variables x$\in \mathbb{R}^d$ proportional to

$$f(x)\ \Delta(f, h(x)) \quad \text{with} \tag{9.30}$$

$$\Delta(f, H) = \exp\left\{-\int \mathrm{d}^d x' f(x')\Theta(h(x') - H)\right\} \tag{9.31}$$

The `true_function` f is however too complicated and we are only able to generate `x` according to the `overestimator` F. This algorithm is described in Appendix B of 0709.2092 and is proven e.g. in 1211.7204 and hep-ph/0606275. Intuitively speaking, we overestimate the emission probability and can therefore set `scale_max = scale` if the emission is rejected.

⟨shower algorithms: procedures⟩≡

```
subroutine generate_vetoed (x, overestimator, true_function, &
       sudakov, inverse_sudakov, scale_min)
  real(default), dimension(:), intent(out) :: x
  !class(rng_t), intent(inout) :: rng
  procedure(XXX_function), pointer, intent(in) :: overestimator, true_function
  procedure(sudakov_p), pointer, intent(in) :: sudakov, inverse_sudakov
  real(default), intent(in) :: scale_min
  real(default) :: random, scale_max, scale
  scale_max = inverse_sudakov (one)
  do while (scale_max > scale_min)
     !call rng%generate (random)
     scale = inverse_sudakov (random * sudakov (scale_max))
     call generate_on_hypersphere (x, overestimator, scale)
     !call rng%generate (random)
     if (random < true_function (x) / overestimator (x)) then
        return !!! accept x
     end if
```

```
        scale_max = scale
      end do
    end subroutine generate_vetoed
```

⟨*shower algorithms: procedures*⟩+≡

```
    subroutine generate_on_hypersphere (x, overestimator, scale)
      real(default), dimension(:), intent(out) :: x
      procedure(XXX_function), pointer, intent(in) :: overestimator
      real(default), intent(in) :: scale
      call msg_bug ("generate_on_hypersphere: not implemented")
    end subroutine generate_on_hypersphere
```

⟨*shower algorithms: interfaces*⟩≡

```
    interface
      pure function XXX_function (x)
        import
        real(default) :: XXX_function
        real(default), dimension(:), intent(in) :: x
      end function XXX_function
    end interface
    interface
      pure function sudakov_p (x)
        import
        real(default) :: sudakov_p
        real(default), intent(in) :: x
      end function sudakov_p
    end interface
```

9.6.1 Unit tests

(Currently unused.)

⟨*XXX shower algorithms: public*⟩≡

```
    public :: shower_algorithms_test
```

⟨*XXX shower algorithms: tests*⟩≡

```
    subroutine shower_algorithms_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨shower algorithms: execute tests⟩
    end subroutine shower_algorithms_test
```

Splitting functions

⟨*XXX shower algorithms: execute tests*⟩≡

```
    call test (shower_algorithms_1, "shower_algorithms_1", &
         "veto technique", &
         u, results)
```

⟨*XXX shower algorithms: tests*⟩+≡

```
    subroutine shower_algorithms_1 (u)
      integer, intent(in) :: u
```

```
  write (u, "(A)")  "* Test output: shower_algorithms_1"
  write (u, "(A)")  "*   Purpose: check veto technique"
  write (u, "(A)")

  write (u, "(A)")  "* Splitting functions:"
  write (u, "(A)")

  !call assert (u, vanishes (p_qqg_pol (z, +1, -1, +1)))
  !call assert (u, nearly_equal ( &
       !p_qqg_pol (z, +1, +1, -1) + p_qqg_pol (z, +1, +1, +1),
       !p_qqg (z))

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: shower_algorithms_1"

end subroutine shower_algorithms_1
```

Chapter 10

Quantum Field Theory Concepts

The objects and methods defined here implement concepts and data for the underlying quantum field theory that we use for computing matrix elements and processes.

model_data Fields and coupling parameters, operators as vertex structures, for a specific model.

model_testbed Provide hooks to deal with a `model_data` extension without referencing it explicitly.

helicities Types and methods for spin density matrices.

colors Dealing with colored particles, using the color-flow representation.

flavors PDG codes and particle properties, depends on the model.

quantum_numbers Quantum numbers and density matrices for entangled particle systems.

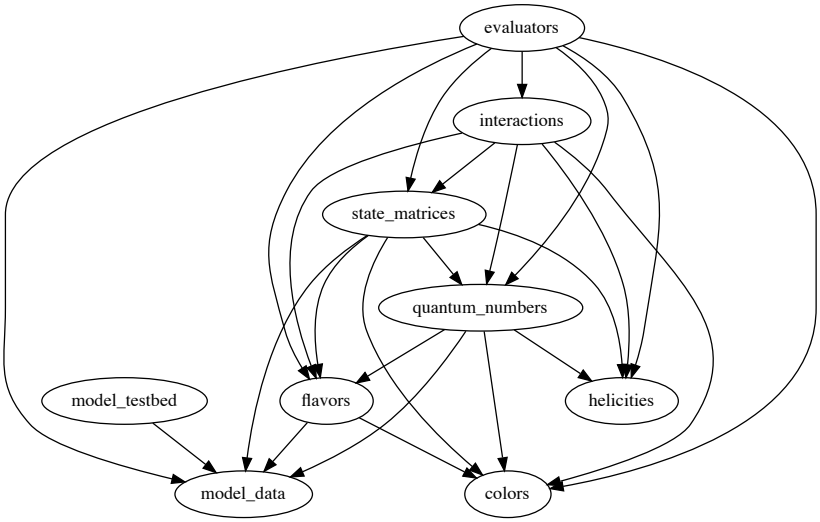

Figure 10.1: Module dependencies in `src/qft`.

10.1 Model Data

These data represent a specific Lagrangian in numeric terms. That is, we have the fields with their quantum numbers, the masses, widths and couplings as numerical values, and the vertices as arrays of fields.

We do not store the relations between coupling parameters. They should be represented by expressions for evaluation, implemented as Sindarin objects in a distinct data structure. Neither do we need the algebraic structure of vertices. The field content of vertices is required for the sole purpose of setting up phase space.

⟨`model_data.f90`⟩≡

```
⟨File header⟩

module model_data

  use, intrinsic :: iso_c_binding !NODEP!

⟨Use kinds⟩
  use kinds, only: i8, i32
  use kinds, only: c_default_float
⟨Use strings⟩
  use format_defs, only: FMT_19
  use io_units
  use diagnostics
  use md5
  use hashes, only: hash
  use physics_defs, only: UNDEFINED, SCALAR

⟨Standard module head⟩

⟨Model data: public⟩

⟨Model data: parameters⟩

⟨Model data: types⟩

contains

⟨Model data: procedures⟩

end module model_data
```

10.1.1 Physics Parameters

Couplings, masses, and widths are physics parameters. Each parameter has a unique name (used, essentially, for diagnostics output and debugging) and a value. The value may be a real or a complex number, so we provide to implementations of an abstract type.

⟨*Model data: public*⟩≡

```
public :: modelpar_data_t
```

⟨*Model data: types*⟩≡

```
type, abstract :: modelpar_data_t
```

```
      private
      type(string_t) :: name
   contains
   ⟨Model data: par data: TBP⟩
end type modelpar_data_t

type, extends (modelpar_data_t) :: modelpar_real_t
   private
   real(default) :: value
end type modelpar_real_t

type, extends (modelpar_data_t) :: modelpar_complex_t
   private
   complex(default) :: value
end type modelpar_complex_t
```

Output for diagnostics. Non-advancing.

⟨*Model data: par data: TBP*⟩≡

```
procedure :: write => par_write
```

⟨*Model data: procedures*⟩≡

```
subroutine par_write (par, unit)
  class(modelpar_data_t), intent(in) :: par
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A,1x,A)", advance="no")  char (par%name), "= "
  select type (par)
  type is (modelpar_real_t)
     write (u, "(" // FMT_19 // ")", advance="no")  par%value
  type is (modelpar_complex_t)
     write (u, "(" // FMT_19 // ",1x,'+',1x," // FMT_19 // ",1x,'I')", &
          advance="no")  par%value
  end select
end subroutine par_write
```

Pretty-printed on separate line, with fixed line length

⟨*Model data: par data: TBP*⟩+≡

```
procedure :: show => par_show
```

⟨*Model data: procedures*⟩+≡

```
subroutine par_show (par, l, u)
  class(modelpar_data_t), intent(in) :: par
  integer, intent(in) :: l, u
  character(len=l) :: buffer
  buffer = par%name
  select type (par)
  type is (modelpar_real_t)
     write (u, "(4x,A,1x,'=',1x," // FMT_19 // ")")  buffer, par%value
  type is (modelpar_complex_t)
     write (u, "(4x,A,1x,'=',1x," // FMT_19 // ",1x,'+',1x," &
          // FMT_19 // ",1x,'I')")  buffer, par%value
  end select
end subroutine par_show
```

Initialize with name and value. The type depends on the argument type. If the type does not match, the value is converted following Fortran rules.

⟨*Model data: par data: TBP*⟩+≡

```
generic :: init => modelpar_data_init_real, modelpar_data_init_complex
procedure, private :: modelpar_data_init_real
procedure, private :: modelpar_data_init_complex
```

⟨*Model data: procedures*⟩+≡

```
subroutine modelpar_data_init_real (par, name, value)
  class(modelpar_data_t), intent(out) :: par
  type(string_t), intent(in) :: name
  real(default), intent(in) :: value
  par%name = name
  par = value
end subroutine modelpar_data_init_real

subroutine modelpar_data_init_complex (par, name, value)
  class(modelpar_data_t), intent(out) :: par
  type(string_t), intent(in) :: name
  complex(default), intent(in) :: value
  par%name = name
  par = value
end subroutine modelpar_data_init_complex
```

Modify the value. We assume that the parameter has been initialized. The type (real or complex) must not be changed, and the name is also fixed.

⟨*Model data: par data: TBP*⟩+≡

```
generic :: assignment(=) => modelpar_data_set_real, modelpar_data_set_complex
procedure, private :: modelpar_data_set_real
procedure, private :: modelpar_data_set_complex
```

⟨*Model data: procedures*⟩+≡

```
elemental subroutine modelpar_data_set_real (par, value)
  class(modelpar_data_t), intent(inout) :: par
  real(default), intent(in) :: value
  select type (par)
  type is (modelpar_real_t)
     par%value = value
  type is (modelpar_complex_t)
     par%value = value
  end select
end subroutine modelpar_data_set_real

elemental subroutine modelpar_data_set_complex (par, value)
  class(modelpar_data_t), intent(inout) :: par
  complex(default), intent(in) :: value
  select type (par)
  type is (modelpar_real_t)
     par%value = value
  type is (modelpar_complex_t)
     par%value = value
  end select
end subroutine modelpar_data_set_complex
```

Return the parameter name.

⟨*Model data: par data: TBP*⟩+≡

```
procedure :: get_name => modelpar_data_get_name
```

⟨*Model data: procedures*⟩+≡

```
function modelpar_data_get_name (par) result (name)
  class(modelpar_data_t), intent(in) :: par
  type(string_t) :: name
  name = par%name
end function modelpar_data_get_name
```

Return the value. In case of a type mismatch, follow Fortran conventions.

⟨*Model data: par data: TBP*⟩+≡

```
procedure, pass :: get_real => modelpar_data_get_real
procedure, pass :: get_complex => modelpar_data_get_complex
```

⟨*Model data: procedures*⟩+≡

```
elemental function modelpar_data_get_real (par) result (value)
  class(modelpar_data_t), intent(in), target :: par
  real(default) :: value
  select type (par)
  type is (modelpar_real_t)
     value = par%value
  type is (modelpar_complex_t)
     value = par%value
  end select
end function modelpar_data_get_real

elemental function modelpar_data_get_complex (par) result (value)
  class(modelpar_data_t), intent(in), target :: par
  complex(default) :: value
  select type (par)
  type is (modelpar_real_t)
     value = par%value
  type is (modelpar_complex_t)
     value = par%value
  end select
end function modelpar_data_get_complex
```

Return a pointer to the value. This makes sense only for matching types.

⟨*Model data: par data: TBP*⟩+≡

```
procedure :: get_real_ptr => modelpar_data_get_real_ptr
procedure :: get_complex_ptr => modelpar_data_get_complex_ptr
```

⟨*Model data: procedures*⟩+≡

```
function modelpar_data_get_real_ptr (par) result (ptr)
  class(modelpar_data_t), intent(in), target :: par
  real(default), pointer :: ptr
  select type (par)
  type is (modelpar_real_t)
     ptr => par%value
  class default
     ptr => null ()
```

```
  end select
end function modelpar_data_get_real_ptr

function modelpar_data_get_complex_ptr (par) result (ptr)
  class(modelpar_data_t), intent(in), target :: par
  complex(default), pointer :: ptr
  select type (par)
  type is (modelpar_complex_t)
     ptr => par%value
  class default
     ptr => null ()
  end select
end function modelpar_data_get_complex_ptr
```

10.1.2 Field Data

The field-data type holds all information that pertains to a particular field (or particle) within a particular model. Information such as spin type, particle code etc. is stored within the object itself, while mass and width are associated to parameters, otherwise assumed zero.

⟨*Model data: public*⟩+≡

```
public :: field_data_t
```

⟨*Model data: types*⟩+≡

```
type :: field_data_t
   private
   type(string_t) :: longname
   integer :: pdg = UNDEFINED
   logical :: visible = .true.
   logical :: parton = .false.
   logical :: gauge = .false.
   logical :: left_handed = .false.
   logical :: right_handed = .false.
   logical :: has_anti = .false.
   logical :: p_is_stable = .true.
   logical :: p_decays_isotropically = .false.
   logical :: p_decays_diagonal = .false.
   logical :: p_has_decay_helicity = .false.
   integer :: p_decay_helicity = 0
   logical :: a_is_stable = .true.
   logical :: a_decays_isotropically = .false.
   logical :: a_decays_diagonal = .false.
   logical :: a_has_decay_helicity = .false.
   integer :: a_decay_helicity = 0
   logical :: p_polarized = .false.
   logical :: a_polarized = .false.
   type(string_t), dimension(:), allocatable :: name, anti
   type(string_t) :: tex_name, tex_anti
   integer :: spin_type = UNDEFINED
   integer :: isospin_type = 1
   integer :: charge_type = 1
   integer :: color_type = 1
   real(default), pointer :: mass_val => null ()
```

```
    class(modelpar_data_t), pointer :: mass_data => null ()
    real(default), pointer :: width_val => null ()
    class(modelpar_data_t), pointer :: width_data => null ()
    integer :: multiplicity = 1
    type(string_t), dimension(:), allocatable :: p_decay
    type(string_t), dimension(:), allocatable :: a_decay
  contains
  ⟨Model data: field data: TBP⟩
end type field_data_t
```

Initialize field data with PDG long name and PDG code. TEX names should be initialized to avoid issues with accessing unallocated string contents.

⟨*Model data: field data: TBP*⟩≡

```
procedure :: init => field_data_init
```

⟨*Model data: procedures*⟩+≡

```
subroutine field_data_init (prt, longname, pdg)
  class(field_data_t), intent(out) :: prt
  type(string_t), intent(in) :: longname
  integer, intent(in) :: pdg
  prt%longname = longname
  prt%pdg = pdg
  prt%tex_name = ""
  prt%tex_anti = ""
end subroutine field_data_init
```

Copy quantum numbers from another particle. Do not compute the multiplicity yet, because this depends on the association of the `mass_data` pointer.

⟨*Model data: field data: TBP*⟩+≡

```
procedure :: copy_from => field_data_copy_from
```

⟨*Model data: procedures*⟩+≡

```
subroutine field_data_copy_from (prt, prt_src)
  class(field_data_t), intent(inout) :: prt
  class(field_data_t), intent(in) :: prt_src
  prt%visible = prt_src%visible
  prt%parton = prt_src%parton
  prt%gauge = prt_src%gauge
  prt%left_handed = prt_src%left_handed
  prt%right_handed = prt_src%right_handed
  prt%p_is_stable =             prt_src%p_is_stable
  prt%p_decays_isotropically =  prt_src%p_decays_isotropically
  prt%p_decays_diagonal =       prt_src%p_decays_diagonal
  prt%p_has_decay_helicity =    prt_src%p_has_decay_helicity
  prt%p_decay_helicity =        prt_src%p_decay_helicity
  prt%p_decays_diagonal =       prt_src%p_decays_diagonal
  prt%a_is_stable =             prt_src%a_is_stable
  prt%a_decays_isotropically =  prt_src%a_decays_isotropically
  prt%a_decays_diagonal =       prt_src%a_decays_diagonal
  prt%a_has_decay_helicity =    prt_src%a_has_decay_helicity
  prt%a_decay_helicity =        prt_src%a_decay_helicity
  prt%p_polarized =             prt_src%p_polarized
  prt%a_polarized =             prt_src%a_polarized
  prt%spin_type = prt_src%spin_type
```

```
    prt%isospin_type = prt_src%isospin_type
    prt%charge_type = prt_src%charge_type
    prt%color_type = prt_src%color_type
    prt%has_anti = prt_src%has_anti
    if (allocated (prt_src%name)) then
       if (allocated (prt%name))  deallocate (prt%name)
       allocate (prt%name (size (prt_src%name)), source = prt_src%name)
    end if
    if (allocated (prt_src%anti)) then
       if (allocated (prt%anti))  deallocate (prt%anti)
       allocate (prt%anti (size (prt_src%anti)), source = prt_src%anti)
    end if
    prt%tex_name = prt_src%tex_name
    prt%tex_anti = prt_src%tex_anti
    if (allocated (prt_src%p_decay)) then
       if (allocated (prt%p_decay))  deallocate (prt%p_decay)
       allocate (prt%p_decay (size (prt_src%p_decay)), source = prt_src%p_decay)
    end if
    if (allocated (prt_src%a_decay)) then
       if (allocated (prt%a_decay))  deallocate (prt%a_decay)
       allocate (prt%a_decay (size (prt_src%a_decay)), source = prt_src%a_decay)
    end if
  end subroutine field_data_copy_from
```

Set particle quantum numbers.

⟨*Model data: field data: TBP*⟩+≡

```
  procedure :: set => field_data_set
```

⟨*Model data: procedures*⟩+≡

```
  subroutine field_data_set (prt, &
       is_visible, is_parton, is_gauge, is_left_handed, is_right_handed, &
       p_is_stable, p_decays_isotropically, p_decays_diagonal, &
       p_decay_helicity, &
       a_is_stable, a_decays_isotropically, a_decays_diagonal, &
       a_decay_helicity, &
       p_polarized, a_polarized, &
       name, anti, tex_name, tex_anti, &
       spin_type, isospin_type, charge_type, color_type, &
       mass_data, width_data, &
       p_decay, a_decay)
    class(field_data_t), intent(inout) :: prt
    logical, intent(in), optional :: is_visible, is_parton, is_gauge
    logical, intent(in), optional :: is_left_handed, is_right_handed
    logical, intent(in), optional :: p_is_stable
    logical, intent(in), optional :: p_decays_isotropically, p_decays_diagonal
    integer, intent(in), optional :: p_decay_helicity
    logical, intent(in), optional :: a_is_stable
    logical, intent(in), optional :: a_decays_isotropically, a_decays_diagonal
    integer, intent(in), optional :: a_decay_helicity
    logical, intent(in), optional :: p_polarized, a_polarized
    type(string_t), dimension(:), intent(in), optional :: name, anti
    type(string_t), intent(in), optional :: tex_name, tex_anti
    integer, intent(in), optional :: spin_type, isospin_type
    integer, intent(in), optional :: charge_type, color_type
```

```
class(modelpar_data_t), intent(in), pointer, optional :: mass_data, width_data
type(string_t), dimension(:), intent(in), optional :: p_decay, a_decay
if (present (is_visible))  prt%visible = is_visible
if (present (is_parton))  prt%parton = is_parton
if (present (is_gauge))  prt%gauge = is_gauge
if (present (is_left_handed))  prt%left_handed = is_left_handed
if (present (is_right_handed))  prt%right_handed = is_right_handed
if (present (p_is_stable))  prt%p_is_stable = p_is_stable
if (present (p_decays_isotropically)) &
      prt%p_decays_isotropically = p_decays_isotropically
if (present (p_decays_diagonal)) &
      prt%p_decays_diagonal = p_decays_diagonal
if (present (p_decay_helicity)) then
   prt%p_has_decay_helicity = .true.
   prt%p_decay_helicity = p_decay_helicity
end if
if (present (a_is_stable))  prt%a_is_stable = a_is_stable
if (present (a_decays_isotropically)) &
      prt%a_decays_isotropically = a_decays_isotropically
if (present (a_decays_diagonal)) &
      prt%a_decays_diagonal = a_decays_diagonal
if (present (a_decay_helicity)) then
   prt%a_has_decay_helicity = .true.
   prt%a_decay_helicity = a_decay_helicity
end if
if (present (p_polarized)) prt%p_polarized = p_polarized
if (present (a_polarized)) prt%a_polarized = a_polarized
if (present (name)) then
   if (allocated (prt%name))  deallocate (prt%name)
   allocate (prt%name (size (name)), source = name)
end if
if (present (anti)) then
   if (allocated (prt%anti))  deallocate (prt%anti)
   allocate (prt%anti (size (anti)), source = anti)
   prt%has_anti = .true.
end if
if (present (tex_name))  prt%tex_name = tex_name
if (present (tex_anti))  prt%tex_anti = tex_anti
if (present (spin_type))  prt%spin_type = spin_type
if (present (isospin_type))  prt%isospin_type = isospin_type
if (present (charge_type))  prt%charge_type = charge_type
if (present (color_type))  prt%color_type = color_type
if (present (mass_data)) then
   prt%mass_data => mass_data
   if (associated (mass_data)) then
      prt%mass_val => mass_data%get_real_ptr ()
   else
      prt%mass_val => null ()
   end if
end if
if (present (width_data)) then
   prt%width_data => width_data
   if (associated (width_data)) then
      prt%width_val => width_data%get_real_ptr ()
```

```
      else
         prt%width_val => null ()
      end if
   end if
   if (present (spin_type) .or. present (mass_data)) then
      call prt%set_multiplicity ()
   end if
   if (present (p_decay)) then
      if (allocated (prt%p_decay))  deallocate (prt%p_decay)
      if (size (p_decay) > 0) &
           allocate (prt%p_decay (size (p_decay)), source = p_decay)
   end if
   if (present (a_decay)) then
      if (allocated (prt%a_decay))  deallocate (prt%a_decay)
      if (size (a_decay) > 0) &
           allocate (prt%a_decay (size (a_decay)), source = a_decay)
   end if
end subroutine field_data_set
```

Calculate the multiplicity given spin type and mass.

⟨*Model data: field data: TBP*⟩+≡

```
procedure, private :: &
     set_multiplicity => field_data_set_multiplicity
```

⟨*Model data: procedures*⟩+≡

```
subroutine field_data_set_multiplicity (prt)
  class(field_data_t), intent(inout) :: prt
  if (prt%spin_type /= SCALAR) then
     if (associated (prt%mass_data)) then
        prt%multiplicity = prt%spin_type
     else if (prt%left_handed .or. prt%right_handed) then
        prt%multiplicity = 1
     else
        prt%multiplicity = 2
     end if
  end if
end subroutine field_data_set_multiplicity
```

Set the mass/width value (not the pointer). The mass/width pointer must be allocated.

⟨*Model data: field data: TBP*⟩+≡

```
procedure, private :: set_mass => field_data_set_mass
procedure, private :: set_width => field_data_set_width
```

⟨*Model data: procedures*⟩+≡

```
subroutine field_data_set_mass (prt, mass)
  class(field_data_t), intent(inout) :: prt
  real(default), intent(in) :: mass
  if (associated (prt%mass_val))  prt%mass_val = mass
end subroutine field_data_set_mass

subroutine field_data_set_width (prt, width)
  class(field_data_t), intent(inout) :: prt
  real(default), intent(in) :: width
```

```
    if (associated (prt%width_val))  prt%width_val = width
  end subroutine field_data_set_width
```

Loose ends: name arrays should be allocated.

⟨*Model data: field data: TBP*⟩+≡

```
  procedure :: freeze => field_data_freeze
```

⟨*Model data: procedures*⟩+≡

```
  elemental subroutine field_data_freeze (prt)
    class(field_data_t), intent(inout) :: prt
    if (.not. allocated (prt%name))  allocate (prt%name (0))
    if (.not. allocated (prt%anti))  allocate (prt%anti (0))
  end subroutine field_data_freeze
```

Output

⟨*Model data: field data: TBP*⟩+≡

```
  procedure :: write => field_data_write
```

⟨*Model data: procedures*⟩+≡

```
  subroutine field_data_write (prt, unit)
    class(field_data_t), intent(in) :: prt
    integer, intent(in), optional :: unit
    integer :: u, i
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(3x,A,1x,A)", advance="no") "particle", char (prt%longname)
    write (u, "(1x,I0)", advance="no") prt%pdg
    if (.not. prt%visible) write (u, "(2x,A)", advance="no") "invisible"
    if (prt%parton)  write (u, "(2x,A)", advance="no") "parton"
    if (prt%gauge)  write (u, "(2x,A)", advance="no") "gauge"
    if (prt%left_handed)  write (u, "(2x,A)", advance="no") "left"
    if (prt%right_handed)  write (u, "(2x,A)", advance="no") "right"
    write (u, *)
    write (u, "(5x,A)", advance="no") "name"
    if (allocated (prt%name)) then
       do i = 1, size (prt%name)
          write (u, "(1x,A)", advance="no")  '"' // char (prt%name(i)) // '"'
       end do
       write (u, *)
       if (prt%has_anti) then
          write (u, "(5x,A)", advance="no") "anti"
          do i = 1, size (prt%anti)
             write (u, "(1x,A)", advance="no")  '"' // char (prt%anti(i)) // '"'
          end do
          write (u, *)
       end if
       if (prt%tex_name /= "") then
          write (u, "(5x,A)")  &
               "tex_name " // '"' // char (prt%tex_name) // '"'
       end if
       if (prt%has_anti .and. prt%tex_anti /= "") then
          write (u, "(5x,A)")  &
               "tex_anti " // '"' // char (prt%tex_anti) // '"'
       end if
    else
```

```
      write (u, "(A)")  "???"
   end if
   write (u, "(5x,A)", advance="no") "spin "
   select case (mod (prt%spin_type - 1, 2))
   case (0);  write (u, "(I0)", advance="no") (prt%spin_type-1) / 2
   case default;  write (u, "(I0,A)", advance="no") prt%spin_type-1, "/2"
   end select
   ! write (u, "(2x,A,I1,A)") "! [multiplicity = ", prt%multiplicity, "]"
   if (abs (prt%isospin_type) /= 1) then
      write (u, "(2x,A)", advance="no") "isospin "
      select case (mod (abs (prt%isospin_type) - 1, 2))
      case (0);  write (u, "(I0)", advance="no") &
           sign (abs (prt%isospin_type) - 1, prt%isospin_type) / 2
      case default;  write (u, "(I0,A)", advance="no") &
           sign (abs (prt%isospin_type) - 1, prt%isospin_type), "/2"
      end select
   end if
   if (abs (prt%charge_type) /= 1) then
      write (u, "(2x,A)", advance="no") "charge "
      select case (mod (abs (prt%charge_type) - 1, 3))
      case (0);  write (u, "(I0)", advance="no") &
           sign (abs (prt%charge_type) - 1, prt%charge_type) / 3
      case default;  write (u, "(I0,A)", advance="no") &
           sign (abs (prt%charge_type) - 1, prt%charge_type), "/3"
      end select
   end if
   if (prt%color_type /= 1) then
      write (u, "(2x,A,I0)", advance="no") "color ", prt%color_type
   end if
   write (u, *)
   if (associated (prt%mass_data)) then
      write (u, "(5x,A)", advance="no") &
           "mass " // char (prt%mass_data%get_name ())
      if (associated (prt%width_data)) then
         write (u, "(2x,A)") &
              "width " // char (prt%width_data%get_name ())
      else
         write (u, *)
      end if
   end if
   call prt%write_decays (u)
end subroutine field_data_write
```

Write decay and polarization data.

⟨*Model data: field data: TBP*⟩+≡

```
procedure :: write_decays => field_data_write_decays
```

⟨*Model data: procedures*⟩+≡

```
subroutine field_data_write_decays (prt, unit)
  class(field_data_t), intent(in) :: prt
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  if (.not. prt%p_is_stable) then
```

```
      if (allocated (prt%p_decay)) then
         write (u, "(5x,A)", advance="no") "p_decay"
         do i = 1, size (prt%p_decay)
            write (u, "(1x,A)", advance="no")  char (prt%p_decay(i))
         end do
         if (prt%p_decays_isotropically) then
            write (u, "(1x,A)", advance="no")  "isotropic"
         else if (prt%p_decays_diagonal) then
            write (u, "(1x,A)", advance="no")  "diagonal"
         else if (prt%p_has_decay_helicity) then
            write (u, "(1x,A,I0)", advance="no")  "helicity = ", &
                 prt%p_decay_helicity
         end if
         write (u, *)
      end if
   else if (prt%p_polarized) then
      write (u, "(5x,A)")  "p_polarized"
   end if
   if (.not. prt%a_is_stable) then
      if (allocated (prt%a_decay)) then
         write (u, "(5x,A)", advance="no") "a_decay"
         do i = 1, size (prt%a_decay)
            write (u, "(1x,A)", advance="no")  char (prt%a_decay(i))
         end do
         if (prt%a_decays_isotropically) then
            write (u, "(1x,A)", advance="no")  "isotropic"
         else if (prt%a_decays_diagonal) then
            write (u, "(1x,A)", advance="no")  "diagonal"
         else if (prt%a_has_decay_helicity) then
            write (u, "(1x,A,I0)", advance="no")  "helicity = ", &
                 prt%a_decay_helicity
         end if
         write (u, *)
      end if
   else if (prt%a_polarized) then
      write (u, "(5x,A)")  "a_polarized"
   end if
end subroutine field_data_write_decays
```

Screen version of output.

⟨*Model data: field data: TBP*⟩+≡

```
procedure :: show => field_data_show
```

⟨*Model data: procedures*⟩+≡

```
subroutine field_data_show (prt, l, u)
  class(field_data_t), intent(in) :: prt
  integer, intent(in) :: l, u
  character(len=l) :: buffer
  integer :: i
  type(string_t), dimension(:), allocatable :: decay
  buffer = prt%get_name (.false.)
  write (u, "(4x,A,1x,I8)", advance="no")  buffer, &
       prt%get_pdg ()
  if (prt%is_polarized ()) then
```

```
      write (u, "(3x,A)")  "polarized"
   else if (.not. prt%is_stable ()) then
      write (u, "(3x,A)", advance="no")  "decays:"
      call prt%get_decays (decay)
      do i = 1, size (decay)
         write (u, "(1x,A)", advance="no")  char (decay(i))
      end do
      write (u, *)
   else
      write (u, *)
   end if
   if (prt%has_antiparticle ()) then
      buffer = prt%get_name (.true.)
      write (u, "(4x,A,1x,I8)", advance="no")  buffer, &
           prt%get_pdg_anti ()
      if (prt%is_polarized (.true.)) then
         write (u, "(3x,A)")  "polarized"
      else if (.not. prt%is_stable (.true.)) then
         write (u, "(3x,A)", advance="no")  "decays:"
         call prt%get_decays (decay, .true.)
         do i = 1, size (decay)
            write (u, "(1x,A)", advance="no")  char (decay(i))
         end do
         write (u, *)
      else
         write (u, *)
      end if
   end if
end subroutine field_data_show
```

Retrieve data:

⟨*Model data: field data: TBP*⟩+≡

```
procedure :: get_pdg => field_data_get_pdg
procedure :: get_pdg_anti => field_data_get_pdg_anti
```

⟨*Model data: procedures*⟩+≡

```
elemental function field_data_get_pdg (prt) result (pdg)
  integer :: pdg
  class(field_data_t), intent(in) :: prt
  pdg = prt%pdg
end function field_data_get_pdg

elemental function field_data_get_pdg_anti (prt) result (pdg)
  integer :: pdg
  class(field_data_t), intent(in) :: prt
  if (prt%has_anti) then
     pdg = - prt%pdg
  else
     pdg = prt%pdg
  end if
end function field_data_get_pdg_anti
```

Predicates:

⟨*Model data: field data: TBP*⟩+≡

```
procedure :: is_visible => field_data_is_visible
procedure :: is_parton => field_data_is_parton
procedure :: is_gauge => field_data_is_gauge
procedure :: is_left_handed => field_data_is_left_handed
procedure :: is_right_handed => field_data_is_right_handed
procedure :: has_antiparticle => field_data_has_antiparticle
procedure :: is_stable => field_data_is_stable
procedure :: get_decays => field_data_get_decays
procedure :: decays_isotropically => field_data_decays_isotropically
procedure :: decays_diagonal => field_data_decays_diagonal
procedure :: has_decay_helicity => field_data_has_decay_helicity
procedure :: decay_helicity => field_data_decay_helicity
procedure :: is_polarized => field_data_is_polarized
```

⟨*Model data: procedures*⟩+≡

```
elemental function field_data_is_visible (prt) result (flag)
  logical :: flag
  class(field_data_t), intent(in) :: prt
  flag = prt%visible
end function field_data_is_visible

elemental function field_data_is_parton (prt) result (flag)
  logical :: flag
  class(field_data_t), intent(in) :: prt
  flag = prt%parton
end function field_data_is_parton

elemental function field_data_is_gauge (prt) result (flag)
  logical :: flag
  class(field_data_t), intent(in) :: prt
  flag = prt%gauge
end function field_data_is_gauge

elemental function field_data_is_left_handed (prt) result (flag)
  logical :: flag
  class(field_data_t), intent(in) :: prt
  flag = prt%left_handed
end function field_data_is_left_handed

elemental function field_data_is_right_handed (prt) result (flag)
  logical :: flag
  class(field_data_t), intent(in) :: prt
  flag = prt%right_handed
end function field_data_is_right_handed

elemental function field_data_has_antiparticle (prt) result (flag)
  logical :: flag
  class(field_data_t), intent(in) :: prt
  flag = prt%has_anti
end function field_data_has_antiparticle

elemental function field_data_is_stable (prt, anti) result (flag)
  logical :: flag
  class(field_data_t), intent(in) :: prt
  logical, intent(in), optional :: anti
```

```
  if (present (anti)) then
     if (anti) then
        flag = prt%a_is_stable
     else
        flag = prt%p_is_stable
     end if
  else
     flag = prt%p_is_stable
  end if
end function field_data_is_stable

subroutine field_data_get_decays (prt, decay, anti)
  class(field_data_t), intent(in) :: prt
  type(string_t), dimension(:), intent(out), allocatable :: decay
  logical, intent(in), optional :: anti
  if (present (anti)) then
     if (anti) then
        allocate (decay (size (prt%a_decay)), source = prt%a_decay)
     else
        allocate (decay (size (prt%p_decay)), source = prt%p_decay)
     end if
  else
     allocate (decay (size (prt%p_decay)), source = prt%p_decay)
  end if
end subroutine field_data_get_decays

elemental function field_data_decays_isotropically &
     (prt, anti) result (flag)
  logical :: flag
  class(field_data_t), intent(in) :: prt
  logical, intent(in), optional :: anti
  if (present (anti)) then
     if (anti) then
        flag = prt%a_decays_isotropically
     else
        flag = prt%p_decays_isotropically
     end if
  else
     flag = prt%p_decays_isotropically
  end if
end function field_data_decays_isotropically

elemental function field_data_decays_diagonal &
     (prt, anti) result (flag)
  logical :: flag
  class(field_data_t), intent(in) :: prt
  logical, intent(in), optional :: anti
  if (present (anti)) then
     if (anti) then
        flag = prt%a_decays_diagonal
     else
        flag = prt%p_decays_diagonal
     end if
  else
```

```
      flag = prt%p_decays_diagonal
   end if
end function field_data_decays_diagonal

elemental function field_data_has_decay_helicity &
      (prt, anti) result (flag)
   logical :: flag
   class(field_data_t), intent(in) :: prt
   logical, intent(in), optional :: anti
   if (present (anti)) then
      if (anti) then
         flag = prt%a_has_decay_helicity
      else
         flag = prt%p_has_decay_helicity
      end if
   else
      flag = prt%p_has_decay_helicity
   end if
end function field_data_has_decay_helicity

elemental function field_data_decay_helicity &
      (prt, anti) result (hel)
   integer :: hel
   class(field_data_t), intent(in) :: prt
   logical, intent(in), optional :: anti
   if (present (anti)) then
      if (anti) then
         hel = prt%a_decay_helicity
      else
         hel = prt%p_decay_helicity
      end if
   else
      hel = prt%p_decay_helicity
   end if
end function field_data_decay_helicity

elemental function field_data_is_polarized (prt, anti) result (flag)
   logical :: flag
   class(field_data_t), intent(in) :: prt
   logical, intent(in), optional :: anti
   logical :: a
   if (present (anti)) then
      a = anti
   else
      a = .false.
   end if
   if (a) then
      flag = prt%a_polarized
   else
      flag = prt%p_polarized
   end if
end function field_data_is_polarized
```

Names. Return the first name in the list (or the first antiparticle name)

⟨*Model data: field data: TBP*⟩+≡

```
procedure :: get_longname => field_data_get_longname
procedure :: get_name => field_data_get_name
procedure :: get_name_array => field_data_get_name_array
```

⟨*Model data: procedures*⟩+≡

```
pure function field_data_get_longname (prt) result (name)
  type(string_t) :: name
  class(field_data_t), intent(in) :: prt
  name = prt%longname
end function field_data_get_longname

pure function field_data_get_name (prt, is_antiparticle) result (name)
  type(string_t) :: name
  class(field_data_t), intent(in) :: prt
  logical, intent(in) :: is_antiparticle
  name = prt%longname
  if (is_antiparticle) then
     if (prt%has_anti) then
        if (allocated (prt%anti)) then
           if (size(prt%anti) > 0) name = prt%anti(1)
        end if
     else
        if (allocated (prt%name)) then
           if (size (prt%name) > 0) name = prt%name(1)
        end if
     end if
  else
     if (allocated (prt%name)) then
        if (size (prt%name) > 0) name = prt%name(1)
     end if
  end if
end function field_data_get_name

subroutine field_data_get_name_array (prt, is_antiparticle, name)
  class(field_data_t), intent(in) :: prt
  logical, intent(in) :: is_antiparticle
  type(string_t), dimension(:), allocatable, intent(inout) :: name
  if (allocated (name))  deallocate (name)
  if (is_antiparticle) then
     if (prt%has_anti) then
        allocate (name (size (prt%anti)))
        name = prt%anti
     else
        allocate (name (0))
     end if
  else
     allocate (name (size (prt%name)))
     name = prt%name
  end if
end subroutine field_data_get_name_array
```

Same for the TEX name.

⟨*Model data: field data: TBP*⟩+≡

```
procedure :: get_tex_name => field_data_get_tex_name
```

⟨*Model data: procedures*⟩+≡

```
elemental function field_data_get_tex_name &
     (prt, is_antiparticle) result (name)
  type(string_t) :: name
  class(field_data_t), intent(in) :: prt
  logical, intent(in) :: is_antiparticle
  if (is_antiparticle) then
     if (prt%has_anti) then
        name = prt%tex_anti
     else
        name = prt%tex_name
     end if
  else
     name = prt%tex_name
  end if
  if (name == "")  name = prt%get_name (is_antiparticle)
end function field_data_get_tex_name
```

Check if any of the field names matches the given string.

⟨*Model data: field data: TBP*⟩+≡

```
procedure, private :: matches_name => field_data_matches_name
```

⟨*Model data: procedures*⟩+≡

```
function field_data_matches_name (field, name, is_antiparticle) result (flag)
  class(field_data_t), intent(in) :: field
  type(string_t), intent(in) :: name
  logical, intent(in) :: is_antiparticle
  logical :: flag
  if (is_antiparticle) then
     if (field%has_anti) then
        flag = any (name == field%anti)
     else
        flag = .false.
     end if
  else
     flag = name == field%longname .or. any (name == field%name)
  end if
end function field_data_matches_name
```

Quantum numbers

⟨*Model data: field data: TBP*⟩+≡

```
procedure :: get_spin_type => field_data_get_spin_type
procedure :: get_multiplicity => field_data_get_multiplicity
procedure :: get_isospin_type => field_data_get_isospin_type
procedure :: get_charge_type => field_data_get_charge_type
procedure :: get_color_type => field_data_get_color_type
```

⟨*Model data: procedures*⟩+≡

```
elemental function field_data_get_spin_type (prt) result (type)
  integer :: type
  class(field_data_t), intent(in) :: prt
  type = prt%spin_type
```

```
end function field_data_get_spin_type

elemental function field_data_get_multiplicity (prt) result (type)
  integer :: type
  class(field_data_t), intent(in) :: prt
  type = prt%multiplicity
end function field_data_get_multiplicity

elemental function field_data_get_isospin_type (prt) result (type)
  integer :: type
  class(field_data_t), intent(in) :: prt
  type = prt%isospin_type
end function field_data_get_isospin_type

elemental function field_data_get_charge_type (prt) result (type)
  integer :: type
  class(field_data_t), intent(in) :: prt
  type = prt%charge_type
end function field_data_get_charge_type

elemental function field_data_get_color_type (prt) result (type)
  integer :: type
  class(field_data_t), intent(in) :: prt
  type = prt%color_type
end function field_data_get_color_type
```

In the MSSM, neutralinos can have a negative mass. This is relevant for computing matrix elements. However, within the WHIZARD main program we are interested only in kinematics, therefore we return the absolute value of the particle mass. If desired, we can extract the sign separately.

⟨*Model data: field data: TBP*⟩+≡

```
procedure :: get_charge => field_data_get_charge
procedure :: get_isospin => field_data_get_isospin
procedure :: get_mass => field_data_get_mass
procedure :: get_mass_sign => field_data_get_mass_sign
procedure :: get_width => field_data_get_width
```

⟨*Model data: procedures*⟩+≡

```
elemental function field_data_get_charge (prt) result (charge)
  real(default) :: charge
  class(field_data_t), intent(in) :: prt
  if (prt%charge_type /= 0) then
     charge = real (sign ((abs(prt%charge_type) - 1), &
           prt%charge_type), default) / 3
  else
     charge = 0
  end if
end function field_data_get_charge

elemental function field_data_get_isospin (prt) result (isospin)
  real(default) :: isospin
  class(field_data_t), intent(in) :: prt
  if (prt%isospin_type /= 0) then
     isospin = real (sign (abs(prt%isospin_type) - 1, &
```

```
          prt%isospin_type), default) / 2
  else
     isospin = 0
  end if
end function field_data_get_isospin

elemental function field_data_get_mass (prt) result (mass)
  real(default) :: mass
  class(field_data_t), intent(in) :: prt
  if (associated (prt%mass_val)) then
     mass = abs (prt%mass_val)
  else
     mass = 0
  end if
end function field_data_get_mass

elemental function field_data_get_mass_sign (prt) result (sgn)
  integer :: sgn
  class(field_data_t), intent(in) :: prt
  if (associated (prt%mass_val)) then
     sgn = sign (1._default, prt%mass_val)
  else
     sgn = 0
  end if
end function field_data_get_mass_sign

elemental function field_data_get_width (prt) result (width)
  real(default) :: width
  class(field_data_t), intent(in) :: prt
  if (associated (prt%width_val)) then
     width = prt%width_val
  else
     width = 0
  end if
end function field_data_get_width
```

Find the `model` containing the `PDG` given two model files.

⟨*Model data: public*⟩+≡

```
public :: find_model
```

⟨*Model data: procedures*⟩+≡

```
subroutine find_model (model, PDG, model_A, model_B)
  class(model_data_t), pointer, intent(out) :: model
  integer, intent(in) :: PDG
  class(model_data_t), intent(in), target :: model_A, model_B
  character(len=5) :: buffer
  if (model_A%test_field (PDG)) then
     model => model_A
  else if (model_B%test_field (PDG)) then
     model => model_B
  else
     write (buffer, "(I5)") PDG
     call msg_fatal ("Parton " // buffer // &
          " not found in the given model files")
```

```
    end if
  end subroutine find_model
```

10.1.3 Vertex data

The vertex object contains an array of particle-data pointers, for which we need a separate type. (We could use the flavor type defined in another module.)

The program does not (yet?) make use of vertex definitions, so they are not stored here.

⟨*Model data: types*⟩+≡

```
  type :: field_data_p
     type(field_data_t), pointer :: p => null ()
  end type field_data_p
```

⟨*Model data: types*⟩+≡

```
  type :: vertex_t
     private
     logical :: trilinear
     integer, dimension(:), allocatable :: pdg
     type(field_data_p), dimension(:), allocatable :: prt
   contains
   ⟨Model data: vertex: TBP⟩
  end type vertex_t
```

⟨*Model data: vertex: TBP*⟩≡

```
  procedure :: write => vertex_write
```

⟨*Model data: procedures*⟩+≡

```
  subroutine vertex_write (vtx, unit)
    class(vertex_t), intent(in) :: vtx
    integer, intent(in), optional :: unit
    integer :: u, i
    u = given_output_unit (unit)
    write (u, "(3x,A)", advance="no")  "vertex"
    do i = 1, size (vtx%prt)
       if (associated (vtx%prt(i)%p)) then
          write (u, "(1x,A)", advance="no") &
               '"' // char (vtx%prt(i)%p%get_name (vtx%pdg(i) < 0)) &
                   // '"'
       else
          write (u, "(1x,I7)", advance="no") vtx%pdg(i)
       end if
    end do
    write (u, *)
  end subroutine vertex_write
```

Initialize using PDG codes. The model is used for finding particle data pointers associated with the pdg codes.

⟨*Model data: vertex: TBP*⟩+≡

```
  procedure :: init => vertex_init
```

⟨*Model data: procedures*⟩+≡

```
subroutine vertex_init (vtx, pdg, model)
  class(vertex_t), intent(out) :: vtx
  integer, dimension(:), intent(in) :: pdg
  type(model_data_t), intent(in), target, optional :: model
  integer :: i
  allocate (vtx%pdg (size (pdg)))
  allocate (vtx%prt (size (pdg)))
  vtx%trilinear = size (pdg) == 3
  vtx%pdg = pdg
  if (present (model)) then
     do i = 1, size (pdg)
        vtx%prt(i)%p => model%get_field_ptr (pdg(i))
     end do
  end if
end subroutine vertex_init
```

Copy vertex: we must reassign the field-data pointer to a new model.

⟨*Model data: vertex: TBP*⟩+≡

```
procedure :: copy_from => vertex_copy_from
```

⟨*Model data: procedures*⟩+≡

```
subroutine vertex_copy_from (vtx, old_vtx, new_model)
  class(vertex_t), intent(out) :: vtx
  class(vertex_t), intent(in) :: old_vtx
  type(model_data_t), intent(in), target, optional :: new_model
  call vtx%init (old_vtx%pdg, new_model)
end subroutine vertex_copy_from
```

Single-particle lookup: Given a particle code, we return matching codes if present, otherwise zero. Actually, we return the antiparticles of the matching codes, as appropriate for computing splittings.

⟨*Model data: vertex: TBP*⟩+≡

```
procedure :: get_match => vertex_get_match
```

⟨*Model data: procedures*⟩+≡

```
subroutine vertex_get_match (vtx, pdg1, pdg2)
  class(vertex_t), intent(in) :: vtx
  integer, intent(in) :: pdg1
  integer, dimension(:), allocatable, intent(out) :: pdg2
  integer :: i, j
  do i = 1, size (vtx%pdg)
     if (vtx%pdg(i) == pdg1) then
        allocate (pdg2 (size (vtx%pdg) - 1))
        do j = 1, i-1
           pdg2(j) = anti (j)
        end do
        do j = i, size (pdg2)
           pdg2(j) = anti (j+1)
        end do
        exit
     end if
  end do
contains
```

```
  function anti (i) result (pdg)
    integer, intent(in) :: i
    integer :: pdg
    if (vtx%prt(i)%p%has_antiparticle ()) then
       pdg = - vtx%pdg(i)
    else
       pdg = vtx%pdg(i)
    end if
  end function anti
end subroutine vertex_get_match
```

To access this from the outside, we create an iterator. The iterator has the sole purpose of returning the matching particles for a given array of PDG codes.

⟨*Model data: public*⟩+≡

```
public :: vertex_iterator_t
```

⟨*Model data: types*⟩+≡

```
type :: vertex_iterator_t
   private
   class(model_data_t), pointer :: model => null ()
   integer, dimension(:), allocatable :: pdg
   integer :: vertex_index = 0
   integer :: pdg_index = 0
   logical :: save_pdg_index
 contains
   procedure :: init => vertex_iterator_init
   procedure :: get_next_match => vertex_iterator_get_next_match
end type vertex_iterator_t
```

We initialize the iterator for a particular model with the `pdg` index of the particle we are looking at.

⟨*Model data: procedures*⟩+≡

```
subroutine vertex_iterator_init (it, model, pdg, save_pdg_index)
  class(vertex_iterator_t), intent(out) :: it
  class(model_data_t), intent(in), target :: model
  integer, dimension(:), intent(in) :: pdg
  logical, intent(in) :: save_pdg_index
  it%model => model
  allocate (it%pdg (size (pdg)), source = pdg)
  it%save_pdg_index = save_pdg_index
end subroutine vertex_iterator_init

subroutine vertex_iterator_get_next_match (it, pdg_match)
  class(vertex_iterator_t), intent(inout) :: it
  integer, dimension(:), allocatable, intent(out) :: pdg_match
  integer :: i, j
  do i = it%vertex_index + 1, size (it%model%vtx)
     do j = it%pdg_index + 1, size (it%pdg)
        call it%model%vtx(i)%get_match (it%pdg(j), pdg_match)
        if (it%save_pdg_index) then
           if (allocated (pdg_match) .and. j < size (it%pdg)) then
              it%pdg_index = j
              return
```

```
            else if (allocated (pdg_match) .and. j == size (it%pdg)) then
               it%vertex_index = i
               it%pdg_index = 0
               return
            end if
         else if (allocated (pdg_match)) then
            it%vertex_index = i
            return
         end if
      end do
   end do
   it%vertex_index = 0
   it%pdg_index = 0
end subroutine vertex_iterator_get_next_match
```

10.1.4 Vertex lookup table

The vertex lookup table is a hash table: given two particle codes, we check which codes are allowed for the third one.

The size of the hash table should be large enough that collisions are rare. We first select a size based on the number of vertices (multiplied by six because all permutations count), with some margin, and then choose the smallest integer power of two larger than this.

⟨*Model data: parameters*⟩≡

```
integer, parameter :: VERTEX_TABLE_SCALE_FACTOR = 60
```

⟨*Model data: procedures*⟩+≡

```
function vertex_table_size (n_vtx) result (n)
  integer(i32) :: n
  integer, intent(in) :: n_vtx
  integer :: i, s
  s = VERTEX_TABLE_SCALE_FACTOR * n_vtx
  n = 1
  do i = 1, 31
     n = ishft (n, 1)
     s = ishft (s,-1)
     if (s == 0)  exit
  end do
end function vertex_table_size
```

The specific hash function takes two particle codes (arbitrary integers) and returns a 32-bit integer. It makes use of the universal function `hash` which operates on a byte array.

⟨*Model data: procedures*⟩+≡

```
function hash2 (pdg1, pdg2)
  integer(i32) :: hash2
  integer, intent(in) :: pdg1, pdg2
  integer(i8), dimension(1) :: mold
  hash2 = hash (transfer ([pdg1, pdg2], mold))
end function hash2
```

Each entry in the vertex table stores the two particle codes and an array of possibilities for the third code.

⟨*Model data: types*⟩+≡

```
type :: vertex_table_entry_t
   private
   integer :: pdg1 = 0, pdg2 = 0
   integer :: n = 0
   integer, dimension(:), allocatable :: pdg3
end type vertex_table_entry_t
```

The vertex table:

⟨*Model data: types*⟩+≡

```
type :: vertex_table_t
   type(vertex_table_entry_t), dimension(:), allocatable :: entry
   integer :: n_collisions = 0
   integer(i32) :: mask
 contains
 ⟨Model data: vertex table: TBP⟩
end type vertex_table_t
```

Output.

⟨*Model data: vertex table: TBP*⟩≡

```
procedure :: write => vertex_table_write
```

⟨*Model data: procedures*⟩+≡

```
subroutine vertex_table_write (vt, unit)
  class(vertex_table_t), intent(in) :: vt
  integer, intent(in), optional :: unit
  integer :: u, i
  character(9) :: size_pdg3
  u = given_output_unit (unit)
  write (u, "(A)") "vertex hash table:"
  write (u, "(A,I7)") "  size = ", size (vt%entry)
  write (u, "(A,I7)") "  used = ", count (vt%entry%n /= 0)
  write (u, "(A,I7)") "  coll = ", vt%n_collisions
  do i = lbound (vt%entry, 1), ubound (vt%entry, 1)
     if (vt%entry(i)%n /= 0) then
        write (size_pdg3, "(I7)") size (vt%entry(i)%pdg3)
        write (u, "(A,1x,I7,1x,A,2(1x,I7),A," // &
             size_pdg3 // "(1x,I7))")  &
             "  ", i, ":", vt%entry(i)%pdg1, &
             vt%entry(i)%pdg2, "->", vt%entry(i)%pdg3
     end if
  end do
end subroutine vertex_table_write
```

Initializing the vertex table: This is done in two passes. First, we scan all permutations for all vertices and count the number of entries in each bucket of the hashtable. Then, the buckets are allocated accordingly and filled.

Collision resolution is done by just incrementing the hash value until an empty bucket is found. The vertex table size is fixed, since we know from the beginning the number of entries.

⟨*Model data: vertex table: TBP*⟩+≡

```
procedure :: init => vertex_table_init
```

⟨*Model data: procedures*⟩+≡

```
subroutine vertex_table_init (vt, prt, vtx)
  class(vertex_table_t), intent(out) :: vt
  type(field_data_t), dimension(:), intent(in) :: prt
  type(vertex_t), dimension(:), intent(in) :: vtx
  integer :: n_vtx, vt_size, i, p1, p2, p3
  integer, dimension(3) :: p
  n_vtx = size (vtx)
  vt_size = vertex_table_size (count (vtx%trilinear))
  vt%mask = vt_size - 1
  allocate (vt%entry (0:vt_size-1))
  do i = 1, n_vtx
     if (vtx(i)%trilinear) then
        p = vtx(i)%pdg
        p1 = p(1);  p2 = p(2)
        call create (hash2 (p1, p2))
        if (p(2) /= p(3)) then
           p2 = p(3)
           call create (hash2 (p1, p2))
        end if
        if (p(1) /= p(2)) then
           p1 = p(2);  p2 = p(1)
           call create (hash2 (p1, p2))
           if (p(1) /= p(3)) then
              p2 = p(3)
              call create (hash2 (p1, p2))
           end if
        end if
        if (p(1) /= p(3)) then
           p1 = p(3);  p2 = p(1)
           call create (hash2 (p1, p2))
           if (p(1) /= p(2)) then
              p2 = p(2)
              call create (hash2 (p1, p2))
           end if
        end if
     end if
  end do
  do i = 0, vt_size - 1
     allocate (vt%entry(i)%pdg3 (vt%entry(i)%n))
  end do
  vt%entry%n = 0
  do i = 1, n_vtx
     if (vtx(i)%trilinear) then
        p = vtx(i)%pdg
        p1 = p(1);  p2 = p(2);  p3 = p(3)
        call register (hash2 (p1, p2))
        if (p(2) /= p(3)) then
           p2 = p(3);  p3 = p(2)
           call register (hash2 (p1, p2))
        end if
        if (p(1) /= p(2)) then
           p1 = p(2);  p2 = p(1);  p3 = p(3)
```

```
            call register (hash2 (p1, p2))
            if (p(1) /= p(3)) then
               p2 = p(3);  p3 = p(1)
               call register (hash2 (p1, p2))
            end if
         end if
         if (p(1) /= p(3)) then
            p1 = p(3);  p2 = p(1);  p3 = p(2)
            call register (hash2 (p1, p2))
            if (p(1) /= p(2)) then
               p2 = p(2);  p3 = p(1)
               call register (hash2 (p1, p2))
            end if
         end if
      end if
   end do
contains
   recursive subroutine create (hashval)
      integer(i32), intent(in) :: hashval
      integer :: h
      h = iand (hashval, vt%mask)
      if (vt%entry(h)%n == 0) then
         vt%entry(h)%pdg1 = p1
         vt%entry(h)%pdg2 = p2
         vt%entry(h)%n = 1
      else if (vt%entry(h)%pdg1 == p1 .and. vt%entry(h)%pdg2 == p2) then
         vt%entry(h)%n = vt%entry(h)%n + 1
      else
         vt%n_collisions = vt%n_collisions + 1
         call create (hashval + 1)
      end if
   end subroutine create
   recursive subroutine register (hashval)
      integer(i32), intent(in) :: hashval
      integer :: h
      h = iand (hashval, vt%mask)
      if (vt%entry(h)%pdg1 == p1 .and. vt%entry(h)%pdg2 == p2) then
         vt%entry(h)%n = vt%entry(h)%n + 1
         vt%entry(h)%pdg3(vt%entry(h)%n) = p3
      else
         call register (hashval + 1)
      end if
   end subroutine register
end subroutine vertex_table_init
```

Return the array of particle codes that match the given pair.

⟨*Model data: vertex table: TBP*⟩+≡

```
procedure :: match => vertex_table_match
```

⟨*Model data: procedures*⟩+≡

```
subroutine vertex_table_match (vt, pdg1, pdg2, pdg3)
   class(vertex_table_t), intent(in) :: vt
   integer, intent(in) :: pdg1, pdg2
   integer, dimension(:), allocatable, intent(out) :: pdg3
```

```
    call match (hash2 (pdg1, pdg2))
  contains
    recursive subroutine match (hashval)
      integer(i32), intent(in) :: hashval
      integer :: h
      h = iand (hashval, vt%mask)
      if (vt%entry(h)%n == 0) then
         allocate (pdg3 (0))
      else if (vt%entry(h)%pdg1 == pdg1 .and. vt%entry(h)%pdg2 == pdg2) then
         allocate (pdg3 (size (vt%entry(h)%pdg3)))
         pdg3 = vt%entry(h)%pdg3
      else
         call match (hashval + 1)
      end if
    end subroutine match
  end subroutine vertex_table_match
```

Return true if the triplet is represented as a vertex.

⟨*Model data: vertex table: TBP*⟩+≡

```
  procedure :: check => vertex_table_check
```

⟨*Model data: procedures*⟩+≡

```
  function vertex_table_check (vt, pdg1, pdg2, pdg3) result (flag)
    class(vertex_table_t), intent(in) :: vt
    integer, intent(in) :: pdg1, pdg2, pdg3
    logical :: flag
    flag = check (hash2 (pdg1, pdg2))
  contains
    recursive function check (hashval) result (flag)
      integer(i32), intent(in) :: hashval
      integer :: h
      logical :: flag
      h = iand (hashval, vt%mask)
      if (vt%entry(h)%n == 0) then
         flag = .false.
      else if (vt%entry(h)%pdg1 == pdg1 .and. vt%entry(h)%pdg2 == pdg2) then
         flag = any (vt%entry(h)%pdg3 == pdg3)
      else
         flag = check (hashval + 1)
      end if
    end function check
  end function vertex_table_check
```

10.1.5 Model Data Record

This type collects the model data as defined above.

We deliberately implement the parameter arrays as pointer arrays. We thus avoid keeping track of TARGET attributes.

The `scheme` identifier provides meta information. It doesn't give the client code an extra parameter, but it tells something about the interpretation of the parameters. If the scheme ID is left as default (zero), it is ignored.

⟨*Model data: public*⟩+≡

```
public :: model_data_t
```

⟨*Model data: types*⟩+≡

```
type :: model_data_t
   private
   type(string_t) :: name
   integer :: scheme = 0
   type(modelpar_real_t), dimension(:), pointer :: par_real => null ()
   type(modelpar_complex_t), dimension(:), pointer :: par_complex => null ()
   type(field_data_t), dimension(:), allocatable :: field
   type(vertex_t), dimension(:), allocatable :: vtx
   type(vertex_table_t) :: vt
 contains
 ⟨Model data: model data: TBP⟩
end type model_data_t
```

Finalizer, deallocate pointer arrays.

⟨*Model data: model data: TBP*⟩≡

```
procedure :: final => model_data_final
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_final (model)
  class(model_data_t), intent(inout) :: model
  deallocate (model%par_real)
  deallocate (model%par_complex)
end subroutine model_data_final
```

Output. The signature matches the signature of the high-level `model_write` procedure, so some of the options don't actually apply.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: write => model_data_write
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_write (model, unit, verbose, &
     show_md5sum, show_variables, show_parameters, &
     show_particles, show_vertices, show_scheme)
  class(model_data_t), intent(in) :: model
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  logical, intent(in), optional :: show_md5sum
  logical, intent(in), optional :: show_variables
  logical, intent(in), optional :: show_parameters
  logical, intent(in), optional :: show_particles
  logical, intent(in), optional :: show_vertices
  logical, intent(in), optional :: show_scheme
  logical :: show_sch, show_par, show_prt, show_vtx
  integer :: u, i
  u = given_output_unit (unit)
  show_sch = .false.;  if (present (show_scheme)) &
       show_sch = show_scheme
  show_par = .true.;  if (present (show_parameters)) &
       show_par = show_parameters
  show_prt = .true.;  if (present (show_particles)) &
       show_prt = show_particles
```

```
    show_vtx = .true.;  if (present (show_vertices)) &
         show_vtx = show_vertices
    if (show_sch) then
       write (u, "(3x,A,1X,I0)")  "scheme =", model%scheme
    end if
    if (show_par) then
       do i = 1, size (model%par_real)
          call model%par_real(i)%write (u)
          write (u, "(A)")
       end do
       do i = 1, size (model%par_complex)
          call model%par_complex(i)%write (u)
          write (u, "(A)")
       end do
    end if
    if (show_prt) then
       write (u, "(A)")
       call model%write_fields (u)
    end if
    if (show_vtx) then
       write (u, "(A)")
       call model%write_vertices (u, verbose)
    end if
  end subroutine model_data_write
```

Initialize, allocating pointer arrays. The second version makes a deep copy.

⟨*Model data: model data: TBP*⟩+≡

```
  generic :: init => model_data_init
  procedure, private :: model_data_init
```

⟨*Model data: procedures*⟩+≡

```
  subroutine model_data_init (model, name, &
       n_par_real, n_par_complex, n_field, n_vtx)
    class(model_data_t), intent(out) :: model
    type(string_t), intent(in) :: name
    integer, intent(in) :: n_par_real, n_par_complex
    integer, intent(in) :: n_field
    integer, intent(in) :: n_vtx
    model%name = name
    allocate (model%par_real (n_par_real))
    allocate (model%par_complex (n_par_complex))
    allocate (model%field (n_field))
    allocate (model%vtx (n_vtx))
  end subroutine model_data_init
```

Set the scheme ID.

⟨*Model data: model data: TBP*⟩+≡

```
  procedure :: set_scheme_num => model_data_set_scheme_num
```

⟨*Model data: procedures*⟩+≡

```
  subroutine model_data_set_scheme_num (model, scheme)
    class(model_data_t), intent(inout) :: model
    integer, intent(in) :: scheme
    model%scheme = scheme
```

```
end subroutine model_data_set_scheme_num
```

Complete model data initialization.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: freeze_fields => model_data_freeze_fields
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_freeze_fields (model)
  class(model_data_t), intent(inout) :: model
  call model%field%freeze ()
end subroutine model_data_freeze_fields
```

Deep copy. The new model should already be initialized, so we do not allocate memory.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: copy_from => model_data_copy
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_copy (model, src)
  class(model_data_t), intent(inout), target :: model
  class(model_data_t), intent(in), target :: src
  class(modelpar_data_t), pointer :: data, src_data
  integer :: i
  model%scheme = src%scheme
  model%par_real = src%par_real
  model%par_complex = src%par_complex
  do i = 1, size (src%field)
     associate (field => model%field(i), src_field => src%field(i))
       call field%init (src_field%get_longname (), src_field%get_pdg ())
       call field%copy_from (src_field)
       src_data => src_field%mass_data
       if (associated (src_data)) then
          data => model%get_par_data_ptr (src_data%get_name ())
          call field%set (mass_data = data)
       end if
       src_data => src_field%width_data
       if (associated (src_data)) then
          data => model%get_par_data_ptr (src_data%get_name ())
          call field%set (width_data = data)
       end if
       call field%set_multiplicity ()
     end associate
  end do
  do i = 1, size (src%vtx)
     call model%vtx(i)%copy_from (src%vtx(i), model)
  end do
  call model%freeze_vertices ()
end subroutine model_data_copy
```

Return the model name and numeric scheme.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: get_name => model_data_get_name
procedure :: get_scheme_num => model_data_get_scheme_num
```

⟨*Model data: procedures*⟩+≡

```
function model_data_get_name (model) result (name)
  class(model_data_t), intent(in) :: model
  type(string_t) :: name
  name = model%name
end function model_data_get_name

function model_data_get_scheme_num (model) result (scheme)
  class(model_data_t), intent(in) :: model
  integer :: scheme
  scheme = model%scheme
end function model_data_get_scheme_num
```

Retrieve a MD5 sum for the current model parameter values and decay/polarization settings. This is done by writing them to a temporary file, using a standard format. If the model scheme is nonzero, it is also written.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: get_parameters_md5sum => model_data_get_parameters_md5sum
```

⟨*Model data: procedures*⟩+≡

```
function model_data_get_parameters_md5sum (model) result (par_md5sum)
  character(32) :: par_md5sum
  class(model_data_t), intent(in) :: model
  real(default), dimension(:), allocatable :: par
  type(field_data_t), pointer :: field
  integer :: unit, i
  allocate (par (model%get_n_real ()))
  call model%real_parameters_to_array (par)
  unit = free_unit ()
  open (unit, status="scratch", action="readwrite")
  if (model%scheme /= 0)  write (unit, "(I0)")  model%scheme
  write (unit, "(" // FMT_19 // ")")  par
  do i = 1, model%get_n_field ()
     field => model%get_field_ptr_by_index (i)
     if (.not. field%is_stable (.false.) .or. .not. field%is_stable (.true.) &
          .or. field%is_polarized (.false.) .or. field%is_polarized (.true.))&
          then
        write (unit, "(3x,A)") char (field%get_longname ())
        call field%write_decays (unit)
     end if
  end do
  rewind (unit)
  par_md5sum = md5sum (unit)
  close (unit)
end function model_data_get_parameters_md5sum
```

Return the MD5 sum. This is a placeholder, to be overwritten for the complete model definition.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: get_md5sum => model_data_get_md5sum
```

⟨*Model data: procedures*⟩+≡

```
function model_data_get_md5sum (model) result (md5sum)
  class(model_data_t), intent(in) :: model
```

```
    character(32) :: md5sum
    md5sum = model%get_parameters_md5sum ()
  end function model_data_get_md5sum
```

Initialize a real or complex parameter.

⟨*Model data: model data: TBP*⟩+≡

```
  generic :: init_par => model_data_init_par_real, model_data_init_par_complex
  procedure, private :: model_data_init_par_real
  procedure, private :: model_data_init_par_complex
```

⟨*Model data: procedures*⟩+≡

```
  subroutine model_data_init_par_real (model, i, name, value)
    class(model_data_t), intent(inout) :: model
    integer, intent(in) :: i
    type(string_t), intent(in) :: name
    real(default), intent(in) :: value
    call model%par_real(i)%init (name, value)
  end subroutine model_data_init_par_real

  subroutine model_data_init_par_complex (model, i, name, value)
    class(model_data_t), intent(inout) :: model
    integer, intent(in) :: i
    type(string_t), intent(in) :: name
    complex(default), intent(in) :: value
    call model%par_complex(i)%init (name, value)
  end subroutine model_data_init_par_complex
```

After initialization, return size of parameter array.

⟨*Model data: model data: TBP*⟩+≡

```
  procedure :: get_n_real => model_data_get_n_real
  procedure :: get_n_complex => model_data_get_n_complex
```

⟨*Model data: procedures*⟩+≡

```
  function model_data_get_n_real (model) result (n)
    class(model_data_t), intent(in) :: model
    integer :: n
    n = size (model%par_real)
  end function model_data_get_n_real

  function model_data_get_n_complex (model) result (n)
    class(model_data_t), intent(in) :: model
    integer :: n
    n = size (model%par_complex)
  end function model_data_get_n_complex
```

After initialization, extract the whole parameter array.

⟨*Model data: model data: TBP*⟩+≡

```
  procedure :: real_parameters_to_array &
       => model_data_real_par_to_array
  procedure :: complex_parameters_to_array &
       => model_data_complex_par_to_array
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_real_par_to_array (model, array)
  class(model_data_t), intent(in) :: model
  real(default), dimension(:), intent(inout) :: array
  array = model%par_real%get_real ()
end subroutine model_data_real_par_to_array

subroutine model_data_complex_par_to_array (model, array)
  class(model_data_t), intent(in) :: model
  complex(default), dimension(:), intent(inout) :: array
  array = model%par_complex%get_complex ()
end subroutine model_data_complex_par_to_array
```

After initialization, set the whole parameter array.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: real_parameters_from_array &
     => model_data_real_par_from_array
procedure :: complex_parameters_from_array &
     => model_data_complex_par_from_array
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_real_par_from_array (model, array)
  class(model_data_t), intent(inout) :: model
  real(default), dimension(:), intent(in) :: array
  model%par_real = array
end subroutine model_data_real_par_from_array

subroutine model_data_complex_par_from_array (model, array)
  class(model_data_t), intent(inout) :: model
  complex(default), dimension(:), intent(in) :: array
  model%par_complex = array
end subroutine model_data_complex_par_from_array
```

Analogous, for a C parameter array.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: real_parameters_to_c_array &
     => model_data_real_par_to_c_array
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_real_par_to_c_array (model, array)
  class(model_data_t), intent(in) :: model
  real(c_default_float), dimension(:), intent(inout) :: array
  array = model%par_real%get_real ()
end subroutine model_data_real_par_to_c_array
```

After initialization, set the whole parameter array.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: real_parameters_from_c_array &
     => model_data_real_par_from_c_array
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_real_par_from_c_array (model, array)
  class(model_data_t), intent(inout) :: model
  real(c_default_float), dimension(:), intent(in) :: array
```

```
    model%par_real = real (array, default)
  end subroutine model_data_real_par_from_c_array
```

After initialization, get pointer to a real or complex parameter, directly by index.

⟨*Model data: model data: TBP*⟩+≡

```
  procedure :: get_par_real_ptr => model_data_get_par_real_ptr_index
  procedure :: get_par_complex_ptr => model_data_get_par_complex_ptr_index
```

⟨*Model data: procedures*⟩+≡

```
  function model_data_get_par_real_ptr_index (model, i) result (ptr)
    class(model_data_t), intent(inout) :: model
    integer, intent(in) :: i
    class(modelpar_data_t), pointer :: ptr
    ptr => model%par_real(i)
  end function model_data_get_par_real_ptr_index

  function model_data_get_par_complex_ptr_index (model, i) result (ptr)
    class(model_data_t), intent(inout) :: model
    integer, intent(in) :: i
    class(modelpar_data_t), pointer :: ptr
    ptr => model%par_complex(i)
  end function model_data_get_par_complex_ptr_index
```

After initialization, get pointer to a parameter by name.

⟨*Model data: model data: TBP*⟩+≡

```
  procedure :: get_par_data_ptr => model_data_get_par_data_ptr_name
```

⟨*Model data: procedures*⟩+≡

```
  function model_data_get_par_data_ptr_name (model, name) result (ptr)
    class(model_data_t), intent(in) :: model
    type(string_t), intent(in) :: name
    class(modelpar_data_t), pointer :: ptr
    integer :: i
    do i = 1, size (model%par_real)
       if (model%par_real(i)%name == name) then
          ptr => model%par_real(i)
          return
       end if
    end do
    do i = 1, size (model%par_complex)
       if (model%par_complex(i)%name == name) then
          ptr => model%par_complex(i)
          return
       end if
    end do
    ptr => null ()
  end function model_data_get_par_data_ptr_name
```

Return the value by name. Again, type conversion is allowed.

⟨*Model data: model data: TBP*⟩+≡

```
  procedure :: get_real => model_data_get_par_real_value
  procedure :: get_complex => model_data_get_par_complex_value
```

⟨*Model data: procedures*⟩+≡

```
function model_data_get_par_real_value (model, name) result (value)
  class(model_data_t), intent(in) :: model
  type(string_t), intent(in) :: name
  class(modelpar_data_t), pointer :: par
  real(default) :: value
  par => model%get_par_data_ptr (name)
  value = par%get_real ()
end function model_data_get_par_real_value

function model_data_get_par_complex_value (model, name) result (value)
  class(model_data_t), intent(in) :: model
  type(string_t), intent(in) :: name
  class(modelpar_data_t), pointer :: par
  complex(default) :: value
  par => model%get_par_data_ptr (name)
  value = par%get_complex ()
end function model_data_get_par_complex_value
```

Modify a real or complex parameter.

⟨*Model data: model data: TBP*⟩+≡

```
generic :: set_par => model_data_set_par_real, model_data_set_par_complex
procedure, private :: model_data_set_par_real
procedure, private :: model_data_set_par_complex
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_set_par_real (model, name, value)
  class(model_data_t), intent(inout) :: model
  type(string_t), intent(in) :: name
  real(default), intent(in) :: value
  class(modelpar_data_t), pointer :: par
  par => model%get_par_data_ptr (name)
  par = value
end subroutine model_data_set_par_real

subroutine model_data_set_par_complex (model, name, value)
  class(model_data_t), intent(inout) :: model
  type(string_t), intent(in) :: name
  complex(default), intent(in) :: value
  class(modelpar_data_t), pointer :: par
  par => model%get_par_data_ptr (name)
  par = value
end subroutine model_data_set_par_complex
```

List all fields in the model.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: write_fields => model_data_write_fields
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_write_fields (model, unit)
  class(model_data_t), intent(in) :: model
  integer, intent(in), optional :: unit
  integer :: i
  do i = 1, size (model%field)
```

```
      call model%field(i)%write (unit)
    end do
  end subroutine model_data_write_fields
```

After initialization, return number of fields (particles):

⟨*Model data: model data: TBP*⟩+≡

```
  procedure :: get_n_field => model_data_get_n_field
```

⟨*Model data: procedures*⟩+≡

```
  function model_data_get_n_field (model) result (n)
    class(model_data_t), intent(in) :: model
    integer :: n
    n = size (model%field)
  end function model_data_get_n_field
```

Return the PDG code of a field. The field is identified by name or by index. If the field is not found, return zero.

⟨*Model data: model data: TBP*⟩+≡

```
  generic :: get_pdg => &
       model_data_get_field_pdg_index, &
       model_data_get_field_pdg_name
  procedure, private :: model_data_get_field_pdg_index
  procedure, private :: model_data_get_field_pdg_name
```

⟨*Model data: procedures*⟩+≡

```
  function model_data_get_field_pdg_index (model, i) result (pdg)
    class(model_data_t), intent(in) :: model
    integer, intent(in) :: i
    integer :: pdg
    pdg = model%field(i)%get_pdg ()
  end function model_data_get_field_pdg_index

  function model_data_get_field_pdg_name (model, name, check) result (pdg)
    class(model_data_t), intent(in) :: model
    type(string_t), intent(in) :: name
    logical, intent(in), optional :: check
    integer :: pdg
    integer :: i
    do i = 1, size (model%field)
       associate (field => model%field(i))
         if (field%matches_name (name, .false.)) then
            pdg = field%get_pdg ()
            return
         else if (field%matches_name (name, .true.)) then
            pdg = - field%get_pdg ()
            return
         end if
       end associate
    end do
    pdg = 0
    call model%field_error (check, name)
  end function model_data_get_field_pdg_name
```

Return an array of all PDG codes, including antiparticles. The antiparticle are sorted after all particles.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: get_all_pdg => model_data_get_all_pdg
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_get_all_pdg (model, pdg)
  class(model_data_t), intent(in) :: model
  integer, dimension(:), allocatable, intent(inout) :: pdg
  integer :: n0, n1, i, k
  n0 = size (model%field)
  n1 = n0 + count (model%field%has_antiparticle ())
  allocate (pdg (n1))
  pdg(1:n0) = model%field%get_pdg ()
  k = n0
  do i = 1, size (model%field)
     associate (field => model%field(i))
       if (field%has_antiparticle ()) then
          k = k + 1
          pdg(k) = - field%get_pdg ()
       end if
     end associate
  end do
end subroutine model_data_get_all_pdg
```

Return pointer to the field array.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: get_field_array_ptr => model_data_get_field_array_ptr
```

⟨*Model data: procedures*⟩+≡

```
function model_data_get_field_array_ptr (model) result (ptr)
  class(model_data_t), intent(in), target :: model
  type(field_data_t), dimension(:), pointer :: ptr
  ptr => model%field
end function model_data_get_field_array_ptr
```

Return pointer to a field. The identifier should be the unique long name, the PDG code, or the index.

We can issue an error message, if the `check` flag is set. We never return an error if the PDG code is zero, this yields just a null pointer.

⟨*Model data: model data: TBP*⟩+≡

```
generic :: get_field_ptr => &
     model_data_get_field_ptr_name, &
     model_data_get_field_ptr_pdg
procedure, private :: model_data_get_field_ptr_name
procedure, private :: model_data_get_field_ptr_pdg
procedure :: get_field_ptr_by_index => model_data_get_field_ptr_index
```

⟨*Model data: procedures*⟩+≡

```
function model_data_get_field_ptr_name (model, name, check) result (ptr)
  class(model_data_t), intent(in), target :: model
  type(string_t), intent(in) :: name
  logical, intent(in), optional :: check
  type(field_data_t), pointer :: ptr
```

```
  integer :: i
  do i = 1, size (model%field)
     if (model%field(i)%matches_name (name, .false.)) then
        ptr => model%field(i)
        return
     else if (model%field(i)%matches_name (name, .true.)) then
        ptr => model%field(i)
        return
     end if
  end do
  ptr => null ()
  call model%field_error (check, name)
end function model_data_get_field_ptr_name

function model_data_get_field_ptr_pdg (model, pdg, check) result (ptr)
  class(model_data_t), intent(in), target :: model
  integer, intent(in) :: pdg
  logical, intent(in), optional :: check
  type(field_data_t), pointer :: ptr
  integer :: i, pdg_abs
  if (pdg == 0) then
     ptr => null ()
     return
  end if
  pdg_abs = abs (pdg)
  do i = 1, size (model%field)
     if (model%field(i)%get_pdg () == pdg_abs) then
        ptr => model%field(i)
        return
     end if
  end do
  ptr => null ()
  call model%field_error (check, pdg=pdg)
end function model_data_get_field_ptr_pdg

function model_data_get_field_ptr_index (model, i) result (ptr)
  class(model_data_t), intent(in), target :: model
  integer, intent(in) :: i
  type(field_data_t), pointer :: ptr
  ptr => model%field(i)
end function model_data_get_field_ptr_index
```

Don't assign a pointer, just check.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: test_field => model_data_test_field_pdg
```

⟨*Model data: procedures*⟩+≡

```
function model_data_test_field_pdg (model, pdg, check) result (exist)
  class(model_data_t), intent(in), target :: model
  integer, intent(in) :: pdg
  logical, intent(in), optional :: check
  logical :: exist
  exist = associated (model%get_field_ptr (pdg, check))
end function model_data_test_field_pdg
```

Error message, if check is set.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: field_error => model_data_field_error
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_field_error (model, check, name, pdg)
  class(model_data_t), intent(in) :: model
  logical, intent(in), optional :: check
  type(string_t), intent(in), optional :: name
  integer, intent(in), optional :: pdg
  if (present (check)) then
     if (check) then
        if (present (name)) then
           write (msg_buffer, "(A,1x,A,1x,A,1x,A)") &
                "No particle with name", char (name), &
                "is contained in model", char (model%name)
        else if (present (pdg)) then
           write (msg_buffer, "(A,1x,I0,1x,A,1x,A)") &
                "No particle with PDG code", pdg, &
                "is contained in model", char (model%name)
        else
           write (msg_buffer, "(A,1x,A,1x,A)") &
                "Particle missing", &
                "in model", char (model%name)
        end if
        call msg_fatal ()
     end if
  end if
end subroutine model_data_field_error
```

Assign mass and width value, which are associated via pointer. Identify the particle via pdg.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: set_field_mass => model_data_set_field_mass_pdg
procedure :: set_field_width => model_data_set_field_width_pdg
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_set_field_mass_pdg (model, pdg, value)
  class(model_data_t), intent(inout) :: model
  integer, intent(in) :: pdg
  real(default), intent(in) :: value
  type(field_data_t), pointer :: field
  field => model%get_field_ptr (pdg, check = .true.)
  call field%set_mass (value)
end subroutine model_data_set_field_mass_pdg

subroutine model_data_set_field_width_pdg (model, pdg, value)
  class(model_data_t), intent(inout) :: model
  integer, intent(in) :: pdg
  real(default), intent(in) :: value
  type(field_data_t), pointer :: field
  field => model%get_field_ptr (pdg, check = .true.)
  call field%set_width (value)
```

```
end subroutine model_data_set_field_width_pdg
```

Mark a particle as unstable and provide a list of names for its decay processes. In contrast with the previous subroutine which is for internal use, we address the particle by its PDG code. If the index is negative, we address the antiparticle.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: set_unstable => model_data_set_unstable
procedure :: set_stable => model_data_set_stable
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_set_unstable &
     (model, pdg, decay, isotropic, diagonal, decay_helicity)
  class(model_data_t), intent(inout), target :: model
  integer, intent(in) :: pdg
  type(string_t), dimension(:), intent(in) :: decay
  logical, intent(in), optional :: isotropic, diagonal
  integer, intent(in), optional :: decay_helicity
  type(field_data_t), pointer :: field
  field => model%get_field_ptr (pdg)
  if (pdg > 0) then
     call field%set ( &
          p_is_stable = .false., p_decay = decay, &
          p_decays_isotropically = isotropic, &
          p_decays_diagonal = diagonal, &
          p_decay_helicity = decay_helicity)
  else
     call field%set ( &
          a_is_stable = .false., a_decay = decay, &
          a_decays_isotropically = isotropic, &
          a_decays_diagonal = diagonal, &
          a_decay_helicity = decay_helicity)
  end if
end subroutine model_data_set_unstable

subroutine model_data_set_stable (model, pdg)
  class(model_data_t), intent(inout), target :: model
  integer, intent(in) :: pdg
  type(field_data_t), pointer :: field
  field => model%get_field_ptr (pdg)
  if (pdg > 0) then
     call field%set (p_is_stable = .true.)
  else
     call field%set (a_is_stable = .true.)
  end if
end subroutine model_data_set_stable
```

Mark a particle as polarized.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: set_polarized => model_data_set_polarized
procedure :: set_unpolarized => model_data_set_unpolarized
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_set_polarized (model, pdg)
  class(model_data_t), intent(inout), target :: model
```

```
    integer, intent(in) :: pdg
    type(field_data_t), pointer :: field
    field => model%get_field_ptr (pdg)
    if (pdg > 0) then
       call field%set (p_polarized = .true.)
    else
       call field%set (a_polarized = .true.)
    end if
  end subroutine model_data_set_polarized

  subroutine model_data_set_unpolarized (model, pdg)
    class(model_data_t), intent(inout), target :: model
    integer, intent(in) :: pdg
    type(field_data_t), pointer :: field
    field => model%get_field_ptr (pdg)
    if (pdg > 0) then
       call field%set (p_polarized = .false.)
    else
       call field%set (a_polarized = .false.)
    end if
  end subroutine model_data_set_unpolarized
```

Revert all polarized (unstable) particles to unpolarized (stable) status, respectively.

⟨*Model data: model data: TBP*⟩+≡

```
  procedure :: clear_unstable => model_clear_unstable
  procedure :: clear_polarized => model_clear_polarized
```

⟨*Model data: procedures*⟩+≡

```
  subroutine model_clear_unstable (model)
    class(model_data_t), intent(inout), target :: model
    integer :: i
    type(field_data_t), pointer :: field
    do i = 1, model%get_n_field ()
       field => model%get_field_ptr_by_index (i)
       call field%set (p_is_stable = .true.)
       if (field%has_antiparticle ()) then
          call field%set (a_is_stable = .true.)
       end if
    end do
  end subroutine model_clear_unstable

  subroutine model_clear_polarized (model)
    class(model_data_t), intent(inout), target :: model
    integer :: i
    type(field_data_t), pointer :: field
    do i = 1, model%get_n_field ()
       field => model%get_field_ptr_by_index (i)
       call field%set (p_polarized = .false.)
       if (field%has_antiparticle ()) then
          call field%set (a_polarized = .false.)
       end if
    end do
  end subroutine model_clear_polarized
```

List all vertices, optionally also the hash table.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: write_vertices => model_data_write_vertices
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_write_vertices (model, unit, verbose)
  class(model_data_t), intent(in) :: model
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: i, u
  u = given_output_unit (unit)
  do i = 1, size (model%vtx)
     call vertex_write (model%vtx(i), unit)
  end do
  if (present (verbose)) then
     if (verbose) then
        write (u, *)
        call vertex_table_write (model%vt, unit)
     end if
  end if
end subroutine model_data_write_vertices
```

Vertex definition.

⟨*Model data: model data: TBP*⟩+≡

```
generic :: set_vertex => &
     model_data_set_vertex_pdg, model_data_set_vertex_names
procedure, private :: model_data_set_vertex_pdg
procedure, private :: model_data_set_vertex_names
```

⟨*Model data: procedures*⟩+≡

```
subroutine model_data_set_vertex_pdg (model, i, pdg)
  class(model_data_t), intent(inout), target :: model
  integer, intent(in) :: i
  integer, dimension(:), intent(in) :: pdg
  call vertex_init (model%vtx(i), pdg, model)
end subroutine model_data_set_vertex_pdg

subroutine model_data_set_vertex_names (model, i, name)
  class(model_data_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(string_t), dimension(:), intent(in) :: name
  integer, dimension(size(name)) :: pdg
  integer :: j
  do j = 1, size (name)
     pdg(j) = model%get_pdg (name(j))
  end do
  call model%set_vertex (i, pdg)
end subroutine model_data_set_vertex_names
```

Finalize vertex definition: set up the hash table.

⟨*Model data: model data: TBP*⟩+≡

```
procedure :: freeze_vertices => model_data_freeze_vertices
```

⟨*Model data: procedures*⟩+≡
```
subroutine model_data_freeze_vertices (model)
  class(model_data_t), intent(inout) :: model
  call model%vt%init (model%field, model%vtx)
end subroutine model_data_freeze_vertices
```

Number of vertices in model

⟨*Model data: model data: TBP*⟩+≡
```
procedure :: get_n_vtx => model_data_get_n_vtx
```

⟨*Model data: procedures*⟩+≡
```
function model_data_get_n_vtx (model) result (n)
  class(model_data_t), intent(in) :: model
  integer :: n
  n = size (model%vtx)
end function model_data_get_n_vtx
```

Lookup functions

⟨*Model data: model data: TBP*⟩+≡
```
procedure :: match_vertex => model_data_match_vertex
```

⟨*Model data: procedures*⟩+≡
```
subroutine model_data_match_vertex (model, pdg1, pdg2, pdg3)
  class(model_data_t), intent(in) :: model
  integer, intent(in) :: pdg1, pdg2
  integer, dimension(:), allocatable, intent(out) :: pdg3
  call model%vt%match (pdg1, pdg2, pdg3)
end subroutine model_data_match_vertex
```

⟨*Model data: model data: TBP*⟩+≡
```
procedure :: check_vertex => model_data_check_vertex
```

⟨*Model data: procedures*⟩+≡
```
function model_data_check_vertex (model, pdg1, pdg2, pdg3) result (flag)
  logical :: flag
  class(model_data_t), intent(in) :: model
  integer, intent(in) :: pdg1, pdg2, pdg3
  flag = model%vt%check (pdg1, pdg2, pdg3)
end function model_data_check_vertex
```

10.1.6 Toy Models

This is a stripped-down version of the (already trivial) model 'Test'.

⟨*Model data: model data: TBP*⟩+≡
```
procedure :: init_test => model_data_init_test
```

⟨*Model data: procedures*⟩+≡
```
subroutine model_data_init_test (model)
  class(model_data_t), intent(out) :: model
  type(field_data_t), pointer :: field
  integer, parameter :: n_real = 4
  integer, parameter :: n_field = 2
  integer, parameter :: n_vertex = 2
```

```
    integer :: i
    call model%init (var_str ("Test"), &
         n_real, 0, n_field, n_vertex)
    i = 0
    i = i + 1
    call model%init_par (i, var_str ("gy"), 1._default)
    i = i + 1
    call model%init_par (i, var_str ("ms"), 125._default)
    i = i + 1
    call model%init_par (i, var_str ("ff"), 1.5_default)
    i = i + 1
    call model%init_par (i, var_str ("mf"), 1.5_default * 125._default)
    i = 0
    i = i + 1
    field => model%get_field_ptr_by_index (i)
    call field%init (var_str ("SCALAR"), 25)
    call field%set (spin_type=1)
    call field%set (mass_data=model%get_par_real_ptr (2))
    call field%set (name = [var_str ("s")])
    i = i + 1
    field => model%get_field_ptr_by_index (i)
    call field%init (var_str ("FERMION"), 6)
    call field%set (spin_type=2)
    call field%set (mass_data=model%get_par_real_ptr (4))
    call field%set (name = [var_str ("f")], anti = [var_str ("fbar")])
    call model%freeze_fields ()
    i = 0
    i = i + 1
    call model%set_vertex (i, [var_str ("fbar"), var_str ("f"), var_str ("s")])
    i = i + 1
    call model%set_vertex (i, [var_str ("s"), var_str ("s"), var_str ("s")])
    call model%freeze_vertices ()
  end subroutine model_data_init_test
```

This procedure prepares a subset of QED for testing purposes.

⟨*Model data: model data: TBP*⟩+≡

```
  procedure :: init_qed_test => model_data_init_qed_test
```

⟨*Model data: procedures*⟩+≡

```
  subroutine model_data_init_qed_test (model)
    class(model_data_t), intent(out) :: model
    type(field_data_t), pointer :: field
    integer, parameter :: n_real = 1
    integer, parameter :: n_field = 2
    integer :: i
    call model%init (var_str ("QED_test"), &
         n_real, 0, n_field, 0)
    i = 0
    i = i + 1
    call model%init_par (i, var_str ("me"), 0.000510997_default)
    i = 0
    i = i + 1
    field => model%get_field_ptr_by_index (i)
    call field%init (var_str ("E_LEPTON"), 11)
```

```
    call field%set (spin_type=2, charge_type=-4)
    call field%set (mass_data=model%get_par_real_ptr (1))
    call field%set (name = [var_str ("e-")], anti = [var_str ("e+")])
    i = i + 1
    field => model%get_field_ptr_by_index (i)
    call field%init (var_str ("PHOTON"), 22)
    call field%set (spin_type=3)
    call field%set (name = [var_str ("A")])
    call model%freeze_fields ()
    call model%freeze_vertices ()
  end subroutine model_data_init_qed_test
```

This procedure prepares a subset of the Standard Model for testing purposes. We can thus avoid dependencies on model I/O, which is not defined here.

⟨*Model data: model data: TBP*⟩+≡

```
  procedure :: init_sm_test => model_data_init_sm_test
```

⟨*Model data: procedures*⟩+≡

```
  subroutine model_data_init_sm_test (model)
    class(model_data_t), intent(out) :: model
    type(field_data_t), pointer :: field
    integer, parameter :: n_real = 11
    integer, parameter :: n_field = 19
    integer, parameter :: n_vtx = 9
    integer :: i
    call model%init (var_str ("SM_test"), &
         n_real, 0, n_field, n_vtx)
    i = 0
    i = i + 1
    call model%init_par (i, var_str ("mZ"), 91.1882_default)
    i = i + 1
    call model%init_par (i, var_str ("mW"), 80.419_default)
    i = i + 1
    call model%init_par (i, var_str ("me"), 0.000510997_default)
    i = i + 1
    call model%init_par (i, var_str ("mmu"), 0.105658389_default)
    i = i + 1
    call model%init_par (i, var_str ("mb"), 4.2_default)
    i = i + 1
    call model%init_par (i, var_str ("mtop"), 173.1_default)
    i = i + 1
    call model%init_par (i, var_str ("wZ"), 2.443_default)
    i = i + 1
    call model%init_par (i, var_str ("wW"), 2.049_default)
    i = i + 1
    call model%init_par (i, var_str ("ee"), 0.3079561542961_default)
    i = i + 1
    call model%init_par (i, var_str ("cw"), 8.819013863636E-01_default)
    i = i + 1
    call model%init_par (i, var_str ("sw"), 4.714339240339E-01_default)
    i = 0
    i = i + 1
    field => model%get_field_ptr_by_index (i)
    call field%init (var_str ("D_QUARK"), 1)
```

```
call field%set (spin_type=2, color_type=3, charge_type=-2, isospin_type=-2)
call field%set (name = [var_str ("d")], anti = [var_str ("dbar")])
i = i + 1
field => model%get_field_ptr_by_index (i)
call field%init (var_str ("U_QUARK"), 2)
call field%set (spin_type=2, color_type=3, charge_type=3, isospin_type=2)
call field%set (name = [var_str ("u")], anti = [var_str ("ubar")])
i = i + 1
field => model%get_field_ptr_by_index (i)
call field%init (var_str ("S_QUARK"), 3)
call field%set (spin_type=2, color_type=3, charge_type=-2, isospin_type=-2)
call field%set (name = [var_str ("s")], anti = [var_str ("sbar")])
i = i + 1
field => model%get_field_ptr_by_index (i)
call field%init (var_str ("C_QUARK"), 4)
call field%set (spin_type=2, color_type=3, charge_type=3, isospin_type=2)
call field%set (name = [var_str ("c")], anti = [var_str ("cbar")])
i = i + 1
field => model%get_field_ptr_by_index (i)
call field%init (var_str ("B_QUARK"), 5)
call field%set (spin_type=2, color_type=3, charge_type=-2, isospin_type=-2)
call field%set (mass_data=model%get_par_real_ptr (5))
call field%set (name = [var_str ("b")], anti = [var_str ("bbar")])
i = i + 1
field => model%get_field_ptr_by_index (i)
call field%init (var_str ("T_QUARK"), 6)
call field%set (spin_type=2, color_type=3, charge_type=3, isospin_type=2)
call field%set (mass_data=model%get_par_real_ptr (6))
call field%set (name = [var_str ("t")], anti = [var_str ("tbar")])
i = i + 1
field => model%get_field_ptr_by_index (i)
call field%init (var_str ("E_LEPTON"), 11)
call field%set (spin_type=2)
call field%set (mass_data=model%get_par_real_ptr (3))
call field%set (name = [var_str ("e-")], anti = [var_str ("e+")])
i = i + 1
field => model%get_field_ptr_by_index (i)
call field%init (var_str ("E_NEUTRINO"), 12)
call field%set (spin_type=2, is_left_handed=.true.)
call field%set (name = [var_str ("nue")], anti = [var_str ("nuebar")])
i = i + 1
field => model%get_field_ptr_by_index (i)
call field%init (var_str ("MU_LEPTON"), 13)
call field%set (spin_type=2)
call field%set (mass_data=model%get_par_real_ptr (4))
call field%set (name = [var_str ("mu-")], anti = [var_str ("mu+")])
i = i + 1
field => model%get_field_ptr_by_index (i)
call field%init (var_str ("MU_NEUTRINO"), 14)
call field%set (spin_type=2, is_left_handed=.true.)
call field%set (name = [var_str ("numu")], anti = [var_str ("numubar")])
i = i + 1
field => model%get_field_ptr_by_index (i)
call field%init (var_str ("GLUON"), 21)
```

```
    call field%set (spin_type=3, color_type=8)
    call field%set (name = [var_str ("gl")])
    i = i + 1
    field => model%get_field_ptr_by_index (i)
    call field%init (var_str ("PHOTON"), 22)
    call field%set (spin_type=3)
    call field%set (name = [var_str ("A")])
    i = i + 1
    field => model%get_field_ptr_by_index (i)
    call field%init (var_str ("Z_BOSON"), 23)
    call field%set (spin_type=3)
    call field%set (mass_data=model%get_par_real_ptr (1))
    call field%set (width_data=model%get_par_real_ptr (7))
    call field%set (name = [var_str ("Z")])
    i = i + 1
    field => model%get_field_ptr_by_index (i)
    call field%init (var_str ("W_BOSON"), 24)
    call field%set (spin_type=3)
    call field%set (mass_data=model%get_par_real_ptr (2))
    call field%set (width_data=model%get_par_real_ptr (8))
    call field%set (name = [var_str ("W+")], anti = [var_str ("W-")])
    i = i + 1
    field => model%get_field_ptr_by_index (i)
    call field%init (var_str ("HIGGS"), 25)
    call field%set (spin_type=1)
!    call field%set (mass_data=model%get_par_real_ptr (2))
!    call field%set (width_data=model%get_par_real_ptr (8))
    call field%set (name = [var_str ("H")])
    i = i + 1
    field => model%get_field_ptr_by_index (i)
    call field%init (var_str ("PROTON"), 2212)
    call field%set (spin_type=2)
    call field%set (name = [var_str ("p")], anti = [var_str ("pbar")])
!    call field%set (mass_data=model%get_par_real_ptr (12))
    i = i + 1
    field => model%get_field_ptr_by_index (i)
    call field%init (var_str ("HADRON_REMNANT_SINGLET"), 91)
    call field%set (color_type=1)
    call field%set (name = [var_str ("hr1")])
    i = i + 1
    field => model%get_field_ptr_by_index (i)
    call field%init (var_str ("HADRON_REMNANT_TRIPLET"), 92)
    call field%set (color_type=3)
    call field%set (name = [var_str ("hr3")], anti = [var_str ("hr3bar")])
    i = i + 1
    field => model%get_field_ptr_by_index (i)
    call field%init (var_str ("HADRON_REMNANT_OCTET"), 93)
    call field%set (color_type=8)
    call field%set (name = [var_str ("hr8")])
    call model%freeze_fields ()
    i = 0
    i = i + 1
    call model%set_vertex (i, [var_str ("dbar"), var_str ("d"), var_str ("A")])
    i = i + 1
```

```
  call model%set_vertex (i, [var_str ("ubar"), var_str ("u"), var_str ("A")])
  i = i + 1
  call model%set_vertex (i, [var_str ("gl"), var_str ("gl"), var_str ("gl")])
  i = i + 1
  call model%set_vertex (i, [var_str ("dbar"), var_str ("d"), var_str ("gl")])
  i = i + 1
  call model%set_vertex (i, [var_str ("ubar"), var_str ("u"), var_str ("gl")])
  i = i + 1
  call model%set_vertex (i, [var_str ("dbar"), var_str ("d"), var_str ("Z")])
  i = i + 1
  call model%set_vertex (i, [var_str ("ubar"), var_str ("u"), var_str ("Z")])
  i = i + 1
  call model%set_vertex (i, [var_str ("ubar"), var_str ("d"), var_str ("W+")])
  i = i + 1
  call model%set_vertex (i, [var_str ("dbar"), var_str ("u"), var_str ("W-")])
  call model%freeze_vertices ()
end subroutine model_data_init_sm_test
```

10.2 Model Testbed

The standard way of defining a model uses concrete variables and expressions to interpret the model file. Some of this is not available at the point of use. This is no problem for the WHIZARD program as a whole, but unit tests are kept local to their respective module and don't access all definitions.

Instead, we introduce a separate module that provides hooks, one for initializing a model and one for finalizing a model. The main program can assign real routines to the hooks (procedure pointers of abstract type) before unit tests are called. The unit tests can call the abstract routines without knowing about their implementation.

⟨model_testbed.f90⟩≡

```
⟨File header⟩

module model_testbed

⟨Use strings⟩
  use model_data
  use var_base

⟨Standard module head⟩

⟨Model testbed: public⟩

⟨Model testbed: variables⟩

⟨Model testbed: interfaces⟩

end module model_testbed
```

10.2.1 Abstract Model Handlers

Both routines take a polymorphic model (data) pointer, which is allocated/deallocated inside the subroutine. The model constructor prepare_model requires the model name as input. It can, optionally, return a link to the variable list of the model.

⟨*Model testbed: public*⟩≡

```
public :: prepare_model
public :: cleanup_model
```

⟨*Model testbed: variables*⟩≡

```
procedure (prepare_model_proc), pointer :: prepare_model => null ()
procedure (cleanup_model_proc), pointer :: cleanup_model => null ()
```

⟨*Model testbed: interfaces*⟩≡

```
abstract interface
   subroutine prepare_model_proc (model, name, vars)
     import
     class(model_data_t), pointer, intent(inout) :: model
     type(string_t), intent(in) :: name
     class(vars_t), pointer, intent(out), optional :: vars
   end subroutine prepare_model_proc
end interface
```

```
abstract interface
   subroutine cleanup_model_proc (model)
     import
     class(model_data_t), pointer, intent(inout) :: model
   end subroutine cleanup_model_proc
end interface
```

10.3 Helicities

This module defines types and tools for dealing with helicity information.

⟨helicities.f90⟩≡

```
⟨File header⟩

module helicities

  use io_units

⟨Standard module head⟩

⟨Helicities: public⟩

⟨Helicities: types⟩

⟨Helicities: interfaces⟩

contains

⟨Helicities: procedures⟩

end module helicities
```

10.3.1 Helicity types

Helicities may be defined or undefined, corresponding to a polarized or unpolarized state. Each helicity is actually a pair of helicities, corresponding to an entry in the spin density matrix. Obviously, diagonal entries are distinguished.

⟨*Helicities: public*⟩≡

```
public :: helicity_t
```

⟨*Helicities: types*⟩≡

```
type :: helicity_t
   private
   logical :: defined = .false.
   integer :: h1, h2
 contains
 ⟨Helicities: helicity: TBP⟩
end type helicity_t
```

Constructor functions, for convenience:

⟨*Helicities: public*⟩+≡

```
public :: helicity
```

⟨*Helicities: interfaces*⟩≡

```
interface helicity
   module procedure helicity0, helicity1, helicity2
end interface helicity
```

⟨*Helicities: procedures*⟩≡

```
pure function helicity0 () result (hel)
  type(helicity_t) :: hel
end function helicity0

elemental function helicity1 (h) result (hel)
  type(helicity_t) :: hel
  integer, intent(in) :: h
  call hel%init (h)
end function helicity1

elemental function helicity2 (h2, h1) result (hel)
  type(helicity_t) :: hel
  integer, intent(in) :: h1, h2
  call hel%init (h2, h1)
end function helicity2
```

Initializers.

Note: conceptually, the argument to initializers should be INTENT(OUT). However, Interp. F08/0033 prohibited this. The reason is that, in principle, the call could result in the execution of an impure finalizer for a type extension of `hel` (ugh).

⟨*Helicities: helicity: TBP*⟩≡

```
generic :: init => helicity_init_empty, helicity_init_same, helicity_init_different
procedure, private :: helicity_init_empty
procedure, private :: helicity_init_same
procedure, private :: helicity_init_different
```

⟨*Helicities: procedures*⟩+≡

```
elemental subroutine helicity_init_empty (hel)
  class(helicity_t), intent(inout) :: hel
  hel%defined = .false.
end subroutine helicity_init_empty

elemental subroutine helicity_init_same (hel, h)
  class(helicity_t), intent(inout) :: hel
  integer, intent(in) :: h
  hel%defined = .true.
  hel%h1 = h
  hel%h2 = h
end subroutine helicity_init_same

elemental subroutine helicity_init_different (hel, h2, h1)
  class(helicity_t), intent(inout) :: hel
  integer, intent(in) :: h1, h2
  hel%defined = .true.
  hel%h2 = h2
  hel%h1 = h1
end subroutine helicity_init_different
```

Undefine:

⟨*Helicities: helicity: TBP*⟩+≡

```
procedure :: undefine => helicity_undefine
```

⟨*Helicities: procedures*⟩+≡

```
elemental subroutine helicity_undefine (hel)
  class(helicity_t), intent(inout) :: hel
  hel%defined = .false.
end subroutine helicity_undefine
```

Diagonalize by removing the second entry (use with care!)

⟨*Helicities: helicity: TBP*⟩+≡

```
procedure :: diagonalize => helicity_diagonalize
```

⟨*Helicities: procedures*⟩+≡

```
elemental subroutine helicity_diagonalize (hel)
  class(helicity_t), intent(inout) :: hel
  hel%h2 = hel%h1
end subroutine helicity_diagonalize
```

⟨*Helicities: helicity: TBP*⟩+≡

```
procedure :: get_indices => helicity_get_indices
```

⟨*Helicities: procedures*⟩+≡

```
subroutine helicity_get_indices (hel, h1, h2)
  class(helicity_t), intent(in) :: hel
  integer, intent(out) :: h1, h2
  h1 = hel%h1; h2 = hel%h2
 end subroutine helicity_get_indices
```

Output (no linebreak). No output if undefined.

⟨*Helicities: helicity: TBP*⟩+≡

```
procedure :: write => helicity_write
```

⟨*Helicities: procedures*⟩+≡

```
subroutine helicity_write (hel, unit)
  class(helicity_t), intent(in) :: hel
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  if (hel%defined) then
     write (u, "(A)", advance="no")  "h("
     write (u, "(I0)", advance="no")  hel%h1
     if (hel%h1 /= hel%h2) then
        write (u, "(A)", advance="no") "|"
        write (u, "(I0)", advance="no")  hel%h2
     end if
     write (u, "(A)", advance="no")  ")"
  end if
end subroutine helicity_write
```

Binary I/O. Write contents only if defined.

⟨*Helicities: helicity: TBP*⟩+≡

```
procedure :: write_raw => helicity_write_raw
procedure :: read_raw => helicity_read_raw
```

⟨*Helicities: procedures*⟩+≡

```
subroutine helicity_write_raw (hel, u)
  class(helicity_t), intent(in) :: hel
  integer, intent(in) :: u
  write (u) hel%defined
  if (hel%defined) then
     write (u) hel%h1, hel%h2
  end if
end subroutine helicity_write_raw

subroutine helicity_read_raw (hel, u, iostat)
  class(helicity_t), intent(out) :: hel
  integer, intent(in) :: u
  integer, intent(out), optional :: iostat
  read (u, iostat=iostat) hel%defined
  if (hel%defined) then
     read (u, iostat=iostat) hel%h1, hel%h2
  end if
end subroutine helicity_read_raw
```

10.3.2 Predicates

Check if the helicity is defined:

⟨*Helicities: helicity: TBP*⟩+≡

```
procedure :: is_defined => helicity_is_defined
```

⟨*Helicities: procedures*⟩+≡

```
elemental function helicity_is_defined (hel) result (defined)
  logical :: defined
  class(helicity_t), intent(in) :: hel
  defined = hel%defined
end function helicity_is_defined
```

Return true if the two helicities are equal or the particle is unpolarized:

⟨*Helicities: helicity: TBP*⟩+≡

```
procedure :: is_diagonal => helicity_is_diagonal
```

⟨*Helicities: procedures*⟩+≡

```
elemental function helicity_is_diagonal (hel) result (diagonal)
  logical :: diagonal
  class(helicity_t), intent(in) :: hel
  if (hel%defined) then
     diagonal = hel%h1 == hel%h2
  else
     diagonal = .true.
  end if
end function helicity_is_diagonal
```

10.3.3 Accessing contents

This returns a two-element array and thus cannot be elemental. The result is unpredictable if the helicity is undefined.

⟨*Helicities: helicity: TBP*⟩+≡

```
procedure :: to_pair => helicity_to_pair
```

⟨*Helicities: procedures*⟩+≡

```
pure function helicity_to_pair (hel) result (h)
  integer, dimension(2) :: h
  class(helicity_t), intent(in) :: hel
  h(1) = hel%h2
  h(2) = hel%h1
end function helicity_to_pair
```

10.3.4 Comparisons

When comparing helicities, if either one is undefined, they are considered to match. In other words, an unpolarized particle matches any polarization. In the `dmatch` variant, it matches only diagonal helicity.

⟨*Helicities: helicity: TBP*⟩+≡

```
generic :: operator(.match.) => helicity_match
generic :: operator(.dmatch.) => helicity_match_diagonal
generic :: operator(==) => helicity_eq
generic :: operator(/=) => helicity_neq
procedure, private ::  helicity_match
procedure, private ::  helicity_match_diagonal
procedure, private ::  helicity_eq
procedure, private ::  helicity_neq
```

⟨*Helicities: procedures*⟩+≡

```
elemental function helicity_match (hel1, hel2) result (eq)
  logical :: eq
  class(helicity_t), intent(in) :: hel1, hel2
  if (hel1%defined .and. hel2%defined) then
     eq = (hel1%h1 == hel2%h1) .and. (hel1%h2 == hel2%h2)
  else
     eq = .true.
  end if
end function helicity_match

elemental function helicity_match_diagonal (hel1, hel2) result (eq)
  logical :: eq
  class(helicity_t), intent(in) :: hel1, hel2
  if (hel1%defined .and. hel2%defined) then
     eq = (hel1%h1 == hel2%h1) .and. (hel1%h2 == hel2%h2)
  else if (hel1%defined) then
     eq = hel1%h1 == hel1%h2
  else if (hel2%defined) then
     eq = hel2%h1 == hel2%h2
  else
     eq = .true.
  end if
```

```
end function helicity_match_diagonal
```

⟨*Helicities: procedures*⟩+≡

```
elemental function helicity_eq (hel1, hel2) result (eq)
  logical :: eq
  class(helicity_t), intent(in) :: hel1, hel2
  if (hel1%defined .and. hel2%defined) then
     eq = (hel1%h1 == hel2%h1) .and. (hel1%h2 == hel2%h2)
  else if (.not. hel1%defined .and. .not. hel2%defined) then
     eq = .true.
  else
     eq = .false.
  end if
end function helicity_eq
```

⟨*Helicities: procedures*⟩+≡

```
elemental function helicity_neq (hel1, hel2) result (neq)
  logical :: neq
  class(helicity_t), intent(in) :: hel1, hel2
  if (hel1%defined .and. hel2%defined) then
     neq = (hel1%h1 /= hel2%h1) .or. (hel1%h2 /= hel2%h2)
  else if (.not. hel1%defined .and. .not. hel2%defined) then
     neq = .false.
  else
     neq = .true.
  end if
end function helicity_neq
```

10.3.5 Tools

Merge two helicity objects by taking the first entry from the first and the second entry from the second argument. Makes sense only if the input helicities were defined and diagonal. The handling of ghost flags is not well-defined; one should verify beforehand that they match.

⟨*Helicities: helicity: TBP*⟩+≡

```
generic :: operator(.merge.) => merge_helicities
procedure, private ::  merge_helicities
```

⟨*Helicities: procedures*⟩+≡

```
elemental function merge_helicities (hel1, hel2) result (hel)
  type(helicity_t) :: hel
  class(helicity_t), intent(in) :: hel1, hel2
  if (hel1%defined .and. hel2%defined) then
     call hel%init (hel2%h1, hel1%h1)
  else if (hel1%defined) then
     call hel%init (hel1%h2, hel1%h1)
  else if (hel2%defined) then
     call hel%init (hel2%h2, hel2%h1)
  end if
end function merge_helicities
```

10.4 Colors

This module defines a type and tools for dealing with color information.

Each particle can have zero or more (in practice, usually not more than two) color indices. Color indices are positive; flow direction can be determined from the particle nature.

While parton shower matrix elements are diagonal in color, some special applications (e.g., subtractions for NLO matrix elements) require non-diagonal color matrices.

⟨colors.f90⟩≡

```
⟨File header⟩

module colors

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use diagnostics

⟨Standard module head⟩

⟨Colors: public⟩

⟨Colors: types⟩

⟨Colors: interfaces⟩

contains

⟨Colors: procedures⟩

end module colors
```

10.4.1 The color type

A particle may have an arbitrary number of color indices (in practice, from zero to two, but more are possible). This object acts as a container. (The current implementation has a fixed array of length two.)

The fact that color comes as an array prohibits elemental procedures in some places. (May add interfaces and multi versions where necessary.)

The color may be undefined.

NOTE: Due to a compiler bug in nagfor 5.2, we do not use allocatable but fixed-size arrays with dimension 2. Only nonzero entries count. This may be more efficient anyway, but gives up some flexibility. However, the squaring algorithm currently works only for singlets, (anti)triplets and octets anyway, so two components are enough.

This type has to be generalized (abstract type and specific implementations) when trying to pursue generalized color flows or Monte Carlo over continuous color.

⟨Colors: public⟩≡

```
public :: color_t
```

⟨*Colors: types*⟩≡

```
type :: color_t
   private
   logical :: defined = .false.
   integer, dimension(2) :: c1 = 0, c2 = 0
   logical :: ghost = .false.
 contains
 ⟨Colors: color: TBP⟩
end type color_t
```

Initializers:

⟨*Colors: color: TBP*⟩≡

```
generic :: init => &
     color_init_trivial, color_init_trivial_ghost, &
     color_init_array, color_init_array_ghost, &
     color_init_arrays, color_init_arrays_ghost
procedure, private :: color_init_trivial
procedure, private :: color_init_trivial_ghost
procedure, private :: color_init_array
procedure, private :: color_init_array_ghost
procedure, private :: color_init_arrays
procedure, private :: color_init_arrays_ghost
```

Undefined color: array remains unallocated

⟨*Colors: procedures*⟩≡

```
pure subroutine color_init_trivial (col)
  class(color_t), intent(inout) :: col
  col%defined = .true.
  col%c1 = 0
  col%c2 = 0
  col%ghost = .false.
end subroutine color_init_trivial

pure subroutine color_init_trivial_ghost (col, ghost)
  class(color_t), intent(inout) :: col
  logical, intent(in) :: ghost
  col%defined = .true.
  col%c1 = 0
  col%c2 = 0
  col%ghost = ghost
end subroutine color_init_trivial_ghost
```

This defines color from an arbitrary length color array, suitable for any representation. We may have two color arrays (non-diagonal matrix elements). This cannot be elemental. The third version assigns an array of colors, using a two-dimensional array as input.

⟨*Colors: procedures*⟩+≡

```
pure subroutine color_init_array (col, c1)
  class(color_t), intent(inout) :: col
  integer, dimension(:), intent(in) :: c1
  col%defined = .true.
  col%c1 = pack (c1, c1 /= 0, [0,0])
  col%c2 = col%c1
```

```
  col%ghost = .false.
end subroutine color_init_array

pure subroutine color_init_array_ghost (col, c1, ghost)
  class(color_t), intent(inout) :: col
  integer, dimension(:), intent(in) :: c1
  logical, intent(in) :: ghost
  call color_init_array (col, c1)
  col%ghost = ghost
end subroutine color_init_array_ghost

pure subroutine color_init_arrays (col, c1, c2)
  class(color_t), intent(inout) :: col
  integer, dimension(:), intent(in) :: c1, c2
  col%defined = .true.
  if (size (c1) == size (c2)) then
     col%c1 = pack (c1, c1 /= 0, [0,0])
     col%c2 = pack (c2, c2 /= 0, [0,0])
  else if (size (c1) /= 0) then
     col%c1 = pack (c1, c1 /= 0, [0,0])
     col%c2 = col%c1
  else if (size (c2) /= 0) then
     col%c1 = pack (c2, c2 /= 0, [0,0])
     col%c2 = col%c1
  end if
  col%ghost = .false.
end subroutine color_init_arrays

pure subroutine color_init_arrays_ghost (col, c1, c2, ghost)
  class(color_t), intent(inout) :: col
  integer, dimension(:), intent(in) :: c1, c2
  logical, intent(in) :: ghost
  call color_init_arrays (col, c1, c2)
  col%ghost = ghost
end subroutine color_init_arrays_ghost
```

This version is restricted to singlets, triplets, antitriplets, and octets: The input contains the color and anticolor index, each of the may be zero.

⟨*Colors: color: TBP*⟩+≡

```
procedure :: init_col_acl => color_init_col_acl
```

⟨*Colors: procedures*⟩+≡

```
elemental subroutine color_init_col_acl (col, col_in, acl_in)
  class(color_t), intent(inout) :: col
  integer, intent(in) :: col_in, acl_in
  integer, dimension(0) :: null_array
  select case (col_in)
  case (0)
     select case (acl_in)
     case (0)
        call color_init_array (col, null_array)
     case default
        call color_init_array (col, [-acl_in])
     end select
```

```
    case default
       select case (acl_in)
       case (0)
          call color_init_array (col, [col_in])
       case default
          call color_init_array (col, [col_in, -acl_in])
       end select
    end select
  end subroutine color_init_col_acl
```

This version is used for the external interface. We convert a fixed-size array of colors (for each particle) to the internal form by packing only the nonzero entries.

Some of these procedures produce an arry, so they can't be all type-bound. We implement them as ordinary procedures.

⟨*Colors: public*⟩+≡

```
  public :: color_init_from_array
```

⟨*Colors: interfaces*⟩≡

```
  interface color_init_from_array
     module procedure color_init_from_array1
     module procedure color_init_from_array1g
     module procedure color_init_from_array2
     module procedure color_init_from_array2g
  end interface color_init_from_array
```

⟨*Colors: procedures*⟩+≡

```
  pure subroutine color_init_from_array1 (col, c1)
    type(color_t), intent(inout) :: col
    integer, dimension(:), intent(in) :: c1
    logical, dimension(size(c1)) :: mask
    mask = c1 /= 0
    col%defined = .true.
    col%c1 = pack (c1, mask, col%c1)
    col%c2 = col%c1
    col%ghost = .false.
  end subroutine color_init_from_array1

  pure subroutine color_init_from_array1g (col, c1, ghost)
    type(color_t), intent(inout) :: col
    integer, dimension(:), intent(in) :: c1
    logical, intent(in) :: ghost
    call color_init_from_array1 (col, c1)
    col%ghost = ghost
  end subroutine color_init_from_array1g

  pure subroutine color_init_from_array2 (col, c1)
    integer, dimension(:,:), intent(in) :: c1
    type(color_t), dimension(:), intent(inout) :: col
    integer :: i
    do i = 1, size (c1,2)
       call color_init_from_array1 (col(i), c1(:,i))
    end do
```

```
end subroutine color_init_from_array2

pure subroutine color_init_from_array2g (col, c1, ghost)
  integer, dimension(:,:), intent(inout) :: c1
  type(color_t), dimension(:), intent(out) :: col
  logical, intent(in), dimension(:) :: ghost
  call color_init_from_array2 (col, c1)
  col%ghost = ghost
end subroutine color_init_from_array2g
```

Set the ghost property

⟨*Colors: color: TBP*⟩+≡

```
procedure :: set_ghost => color_set_ghost
```

⟨*Colors: procedures*⟩+≡

```
elemental subroutine color_set_ghost (col, ghost)
  class(color_t), intent(inout) :: col
  logical, intent(in) :: ghost
  col%ghost = ghost
end subroutine color_set_ghost
```

Undefine the color state:

⟨*Colors: color: TBP*⟩+≡

```
procedure :: undefine => color_undefine
```

⟨*Colors: procedures*⟩+≡

```
elemental subroutine color_undefine (col, undefine_ghost)
  class(color_t), intent(inout) :: col
  logical, intent(in), optional :: undefine_ghost
  col%defined = .false.
  if (present (undefine_ghost)) then
     if (undefine_ghost)  col%ghost = .false.
  else
     col%ghost = .false.
  end if
end subroutine color_undefine
```

Output. As dense as possible, no linebreak. If color is undefined, no output.

The separate version for a color array suggest two distinct interfaces.

⟨*Colors: public*⟩+≡

```
public :: color_write
```

⟨*Colors: interfaces*⟩+≡

```
interface color_write
   module procedure color_write_single
   module procedure color_write_array
end interface color_write
```

⟨*Colors: color: TBP*⟩+≡

```
procedure :: write => color_write_single
```

⟨*Colors: procedures*⟩+≡

```
subroutine color_write_single (col, unit)
  class(color_t), intent(in) :: col
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  if (col%ghost) then
     write (u, "(A)", advance="no")  "c*"
  else if (col%defined) then
     write (u, "(A)", advance="no")  "c("
     if (col%c1(1) /= 0)  write (u, "(I0)", advance="no")  col%c1(1)
     if (any (col%c1 /= 0))  write (u, "(1x)", advance="no")
     if (col%c1(2) /= 0)  write (u, "(I0)", advance="no")  col%c1(2)
     if (.not. col%is_diagonal ()) then
        write (u, "(A)", advance="no")  "|"
        if (col%c2(1) /= 0)  write (u, "(I0)", advance="no")  col%c2(1)
        if (any (col%c2 /= 0))  write (u, "(1x)", advance="no")
        if (col%c2(2) /= 0)  write (u, "(I0)", advance="no")  col%c2(2)
     end if
     write (u, "(A)", advance="no") ")"
  end if
end subroutine color_write_single

subroutine color_write_array (col, unit)
  type(color_t), dimension(:), intent(in) :: col
  integer, intent(in), optional :: unit
  integer :: u
  integer :: i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(A)", advance="no") "["
  do i = 1, size (col)
     if (i > 1)  write (u, "(1x)", advance="no")
     call color_write_single (col(i), u)
  end do
  write (u, "(A)", advance="no") "]"
end subroutine color_write_array
```

Binary I/O. For allocatable colors, this would have to be modified.

⟨*Colors: color: TBP*⟩+≡

```
procedure :: write_raw => color_write_raw
procedure :: read_raw => color_read_raw
```

⟨*Colors: procedures*⟩+≡

```
subroutine color_write_raw (col, u)
  class(color_t), intent(in) :: col
  integer, intent(in) :: u
  logical :: defined
  defined = col%is_defined () .or. col%is_ghost ()
  write (u) defined
  if (defined) then
     write (u) col%c1, col%c2
     write (u) col%ghost
  end if
end subroutine color_write_raw
```

```
subroutine color_read_raw (col, u, iostat)
  class(color_t), intent(inout) :: col
  integer, intent(in) :: u
  integer, intent(out), optional :: iostat
  logical :: defined
  read (u, iostat=iostat) col%defined
  if (col%defined) then
     read (u, iostat=iostat) col%c1, col%c2
     read (u, iostat=iostat) col%ghost
  end if
end subroutine color_read_raw
```

10.4.2 Predicates

Return the definition status. A color state may be defined but trivial.

⟨*Colors: color: TBP*⟩+≡

```
procedure :: is_defined => color_is_defined
procedure :: is_nonzero => color_is_nonzero
```

⟨*Colors: procedures*⟩+≡

```
elemental function color_is_defined (col) result (defined)
  logical :: defined
  class(color_t), intent(in) :: col
  defined = col%defined
end function color_is_defined

elemental function color_is_nonzero (col) result (flag)
  logical :: flag
  class(color_t), intent(in) :: col
  flag = col%defined &
       .and. .not. col%ghost &
       .and. any (col%c1 /= 0 .or. col%c2 /= 0)
end function color_is_nonzero
```

Diagonal color objects have only one array allocated:

⟨*Colors: color: TBP*⟩+≡

```
procedure :: is_diagonal => color_is_diagonal
```

⟨*Colors: procedures*⟩+≡

```
elemental function color_is_diagonal (col) result (diagonal)
  logical :: diagonal
  class(color_t), intent(in) :: col
  if (col%defined) then
     diagonal = all (col%c1 == col%c2)
  else
     diagonal = .true.
  end if
end function color_is_diagonal
```

Return the ghost flag

⟨*Colors: color: TBP*⟩+≡

```
procedure :: is_ghost => color_is_ghost
```

⟨*Colors: procedures*⟩+≡
```
elemental function color_is_ghost (col) result (ghost)
  logical :: ghost
  class(color_t), intent(in) :: col
  ghost = col%ghost
end function color_is_ghost
```

The ghost parity: true if the color-ghost flag is set. Again, no TBP since this is an array.

⟨*Colors: procedures*⟩+≡
```
pure function color_ghost_parity (col) result (parity)
  type(color_t), dimension(:), intent(in) :: col
  logical :: parity
  parity = mod (count (col%ghost), 2) == 1
end function color_ghost_parity
```

10.4.3 Accessing contents

Return the number of color indices. We assume that it is identical for both arrays.

⟨*Colors: color: TBP*⟩+≡
```
procedure, private :: get_number_of_indices => color_get_number_of_indices
```

⟨*Colors: procedures*⟩+≡
```
elemental function color_get_number_of_indices (col) result (n)
  integer :: n
  class(color_t), intent(in) :: col
  if (col%defined .and. .not. col%ghost) then
     n = count (col%c1 /= 0)
  else
     n = 0
  end if
end function color_get_number_of_indices
```

Return the (first) color/anticolor entry (assuming that color is diagonal). The result is a positive color index.

⟨*Colors: color: TBP*⟩+≡
```
procedure :: get_col => color_get_col
procedure :: get_acl => color_get_acl
```

⟨*Colors: procedures*⟩+≡
```
elemental function color_get_col (col) result (c)
  integer :: c
  class(color_t), intent(in) :: col
  integer :: i
  if (col%defined .and. .not. col%ghost) then
     do i = 1, size (col%c1)
        if (col%c1(i) > 0) then
           c = col%c1(i)
           return
        end if
     end do
```

```
    end if
    c = 0
  end function color_get_col

  elemental function color_get_acl (col) result (c)
    integer :: c
    class(color_t), intent(in) :: col
    integer :: i
    if (col%defined .and. .not. col%ghost) then
       do i = 1, size (col%c1)
          if (col%c1(i) < 0) then
             c = - col%c1(i)
             return
          end if
       end do
    end if
    c = 0
  end function color_get_acl
```

Return the color index with highest absolute value

⟨*Colors: public*⟩+≡

```
  public :: color_get_max_value
```

⟨*Colors: interfaces*⟩+≡

```
  interface color_get_max_value
     module procedure color_get_max_value0
     module procedure color_get_max_value1
     module procedure color_get_max_value2
  end interface color_get_max_value
```

⟨*Colors: procedures*⟩+≡

```
  elemental function color_get_max_value0 (col) result (cmax)
    integer :: cmax
    type(color_t), intent(in) :: col
    if (col%defined .and. .not. col%ghost) then
       cmax = maxval (abs (col%c1))
    else
       cmax = 0
    end if
  end function color_get_max_value0

  pure function color_get_max_value1 (col) result (cmax)
    integer :: cmax
    type(color_t), dimension(:), intent(in) :: col
    cmax = maxval (color_get_max_value0 (col))
  end function color_get_max_value1

  pure function color_get_max_value2 (col) result (cmax)
    integer :: cmax
    type(color_t), dimension(:,:), intent(in) :: col
    integer, dimension(size(col, 2)) :: cm
    integer :: i
    forall (i = 1:size(col, 2))
       cm(i) = color_get_max_value1 (col(:,i))
```

```
    end forall
    cmax = maxval (cm)
  end function color_get_max_value2
```

10.4.4 Comparisons

Similar to helicities, colors match if they are equal, or if either one is undefined.

⟨*Colors: color: TBP*⟩+≡

```
  generic :: operator(.match.) => color_match
  generic :: operator(==) => color_eq
  generic :: operator(/=) => color_neq
  procedure, private ::  color_match
  procedure, private ::  color_eq
  procedure, private ::  color_neq
```

⟨*Colors: procedures*⟩+≡

```
  elemental function color_match (col1, col2) result (eq)
    logical :: eq
    class(color_t), intent(in) :: col1, col2
    if (col1%defined .and. col2%defined) then
       if (col1%ghost .and. col2%ghost) then
          eq = .true.
       else if (.not. col1%ghost .and. .not. col2%ghost) then
          eq = all (col1%c1 == col2%c1) .and. all (col1%c2 == col2%c2)
       else
          eq = .false.
       end if
    else
       eq = .true.
    end if
  end function color_match

  elemental function color_eq (col1, col2) result (eq)
    logical :: eq
    class(color_t), intent(in) :: col1, col2
    if (col1%defined .and. col2%defined) then
       if (col1%ghost .and. col2%ghost) then
          eq = .true.
       else if (.not. col1%ghost .and. .not. col2%ghost) then
          eq = all (col1%c1 == col2%c1) .and. all (col1%c2 == col2%c2)
       else
          eq = .false.
       end if
    else if (.not. col1%defined &
       .and. .not. col2%defined) then
       eq = col1%ghost .eqv. col2%ghost
    else
       eq = .false.
    end if
  end function color_eq
```

⟨*Colors: procedures*⟩+≡

```
elemental function color_neq (col1, col2) result (neq)
  logical :: neq
  class(color_t), intent(in) :: col1, col2
  if (col1%defined .and. col2%defined) then
     if (col1%ghost .and. col2%ghost) then
        neq = .false.
     else if (.not. col1%ghost .and. .not. col2%ghost) then
        neq = any (col1%c1 /= col2%c1) .or. any (col1%c2 /= col2%c2)
     else
        neq = .true.
     end if
  else if (.not. col1%defined &
        .and. .not. col2%defined) then
     neq = col1%ghost .neqv. col2%ghost
  else
     neq = .true.
  end if
end function color_neq
```

10.4.5 Tools

Shift color indices by a common offset.

⟨*Colors: color: TBP*⟩+≡

```
procedure :: add_offset => color_add_offset
```

⟨*Colors: procedures*⟩+≡

```
elemental subroutine color_add_offset (col, offset)
  class(color_t), intent(inout) :: col
  integer, intent(in) :: offset
  if (col%defined .and. .not. col%ghost) then
     where (col%c1 /= 0)  col%c1 = col%c1 + sign (offset, col%c1)
     where (col%c2 /= 0)  col%c2 = col%c2 + sign (offset, col%c2)
  end if
end subroutine color_add_offset
```

Reassign color indices for an array of colored particle in canonical order. The allocated size of the color map is such that two colors per particle can be accomodated.

The algorithm works directly on the contents of the color objects, it

⟨*Colors: public*⟩+≡

```
public :: color_canonicalize
```

⟨*Colors: procedures*⟩+≡

```
subroutine color_canonicalize (col)
  type(color_t), dimension(:), intent(inout) :: col
  integer, dimension(2*size(col)) :: map
  integer :: n_col, i, j, k
  n_col = 0
  do i = 1, size (col)
     if (col(i)%defined .and. .not. col(i)%ghost) then
        do j = 1, size (col(i)%c1)
```

```
            if (col(i)%c1(j) /= 0) then
               k = find (abs (col(i)%c1(j)), map(:n_col))
               if (k == 0) then
                  n_col = n_col + 1
                  map(n_col) = abs (col(i)%c1(j))
                  k = n_col
               end if
               col(i)%c1(j) = sign (k, col(i)%c1(j))
            end if
            if (col(i)%c2(j) /= 0) then
               k = find (abs (col(i)%c2(j)), map(:n_col))
               if (k == 0) then
                  n_col = n_col + 1
                  map(n_col) = abs (col(i)%c2(j))
                  k = n_col
               end if
               col(i)%c2(j) = sign (k, col(i)%c2(j))
            end if
         end do
      end if
   end do
contains
   function find (c, array) result (k)
      integer :: k
      integer, intent(in) :: c
      integer, dimension(:), intent(in) :: array
      integer :: i
      k = 0
      do i = 1, size (array)
         if (c == array (i)) then
            k = i
            return
         end if
      end do
   end function find
end subroutine color_canonicalize
```

Return an array of different color indices from an array of colors. The last argument is a pseudo-color array, where the color entries correspond to the position of the corresponding index entry in the index array. The colors are assumed to be diagonal.

The algorithm works directly on the contents of the color objects.

⟨*Colors: procedures*⟩+≡

```
subroutine extract_color_line_indices (col, c_index, col_pos)
   type(color_t), dimension(:), intent(in) :: col
   integer, dimension(:), intent(out), allocatable :: c_index
   type(color_t), dimension(size(col)), intent(out) :: col_pos
   integer, dimension(:), allocatable :: c_tmp
   integer :: i, j, k, n, c
   allocate (c_tmp (sum (col%get_number_of_indices ())), source=0)
   n = 0
   SCAN1: do i = 1, size (col)
      if (col(i)%defined .and. .not. col(i)%ghost) then
```

```
      SCAN2: do j = 1, 2
         c = abs (col(i)%c1(j))
         if (c /= 0) then
            do k = 1, n
               if (c_tmp(k) == c) then
                  col_pos(i)%c1(j) = k
                  cycle SCAN2
               end if
            end do
            n = n + 1
            c_tmp(n) = c
            col_pos(i)%c1(j) = n
         end if
      end do SCAN2
   end if
end do SCAN1
allocate (c_index (n))
c_index = c_tmp(1:n)
end subroutine extract_color_line_indices
```

Given a color array, pairwise contract the color lines in all possible ways and return the resulting array of arrays. The input color array must be diagonal, and each color should occur exactly twice, once as color and once as anticolor.

Gluon entries with equal color and anticolor are explicitly excluded.

This algorithm is generic, but for long arrays it is neither efficient, nor does it avoid duplicates. It is intended for small arrays, in particular for the state matrix of a structure-function pair.

The algorithm works directly on the contents of the color objects, it thus depends on the implementation.

⟨*Colors: public*⟩+≡

```
public :: color_array_make_contractions
```

⟨*Colors: procedures*⟩+≡

```
subroutine color_array_make_contractions (col_in, col_out)
  type(color_t), dimension(:), intent(in) :: col_in
  type(color_t), dimension(:,:), intent(out), allocatable :: col_out
  type :: entry_t
     integer, dimension(:), allocatable :: map
     type(color_t), dimension(:), allocatable :: col
     type(entry_t), pointer :: next => null ()
     logical :: nlo_event = .false.
  end type entry_t
  type :: list_t
     integer :: n = 0
     type(entry_t), pointer :: first => null ()
     type(entry_t), pointer :: last => null ()
  end type list_t
  type(list_t) :: list
  type(entry_t), pointer :: entry
  integer, dimension(:), allocatable :: c_index
  type(color_t), dimension(size(col_in)) :: col_pos
  integer :: n_prt, n_c_index
  integer, dimension(:), allocatable :: map
```

```
  integer :: i, j, c
  n_prt = size (col_in)
  call extract_color_line_indices (col_in, c_index, col_pos)
  n_c_index = size (c_index)
  allocate (map (n_c_index))
  map = 0
  call list_append_if_valid (list, map)
  entry => list%first
  do while (associated (entry))
     do i = 1, n_c_index
        if (entry%map(i) == 0) then
           c = c_index(i)
           do j = i + 1, n_c_index
              if (entry%map(j) == 0) then
                 map = entry%map
                 map(i) = c
                 map(j) = c
                 call list_append_if_valid (list, map)
              end if
           end do
        end if
     end do
     entry => entry%next
  end do
  call list_to_array (list, col_out)
contains
  subroutine list_append_if_valid (list, map)
    type(list_t), intent(inout) :: list
    integer, dimension(:), intent(in) :: map
    type(entry_t), pointer :: entry
    integer :: i, j, c, p
    entry => list%first
    do while (associated (entry))
       if (all (map == entry%map))  return
       entry => entry%next
    end do
    allocate (entry)
    allocate (entry%map (n_c_index))
    entry%map = map
    allocate (entry%col (n_prt))
    do i = 1, n_prt
       do j = 1, 2
          c = col_in(i)%c1(j)
          if (c /= 0) then
             p = col_pos(i)%c1(j)
             entry%col(i)%defined = .true.
             if (map(p) /= 0) then
                entry%col(i)%c1(j) = sign (map(p), c)
             else
                entry%col(i)%c1(j) = c
             endif
             entry%col(i)%c2(j) = entry%col(i)%c1(j)
          end if
       end do
```

```
      if (any (entry%col(i)%c1 /= 0) .and. &
           entry%col(i)%c1(1) == - entry%col(i)%c1(2))  return
    end do
    if (associated (list%last)) then
       list%last%next => entry
    else
       list%first => entry
    end if
    list%last => entry
    list%n = list%n + 1
  end subroutine list_append_if_valid
  subroutine list_to_array (list, col)
    type(list_t), intent(inout) :: list
    type(color_t), dimension(:,:), intent(out), allocatable :: col
    type(entry_t), pointer :: entry
    integer :: i
    allocate (col (n_prt, list%n - 1))
    do i = 0, list%n - 1
       entry => list%first
       list%first => list%first%next
       if (i /= 0)  col(:,i) = entry%col
       deallocate (entry)
    end do
    list%last => null ()
  end subroutine list_to_array
end subroutine color_array_make_contractions
```

Invert the color index, switching from particle to antiparticle. For gluons, we have to swap the order of color entries.

⟨*Colors: color: TBP*⟩+≡

```
procedure :: invert => color_invert
```

⟨*Colors: procedures*⟩+≡

```
elemental subroutine color_invert (col)
  class(color_t), intent(inout) :: col
  if (col%defined .and. .not. col%ghost) then
     col%c1 = - col%c1
     col%c2 = - col%c2
     if (col%c1(1) < 0 .and. col%c1(2) > 0) then
        col%c1 = col%c1(2:1:-1)
        col%c2 = col%c2(2:1:-1)
     end if
  end if
end subroutine color_invert
```

Make a color map for two matching color arrays. The result is an array of integer pairs.

⟨*Colors: public*⟩+≡

```
public :: make_color_map
```

⟨*Colors: interfaces*⟩+≡

```
interface make_color_map
   module procedure color_make_color_map
end interface make_color_map
```

⟨*Colors: procedures*⟩+≡

```
subroutine color_make_color_map (map, col1, col2)
  integer, dimension(:,:), intent(out), allocatable :: map
  type(color_t), dimension(:), intent(in) :: col1, col2
  integer, dimension(:,:), allocatable :: map1
  integer :: i, j, k
  allocate (map1 (2, 2 * sum (col1%get_number_of_indices ())))
  k = 0
  do i = 1, size (col1)
     if (col1(i)%defined .and. .not. col1(i)%ghost) then
        do j = 1, size (col1(i)%c1)
           if (col1(i)%c1(j) /= 0 &
                .and. all (map1(1,:k) /= abs (col1(i)%c1(j)))) then
              k = k + 1
              map1(1,k) = abs (col1(i)%c1(j))
              map1(2,k) = abs (col2(i)%c1(j))
           end if
           if (col1(i)%c2(j) /= 0 &
                .and. all (map1(1,:k) /= abs (col1(i)%c2(j)))) then
              k = k + 1
              map1(1,k) = abs (col1(i)%c2(j))
              map1(2,k) = abs (col2(i)%c2(j))
           end if
        end do
     end if
  end do
  allocate (map (2, k))
  map(:,:) = map1(:,:k)
end subroutine color_make_color_map
```

Translate colors which have a match in the translation table (an array of integer pairs). Color that do not match an entry are simply transferred; this is done by first transferring all components, then modifiying entries where appropriate.

⟨*Colors: public*⟩+≡

```
public :: color_translate
```

⟨*Colors: interfaces*⟩+≡

```
interface color_translate
   module procedure color_translate0
   module procedure color_translate0_offset
   module procedure color_translate1
end interface color_translate
```

⟨*Colors: procedures*⟩+≡

```
subroutine color_translate0 (col, map)
  type(color_t), intent(inout) :: col
  integer, dimension(:,:), intent(in) :: map
  type(color_t) :: col_tmp
  integer :: i
  if (col%defined .and. .not. col%ghost) then
     col_tmp = col
     do i = 1, size (map,2)
```

```
        where (abs (col%c1) == map(1,i))
           col_tmp%c1 = sign (map(2,i), col%c1)
        end where
        where (abs (col%c2) == map(1,i))
           col_tmp%c2 = sign (map(2,i), col%c2)
        end where
     end do
     col = col_tmp
  end if
end subroutine color_translate0

subroutine color_translate0_offset (col, map, offset)
  type(color_t), intent(inout) :: col
  integer, dimension(:,:), intent(in) :: map
  integer, intent(in) :: offset
  logical, dimension(size(col%c1)) :: mask1, mask2
  type(color_t) :: col_tmp
  integer :: i
  if (col%defined .and. .not. col%ghost) then
     col_tmp = col
     mask1 = col%c1 /= 0
     mask2 = col%c2 /= 0
     do i = 1, size (map,2)
        where (abs (col%c1) == map(1,i))
           col_tmp%c1 = sign (map(2,i), col%c1)
           mask1 = .false.
        end where
        where (abs (col%c2) == map(1,i))
           col_tmp%c2 = sign (map(2,i), col%c2)
           mask2 = .false.
        end where
     end do
     col = col_tmp
     where (mask1)  col%c1 = sign (abs (col%c1) + offset, col%c1)
     where (mask2)  col%c2 = sign (abs (col%c2) + offset, col%c2)
  end if
end subroutine color_translate0_offset

subroutine color_translate1 (col, map, offset)
  type(color_t), dimension(:), intent(inout) :: col
  integer, dimension(:,:), intent(in) :: map
  integer, intent(in), optional :: offset
  integer :: i
  if (present (offset)) then
     do i = 1, size (col)
        call color_translate0_offset (col(i), map, offset)
     end do
  else
     do i = 1, size (col)
        call color_translate0 (col(i), map)
     end do
  end if
end subroutine color_translate1
```

Merge two color objects by taking the first entry from the first and the first entry from the second argument. Makes sense only if the input colors are defined (and diagonal). If either one is undefined, transfer the defined one.

⟨*Colors: color: TBP*⟩+≡

```
generic :: operator(.merge.) => merge_colors
procedure, private ::  merge_colors
```

⟨*Colors: procedures*⟩+≡

```
elemental function merge_colors (col1, col2) result (col)
  type(color_t) :: col
  class(color_t), intent(in) :: col1, col2
  if (color_is_defined (col1) .and. color_is_defined (col2)) then
     if (color_is_ghost (col1) .and. color_is_ghost (col2)) then
        call color_init_trivial_ghost (col, .true.)
     else
        call color_init_arrays (col, col1%c1, col2%c1)
     end if
  else if (color_is_defined (col1)) then
     call color_init_array (col, col1%c1)
  else if (color_is_defined (col2)) then
     call color_init_array (col, col2%c1)
  end if
end function merge_colors
```

Compute the color factor, given two interfering color arrays.

⟨*Colors: public*⟩+≡

```
public :: compute_color_factor
```

⟨*Colors: procedures*⟩+≡

```
function compute_color_factor (col1, col2, nc) result (factor)
  real(default) :: factor
  type(color_t), dimension(:), intent(in) :: col1, col2
  integer, intent(in), optional :: nc
  type(color_t), dimension(size(col1)) :: col
  integer :: ncol, nloops, nghost
  ncol = 3;  if (present (nc))  ncol = nc
  col = col1 .merge. col2
  nloops = count_color_loops (col)
  nghost = count (col%is_ghost ())
  factor = real (ncol, default) ** (nloops - nghost)
  if (color_ghost_parity (col))  factor = - factor
end function compute_color_factor
```

We have a pair of color index arrays which corresponds to a squared matrix element. We want to determine the number of color loops in this square matrix element. So we first copy the colors (stored in a single color array with a pair of color lists in each entry) to a temporary where the color indices are shifted by some offset. We then recursively follow each loop, starting at the first color that has the offset, resetting the first color index to the loop index and each further index to zero as we go. We check that (a) each color index occurs twice within the left (right) color array, (b) the loops are closed, so we always come back to a line which has the loop index.

In order for the algorithm to work we have to conjugate the colors of initial state particles (one for decays, two for scatterings) into their corresponding anticolors of outgoing particles.

⟨*Colors: public*⟩+≡

```
public :: count_color_loops
```

⟨*Colors: procedures*⟩+≡

```
function count_color_loops (col) result (count)
  integer :: count
  type(color_t), dimension(:), intent(in) :: col
  type(color_t), dimension(size(col)) :: cc
  integer :: i, n, offset
  cc = col
  n = size (cc)
  offset = n
  call color_add_offset (cc, offset)
  count = 0
  SCAN_LOOPS: do
     do i = 1, n
        if (color_is_nonzero (cc(i))) then
           if (any (cc(i)%c1 > offset)) then
              count = count + 1
              call follow_line1 (pick_new_line (cc(i)%c1, count, 1))
              cycle SCAN_LOOPS
           end if
        end if
     end do
     exit SCAN_LOOPS
  end do SCAN_LOOPS
contains
  function pick_new_line (c, reset_val, sgn) result (line)
    integer :: line
    integer, dimension(:), intent(inout) :: c
    integer, intent(in) :: reset_val
    integer, intent(in) :: sgn
    integer :: i
    if (any (c == count)) then
       line = count
    else
       do i = 1, size (c)
          if (sign (1, c(i)) == sgn .and. abs (c(i)) > offset) then
             line = c(i)
             c(i) = reset_val
             return
          end if
       end do
       call color_mismatch
    end if
  end function pick_new_line

  subroutine reset_line (c, line)
    integer, dimension(:), intent(inout) :: c
    integer, intent(in) :: line
    integer :: i
```

```
    do i = 1, size (c)
       if (c(i) == line) then
          c(i) = 0
          return
       end if
    end do
  end subroutine reset_line

  recursive subroutine follow_line1 (line)
    integer, intent(in) :: line
    integer :: i
    if (line == count) return
    do i = 1, n
       if (any (cc(i)%c1 == -line)) then
          call reset_line (cc(i)%c1, -line)
          call follow_line2 (pick_new_line (cc(i)%c2, 0, sign (1, -line)))
          return
       end if
    end do
    call color_mismatch ()
  end subroutine follow_line1

  recursive subroutine follow_line2 (line)
    integer, intent(in) :: line
    integer :: i
    do i = 1, n
       if (any (cc(i)%c2 == -line)) then
          call reset_line (cc(i)%c2, -line)
          call follow_line1 (pick_new_line (cc(i)%c1, 0, sign (1, -line)))
          return
       end if
    end do
    call color_mismatch ()
  end subroutine follow_line2

  subroutine color_mismatch ()
    call color_write (col)
    print *
    call msg_fatal ("Color flow mismatch: Non-closed color lines appear during ", &
       [var_str ("the evaluation of color correlations. This can happen if there "), &
        var_str ("are different color structures in the initial or final state of "), &
        var_str ("the process definition. If so, please use separate processes for "), &
        var_str ("the different initial / final states. In a future WHIZARD version "), &
        var_str ("this will be fixed.")])
  end subroutine color_mismatch
end function count_color_loops
```

10.4.6 Unit tests

Test module, followed by the corresponding implementation module.

⟨colors_ut.f90⟩≡
  ⟨*File header*⟩

```
module colors_ut
  use unit_tests
  use colors_uti

⟨Standard module head⟩

⟨Colors: public test⟩

contains

⟨Colors: test driver⟩

end module colors_ut
```

⟨colors_uti.f90⟩≡

```
⟨File header⟩

module colors_uti

  use colors

⟨Standard module head⟩

⟨Colors: test declarations⟩

contains

⟨Colors: tests⟩

end module colors_uti
```

API: driver for the unit tests below.

⟨*Colors: public test*⟩≡

```
public :: color_test
```

⟨*Colors: test driver*⟩≡

```
subroutine color_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨Colors: execute tests⟩
end subroutine color_test
```

This is a color counting test.

⟨*Colors: execute tests*⟩≡

```
call test (color_1, "color_1", &
     "check color counting", &
     u, results)
```

⟨*Colors: test declarations*⟩≡

```
public :: color_1
```

⟨*Colors: tests*⟩≡

```
subroutine color_1 (u)
  integer, intent(in) :: u
  type(color_t), dimension(4) :: col1, col2, col
```

```
  type(color_t), dimension(:), allocatable :: col3
  type(color_t), dimension(:,:), allocatable :: col_array
  integer :: count, i
  call col1%init_col_acl ([1, 0, 2, 3], [0, 1, 3, 2])
  col2 = col1
  call color_write (col1, u)
  write (u, "(A)")
  call color_write (col2, u)
  write (u, "(A)")
  col = col1 .merge. col2
  call color_write (col, u)
  write (u, "(A)")
  count = count_color_loops (col)
  write (u, "(A,I1)") "Number of color loops (3): ", count
  call col2%init_col_acl ([1, 0, 2, 3], [0, 2, 3, 1])
  call color_write (col1, u)
  write (u, "(A)")
  call color_write (col2, u)
  write (u, "(A)")
  col = col1 .merge. col2
  call color_write (col, u)
  write (u, "(A)")
  count = count_color_loops (col)
  write (u, "(A,I1)")  "Number of color loops (2): ", count
  write (u, "(A)")
  allocate (col3 (4))
  call color_init_from_array (col3, &
       reshape ([1, 0,   0, -1,  2, -3,  3, -2], &
                [2, 4]))
  call color_write (col3, u)
  write (u, "(A)")
  call color_array_make_contractions (col3, col_array)
  write (u, "(A)")  "Contractions:"
  do i = 1, size (col_array, 2)
     call color_write (col_array(:,i), u)
     write (u, "(A)")
  end do
  deallocate (col3)
  write (u, "(A)")
  allocate (col3 (6))
  call color_init_from_array (col3, &
       reshape ([1, -2,   3, 0,  0, -1,  2, -4,  -3, 0,  4, 0], &
                [2, 6]))
  call color_write (col3, u)
  write (u, "(A)")
  call color_array_make_contractions (col3, col_array)
  write (u, "(A)")  "Contractions:"
  do i = 1, size (col_array, 2)
     call color_write (col_array(:,i), u)
     write (u, "(A)")
  end do
end subroutine color_1
```

10.4.7 The Madgraph color model

This section describes the method for matrix element and color flow calculation within Madgraph.

For each Feynman diagram, the colorless amplitude for a specified helicity and momentum configuration (in- and out- combined) is computed:

$$A_d(p,h) \tag{10.1}$$

Inserting color, the squared matrix element for definite helicity and momentum is

$$M^2(p,h) = \sum_{dd'} A_d(p,h)\, C_{dd'}\, A^*_{d'}(p,h) \tag{10.2}$$

where $C_{dd'}$ describes the color interference of the two diagrams A_d and A'_d, which is independent of momentum and helicity and can be calculated for each Feynman diagram pair by reducing it to the corresponding color graph. Obviously, one could combine all diagrams with identical color structure, such that the index d runs only over different color graphs. For colorless diagrams all elements of $C_{dd'}$ are equal to unity.

The hermitian matrix $C_{dd'}$ is diagonalized once and for all, such that it can be written in the form

$$C_{dd'} = \sum_\lambda c_d^\lambda \lambda\, c_d^{\lambda *}, \tag{10.3}$$

where the eigenvectors c_d are normalized,

$$\sum_d |c_d^\lambda|^2 = 1, \tag{10.4}$$

and the λ values are the corresponding eigenvalues. In the colorless case, this means $c_d = 1/\sqrt{N_d}$ for all diagrams (N_d = number of diagrams), and $\lambda = N_d$ is the only nonzero eigenvalue.

Consequently, the squared matrix element for definite helicity and momentum can also be written as

$$M^2(p,h) = \sum_\lambda A_\lambda(p,h)\, \lambda\, A_\lambda(p,h)^* \tag{10.5}$$

with

$$A_\lambda(p,h) = \sum_d c_d^\lambda A_d(p,h). \tag{10.6}$$

For generic spin density matrices, this is easily generalized to

$$M^2(p,h,h') = \sum_\lambda A_\lambda(p,h)\, \lambda\, A_\lambda(p,h')^* \tag{10.7}$$

To determine the color flow probabilities of a given momentum-helicity configuration, the color flow amplitudes are calculated as

$$a_f(p,h) = \sum_d \beta_d^f A_d(p,h), \tag{10.8}$$

where the coefficients β_d^f describe the amplitude for a given Feynman diagram (or color graph) d to correspond to a definite color flow f. They are computed

from $C_{dd'}$ by transforming this matrix into the color flow basis and neglecting all off-diagonal elements. Again, these coefficients do not depend on momentum or helicity and can therefore be calculated in advance. This gives the color flow transition matrix

$$F^f(p,h,h') = a_f(p,h)\, a_f^*(p,h') \tag{10.9}$$

which is assumed diagonal in color flow space and is separate from the color-summed transition matrix M^2. They are, however, equivalent (up to a factor) to leading order in $1/N_c$, and using the color flow transition matrix is appropriate for matching to hadronization.

Note that the color flow transition matrix is not normalized at this stage. To make use of it, we have to fold it with the in-state density matrix to get a pseudo density matrix

$$\hat{\rho}^f_{\text{out}}(p,h_{\text{out}},h'_{\text{out}}) = \sum_{h_{\text{in}}h'_{\text{in}}} F^f(p,h,h')\,\rho_{\text{in}}(p,h_{\text{in}},h'_{\text{in}}) \tag{10.10}$$

which gets a meaning only after contracted with projections on the outgoing helicity states k_{out}, given as linear combinations of helicity states with the unitary coefficient matrix $c(k_{\text{out}},h_{\text{out}})$. Then the probability of finding color flow f when the helicity state k_{out} is measured is given by

$$P^f(p,k_{\text{out}}) = Q^f(p,k_{\text{out}})/\sum_f Q^f(p,k_{\text{out}}) \tag{10.11}$$

where

$$Q^f(p,k_{\text{out}}) = \sum_{h_{\text{out}}h'_{\text{out}}} c(k_{\text{out}},h_{\text{out}})\,\hat{\rho}^f_{\text{out}}(p,h_{\text{out}},h'_{\text{out}})\,c^*(k_{\text{out}},h'_{\text{out}}) \tag{10.12}$$

However, if we can assume that the out-state helicity basis is the canonical one, we can throw away the off diagonal elements in the color flow density matrix and normalize the ones on the diagonal to obtain

$$P^f(p,h_{\text{out}}) = \hat{\rho}^f_{\text{out}}(p,h_{\text{out}},h_{\text{out}})/\sum_f \hat{\rho}^f_{\text{out}}(p,h_{\text{out}},h_{\text{out}}) \tag{10.13}$$

Finally, the color-summed out-state density matrix is computed by the scattering formula

$$\rho_{\text{out}}(p,h_{\text{out}},h'_{\text{out}}) = \sum_{h_{\text{in}}h'_{\text{in}}} M^2(p,h,h')\,\rho_{\text{in}}(p,h_{\text{in}},h'_{\text{in}}) \tag{10.14}$$

$$= \sum_{h_{\text{in}}h'_{\text{in}}\lambda} A_\lambda(p,h)\,\lambda\,A_\lambda(p,h')^*\rho_{\text{in}}(p,h_{\text{in}},h'_{\text{in}}), \tag{10.15}$$

The trace of ρ_{out} is the squared matrix element, summed over all internal degrees of freedom. To get the squared matrix element for a definite helicity k_{out} and color flow f, one has to project the density matrix onto the given helicity state and multiply with $P^f(p,k_{\text{out}})$.

For diagonal helicities the out-state density reduces to

$$\rho_{\text{out}}(p,h_{\text{out}}) = \sum_{h_{\text{in}}\lambda} \lambda|A_\lambda(p,h)|^2\rho_{\text{in}}(p,h_{\text{in}}). \tag{10.16}$$

Since no basis transformation is involved, we can use the normalized color flow probability $P^f(p, h_{\rm out})$ and express the result as

$$\rho^f_{\rm out}(p, h_{\rm out}) = \rho_{\rm out}(p, h_{\rm out})\, P^f(p, h_{\rm out}) \tag{10.17}$$

$$= \sum_{h_{\rm in}\lambda} \frac{|a^f(p,h)|^2}{\sum_f |a^f(p,h)|^2} \lambda |A_\lambda(p,h)|^2 \rho_{\rm in}(p, h_{\rm in}). \tag{10.18}$$

From these considerations, the following calculation strategy can be derived:

- Before the first event is generated, the color interference matrix $C_{dd'}$ is computed and diagonalized, so the eigenvectors c^λ_d, eigenvalues λ and color flow coefficients β^f_d are obtained. In practice, these calculations are done when the matrix element code is generated, and the results are hardcoded in the matrix element subroutine as `DATA` statements.
- For each event, one loops over helicities once and stores the matrices $A_\lambda(p,h)$ and $a^f(p,h)$. The allowed color flows, helicity combinations and eigenvalues are each labeled by integer indices, so one has to store complex matrices of dimension $N_\lambda \times N_h$ and $N_f \times N_h$, respectively.
- The further strategy depends on the requested information.
  1. If colorless diagonal helicity amplitudes are required, the eigenvalues $A_\lambda(p,h)$ are squared, summed with weight λ, and the result contracted with the in-state probability vector $\rho_{\rm in}(p, h_{\rm in})$. The result is a probability vector $\rho_{\rm out}(p, h_{\rm out})$.
  2. For colored diagonal helicity amplitudes, the color coefficients $a^f(p,h)$ are also squared and used as weights to obtain the color-flow probability vector $\rho^f_{\rm out}(p, h_{\rm out})$.
  3. For colorless non-diagonal helicity amplitudes, we contract the tensor product of $A_\lambda(p,h)$ with $A_\lambda(p,h')$, weighted with λ, with the correlated in-state density matrix, to obtain a correlated out-state density matrix.
  4. In the general (colored, non-diagonal) case, we do the same as in the colorless case, but return the un-normalized color flow density matrix $\hat\rho^f_{\rm out}(p, h_{\rm out}, h'_{\rm out})$ in addition. When the relevant helicity basis is known, the latter can be used by the caller program to determine flow probabilities. (In reality, we assume the canonical basis and reduce the correlated out-state density to its diagonal immediately.)

10.5 Flavors: Particle properties

This module contains a type for holding the flavor code, and all functions that depend on the model, i.e., that determine particle properties.

The PDG code is packed in a special `flavor` type. (This prohibits meaningless operations, and it allows for a different implementation, e.g., some non-PDG scheme internally, if appropiate at some point.)

There are lots of further particle properties that depend on the model. Implementing a flyweight pattern, the associated field data object is to be stored in a central area, the `flavor` object just receives a pointer to this, so all queries can be delegated.

⟨`flavors.f90`⟩≡

```
⟨File header⟩

module flavors

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use diagnostics
  use physics_defs, only: UNDEFINED
  use physics_defs, only: INVALID
  use physics_defs, only: HADRON_REMNANT
  use physics_defs, only: HADRON_REMNANT_SINGLET
  use physics_defs, only: HADRON_REMNANT_TRIPLET
  use physics_defs, only: HADRON_REMNANT_OCTET
  use model_data
  use colors, only: color_t

⟨Standard module head⟩

⟨Flavors: public⟩

⟨Flavors: types⟩

⟨Flavors: interfaces⟩

contains

⟨Flavors: procedures⟩

end module flavors
```

10.5.1 The flavor type

The flavor type is an integer representing the PDG code, or undefined (zero). Negative codes represent antiflavors. They should be used only for particles which do have a distinct antiparticle.

The `radiated` flag can be set for particles that are the result of a beam-structure interaction (hadron beam remnant, ISR photon, etc.), not of the hard interaction itself.

Further properties of the given flavor can be retrieved via the particle-data pointer, if it is associated.

⟨*Flavors: public*⟩≡
```
public :: flavor_t
```

⟨*Flavors: types*⟩≡
```
type :: flavor_t
   private
   integer :: f = UNDEFINED
   logical :: radiated = .false.
   type(field_data_t), pointer :: field_data => null ()
 contains
 ⟨Flavors: flavor: TBP⟩
end type flavor_t
```

Initializer form. If the model is assigned, the procedure is impure, therefore we have to define a separate array version.

Note: The pure elemental subroutines can't have an intent(out) CLASS argument (because of the potential for an impure finalizer in a type extension), so we stick to intent(inout) and (re)set all components explicitly.

⟨*Flavors: flavor: TBP*⟩≡
```
generic :: init => &
     flavor_init_empty, &
     flavor_init, &
     flavor_init_field_data, &
     flavor_init_model, &
     flavor_init_model_alt, &
     flavor_init_name_model
procedure, private :: flavor_init_empty
procedure, private :: flavor_init
procedure, private :: flavor_init_field_data
procedure, private :: flavor_init_model
procedure, private :: flavor_init_model_alt
procedure, private :: flavor_init_name_model
```

⟨*Flavors: procedures*⟩≡
```
elemental subroutine flavor_init_empty (flv)
  class(flavor_t), intent(inout) :: flv
  flv%f = UNDEFINED
  flv%radiated = .false.
  flv%field_data => null ()
end subroutine flavor_init_empty

elemental subroutine flavor_init (flv, f)
  class(flavor_t), intent(inout) :: flv
  integer, intent(in) :: f
  flv%f = f
  flv%radiated = .false.
  flv%field_data => null ()
end subroutine flavor_init

impure elemental subroutine flavor_init_field_data (flv, field_data)
  class(flavor_t), intent(inout) :: flv
  type(field_data_t), intent(in), target :: field_data
```

```
  flv%f = field_data%get_pdg ()
  flv%radiated = .false.
  flv%field_data => field_data
end subroutine flavor_init_field_data

impure elemental subroutine flavor_init_model (flv, f, model)
  class(flavor_t), intent(inout) :: flv
  integer, intent(in) :: f
  class(model_data_t), intent(in), target :: model
  flv%f = f
  flv%radiated = .false.
  flv%field_data => model%get_field_ptr (f, check=.true.)
end subroutine flavor_init_model

impure elemental subroutine flavor_init_model_alt (flv, f, model, alt_model)
  class(flavor_t), intent(inout) :: flv
  integer, intent(in) :: f
  class(model_data_t), intent(in), target :: model, alt_model
  flv%f = f
  flv%radiated = .false.
  flv%field_data => model%get_field_ptr (f, check=.false.)
  if (.not. associated (flv%field_data)) then
     flv%field_data => alt_model%get_field_ptr (f, check=.false.)
     if (.not. associated (flv%field_data)) then
        write (msg_buffer, "(A,1x,I0,1x,A,1x,A,1x,A,1x,A)") &
             "Particle with code", f, &
             "found neither in model", char (model%get_name ()), &
             "nor in model", char (alt_model%get_name ())
        call msg_fatal ()
     end if
  end if
end subroutine flavor_init_model_alt

impure elemental subroutine flavor_init_name_model (flv, name, model)
  class(flavor_t), intent(inout) :: flv
  type(string_t), intent(in) :: name
  class(model_data_t), intent(in), target :: model
  flv%f = model%get_pdg (name)
  flv%radiated = .false.
  flv%field_data => model%get_field_ptr (name, check=.true.)
end subroutine flavor_init_name_model
```

Set the `radiated` flag.

⟨*Flavors: flavor: TBP*⟩+≡

```
procedure :: tag_radiated => flavor_tag_radiated
```

⟨*Flavors: procedures*⟩+≡

```
elemental subroutine flavor_tag_radiated (flv)
  class(flavor_t), intent(inout) :: flv
  flv%radiated = .true.
end subroutine flavor_tag_radiated
```

Undefine the flavor state:

⟨*Flavors: flavor: TBP*⟩+≡

```
procedure :: undefine => flavor_undefine
```

⟨*Flavors: procedures*⟩+≡

```
elemental subroutine flavor_undefine (flv)
  class(flavor_t), intent(inout) :: flv
  flv%f = UNDEFINED
  flv%field_data => null ()
end subroutine flavor_undefine
```

Output: dense, no linebreak

⟨*Flavors: flavor: TBP*⟩+≡

```
procedure :: write => flavor_write
```

⟨*Flavors: procedures*⟩+≡

```
subroutine flavor_write (flv, unit)
  class(flavor_t), intent(in) :: flv
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  if (associated (flv%field_data)) then
     write (u, "(A)", advance="no")  "f("
  else
     write (u, "(A)", advance="no")  "p("
  end if
  write (u, "(I0)", advance="no")  flv%f
  if (flv%radiated) then
     write (u, "('*')", advance="no")
  end if
  write (u, "(A)", advance="no")  ")"
end subroutine flavor_write
```

⟨*Flavors: public*⟩+≡

```
public :: flavor_write_array
```

⟨*Flavors: procedures*⟩+≡

```
subroutine flavor_write_array (flv, unit)
  type(flavor_t), intent(in), dimension(:) :: flv
  integer, intent(in), optional :: unit
  integer :: u, i_flv
  u = given_output_unit (unit); if (u < 0) return
  do i_flv = 1, size (flv)
     call flv(i_flv)%write (u)
     if (i_flv /= size (flv)) write (u,"(A)", advance = "no") " / "
  end do
  write (u,"(A)")
end subroutine flavor_write_array
```

Binary I/O. Currently, the model information is not written/read, so after reading the particle-data pointer is empty.

⟨*Flavors: flavor: TBP*⟩+≡

```
procedure :: write_raw => flavor_write_raw
procedure :: read_raw => flavor_read_raw
```

⟨*Flavors: procedures*⟩+≡

```
subroutine flavor_write_raw (flv, u)
  class(flavor_t), intent(in) :: flv
  integer, intent(in) :: u
  write (u) flv%f
  write (u) flv%radiated
end subroutine flavor_write_raw

subroutine flavor_read_raw (flv, u, iostat)
  class(flavor_t), intent(out) :: flv
  integer, intent(in) :: u
  integer, intent(out), optional :: iostat
  read (u, iostat=iostat) flv%f
  if (present (iostat)) then
     if (iostat /= 0)  return
  end if
  read (u, iostat=iostat)  flv%radiated
end subroutine flavor_read_raw
```

Assignment

Default assignment of flavor objects is possible, but cannot be used in pure procedures, because a pointer assignment is involved.

Assign the particle pointer separately. This cannot be elemental, so we define a scalar and an array version explicitly. We refer to an array of flavors, not an array of models.

⟨*Flavors: flavor: TBP*⟩+≡

```
procedure :: set_model => flavor_set_model_single
```

⟨*Flavors: procedures*⟩+≡

```
impure elemental subroutine flavor_set_model_single (flv, model)
  class(flavor_t), intent(inout) :: flv
  class(model_data_t), intent(in), target :: model
  if (flv%f /= UNDEFINED) &
       flv%field_data => model%get_field_ptr (flv%f)
end subroutine flavor_set_model_single
```

Predicates

Return the definition status. By definition, the flavor object is defined if the flavor PDG code is nonzero.

⟨*Flavors: flavor: TBP*⟩+≡

```
procedure :: is_defined => flavor_is_defined
```

⟨*Flavors: procedures*⟩+≡

```
elemental function flavor_is_defined (flv) result (defined)
  class(flavor_t), intent(in) :: flv
  logical :: defined
  defined = flv%f /= UNDEFINED
end function flavor_is_defined
```

Check for valid flavor (including undefined). This is distinct from the `is_defined` status. Invalid flavor is actually a specific PDG code.

⟨*Flavors: flavor: TBP*⟩+≡
```
procedure :: is_valid => flavor_is_valid
```

⟨*Flavors: procedures*⟩+≡
```
elemental function flavor_is_valid (flv) result (valid)
  class(flavor_t), intent(in) :: flv
  logical :: valid
  valid = flv%f /= INVALID
end function flavor_is_valid
```

Return true if the particle-data pointer is associated. (Debugging aid)

⟨*Flavors: flavor: TBP*⟩+≡
```
procedure :: is_associated => flavor_is_associated
```

⟨*Flavors: procedures*⟩+≡
```
elemental function flavor_is_associated (flv) result (flag)
  class(flavor_t), intent(in) :: flv
  logical :: flag
  flag = associated (flv%field_data)
end function flavor_is_associated
```

Check the `radiated` flag. A radiated particle has a definite PDG flavor status, but it is actually a pseudoparticle (a beam remnant) which may be subject to fragmentation.

⟨*Flavors: flavor: TBP*⟩+≡
```
procedure :: is_radiated => flavor_is_radiated
```

⟨*Flavors: procedures*⟩+≡
```
elemental function flavor_is_radiated (flv) result (flag)
  class(flavor_t), intent(in) :: flv
  logical :: flag
  flag = flv%radiated
end function flavor_is_radiated
```

Accessing contents

With the exception of the PDG code, all particle property enquiries are delegated to the `field_data` pointer. If this is unassigned, some access function will crash.

Return the flavor as an integer

⟨*Flavors: flavor: TBP*⟩+≡
```
procedure :: get_pdg => flavor_get_pdg
```

⟨*Flavors: procedures*⟩+≡
```
elemental function flavor_get_pdg (flv) result (f)
  integer :: f
  class(flavor_t), intent(in) :: flv
  f = flv%f
end function flavor_get_pdg
```

Return the flavor of the antiparticle

⟨*Flavors: flavor: TBP*⟩+≡

```
procedure :: get_pdg_anti => flavor_get_pdg_anti
```

⟨*Flavors: procedures*⟩+≡

```
elemental function flavor_get_pdg_anti (flv) result (f)
  integer :: f
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     if (flv%field_data%has_antiparticle ()) then
        f = -flv%f
     else
        f = flv%f
     end if
  else
     f = 0
  end if
end function flavor_get_pdg_anti
```

Absolute value:

⟨*Flavors: flavor: TBP*⟩+≡

```
procedure :: get_pdg_abs => flavor_get_pdg_abs
```

⟨*Flavors: procedures*⟩+≡

```
elemental function flavor_get_pdg_abs (flv) result (f)
  integer :: f
  class(flavor_t), intent(in) :: flv
  f = abs (flv%f)
end function flavor_get_pdg_abs
```

Generic properties

⟨*Flavors: flavor: TBP*⟩+≡

```
procedure :: is_visible => flavor_is_visible
procedure :: is_parton => flavor_is_parton
procedure :: is_beam_remnant => flavor_is_beam_remnant
procedure :: is_gauge => flavor_is_gauge
procedure :: is_left_handed => flavor_is_left_handed
procedure :: is_right_handed => flavor_is_right_handed
procedure :: is_antiparticle => flavor_is_antiparticle
procedure :: has_antiparticle => flavor_has_antiparticle
procedure :: is_stable => flavor_is_stable
procedure :: get_decays => flavor_get_decays
procedure :: decays_isotropically => flavor_decays_isotropically
procedure :: decays_diagonal => flavor_decays_diagonal
procedure :: has_decay_helicity => flavor_has_decay_helicity
procedure :: get_decay_helicity => flavor_get_decay_helicity
procedure :: is_polarized => flavor_is_polarized
```

⟨*Flavors: procedures*⟩+≡

```
elemental function flavor_is_visible (flv) result (flag)
  logical :: flag
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     flag = flv%field_data%is_visible ()
```

```
  else
     flag = .false.
  end if
end function flavor_is_visible

elemental function flavor_is_parton (flv) result (flag)
  logical :: flag
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     flag = flv%field_data%is_parton ()
  else
     flag = .false.
  end if
end function flavor_is_parton

elemental function flavor_is_beam_remnant (flv) result (flag)
  logical :: flag
  class(flavor_t), intent(in) :: flv
  select case (abs (flv%f))
  case (HADRON_REMNANT, &
       HADRON_REMNANT_SINGLET, HADRON_REMNANT_TRIPLET, HADRON_REMNANT_OCTET)
     flag = .true.
  case default
     flag = .false.
  end select
end function flavor_is_beam_remnant

elemental function flavor_is_gauge (flv) result (flag)
  logical :: flag
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     flag = flv%field_data%is_gauge ()
  else
     flag = .false.
  end if
end function flavor_is_gauge

elemental function flavor_is_left_handed (flv) result (flag)
  logical :: flag
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     if (flv%f > 0) then
        flag = flv%field_data%is_left_handed ()
     else
        flag = flv%field_data%is_right_handed ()
     end if
  else
     flag = .false.
  end if
end function flavor_is_left_handed

elemental function flavor_is_right_handed (flv) result (flag)
  logical :: flag
  class(flavor_t), intent(in) :: flv
```

```
  if (associated (flv%field_data)) then
     if (flv%f > 0) then
        flag = flv%field_data%is_right_handed ()
     else
        flag = flv%field_data%is_left_handed ()
     end if
  else
     flag = .false.
  end if
end function flavor_is_right_handed

elemental function flavor_is_antiparticle (flv) result (flag)
  logical :: flag
  class(flavor_t), intent(in) :: flv
  flag = flv%f < 0
end function flavor_is_antiparticle

elemental function flavor_has_antiparticle (flv) result (flag)
  logical :: flag
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     flag = flv%field_data%has_antiparticle ()
  else
     flag = .false.
  end if
end function flavor_has_antiparticle

elemental function flavor_is_stable (flv) result (flag)
  logical :: flag
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     flag = flv%field_data%is_stable (anti = flv%f < 0)
  else
     flag = .true.
  end if
end function flavor_is_stable

subroutine flavor_get_decays (flv, decay)
  class(flavor_t), intent(in) :: flv
  type(string_t), dimension(:), intent(out), allocatable :: decay
  logical :: anti
  anti = flv%f < 0
  if (.not. flv%field_data%is_stable (anti)) then
     call flv%field_data%get_decays (decay, anti)
  end if
end subroutine flavor_get_decays

elemental function flavor_decays_isotropically (flv) result (flag)
  logical :: flag
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     flag = flv%field_data%decays_isotropically (anti = flv%f < 0)
  else
     flag = .true.
```

```
  end if
end function flavor_decays_isotropically

elemental function flavor_decays_diagonal (flv) result (flag)
  logical :: flag
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     flag = flv%field_data%decays_diagonal (anti = flv%f < 0)
  else
     flag = .true.
  end if
end function flavor_decays_diagonal

elemental function flavor_has_decay_helicity (flv) result (flag)
  logical :: flag
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     flag = flv%field_data%has_decay_helicity (anti = flv%f < 0)
  else
     flag = .false.
  end if
end function flavor_has_decay_helicity

elemental function flavor_get_decay_helicity (flv) result (hel)
  integer :: hel
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     hel = flv%field_data%decay_helicity (anti = flv%f < 0)
  else
     hel = 0
  end if
end function flavor_get_decay_helicity

elemental function flavor_is_polarized (flv) result (flag)
  logical :: flag
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     flag = flv%field_data%is_polarized (anti = flv%f < 0)
  else
     flag = .false.
  end if
end function flavor_is_polarized
```

Names:

⟨*Flavors: flavor: TBP*⟩+≡

```
procedure :: get_name => flavor_get_name
procedure :: get_tex_name => flavor_get_tex_name
```

⟨*Flavors: procedures*⟩+≡

```
elemental function flavor_get_name (flv) result (name)
  type(string_t) :: name
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     name = flv%field_data%get_name (flv%f < 0)
```

```
    else
       name = "?"
    end if
  end function flavor_get_name

  elemental function flavor_get_tex_name (flv) result (name)
    type(string_t) :: name
    class(flavor_t), intent(in) :: flv
    if (associated (flv%field_data)) then
       name = flv%field_data%get_tex_name (flv%f < 0)
    else
       name = "?"
    end if
  end function flavor_get_tex_name
```

⟨*Flavors: flavor: TBP*⟩+≡

```
  procedure :: get_spin_type => flavor_get_spin_type
  procedure :: get_multiplicity => flavor_get_multiplicity
  procedure :: get_isospin_type => flavor_get_isospin_type
  procedure :: get_charge_type => flavor_get_charge_type
  procedure :: get_color_type => flavor_get_color_type
```

⟨*Flavors: procedures*⟩+≡

```
  elemental function flavor_get_spin_type (flv) result (type)
    integer :: type
    class(flavor_t), intent(in) :: flv
    if (associated (flv%field_data)) then
       type = flv%field_data%get_spin_type ()
    else
       type = 1
    end if
  end function flavor_get_spin_type

  elemental function flavor_get_multiplicity (flv) result (type)
    integer :: type
    class(flavor_t), intent(in) :: flv
    if (associated (flv%field_data)) then
       type = flv%field_data%get_multiplicity ()
    else
       type = 1
    end if
  end function flavor_get_multiplicity

  elemental function flavor_get_isospin_type (flv) result (type)
    integer :: type
    class(flavor_t), intent(in) :: flv
    if (associated (flv%field_data)) then
       type = flv%field_data%get_isospin_type ()
    else
       type = 1
    end if
  end function flavor_get_isospin_type

  elemental function flavor_get_charge_type (flv) result (type)
```

```
    integer :: type
    class(flavor_t), intent(in) :: flv
    if (associated (flv%field_data)) then
       type = flv%field_data%get_charge_type ()
    else
       type = 1
    end if
  end function flavor_get_charge_type

  elemental function flavor_get_color_type (flv) result (type)
    integer :: type
    class(flavor_t), intent(in) :: flv
    if (associated (flv%field_data)) then
       if (flavor_is_antiparticle (flv)) then
          type = - flv%field_data%get_color_type ()
       else
          type = flv%field_data%get_color_type ()
       end if
       select case (type)
       case (-1,-8);  type = abs (type)
       end select
    else
       type = 1
    end if
  end function flavor_get_color_type
```

These functions return real values:

⟨*Flavors: flavor: TBP*⟩+≡

```
  procedure :: get_charge => flavor_get_charge
  procedure :: get_mass => flavor_get_mass
  procedure :: get_width => flavor_get_width
  procedure :: get_isospin => flavor_get_isospin
```

⟨*Flavors: procedures*⟩+≡

```
  elemental function flavor_get_charge (flv) result (charge)
    real(default) :: charge
    class(flavor_t), intent(in) :: flv
    integer :: charge_type
    if (associated (flv%field_data)) then
       charge_type = flv%get_charge_type ()
       if (charge_type == 0 .or. charge_type == 1) then
          charge = 0
       else
          if (flavor_is_antiparticle (flv)) then
             charge = - flv%field_data%get_charge ()
          else
             charge = flv%field_data%get_charge ()
          end if
       end if
    else
       charge = 0
    end if
  end function flavor_get_charge
```

```
elemental function flavor_get_mass (flv) result (mass)
  real(default) :: mass
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     mass = flv%field_data%get_mass ()
  else
     mass = 0
  end if
end function flavor_get_mass

elemental function flavor_get_width (flv) result (width)
  real(default) :: width
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     width = flv%field_data%get_width ()
  else
     width = 0
  end if
end function flavor_get_width

elemental function flavor_get_isospin (flv) result (isospin)
  real(default) :: isospin
  class(flavor_t), intent(in) :: flv
  if (associated (flv%field_data)) then
     if (flavor_is_antiparticle (flv)) then
        isospin = - flv%field_data%get_isospin ()
     else
        isospin = flv%field_data%get_isospin ()
     end if
  else
     isospin = 0
  end if
end function flavor_get_isospin
```

Comparisons

If one of the flavors is undefined, the other defined, they match.

⟨*Flavors: flavor: TBP*⟩+≡

```
generic :: operator(.match.) => flavor_match
generic :: operator(==) => flavor_eq
generic :: operator(/=) => flavor_neq
procedure, private :: flavor_match
procedure, private :: flavor_eq
procedure, private :: flavor_neq
```

⟨*Flavors: procedures*⟩+≡

```
elemental function flavor_match (flv1, flv2) result (eq)
  logical :: eq
  class(flavor_t), intent(in) :: flv1, flv2
  if (flv1%f /= UNDEFINED .and. flv2%f /= UNDEFINED) then
     eq = flv1%f == flv2%f
  else
     eq = .true.
```

```
    end if
  end function flavor_match

  elemental function flavor_eq (flv1, flv2) result (eq)
    logical :: eq
    class(flavor_t), intent(in) :: flv1, flv2
    if (flv1%f /= UNDEFINED .and. flv2%f /= UNDEFINED) then
       eq = flv1%f == flv2%f
    else if (flv1%f == UNDEFINED .and. flv2%f == UNDEFINED) then
       eq = .true.
    else
       eq = .false.
    end if
  end function flavor_eq
```

⟨*Flavors: procedures*⟩+≡

```
  elemental function flavor_neq (flv1, flv2) result (neq)
    logical :: neq
    class(flavor_t), intent(in) :: flv1, flv2
    if (flv1%f /= UNDEFINED .and. flv2%f /= UNDEFINED) then
       neq = flv1%f /= flv2%f
    else if (flv1%f == UNDEFINED .and. flv2%f == UNDEFINED) then
       neq = .false.
    else
       neq = .true.
    end if
  end function flavor_neq
```

Tools

Merge two flavor indices. This works only if both are equal or either one is undefined, because we have no off-diagonal flavor entries. Otherwise, generate an invalid flavor.

We cannot use elemental procedures because of the pointer component.

⟨*Flavors: public*⟩+≡

```
  public :: operator(.merge.)
```

⟨*Flavors: interfaces*⟩≡

```
  interface operator(.merge.)
     module procedure merge_flavors0
     module procedure merge_flavors1
  end interface
```

⟨*Flavors: procedures*⟩+≡

```
  function merge_flavors0 (flv1, flv2) result (flv)
    type(flavor_t) :: flv
    type(flavor_t), intent(in) :: flv1, flv2
    if (flavor_is_defined (flv1) .and. flavor_is_defined (flv2)) then
       if (flv1 == flv2) then
          flv = flv1
       else
          flv%f = INVALID
```

```
      end if
    else if (flavor_is_defined (flv1)) then
      flv = flv1
    else if (flavor_is_defined (flv2)) then
      flv = flv2
    end if
  end function merge_flavors0

  function merge_flavors1 (flv1, flv2) result (flv)
    type(flavor_t), dimension(:), intent(in) :: flv1, flv2
    type(flavor_t), dimension(size(flv1)) :: flv
    integer :: i
    do i = 1, size (flv1)
      flv(i) = flv1(i) .merge. flv2(i)
    end do
  end function merge_flavors1
```

Generate consecutive color indices for a given flavor. The indices are counted starting with the stored value of c, so new indices are created each time this (impure) function is called. The counter can be reset by the optional argument `c_seed` if desired. The optional flag `reverse` is used only for octets. If set, the color and anticolor entries of the octet particle are exchanged.

⟨*Flavors: public*⟩+≡

```
  public :: color_from_flavor
```

⟨*Flavors: interfaces*⟩+≡

```
  interface color_from_flavor
     module procedure color_from_flavor0
     module procedure color_from_flavor1
  end interface
```

⟨*Flavors: procedures*⟩+≡

```
  function color_from_flavor0 (flv, c_seed, reverse) result (col)
    type(color_t) :: col
    type(flavor_t), intent(in) :: flv
    integer, intent(in), optional :: c_seed
    logical, intent(in), optional :: reverse
    integer, save :: c = 1
    logical :: rev
    if (present (c_seed))  c = c_seed
    rev = .false.;  if (present (reverse)) rev = reverse
    select case (flavor_get_color_type (flv))
    case (1)
      call col%init ()
    case (3)
      call col%init ([c]);  c = c + 1
    case (-3)
      call col%init ([-c]);  c = c + 1
    case (8)
      if (rev) then
        call col%init ([c+1, -c]);  c = c + 2
      else
        call col%init ([c, -(c+1)]);  c = c + 2
      end if
```

```
    end select
  end function color_from_flavor0

  function color_from_flavor1 (flv, c_seed, reverse) result (col)
    type(flavor_t), dimension(:), intent(in) :: flv
    integer, intent(in), optional :: c_seed
    logical, intent(in), optional :: reverse
    type(color_t), dimension(size(flv)) :: col
    integer :: i
    col(1) = color_from_flavor0 (flv(1), c_seed, reverse)
    do i = 2, size (flv)
       col(i) = color_from_flavor0 (flv(i), reverse=reverse)
    end do
  end function color_from_flavor1
```

This procedure returns the flavor object for the antiparticle. The antiparticle code may either be the same code or its negative.

⟨*Flavors: flavor: TBP*⟩+≡

```
  procedure :: anti => flavor_anti
```

⟨*Flavors: procedures*⟩+≡

```
  function flavor_anti (flv) result (aflv)
    type(flavor_t) :: aflv
    class(flavor_t), intent(in) :: flv
    if (flavor_has_antiparticle (flv)) then
       aflv%f = - flv%f
    else
       aflv%f = flv%f
    end if
    aflv%field_data => flv%field_data
  end function flavor_anti
```

10.6 Quantum numbers

This module collects helicity, color, and flavor in a single type and defines procedures

⟨quantum_numbers.f90⟩≡
⟨*File header*⟩

```
module quantum_numbers

  use io_units
  use model_data
  use helicities
  use colors
  use flavors
```

⟨*Standard module head*⟩

⟨*Quantum numbers: public*⟩

⟨*Quantum numbers: types*⟩

⟨*Quantum numbers: interfaces*⟩

```
contains
```

⟨*Quantum numbers: procedures*⟩

```
end module quantum_numbers
```

10.6.1 The quantum number type

⟨*Quantum numbers: public*⟩≡

```
public :: quantum_numbers_t
```

⟨*Quantum numbers: types*⟩≡

```
type :: quantum_numbers_t
   private
   type(flavor_t) :: f
   type(color_t) :: c
   type(helicity_t) :: h
   integer :: sub = 0
 contains
 ⟨Quantum numbers: quantum numbers: TBP⟩
end type quantum_numbers_t
```

Define quantum numbers: Initializer form. All arguments may be present or absent.

Some elemental initializers are impure because they set the `flv` component. This implies transfer of a pointer behind the scenes.

⟨*Quantum numbers: quantum numbers: TBP*⟩≡

```
generic :: init => &
   quantum_numbers_init_f, &
   quantum_numbers_init_c, &
```

```
    quantum_numbers_init_h, &
    quantum_numbers_init_fc, &
    quantum_numbers_init_fh, &
    quantum_numbers_init_ch, &
    quantum_numbers_init_fch, &
    quantum_numbers_init_fs, &
    quantum_numbers_init_fhs, &
    quantum_numbers_init_fcs, &
    quantum_numbers_init_fhcs
procedure, private :: quantum_numbers_init_f
procedure, private :: quantum_numbers_init_c
procedure, private :: quantum_numbers_init_h
procedure, private :: quantum_numbers_init_fc
procedure, private :: quantum_numbers_init_fh
procedure, private :: quantum_numbers_init_ch
procedure, private :: quantum_numbers_init_fch
procedure, private :: quantum_numbers_init_fs
procedure, private :: quantum_numbers_init_fhs
procedure, private :: quantum_numbers_init_fcs
procedure, private :: quantum_numbers_init_fhcs
```

⟨*Quantum numbers: procedures*⟩≡

```
impure elemental subroutine quantum_numbers_init_f (qn, flv)
  class(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  qn%f = flv
  call qn%c%undefine ()
  call qn%h%undefine ()
  qn%sub = 0
end subroutine quantum_numbers_init_f

impure elemental subroutine quantum_numbers_init_c (qn, col)
  class(quantum_numbers_t), intent(out) :: qn
  type(color_t), intent(in) :: col
  call qn%f%undefine ()
  qn%c = col
  call qn%h%undefine ()
  qn%sub = 0
end subroutine quantum_numbers_init_c

impure elemental subroutine quantum_numbers_init_h (qn, hel)
  class(quantum_numbers_t), intent(out) :: qn
  type(helicity_t), intent(in) :: hel
  call qn%f%undefine ()
  call qn%c%undefine ()
  qn%h = hel
  qn%sub = 0
end subroutine quantum_numbers_init_h

impure elemental subroutine quantum_numbers_init_fc (qn, flv, col)
  class(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  type(color_t), intent(in) :: col
  qn%f = flv
  qn%c = col
```

```
  call qn%h%undefine ()
  qn%sub = 0
end subroutine quantum_numbers_init_fc

impure elemental subroutine quantum_numbers_init_fh (qn, flv, hel)
  class(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  type(helicity_t), intent(in) :: hel
  qn%f = flv
  call qn%c%undefine ()
  qn%h = hel
  qn%sub = 0
end subroutine quantum_numbers_init_fh

impure elemental subroutine quantum_numbers_init_ch (qn, col, hel)
  class(quantum_numbers_t), intent(out) :: qn
  type(color_t), intent(in) :: col
  type(helicity_t), intent(in) :: hel
  call qn%f%undefine ()
  qn%c = col
  qn%h = hel
  qn%sub = 0
end subroutine quantum_numbers_init_ch

impure elemental subroutine quantum_numbers_init_fch (qn, flv, col, hel)
  class(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  type(color_t), intent(in) :: col
  type(helicity_t), intent(in) :: hel
  qn%f = flv
  qn%c = col
  qn%h = hel
  qn%sub = 0
end subroutine quantum_numbers_init_fch

impure elemental subroutine quantum_numbers_init_fs (qn, flv, sub)
  class(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  integer, intent(in) :: sub
  qn%f = flv; qn%sub = sub
end subroutine quantum_numbers_init_fs

impure elemental subroutine quantum_numbers_init_fhs (qn, flv, hel, sub)
  class(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  type(helicity_t), intent(in) :: hel
  integer, intent(in) :: sub
  qn%f = flv; qn%h = hel; qn%sub = sub
end subroutine quantum_numbers_init_fhs

impure elemental subroutine quantum_numbers_init_fcs (qn, flv, col, sub)
  class(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  type(color_t), intent(in) :: col
```

```
  integer, intent(in) :: sub
  qn%f = flv; qn%c = col; qn%sub = sub
end subroutine quantum_numbers_init_fcs

impure elemental subroutine quantum_numbers_init_fhcs (qn, flv, hel, col, sub)
  class(quantum_numbers_t), intent(out) :: qn
  type(flavor_t), intent(in) :: flv
  type(helicity_t), intent(in) :: hel
  type(color_t), intent(in) :: col
  integer, intent(in) :: sub
  qn%f = flv; qn%h = hel; qn%c = col; qn%sub = sub
end subroutine quantum_numbers_init_fhcs
```

10.6.2 I/O

Write the quantum numbers in condensed form, enclosed by square brackets. Color is written only if nontrivial. For convenience, introduce also an array version.

If the `col_verbose` option is set, show the quantum number color also if it is zero, but defined. Otherwise, suppress zero color.

⟨*Quantum numbers: public*⟩+≡

```
public :: quantum_numbers_write
```

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡

```
procedure :: write => quantum_numbers_write_single
```

⟨*Quantum numbers: interfaces*⟩≡

```
interface quantum_numbers_write
   module procedure quantum_numbers_write_single
   module procedure quantum_numbers_write_array
end interface
```

⟨*Quantum numbers: procedures*⟩+≡

```
subroutine quantum_numbers_write_single (qn, unit, col_verbose)
  class(quantum_numbers_t), intent(in) :: qn
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: col_verbose
  integer :: u
  logical :: col_verb
  u = given_output_unit (unit);  if (u < 0)  return
  col_verb = .false.;  if (present (col_verbose)) col_verb = col_verbose
  write (u, "(A)", advance = "no")  "["
  if (qn%f%is_defined ()) then
     call qn%f%write (u)
     if (qn%c%is_nonzero () .or. qn%h%is_defined ()) &
          write (u, "(1x)", advance = "no")
  end if
  if (col_verb) then
     if (qn%c%is_defined () .or. qn%c%is_ghost ()) then
        call color_write (qn%c, u)
        if (qn%h%is_defined ())  write (u, "(1x)", advance = "no")
     end if
  else
     if (qn%c%is_nonzero () .or. qn%c%is_ghost ()) then
```

```
         call color_write (qn%c, u)
         if (qn%h%is_defined ())  write (u, "(1x)", advance = "no")
      end if
   end if
   if (qn%h%is_defined ()) then
      call qn%h%write (u)
   end if
   if (qn%sub > 0) &
      write (u, "(A,I1)", advance = "no") " SUB = ", qn%sub
   write (u, "(A)", advance="no")  "]"
end subroutine quantum_numbers_write_single

subroutine quantum_numbers_write_array (qn, unit, col_verbose)
   type(quantum_numbers_t), dimension(:), intent(in) :: qn
   integer, intent(in), optional :: unit
   logical, intent(in), optional :: col_verbose
   integer :: i
   integer :: u
   logical :: col_verb
   u = given_output_unit (unit);  if (u < 0)  return
   col_verb = .false.;  if (present (col_verbose)) col_verb = col_verbose
   write (u, "(A)", advance="no")  "["
   do i = 1, size (qn)
      if (i > 1)  write (u, "(A)", advance="no")  " / "
      if (qn(i)%f%is_defined ()) then
         call qn(i)%f%write (u)
         if (qn(i)%c%is_nonzero () .or. qn(i)%h%is_defined ()) &
              write (u, "(1x)", advance="no")
      end if
      if (col_verb) then
         if (qn(i)%c%is_defined () .or. qn(i)%c%is_ghost ()) then
            call color_write (qn(i)%c, u)
            if (qn(i)%h%is_defined ())  write (u, "(1x)", advance="no")
         end if
      else
         if (qn(i)%c%is_nonzero () .or. qn(i)%c%is_ghost ()) then
            call color_write (qn(i)%c, u)
            if (qn(i)%h%is_defined ())  write (u, "(1x)", advance="no")
         end if
      end if
      if (qn(i)%h%is_defined ()) then
         call qn(i)%h%write (u)
      end if
      if (qn(i)%sub > 0) &
         write (u, "(A,I1)", advance = "no") " SUB = ", qn(i)%sub
   end do
   write (u, "(A)", advance = "no")  "]"
end subroutine quantum_numbers_write_array
```

Binary I/O.

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡

```
procedure :: write_raw => quantum_numbers_write_raw
procedure :: read_raw => quantum_numbers_read_raw
```

⟨*Quantum numbers: procedures*⟩+≡

```
subroutine quantum_numbers_write_raw (qn, u)
  class(quantum_numbers_t), intent(in) :: qn
  integer, intent(in) :: u
  call qn%f%write_raw (u)
  call qn%c%write_raw (u)
  call qn%h%write_raw (u)
end subroutine quantum_numbers_write_raw

subroutine quantum_numbers_read_raw (qn, u, iostat)
  class(quantum_numbers_t), intent(out) :: qn
  integer, intent(in) :: u
  integer, intent(out), optional :: iostat
  call qn%f%read_raw (u, iostat=iostat)
  call qn%c%read_raw (u, iostat=iostat)
  call qn%h%read_raw (u, iostat=iostat)
end subroutine quantum_numbers_read_raw
```

10.6.3 Accessing contents

Color and helicity can be done by elemental functions. Flavor needs impure elemental. We export also the functions directly, this allows us to avoid temporaries in some places.

⟨*Quantum numbers: public*⟩+≡

```
public :: quantum_numbers_get_flavor
public :: quantum_numbers_get_color
public :: quantum_numbers_get_helicity
```

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡

```
procedure :: get_flavor => quantum_numbers_get_flavor
procedure :: get_color => quantum_numbers_get_color
procedure :: get_helicity => quantum_numbers_get_helicity
procedure :: get_sub => quantum_numbers_get_sub
```

⟨*Quantum numbers: procedures*⟩+≡

```
impure elemental function quantum_numbers_get_flavor (qn) result (flv)
  type(flavor_t) :: flv
  class(quantum_numbers_t), intent(in) :: qn
  flv = qn%f
end function quantum_numbers_get_flavor

elemental function quantum_numbers_get_color (qn) result (col)
  type(color_t) :: col
  class(quantum_numbers_t), intent(in) :: qn
  col = qn%c
end function quantum_numbers_get_color

elemental function quantum_numbers_get_helicity (qn) result (hel)
  type(helicity_t) :: hel
  class(quantum_numbers_t), intent(in) :: qn
  hel = qn%h
end function quantum_numbers_get_helicity
```

```
elemental function quantum_numbers_get_sub (qn) result (sub)
  integer :: sub
  class(quantum_numbers_t), intent(in) :: qn
  sub = qn%sub
end function quantum_numbers_get_sub
```

This just resets the ghost property of the color part:

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡

```
procedure :: set_color_ghost => quantum_numbers_set_color_ghost
```

⟨*Quantum numbers: procedures*⟩+≡

```
elemental subroutine quantum_numbers_set_color_ghost (qn, ghost)
  class(quantum_numbers_t), intent(inout) :: qn
  logical, intent(in) :: ghost
  call qn%c%set_ghost (ghost)
end subroutine quantum_numbers_set_color_ghost
```

Assign a model to the flavor part of quantum numbers.

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡

```
procedure :: set_model => quantum_numbers_set_model
```

⟨*Quantum numbers: procedures*⟩+≡

```
impure elemental subroutine quantum_numbers_set_model (qn, model)
  class(quantum_numbers_t), intent(inout) :: qn
  class(model_data_t), intent(in), target :: model
  call qn%f%set_model (model)
end subroutine quantum_numbers_set_model
```

Set the `radiated` flag for the flavor component.

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡

```
procedure :: tag_radiated => quantum_numbers_tag_radiated
```

⟨*Quantum numbers: procedures*⟩+≡

```
elemental subroutine quantum_numbers_tag_radiated (qn)
  class(quantum_numbers_t), intent(inout) :: qn
  call qn%f%tag_radiated ()
end subroutine quantum_numbers_tag_radiated
```

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡

```
procedure :: set_subtraction_index => quantum_numbers_set_subtraction_index
```

⟨*Quantum numbers: procedures*⟩+≡

```
elemental subroutine quantum_numbers_set_subtraction_index (qn, i)
  class(quantum_numbers_t), intent(inout) :: qn
  integer, intent(in) :: i
  qn%sub = i
end subroutine quantum_numbers_set_subtraction_index
```

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡

```
procedure :: get_subtraction_index => quantum_numbers_get_subtraction_index
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function quantum_numbers_get_subtraction_index (qn) result (sub)
  integer :: sub
  class(quantum_numbers_t), intent(in) :: qn
  sub = qn%sub
end function quantum_numbers_get_subtraction_index
```

This is a convenience function: return the color type for the flavor (array).

Note: keep the public version temporarily, this will be used in a complicated expression which triggers a compiler bug (nagfor 5.3) in the TBP version.

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_get_color_type
```

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡
```
procedure :: get_color_type => quantum_numbers_get_color_type
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function quantum_numbers_get_color_type (qn) result (color_type)
  integer :: color_type
  class(quantum_numbers_t), intent(in) :: qn
  color_type = qn%f%get_color_type ()
end function quantum_numbers_get_color_type
```

10.6.4 Predicates

Check if the flavor index is valid (including UNDEFINED).

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡
```
procedure :: are_valid => quantum_numbers_are_valid
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function quantum_numbers_are_valid (qn) result (valid)
  logical :: valid
  class(quantum_numbers_t), intent(in) :: qn
  valid = qn%f%is_valid ()
end function quantum_numbers_are_valid
```

Check if the flavor part has its particle-data pointer associated (debugging aid).

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡
```
procedure :: are_associated => quantum_numbers_are_associated
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function quantum_numbers_are_associated (qn) result (flag)
  logical :: flag
  class(quantum_numbers_t), intent(in) :: qn
  flag = qn%f%is_associated ()
end function quantum_numbers_are_associated
```

Check if the helicity and color quantum numbers are diagonal. (Unpolarized/colorless also counts as diagonal.) Flavor is diagonal by definition.

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡
```
procedure :: are_diagonal => quantum_numbers_are_diagonal
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function quantum_numbers_are_diagonal (qn) result (diagonal)
  logical :: diagonal
  class(quantum_numbers_t), intent(in) :: qn
  diagonal = qn%h%is_diagonal () .and. qn%c%is_diagonal ()
end function quantum_numbers_are_diagonal
```

Check if the color part has the ghost property.

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡
```
procedure :: is_color_ghost => quantum_numbers_is_color_ghost
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function quantum_numbers_is_color_ghost (qn) result (ghost)
  logical :: ghost
  class(quantum_numbers_t), intent(in) :: qn
  ghost = qn%c%is_ghost ()
end function quantum_numbers_is_color_ghost
```

10.6.5 Comparisons

Matching and equality is derived from the individual quantum numbers. The variant `fhmatch` matches only flavor and helicity. The variant `dhmatch` matches only diagonal helicity, if the matching helicity is undefined.

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡
```
generic :: operator(.match.) => quantum_numbers_match
generic :: operator(.fmatch.) => quantum_numbers_match_f
generic :: operator(.fhmatch.) => quantum_numbers_match_fh
generic :: operator(.dhmatch.) => quantum_numbers_match_hel_diag
generic :: operator(==) => quantum_numbers_eq
generic :: operator(/=) => quantum_numbers_neq
procedure, private :: quantum_numbers_match
procedure, private :: quantum_numbers_match_f
procedure, private :: quantum_numbers_match_fh
procedure, private :: quantum_numbers_match_hel_diag
procedure, private :: quantum_numbers_eq
procedure, private :: quantum_numbers_neq
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function quantum_numbers_match (qn1, qn2) result (match)
  logical :: match
  class(quantum_numbers_t), intent(in) :: qn1, qn2
  match = (qn1%f .match. qn2%f) .and. &
        (qn1%c .match. qn2%c) .and. &
        (qn1%h .match. qn2%h)
end function quantum_numbers_match

elemental function quantum_numbers_match_f (qn1, qn2) result (match)
  logical :: match
  class(quantum_numbers_t), intent(in) :: qn1, qn2
  match = (qn1%f .match. qn2%f)
end function quantum_numbers_match_f
```

```
elemental function quantum_numbers_match_fh (qn1, qn2) result (match)
  logical :: match
  class(quantum_numbers_t), intent(in) :: qn1, qn2
  match = (qn1%f .match. qn2%f) .and. &
       (qn1%h .match. qn2%h)
end function quantum_numbers_match_fh

elemental function quantum_numbers_match_hel_diag (qn1, qn2) result (match)
  logical :: match
  class(quantum_numbers_t), intent(in) :: qn1, qn2
  match = (qn1%f .match. qn2%f) .and. &
       (qn1%c .match. qn2%c) .and. &
       (qn1%h .dmatch. qn2%h)
end function quantum_numbers_match_hel_diag

elemental function quantum_numbers_eq (qn1, qn2) result (eq)
  logical :: eq
  class(quantum_numbers_t), intent(in) :: qn1, qn2
  eq = (qn1%f == qn2%f) .and. &
       (qn1%c == qn2%c) .and. &
       (qn1%h == qn2%h) .and. &
       (qn1%sub == qn2%sub)
end function quantum_numbers_eq

elemental function quantum_numbers_neq (qn1, qn2) result (neq)
  logical :: neq
  class(quantum_numbers_t), intent(in) :: qn1, qn2
  neq = (qn1%f /= qn2%f) .or. &
       (qn1%c /= qn2%c) .or. &
       (qn1%h /= qn2%h) .or. &
       (qn1%sub /= qn2%sub)
end function quantum_numbers_neq
```

⟨*Quantum numbers: public*⟩+≡

```
public :: assignment(=)
```

⟨*Quantum numbers: interfaces*⟩+≡

```
interface assignment(=)
   module procedure quantum_numbers_assign
end interface
```

⟨*Quantum numbers: procedures*⟩+≡

```
subroutine quantum_numbers_assign (qn_out, qn_in)
  type(quantum_numbers_t), intent(out) :: qn_out
  type(quantum_numbers_t), intent(in) :: qn_in
  qn_out%f = qn_in%f
  qn_out%c = qn_in%c
  qn_out%h = qn_in%h
  qn_out%sub = qn_in%sub
end subroutine quantum_numbers_assign
```

Two sets of quantum numbers are compatible if the individual quantum numbers are compatible, depending on the mask. Flavor has to match, regardless of the flavor mask.

If the color flag is set, color is compatible if the ghost property is identical. If the color flag is unset, color has to be identical. I.e., if the flag is set, the color amplitudes can interfere. If it is not set, they must be identical, and there must be no ghost. The latter property is used for expanding physical color flows.

Helicity is compatible if the mask is unset, otherwise it has to match. This determines if two amplitudes can be multiplied (no mask) or traced (mask).

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_are_compatible
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function quantum_numbers_are_compatible (qn1, qn2, mask) &
    result (flag)
  logical :: flag
  type(quantum_numbers_t), intent(in) :: qn1, qn2
  type(quantum_numbers_mask_t), intent(in) :: mask
  if (mask%h .or. mask%hd) then
     flag = (qn1%f .match. qn2%f) .and. (qn1%h .match. qn2%h)
  else
     flag = (qn1%f .match. qn2%f)
  end if
  if (mask%c) then
     flag = flag .and. (qn1%c%is_ghost () .eqv. qn2%c%is_ghost ())
  else
     flag = flag .and. &
          .not. (qn1%c%is_ghost () .or. qn2%c%is_ghost ()) .and. &
          (qn1%c == qn2%c)
  end if
end function quantum_numbers_are_compatible
```

This is the analog for a single quantum-number set. We just check for color ghosts; they are excluded if the color mask is unset (color-flow expansion).

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_are_physical
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental function quantum_numbers_are_physical (qn, mask) result (flag)
  logical :: flag
  type(quantum_numbers_t), intent(in) :: qn
  type(quantum_numbers_mask_t), intent(in) :: mask
  if (mask%c) then
     flag = .true.
  else
     flag = .not. qn%c%is_ghost ()
  end if
end function quantum_numbers_are_physical
```

10.6.6 Operations

Inherited from the color component: reassign color indices in canonical order.

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_canonicalize_color
```

⟨*Quantum numbers: procedures*⟩+≡
```
subroutine quantum_numbers_canonicalize_color (qn)
  type(quantum_numbers_t), dimension(:), intent(inout) :: qn
  call color_canonicalize (qn%c)
end subroutine quantum_numbers_canonicalize_color
```

Inherited from the color component: make a color map for two matching quantum-number arrays.

⟨*Quantum numbers: public*⟩+≡
```
public :: make_color_map
```

⟨*Quantum numbers: interfaces*⟩+≡
```
interface make_color_map
   module procedure quantum_numbers_make_color_map
end interface make_color_map
```

⟨*Quantum numbers: procedures*⟩+≡
```
subroutine quantum_numbers_make_color_map (map, qn1, qn2)
  integer, dimension(:,:), intent(out), allocatable :: map
  type(quantum_numbers_t), dimension(:), intent(in) :: qn1, qn2
  call make_color_map (map, qn1%c, qn2%c)
end subroutine quantum_numbers_make_color_map
```

Inherited from the color component: translate the color part using a color-map array

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_translate_color
```

⟨*Quantum numbers: interfaces*⟩+≡
```
interface quantum_numbers_translate_color
   module procedure quantum_numbers_translate_color0
   module procedure quantum_numbers_translate_color1
end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
subroutine quantum_numbers_translate_color0 (qn, map, offset)
  type(quantum_numbers_t), intent(inout) :: qn
  integer, dimension(:,:), intent(in) :: map
  integer, intent(in), optional :: offset
  call color_translate (qn%c, map, offset)
end subroutine quantum_numbers_translate_color0

subroutine quantum_numbers_translate_color1 (qn, map, offset)
  type(quantum_numbers_t), dimension(:), intent(inout) :: qn
  integer, dimension(:,:), intent(in) :: map
  integer, intent(in), optional :: offset
  call color_translate (qn%c, map, offset)
end subroutine quantum_numbers_translate_color1
```

Inherited from the color component: return the color index with highest absolute value.

Since the algorithm is not elemental, we keep the separate procedures for different array rank.

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_numbers_get_max_color_value
```

⟨*Quantum numbers: interfaces*⟩+≡
```
interface quantum_numbers_get_max_color_value
   module procedure quantum_numbers_get_max_color_value0
   module procedure quantum_numbers_get_max_color_value1
   module procedure quantum_numbers_get_max_color_value2
end interface
```

⟨*Quantum numbers: procedures*⟩+≡
```
pure function quantum_numbers_get_max_color_value0 (qn) result (cmax)
  integer :: cmax
  type(quantum_numbers_t), intent(in) :: qn
  cmax = color_get_max_value (qn%c)
end function quantum_numbers_get_max_color_value0

pure function quantum_numbers_get_max_color_value1 (qn) result (cmax)
  integer :: cmax
  type(quantum_numbers_t), dimension(:), intent(in) :: qn
  cmax = color_get_max_value (qn%c)
end function quantum_numbers_get_max_color_value1

pure function quantum_numbers_get_max_color_value2 (qn) result (cmax)
  integer :: cmax
  type(quantum_numbers_t), dimension(:,:), intent(in) :: qn
  cmax = color_get_max_value (qn%c)
end function quantum_numbers_get_max_color_value2
```

Inherited from the color component: add an offset to the indices of the color part

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡
```
procedure :: add_color_offset => quantum_numbers_add_color_offset
```

⟨*Quantum numbers: procedures*⟩+≡
```
elemental subroutine quantum_numbers_add_color_offset (qn, offset)
  class(quantum_numbers_t), intent(inout) :: qn
  integer, intent(in) :: offset
  call qn%c%add_offset (offset)
end subroutine quantum_numbers_add_color_offset
```

Given a quantum number array, return all possible color contractions, leaving the other quantum numbers intact.

⟨*Quantum numbers: public*⟩+≡
```
public :: quantum_number_array_make_color_contractions
```

⟨*Quantum numbers: procedures*⟩+≡
```
subroutine quantum_number_array_make_color_contractions (qn_in, qn_out)
  type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
  type(quantum_numbers_t), dimension(:,:), intent(out), allocatable :: qn_out
  type(color_t), dimension(:,:), allocatable :: col
```

```
    integer :: i
    call color_array_make_contractions (qn_in%c, col)
    allocate (qn_out (size (col, 1), size (col, 2)))
    do i = 1, size (qn_out, 2)
       qn_out(:,i)%f = qn_in%f
       qn_out(:,i)%c = col(:,i)
       qn_out(:,i)%h = qn_in%h
    end do
  end subroutine quantum_number_array_make_color_contractions
```

Inherited from the color component: invert the color, switching particle/antiparticle.

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡

```
  procedure :: invert_color => quantum_numbers_invert_color
```

⟨*Quantum numbers: procedures*⟩+≡

```
  elemental subroutine quantum_numbers_invert_color (qn)
    class(quantum_numbers_t), intent(inout) :: qn
    call qn%c%invert ()
  end subroutine quantum_numbers_invert_color
```

Merge two quantum number sets: for each entry, if both are defined, combine them to an off-diagonal entry (meaningful only if the input was diagonal). If either entry is undefined, take the defined one.

For flavor, off-diagonal entries are invalid, so both flavors must be equal, otherwise an invalid flavor is inserted.

⟨*Quantum numbers: public*⟩+≡

```
  public :: operator(.merge.)
```

⟨*Quantum numbers: interfaces*⟩+≡

```
  interface operator(.merge.)
     module procedure merge_quantum_numbers0
     module procedure merge_quantum_numbers1
  end interface
```

⟨*Quantum numbers: procedures*⟩+≡

```
  function merge_quantum_numbers0 (qn1, qn2) result (qn3)
    type(quantum_numbers_t) :: qn3
    type(quantum_numbers_t), intent(in) :: qn1, qn2
    qn3%f = qn1%f .merge. qn2%f
    qn3%c = qn1%c .merge. qn2%c
    qn3%h = qn1%h .merge. qn2%h
    qn3%sub = merge_subtraction_index (qn1%sub, qn2%sub)
  end function merge_quantum_numbers0

  function merge_quantum_numbers1 (qn1, qn2) result (qn3)
    type(quantum_numbers_t), dimension(:), intent(in) :: qn1, qn2
    type(quantum_numbers_t), dimension(size(qn1)) :: qn3
    qn3%f = qn1%f .merge. qn2%f
    qn3%c = qn1%c .merge. qn2%c
    qn3%h = qn1%h .merge. qn2%h
    qn3%sub = merge_subtraction_index (qn1%sub, qn2%sub)
  end function merge_quantum_numbers1
```

⟨*Quantum numbers: procedures*⟩+≡

```
elemental function merge_subtraction_index (sub1, sub2) result (sub3)
  integer :: sub3
  integer, intent(in) :: sub1, sub2
  if (sub1 > 0 .and. sub2 > 0) then
     if (sub1 == sub2) then
        sub3 = sub1
     else
        sub3 = 0
     end if
  else if (sub1 > 0) then
     sub3 = sub1
  else if (sub2 > 0) then
     sub3 = sub2
  else
     sub3 = 0
  end if
end function merge_subtraction_index
```

10.6.7 The quantum number mask

The quantum numbers mask is true for quantum numbers that should be ignored or summed over. The three mandatory entries correspond to flavor, color, and helicity, respectively.

There is an additional entry `cg`: If false, the color-ghosts property should be kept even if color is ignored. This is relevant only if `c` is set, otherwise it is always false.

The flag `hd` tells that only diagonal entries in helicity should be kept. If `h` is set, `hd` is irrelevant and will be kept `.false.`

⟨*Quantum numbers: public*⟩+≡

```
public :: quantum_numbers_mask_t
```

⟨*Quantum numbers: types*⟩+≡

```
type :: quantum_numbers_mask_t
   private
   logical :: f = .false.
   logical :: c = .false.
   logical :: cg = .false.
   logical :: h = .false.
   logical :: hd = .false.
   integer :: sub = 0
 contains
 ⟨Quantum numbers: quantum numbers mask: TBP⟩
end type quantum_numbers_mask_t
```

Define a quantum number mask: Constructor form

⟨*Quantum numbers: public*⟩+≡

```
public :: quantum_numbers_mask
```

⟨*Quantum numbers: procedures*⟩+≡

```
elemental function quantum_numbers_mask &
     (mask_f, mask_c, mask_h, mask_cg, mask_hd) result (mask)
```

```
    type(quantum_numbers_mask_t) :: mask
    logical, intent(in) :: mask_f, mask_c, mask_h
    logical, intent(in), optional :: mask_cg
    logical, intent(in), optional :: mask_hd
    call quantum_numbers_mask_init &
         (mask, mask_f, mask_c, mask_h, mask_cg, mask_hd)
  end function quantum_numbers_mask
```

Define quantum numbers: Initializer form

⟨*Quantum numbers: quantum numbers mask: TBP*⟩≡

```
  procedure :: init => quantum_numbers_mask_init
```

⟨*Quantum numbers: procedures*⟩+≡

```
  elemental subroutine quantum_numbers_mask_init &
       (mask, mask_f, mask_c, mask_h, mask_cg, mask_hd)
    class(quantum_numbers_mask_t), intent(inout) :: mask
    logical, intent(in) :: mask_f, mask_c, mask_h
    logical, intent(in), optional :: mask_cg, mask_hd
    mask%f = mask_f
    mask%c = mask_c
    mask%h = mask_h
    mask%cg = .false.
    if (present (mask_cg)) then
       if (mask%c)  mask%cg = mask_cg
    else
       mask%cg = mask_c
    end if
    mask%hd = .false.
    if (present (mask_hd)) then
       if (.not. mask%h)  mask%hd = mask_hd
    end if
  end subroutine quantum_numbers_mask_init
```

Write a quantum numbers mask. We need the stand-alone subroutine for the array case.

⟨*Quantum numbers: public*⟩+≡

```
  public :: quantum_numbers_mask_write
```

⟨*Quantum numbers: interfaces*⟩+≡

```
  interface quantum_numbers_mask_write
     module procedure quantum_numbers_mask_write_single
     module procedure quantum_numbers_mask_write_array
  end interface
```

⟨*Quantum numbers: quantum numbers mask: TBP*⟩+≡

```
  procedure :: write => quantum_numbers_mask_write_single
```

⟨*Quantum numbers: procedures*⟩+≡

```
  subroutine quantum_numbers_mask_write_single (mask, unit)
    class(quantum_numbers_mask_t), intent(in) :: mask
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(A)", advance="no") "["
    write (u, "(L1)", advance="no")  mask%f
```

```
  write (u, "(L1)", advance="no")  mask%c
  if (.not.mask%cg)  write (u, "('g')", advance="no")
  write (u, "(L1)", advance="no")  mask%h
  if (mask%hd)  write (u, "('d')", advance="no")
  write (u, "(A)", advance="no") "]"
end subroutine quantum_numbers_mask_write_single

subroutine quantum_numbers_mask_write_array (mask, unit)
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(A)", advance="no") "["
  do i = 1, size (mask)
     if (i > 1)  write (u, "(A)", advance="no")  "/"
     write (u, "(L1)", advance="no")  mask(i)%f
     write (u, "(L1)", advance="no")  mask(i)%c
     if (.not.mask(i)%cg)  write (u, "('g')", advance="no")
     write (u, "(L1)", advance="no")  mask(i)%h
     if (mask(i)%hd)  write (u, "('d')", advance="no")
  end do
  write (u, "(A)", advance="no") "]"
end subroutine quantum_numbers_mask_write_array
```

10.6.8 Setting mask components

⟨*Quantum numbers: quantum numbers mask: TBP*⟩+≡

```
procedure :: set_flavor => quantum_numbers_mask_set_flavor
procedure :: set_color => quantum_numbers_mask_set_color
procedure :: set_helicity => quantum_numbers_mask_set_helicity
procedure :: set_sub => quantum_numbers_mask_set_sub
```

⟨*Quantum numbers: procedures*⟩+≡

```
elemental subroutine quantum_numbers_mask_set_flavor (mask, mask_f)
  class(quantum_numbers_mask_t), intent(inout) :: mask
  logical, intent(in) :: mask_f
  mask%f = mask_f
end subroutine quantum_numbers_mask_set_flavor

elemental subroutine quantum_numbers_mask_set_color (mask, mask_c, mask_cg)
  class(quantum_numbers_mask_t), intent(inout) :: mask
  logical, intent(in) :: mask_c
  logical, intent(in), optional :: mask_cg
  mask%c = mask_c
  if (present (mask_cg)) then
     if (mask%c)  mask%cg = mask_cg
  else
     mask%cg = mask_c
  end if
end subroutine quantum_numbers_mask_set_color

elemental subroutine quantum_numbers_mask_set_helicity (mask, mask_h, mask_hd)
  class(quantum_numbers_mask_t), intent(inout) :: mask
```

```
    logical, intent(in) :: mask_h
    logical, intent(in), optional :: mask_hd
    mask%h = mask_h
    if (present (mask_hd)) then
       if (.not. mask%h)  mask%hd = mask_hd
    end if
  end subroutine quantum_numbers_mask_set_helicity

  elemental subroutine quantum_numbers_mask_set_sub (mask, sub)
    class(quantum_numbers_mask_t), intent(inout) :: mask
    integer, intent(in) :: sub
    mask%sub = sub
  end subroutine quantum_numbers_mask_set_sub
```

The following routines assign part of a mask, depending on the flags given.

⟨*Quantum numbers: quantum numbers mask: TBP*⟩+≡

```
  procedure :: assign => quantum_numbers_mask_assign
```

⟨*Quantum numbers: procedures*⟩+≡

```
  elemental subroutine quantum_numbers_mask_assign &
       (mask, mask_in, flavor, color, helicity)
    class(quantum_numbers_mask_t), intent(inout) :: mask
    class(quantum_numbers_mask_t), intent(in) :: mask_in
    logical, intent(in), optional :: flavor, color, helicity
    if (present (flavor)) then
       if (flavor) then
          mask%f = mask_in%f
       end if
    end if
    if (present (color)) then
       if (color) then
          mask%c = mask_in%c
          mask%cg = mask_in%cg
       end if
    end if
    if (present (helicity)) then
       if (helicity) then
          mask%h = mask_in%h
          mask%hd = mask_in%hd
       end if
    end if
  end subroutine quantum_numbers_mask_assign
```

10.6.9 Mask predicates

Return true if either one of the entries is set:

⟨*Quantum numbers: public*⟩+≡

```
  public :: any
```

⟨*Quantum numbers: interfaces*⟩+≡

```
  interface any
     module procedure quantum_numbers_mask_any
  end interface
```

⟨*Quantum numbers: procedures*⟩+≡

```
function quantum_numbers_mask_any (mask) result (match)
  logical :: match
  type(quantum_numbers_mask_t), intent(in) :: mask
  match = mask%f .or. mask%c .or. mask%h .or. mask%hd
end function quantum_numbers_mask_any
```

10.6.10 Operators

The OR operation is applied to all components.

⟨*Quantum numbers: quantum numbers mask: TBP*⟩+≡

```
generic :: operator(.or.) => quantum_numbers_mask_or
procedure, private :: quantum_numbers_mask_or
```

⟨*Quantum numbers: procedures*⟩+≡

```
elemental function quantum_numbers_mask_or (mask1, mask2) result (mask)
  type(quantum_numbers_mask_t) :: mask
  class(quantum_numbers_mask_t), intent(in) :: mask1, mask2
  mask%f = mask1%f .or. mask2%f
  mask%c = mask1%c .or. mask2%c
  if (mask%c)  mask%cg = mask1%cg .or. mask2%cg
  mask%h = mask1%h .or. mask2%h
  if (.not. mask%h)  mask%hd = mask1%hd .or. mask2%hd
end function quantum_numbers_mask_or
```

10.6.11 Mask comparisons

Return true if the two masks are equivalent / differ:

⟨*Quantum numbers: quantum numbers mask: TBP*⟩+≡

```
generic :: operator(.eqv.) => quantum_numbers_mask_eqv
generic :: operator(.neqv.) => quantum_numbers_mask_neqv
procedure, private :: quantum_numbers_mask_eqv
procedure, private :: quantum_numbers_mask_neqv
```

⟨*Quantum numbers: procedures*⟩+≡

```
elemental function quantum_numbers_mask_eqv (mask1, mask2) result (eqv)
  logical :: eqv
  class(quantum_numbers_mask_t), intent(in) :: mask1, mask2
  eqv = (mask1%f .eqv. mask2%f) .and. &
        (mask1%c .eqv. mask2%c) .and. &
        (mask1%cg .eqv. mask2%cg) .and. &
        (mask1%h .eqv. mask2%h) .and. &
        (mask1%hd .eqv. mask2%hd)
end function quantum_numbers_mask_eqv

elemental function quantum_numbers_mask_neqv (mask1, mask2) result (neqv)
  logical :: neqv
  class(quantum_numbers_mask_t), intent(in) :: mask1, mask2
  neqv = (mask1%f .neqv. mask2%f) .or. &
        (mask1%c .neqv. mask2%c) .or. &
        (mask1%cg .neqv. mask2%cg) .or. &
```

```
        (mask1%h .neqv. mask2%h) .or. &
        (mask1%hd .neqv. mask2%hd)
end function quantum_numbers_mask_neqv
```

10.6.12 Apply a mask

Applying a mask to the quantum number object means undefining those entries where the mask is set. The others remain unaffected.

The hd mask has the special property that it "diagonalizes" helicity, i.e., the second helicity entry is dropped and the result is a diagonal helicity quantum number.

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡

```
procedure :: undefine => quantum_numbers_undefine
procedure :: undefined => quantum_numbers_undefined0
```

⟨*Quantum numbers: public*⟩+≡

```
public :: quantum_numbers_undefined
```

⟨*Quantum numbers: interfaces*⟩+≡

```
interface quantum_numbers_undefined
   module procedure quantum_numbers_undefined0
   module procedure quantum_numbers_undefined1
   module procedure quantum_numbers_undefined11
end interface
```

⟨*Quantum numbers: procedures*⟩+≡

```
elemental subroutine quantum_numbers_undefine (qn, mask)
  class(quantum_numbers_t), intent(inout) :: qn
  type(quantum_numbers_mask_t), intent(in) :: mask
  if (mask%f)  call qn%f%undefine ()
  if (mask%c)  call qn%c%undefine (undefine_ghost = mask%cg)
  if (mask%h) then
     call qn%h%undefine ()
  else if (mask%hd) then
     if (.not. qn%h%is_diagonal ()) then
        call qn%h%diagonalize ()
     end if
  end if
  if (mask%sub > 0) qn%sub = 0
end subroutine quantum_numbers_undefine

function quantum_numbers_undefined0 (qn, mask) result (qn_new)
  class(quantum_numbers_t), intent(in) :: qn
  type(quantum_numbers_mask_t), intent(in) :: mask
  type(quantum_numbers_t) :: qn_new
  select type (qn)
  type is (quantum_numbers_t);  qn_new = qn
  end select
  call quantum_numbers_undefine (qn_new, mask)
end function quantum_numbers_undefined0

function quantum_numbers_undefined1 (qn, mask) result (qn_new)
  type(quantum_numbers_t), dimension(:), intent(in) :: qn
```

```
    type(quantum_numbers_mask_t), intent(in) :: mask
    type(quantum_numbers_t), dimension(size(qn)) :: qn_new
    qn_new = qn
    call quantum_numbers_undefine (qn_new, mask)
  end function quantum_numbers_undefined1

  function quantum_numbers_undefined11 (qn, mask) result (qn_new)
    type(quantum_numbers_t), dimension(:), intent(in) :: qn
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
    type(quantum_numbers_t), dimension(size(qn)) :: qn_new
    qn_new = qn
    call quantum_numbers_undefine (qn_new, mask)
  end function quantum_numbers_undefined11
```

Return true if the input quantum number set has entries that would be removed by the applied mask, e.g., if polarization is defined but `mask%h` is set:

⟨*Quantum numbers: quantum numbers: TBP*⟩+≡

```
  procedure :: are_redundant => quantum_numbers_are_redundant
```

⟨*Quantum numbers: procedures*⟩+≡

```
  elemental function quantum_numbers_are_redundant (qn, mask) &
      result (redundant)
    logical :: redundant
    class(quantum_numbers_t), intent(in) :: qn
    type(quantum_numbers_mask_t), intent(in) :: mask
    redundant = .false.
    if (mask%f) then
       redundant = qn%f%is_defined ()
    end if
    if (mask%c) then
       redundant = qn%c%is_defined ()
    end if
    if (mask%h) then
       redundant = qn%h%is_defined ()
    else if (mask%hd) then
       redundant = .not. qn%h%is_diagonal ()
    end if
    if (mask%sub > 0) redundant = qn%sub >= mask%sub
  end function quantum_numbers_are_redundant
```

Return true if the helicity flag is set or the diagonal-helicity flag is set.

⟨*Quantum numbers: quantum numbers mask: TBP*⟩+≡

```
  procedure :: diagonal_helicity => quantum_numbers_mask_diagonal_helicity
```

⟨*Quantum numbers: procedures*⟩+≡

```
  elemental function quantum_numbers_mask_diagonal_helicity (mask) &
      result (flag)
    logical :: flag
    class(quantum_numbers_mask_t), intent(in) :: mask
    flag = mask%h .or. mask%hd
  end function quantum_numbers_mask_diagonal_helicity
```

Chapter 11

Transition Matrices and Evaluation

The modules in this chapter implement transition matrices and calculations. The functionality is broken down in three modules

state_matrices represent state and transition density matrices built from particle quantum numbers (helicity, color, flavor)

interactions extend state matrices with the record of particle momenta. They also distinguish in- and out-particles and store parent-child relations.

evaluators These objects extend interaction objects by the information how to calculate matrix elements from products and squares of other interactions. They implement the methods to actually compute those matrix elements.

11.1 State matrices

This module deals with the internal state of a particle system, i.e., with its density matrix in flavor, color, and helicity space.

⟨state_matrices.f90⟩≡

```
⟨File header⟩

module state_matrices

⟨Use kinds⟩
  use io_units
  use format_utils, only: pac_fmt
  use format_defs, only: FMT_17, FMT_19
  use diagnostics
  use sorting
  use model_data
  use flavors
  use colors
  use helicities
  use quantum_numbers

⟨Standard module head⟩

⟨State matrices: public⟩

⟨State matrices: parameters⟩

⟨State matrices: types⟩

⟨State matrices: interfaces⟩

contains

⟨State matrices: procedures⟩

end module state_matrices
```

11.1.1 Nodes of the quantum state trie

A quantum state object represents an unnormalized density matrix, i.e., an array of possibilities for flavor, color, and helicity indices with associated complex values. Physically, the trace of this matrix is the summed squared matrix element for an interaction, and the matrix elements divided by this value correspond to the flavor-color-helicity density matrix. (Flavor and color are diagonal.)

We store density matrices as tries, that is, as trees where each branching represents the possible quantum numbers of a particle. The first branching is the first particle in the system. A leaf (the node corresponding to the last particle) contains the value of the matrix element.

Each node contains a flavor, color, and helicity entry. Note that each of those entries may be actually undefined, so we can also represent, e.g., unpolarized particles.

The value is meaningful only for leaves, which have no child nodes. There is a pointer to the parent node which allows for following the trie downwards from a leaf, it is null for a root node. The child nodes are implemented as a list, so there is a pointer to the first and last child, and each node also has a `next` pointer to the next sibling.

The root node does not correspond to a particle, only its children do. The quantum numbers of the root node are irrelevant and will not be set. However, we use a common type for the three classes (root, branch, leaf); they may easily be distinguished by the association status of parent and child.

Node type

The node is linked in all directions: the parent, the first and last in the list of children, and the previous and next sibling. This allows us for adding and removing nodes and whole branches anywhere in the trie. (Circular links are not allowed, however.). The node holds its associated set of quantum numbers. The integer index, which is set only for leaf nodes, is the index of the corresponding matrix element value within the state matrix.

Temporarily, matrix-element values may be stored within a leaf node. This is used during state-matrix factorization. When the state matrix is `freeze`d, these values are transferred to the matrix-element array within the host state matrix.

⟨*State matrices: types*⟩≡

```
type :: node_t
   private
   type(quantum_numbers_t) :: qn
   type(node_t), pointer :: parent => null ()
   type(node_t), pointer :: child_first => null ()
   type(node_t), pointer :: child_last => null ()
   type(node_t), pointer :: next => null ()
   type(node_t), pointer :: previous => null ()
   integer :: me_index = 0
   integer, dimension(:), allocatable :: me_count
   complex(default) :: me = 0
end type node_t
```

Operations on nodes

Recursively deallocate all children of the current node. This includes any values associated with the children.

⟨*State matrices: procedures*⟩≡

```
pure recursive subroutine node_delete_offspring (node)
  type(node_t), pointer :: node
  type(node_t), pointer :: child
  child => node%child_first
  do while (associated (child))
     node%child_first => node%child_first%next
     call node_delete_offspring (child)
     deallocate (child)
     child => node%child_first
  end do
```

```
    node%child_last => null ()
  end subroutine node_delete_offspring
```

Remove a node including its offspring. Adjust the pointers of parent and siblings, if necessary.

⟨*State matrices: procedures*⟩+≡

```
  pure subroutine node_delete (node)
    type(node_t), pointer :: node
    call node_delete_offspring (node)
    if (associated (node%previous)) then
       node%previous%next => node%next
    else if (associated (node%parent)) then
       node%parent%child_first => node%next
    end if
    if (associated (node%next)) then
       node%next%previous => node%previous
    else if (associated (node%parent)) then
       node%parent%child_last => node%previous
    end if
    deallocate (node)
  end subroutine node_delete
```

Append a child node

⟨*State matrices: procedures*⟩+≡

```
  subroutine node_append_child (node, child)
    type(node_t), target, intent(inout) :: node
    type(node_t), pointer :: child
    allocate (child)
    if (associated (node%child_last)) then
       node%child_last%next => child
       child%previous => node%child_last
    else
       node%child_first => child
    end if
    node%child_last => child
    child%parent => node
  end subroutine node_append_child
```

I/O

Output of a single node, no recursion. We print the quantum numbers in square brackets, then the value (if any).

⟨*State matrices: procedures*⟩+≡

```
  subroutine node_write (node, me_array, verbose, unit, col_verbose, testflag)
    type(node_t), intent(in) :: node
    complex(default), dimension(:), intent(in), optional :: me_array
    logical, intent(in), optional :: verbose, col_verbose, testflag
    integer, intent(in), optional :: unit
    logical :: verb
    integer :: u
    character(len=7) :: fmt
```

```
    call pac_fmt (fmt, FMT_19, FMT_17, testflag)
    verb = .false.;  if (present (verbose)) verb = verbose
    u = given_output_unit (unit);  if (u < 0)  return
    call node%qn%write (u, col_verbose)
    if (node%me_index /= 0) then
       write (u, "(A,I0,A)", advance="no")  " => ME(", node%me_index, ")"
       if (present (me_array)) then
          write (u, "(A)", advance="no")  " = "
          write (u, "('(',' // fmt // ',',' // fmt // ',')')", &
               advance="no") pacify_complex (me_array(node%me_index))
       end if
    end if
    write (u, *)
    if (verb) then
       call ptr_write ("parent     ", node%parent)
       call ptr_write ("child_first", node%child_first)
       call ptr_write ("child_last ", node%child_last)
       call ptr_write ("next       ", node%next)
       call ptr_write ("previous   ", node%previous)
    end if
  contains
    subroutine ptr_write (label, node)
      character(*), intent(in) :: label
      type(node_t), pointer :: node
      if (associated (node)) then
         write (u, "(10x,A,1x,'->',1x)", advance="no") label
         call node%qn%write (u, col_verbose)
         write (u, *)
      end if
    end subroutine ptr_write
  end subroutine node_write
```

Recursive output of a node:

⟨*State matrices: procedures*⟩+≡

```
  recursive subroutine node_write_rec (node, me_array, verbose, &
        indent, unit, col_verbose, testflag)
    type(node_t), intent(in), target :: node
    complex(default), dimension(:), intent(in), optional :: me_array
    logical, intent(in), optional :: verbose, col_verbose, testflag
    integer, intent(in), optional :: indent
    integer, intent(in), optional :: unit
    type(node_t), pointer :: current
    logical :: verb
    integer :: i, u
    verb = .false.;  if (present (verbose))  verb = verbose
    i = 0;  if (present (indent)) i = indent
    u = given_output_unit (unit);  if (u < 0)  return
    current => node%child_first
    do while (associated (current))
       write (u, "(A)", advance="no")  repeat (" ", i)
       call node_write (current, me_array, verbose = verb, &
          unit = u, col_verbose = col_verbose, testflag = testflag)
       call node_write_rec (current, me_array, verbose = verb, &
          indent = i + 2, unit = u, col_verbose = col_verbose, testflag = testflag)
```

```
      current => current%next
    end do
  end subroutine node_write_rec
```

Binary I/O. Matrix elements are written only for leaf nodes.

⟨*State matrices: procedures*⟩+≡

```
  recursive subroutine node_write_raw_rec (node, u)
    type(node_t), intent(in), target :: node
    integer, intent(in) :: u
    logical :: associated_child_first, associated_next
    call node%qn%write_raw (u)
    associated_child_first = associated (node%child_first)
    write (u) associated_child_first
    associated_next = associated (node%next)
    write (u) associated_next
    if (associated_child_first) then
       call node_write_raw_rec (node%child_first, u)
    else
       write (u)  node%me_index
       write (u)  node%me
    end if
    if (associated_next) then
       call node_write_raw_rec (node%next, u)
    end if
  end subroutine node_write_raw_rec

  recursive subroutine node_read_raw_rec (node, u, parent, iostat)
    type(node_t), intent(out), target :: node
    integer, intent(in) :: u
    type(node_t), intent(in), optional, target :: parent
    integer, intent(out), optional :: iostat
    logical :: associated_child_first, associated_next
    type(node_t), pointer :: child
    call node%qn%read_raw (u, iostat=iostat)
    read (u, iostat=iostat) associated_child_first
    read (u, iostat=iostat) associated_next
    if (present (parent))  node%parent => parent
    if (associated_child_first) then
       allocate (child)
       node%child_first => child
       node%child_last => null ()
       call node_read_raw_rec (child, u, node, iostat=iostat)
       do while (associated (child))
          child%previous => node%child_last
          node%child_last => child
          child => child%next
       end do
    else
       read (u, iostat=iostat)  node%me_index
       read (u, iostat=iostat)  node%me
    end if
    if (associated_next) then
       allocate (node%next)
       call node_read_raw_rec (node%next, u, parent, iostat=iostat)
```

```
    end if
  end subroutine node_read_raw_rec
```

11.1.2 State matrix

Definition

The quantum state object is a container that keeps and hides the root node. For direct accessibility of values, they are stored in a separate array. The leaf nodes of the quantum-number tree point to those values, once the state matrix is finalized.

The `norm` component is redefined if a common factor is extracted from all nodes.

⟨*State matrices: public*⟩≡

```
  public :: state_matrix_t
```

⟨*State matrices: types*⟩+≡

```
  type :: state_matrix_t
     private
     type(node_t), pointer :: root => null ()
     integer :: depth = 0
     integer :: n_matrix_elements = 0
     logical :: leaf_nodes_store_values = .false.
     integer :: n_counters = 0
     complex(default), dimension(:), allocatable :: me
     real(default) :: norm = 1
     integer :: n_sub = 0
   contains
   ⟨State matrices: state matrix: TBP⟩
  end type state_matrix_t
```

This initializer allocates the root node but does not fill anything. We declare whether values are stored within the nodes during state-matrix construction, and how many counters should be maintained (default: none).

⟨*State matrices: state matrix: TBP*⟩≡

```
  procedure :: init => state_matrix_init
```

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_matrix_init (state, store_values, n_counters)
    class(state_matrix_t), intent(out) :: state
    logical, intent(in), optional :: store_values
    integer, intent(in), optional :: n_counters
    allocate (state%root)
    if (present (store_values)) &
       state%leaf_nodes_store_values = store_values
    if (present (n_counters)) state%n_counters = n_counters
  end subroutine state_matrix_init
```

This recursively deletes all children of the root node, restoring the initial state. The matrix element array is not finalized, since it does not contain physical entries, just pointers.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: final => state_matrix_final
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_final (state)
  class(state_matrix_t), intent(inout) :: state
  if (allocated (state%me))  deallocate (state%me)
  if (associated (state%root))  call node_delete (state%root)
  state%depth = 0
  state%n_matrix_elements = 0
end subroutine state_matrix_final
```

Output: Present the tree as a nested list with appropriate indentation.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: write => state_matrix_write
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_write (state, unit, write_value_list, &
      verbose, col_verbose, testflag)
  class(state_matrix_t), intent(in) :: state
  logical, intent(in), optional :: write_value_list, verbose, col_verbose
  logical, intent(in), optional :: testflag
  integer, intent(in), optional :: unit
  complex(default) :: me_dum
  character(len=7) :: fmt
  integer :: u
  integer :: i
  call pac_fmt (fmt, FMT_19, FMT_17, testflag)
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A," // fmt // ")") "State matrix:  norm = ", state%norm
  if (associated (state%root)) then
     if (allocated (state%me)) then
        call node_write_rec (state%root, state%me, verbose = verbose, &
           indent = 1, unit = u, col_verbose = col_verbose, &
           testflag = testflag)
     else
        call node_write_rec (state%root, verbose = verbose, indent = 1, &
           unit = u, col_verbose = col_verbose, testflag = testflag)
     end if
  end if
  if (present (write_value_list)) then
     if (write_value_list .and. allocated (state%me)) then
        do i = 1, size (state%me)
           write (u, "(1x,I0,A)", advance="no")  i, ":"
           me_dum = state%me(i)
           if (real(state%me(i)) == -real(state%me(i))) then
              me_dum = &
                    cmplx (0._default, aimag(me_dum), kind=default)
           end if
           if (aimag(me_dum) == -aimag(me_dum)) then
              me_dum = &
                    cmplx (real(me_dum), 0._default, kind=default)
           end if
           write (u, "('(','," // fmt // ",',','," // fmt // &
                ",')')")  me_dum
        end do
```

```
      end if
    end if
  end subroutine state_matrix_write
```

Binary I/O. The auxiliary matrix-element array is not written, but reconstructed after reading the tree.

Note: To be checked. Might be broken, don't use (unless trivial).

⟨*State matrices: state matrix: TBP*⟩+≡

```
  procedure :: write_raw => state_matrix_write_raw
  procedure :: read_raw => state_matrix_read_raw
```

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_matrix_write_raw (state, u)
    class(state_matrix_t), intent(in), target :: state
    integer, intent(in) :: u
    logical :: is_defined
    integer :: depth, j
    type(state_iterator_t) :: it
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    is_defined = state%is_defined ()
    write (u)  is_defined
    if (is_defined) then
       write (u)  state%get_norm ()
       write (u)  state%get_n_leaves ()
       depth = state%get_depth ()
       write (u)  depth
       allocate (qn (depth))
       call it%init (state)
       do while (it%is_valid ())
          qn = it%get_quantum_numbers ()
          do j = 1, depth
             call qn(j)%write_raw (u)
          end do
          write (u)  it%get_me_index ()
          write (u)  it%get_matrix_element ()
          call it%advance ()
       end do
    end if
  end subroutine state_matrix_write_raw

  subroutine state_matrix_read_raw (state, u, iostat)
    class(state_matrix_t), intent(out) :: state
    integer, intent(in) :: u
    integer, intent(out) :: iostat
    logical :: is_defined
    real(default) :: norm
    integer :: n_leaves, depth, i, j
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    integer :: me_index
    complex(default) :: me
    read (u, iostat=iostat)  is_defined
    if (iostat /= 0)  goto 1
    if (is_defined) then
       call state%init (store_values = .true.)
```

```
      read (u, iostat=iostat)  norm
      if (iostat /= 0)  goto 1
      call state_matrix_set_norm (state, norm)
      read (u)  n_leaves
      if (iostat /= 0)  goto 1
      read (u)  depth
      if (iostat /= 0)  goto 1
      allocate (qn (depth))
      do i = 1, n_leaves
         do j = 1, depth
            call qn(j)%read_raw (u, iostat=iostat)
            if (iostat /= 0)  goto 1
         end do
         read (u, iostat=iostat)  me_index
         if (iostat /= 0)  goto 1
         read (u, iostat=iostat)  me
         if (iostat /= 0)  goto 1
         call state%add_state (qn, index = me_index, value = me)
      end do
      call state_matrix_freeze (state)
   end if
   return

   ! Clean up on error
1  continue
   call state%final ()
end subroutine state_matrix_read_raw
```

Assign a model pointer to all flavor entries. This will become necessary when we have read a state matrix from file.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: set_model => state_matrix_set_model
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_set_model (state, model)
  class(state_matrix_t), intent(inout), target :: state
  class(model_data_t), intent(in), target :: model
  type(state_iterator_t) :: it
  call it%init (state)
  do while (it%is_valid ())
     call it%set_model (model)
     call it%advance ()
  end do
end subroutine state_matrix_set_model
```

Properties of the quantum state

A state is defined if its root is allocated:

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: is_defined => state_matrix_is_defined
```

⟨*State matrices: procedures*⟩+≡

```
elemental function state_matrix_is_defined (state) result (defined)
  logical :: defined
  class(state_matrix_t), intent(in) :: state
  defined = associated (state%root)
end function state_matrix_is_defined
```

A state is empty if its depth is zero:

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: is_empty => state_matrix_is_empty
```

⟨*State matrices: procedures*⟩+≡

```
elemental function state_matrix_is_empty (state) result (flag)
  logical :: flag
  class(state_matrix_t), intent(in) :: state
  flag = state%depth == 0
end function state_matrix_is_empty
```

Return the number of matrix-element values.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: get_n_matrix_elements => state_matrix_get_n_matrix_elements
```

⟨*State matrices: procedures*⟩+≡

```
pure function state_matrix_get_n_matrix_elements (state) result (n)
  integer :: n
  class(state_matrix_t), intent(in) :: state
  n = state%n_matrix_elements
end function state_matrix_get_n_matrix_elements
```

Return the size of the `me`-array for debugging purposes.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: get_me_size => state_matrix_get_me_size
```

⟨*State matrices: procedures*⟩+≡

```
pure function state_matrix_get_me_size (state) result (n)
  integer :: n
  class(state_matrix_t), intent(in) :: state
  if (allocated (state%me)) then
     n = size (state%me)
  else
     n = 0
  end if
end function state_matrix_get_me_size
```

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: compute_n_sub => state_matrix_compute_n_sub
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_compute_n_sub (state)
  class(state_matrix_t), intent(inout) :: state
  type(state_iterator_t) :: it
  type(quantum_numbers_t), dimension(state%depth) :: qn
  integer :: sub
```

```
    call it%init (state)
    do while (it%is_valid ())
       qn = it%get_quantum_numbers ()
       sub = qn(1)%get_sub ()
       if (sub > state%n_sub) state%n_sub = sub
       call it%advance ()
    end do
  end subroutine state_matrix_compute_n_sub
```

⟨*State matrices: state matrix: TBP*⟩+≡

```
  procedure :: get_n_sub => state_matrix_get_n_sub
```

⟨*State matrices: procedures*⟩+≡

```
  pure function state_matrix_get_n_sub (state) result (n)
    integer :: n
    class(state_matrix_t), intent(in) :: state
    n = state%n_sub
  end function state_matrix_get_n_sub
```

Return the number of leaves. This can be larger than the number of independent matrix elements.

⟨*State matrices: state matrix: TBP*⟩+≡

```
  procedure :: get_n_leaves => state_matrix_get_n_leaves
```

⟨*State matrices: procedures*⟩+≡

```
  function state_matrix_get_n_leaves (state) result (n)
    integer :: n
    class(state_matrix_t), intent(in) :: state
    type(state_iterator_t) :: it
    n = 0
    call it%init (state)
    do while (it%is_valid ())
       n = n + 1
       call it%advance ()
    end do
  end function state_matrix_get_n_leaves
```

Return the depth:

⟨*State matrices: state matrix: TBP*⟩+≡

```
  procedure :: get_depth => state_matrix_get_depth
```

⟨*State matrices: procedures*⟩+≡

```
  pure function state_matrix_get_depth (state) result (depth)
    integer :: depth
    class(state_matrix_t), intent(in) :: state
    depth = state%depth
  end function state_matrix_get_depth
```

Return the norm:

⟨*State matrices: state matrix: TBP*⟩+≡

```
  procedure :: get_norm => state_matrix_get_norm
```

⟨*State matrices: procedures*⟩+≡

```
pure function state_matrix_get_norm (state) result (norm)
  real(default) :: norm
  class(state_matrix_t), intent(in) :: state
  norm = state%norm
end function state_matrix_get_norm
```

Retrieving contents

Return the quantum number array, using an index. We have to scan the state matrix since there is no shortcut.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: get_quantum_number => &
   state_matrix_get_quantum_number
```

⟨*State matrices: procedures*⟩+≡

```
function state_matrix_get_quantum_number (state, i) result (qn)
  class(state_matrix_t), intent(in), target :: state
  integer, intent(in) :: i
  type(quantum_numbers_t), dimension(state%depth) :: qn
  type(state_iterator_t) :: it
  integer :: k
  k = 0
  call it%init (state)
  do while (it%is_valid ())
     k = k + 1
     if (k == i) then
        qn = it%get_quantum_numbers ()
        return
     end if
     call it%advance ()
  end do
end function state_matrix_get_quantum_number
```

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: get_quantum_numbers => &
   state_matrix_get_quantum_numbers
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_get_quantum_numbers (state, qn)
  type(quantum_numbers_t), dimension(:,:), allocatable, intent(out) :: qn
  class(state_matrix_t), intent(in), target :: state
  integer :: i
  allocate (qn (state%get_n_matrix_elements (), &
     state%get_depth()))
  do i = 1, state%get_n_matrix_elements ()
     qn (i, :) = state%get_quantum_number (i)
  end do
end subroutine state_matrix_get_quantum_numbers
```

Return a single matrix element using its index. Works only if the shortcut array is allocated.

⟨*State matrices: state matrix: TBP*⟩+≡

```
generic :: get_matrix_element => get_matrix_element_single
generic :: get_matrix_element => get_matrix_element_array
procedure :: get_matrix_element_single => &
  state_matrix_get_matrix_element_single
procedure :: get_matrix_element_array => &
  state_matrix_get_matrix_element_array
```

⟨*State matrices: procedures*⟩+≡

```
elemental function state_matrix_get_matrix_element_single (state, i) result (me)
  complex(default) :: me
  class(state_matrix_t), intent(in) :: state
  integer, intent(in) :: i
  if (allocated (state%me)) then
     me = state%me(i)
  else
     me = 0
  end if
end function state_matrix_get_matrix_element_single
```

⟨*State matrices: procedures*⟩+≡

```
function state_matrix_get_matrix_element_array (state) result (me)
   complex(default), dimension(:), allocatable :: me
   class(state_matrix_t), intent(in) :: state
   if (allocated (state%me)) then
      allocate (me (size (state%me)))
      me = state%me
   else
      me = 0
   end if
end function state_matrix_get_matrix_element_array
```

Return the color index with maximum absolute value that is present within the state matrix.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: get_max_color_value => state_matrix_get_max_color_value
```

⟨*State matrices: procedures*⟩+≡

```
function state_matrix_get_max_color_value (state) result (cmax)
  integer :: cmax
  class(state_matrix_t), intent(in) :: state
  if (associated (state%root)) then
     cmax = node_get_max_color_value (state%root)
  else
     cmax = 0
  end if
contains
  recursive function node_get_max_color_value (node) result (cmax)
    integer :: cmax
    type(node_t), intent(in), target :: node
    type(node_t), pointer :: current
    cmax = quantum_numbers_get_max_color_value (node%qn)
    current => node%child_first
    do while (associated (current))
       cmax = max (cmax, node_get_max_color_value (current))
```

```
        current => current%next
      end do
    end function node_get_max_color_value
  end function state_matrix_get_max_color_value
```

Building the quantum state

The procedure generates a branch associated to the input array of quantum numbers. If the branch exists already, it is used.

Optionally, we set the matrix-element index, a value (which may be added to the previous one), and increment one of the possible counters. We may also return the matrix element index of the current node.

⟨*State matrices: state matrix: TBP*⟩+≡

```
  procedure :: add_state => state_matrix_add_state
```

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_matrix_add_state &
       (state, qn, index, value, sum_values, counter_index, me_index)
    class(state_matrix_t), intent(inout) :: state
    type(quantum_numbers_t), dimension(:), intent(in) :: qn
    integer, intent(in), optional :: index
    complex(default), intent(in), optional :: value
    logical, intent(in), optional :: sum_values
    integer, intent(in), optional :: counter_index
    integer, intent(out), optional :: me_index
    logical :: set_index, get_index, add
    set_index = present (index)
    get_index = present (me_index)
    add = .false.;  if (present (sum_values))  add = sum_values
    if (state%depth == 0) then
       state%depth = size (qn)
    else if (state%depth /= size (qn)) then
       call state%write ()
       call msg_bug ("State matrix: depth mismatch")
    end if
    if (size (qn) > 0)  call node_make_branch (state%root, qn)
  contains
     recursive subroutine node_make_branch (parent, qn)
       type(node_t), pointer :: parent
       type(quantum_numbers_t), dimension(:), intent(in) :: qn
       type(node_t), pointer :: child
       logical :: match
       match = .false.
       child => parent%child_first
       SCAN_CHILDREN: do while (associated (child))
          match = child%qn == qn(1)
          if (match)  exit SCAN_CHILDREN
          child => child%next
       end do SCAN_CHILDREN
       if (.not. match) then
          call node_append_child (parent, child)
          child%qn = qn(1)
       end if
```

```
      select case (size (qn))
      case (1)
         if (.not. match) then
            state%n_matrix_elements = state%n_matrix_elements + 1
            child%me_index = state%n_matrix_elements
         end if
         if (set_index) then
            child%me_index = index
         end if
         if (get_index) then
            me_index = child%me_index
         end if
         if (present (counter_index)) then
            if (.not. allocated (child%me_count)) then
               allocate (child%me_count (state%n_counters))
               child%me_count = 0
            end if
            child%me_count(counter_index) = child%me_count(counter_index) + 1
         end if
         if (present (value)) then
            if (add) then
               child%me = child%me + value
            else
               child%me = value
            end if
         end if
      case (2:)
         call node_make_branch (child, qn(2:))
      end select
    end subroutine node_make_branch
  end subroutine state_matrix_add_state
```

Remove irrelevant flavor/color/helicity labels and the corresponding branchings. The masks indicate which particles are affected; the masks length should coincide with the depth of the trie (without the root node). Recursively scan the whole tree, starting from the leaf nodes and working up to the root node. If a mask entry is set for the current tree level, scan the children there. For each child within that level make a new empty branch where the masked quantum number is undefined. Then recursively combine all following children with matching quantum number into this new node and move on.

⟨*State matrices: state matrix: TBP*⟩+≡

```
  procedure :: collapse => state_matrix_collapse
```

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_matrix_collapse (state, mask)
    class(state_matrix_t), intent(inout) :: state
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
    type(state_matrix_t) :: red_state
    if (state%is_defined ()) then
       call state%reduce (mask, red_state)
       call state%final ()
       state = red_state
    end if
```

```
end subroutine state_matrix_collapse
```

Transform the given state matrix into a reduced state matrix where some quantum numbers are removed, as indicated by the mask. The procedure creates a new state matrix, so the old one can be deleted after this if it is no longer used.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: reduce => state_matrix_reduce
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_reduce (state, mask, red_state)
  class(state_matrix_t), intent(in), target :: state
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
  type(state_matrix_t), intent(out) :: red_state
  type(state_iterator_t) :: it
  type(quantum_numbers_t), dimension(size(mask)) :: qn
  call red_state%init ()
  call it%init (state)
  do while (it%is_valid ())
     qn = it%get_quantum_numbers ()
     call qn%undefine (mask)
     call red_state%add_state (qn)
     call it%advance ()
  end do
  call red_state%freeze ()
end subroutine state_matrix_reduce
```

This subroutine sets up the matrix-element array. The leaf nodes aquire the index values that point to the appropriate matrix-element entry.

We recursively scan the trie. Once we arrive at a leaf node, the index is increased and associated to that node. Finally, we allocate the matrix-element array with the appropriate size.

If matrix element values are temporarily stored within the leaf nodes, we scan the state again and transfer them to the matrix-element array.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: freeze => state_matrix_freeze
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_freeze (state)
  class(state_matrix_t), intent(inout), target :: state
  type(state_iterator_t) :: it
  if (associated (state%root)) then
     if (allocated (state%me))  deallocate (state%me)
     allocate (state%me (state%n_matrix_elements))
     state%me = 0
     call state%compute_n_sub ()
  end if
  if (state%leaf_nodes_store_values) then
     call it%init (state)
     do while (it%is_valid ())
        state%me(it%get_me_index ()) = it%get_matrix_element ()
        call it%advance ()
     end do
     state%leaf_nodes_store_values = .false.
```

```
    end if
  end subroutine state_matrix_freeze
```

Direct access to the value array

Several methods for setting a value directly are summarized in this generic:

⟨*State matrices: state matrix: TBP*⟩+≡

```
  generic :: set_matrix_element => set_matrix_element_qn
  generic :: set_matrix_element => set_matrix_element_all
  generic :: set_matrix_element => set_matrix_element_array
  generic :: set_matrix_element => set_matrix_element_single
  generic :: set_matrix_element => set_matrix_element_clone
  procedure :: set_matrix_element_qn => state_matrix_set_matrix_element_qn
  procedure :: set_matrix_element_all => state_matrix_set_matrix_element_all
  procedure :: set_matrix_element_array => &
       state_matrix_set_matrix_element_array
  procedure :: set_matrix_element_single => &
       state_matrix_set_matrix_element_single
  procedure :: set_matrix_element_clone => &
     state_matrix_set_matrix_element_clone
```

Set a value that corresponds to a quantum number array:

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_matrix_set_matrix_element_qn (state, qn, value)
    class(state_matrix_t), intent(inout), target :: state
    type(quantum_numbers_t), dimension(:), intent(in) :: qn
    complex(default), intent(in) :: value
    type(state_iterator_t) :: it
    if (.not. allocated (state%me)) then
       allocate (state%me (size(qn)))
    end if
    call it%init (state)
    call it%go_to_qn (qn)
    call it%set_matrix_element (value)
  end subroutine state_matrix_set_matrix_element_qn
```

Set all matrix elements to a single value

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_matrix_set_matrix_element_all (state, value)
    class(state_matrix_t), intent(inout) :: state
    complex(default), intent(in) :: value
    if (.not. allocated (state%me)) then
       allocate (state%me (state%n_matrix_elements))
    end if
    state%me = value
  end subroutine state_matrix_set_matrix_element_all
```

Set the matrix-element array directly.

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_matrix_set_matrix_element_array (state, value)
    class(state_matrix_t), intent(inout) :: state
    complex(default), dimension(:), intent(in) :: value
```

```
    if (.not. allocated (state%me)) then
       allocate (state%me (size (value)))
    end if
    state%me = value
  end subroutine state_matrix_set_matrix_element_array

  pure subroutine state_matrix_set_matrix_element_single (state, i, value)
    class(state_matrix_t), intent(inout) :: state
    integer, intent(in) :: i
    complex(default), intent(in) :: value
    if (.not. allocated (state%me)) then
       allocate (state%me (state%n_matrix_elements))
    end if
    state%me(i) = value
  end subroutine state_matrix_set_matrix_element_single
```

Clone the matrix elements from another (matching) state matrix.

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_matrix_set_matrix_element_clone (state, state1)
    class(state_matrix_t), intent(inout) :: state
    type(state_matrix_t), intent(in) :: state1
    if (.not. allocated (state1%me)) return
    if (.not. allocated (state%me)) allocate (state%me (size (state1%me)))
    state%me = state1%me
  end subroutine state_matrix_set_matrix_element_clone
```

Add a value to a matrix element

⟨*State matrices: state matrix: TBP*⟩+≡

```
  procedure :: add_to_matrix_element => state_matrix_add_to_matrix_element
```

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_matrix_add_to_matrix_element (state, qn, value, match_only_flavor)
    class(state_matrix_t), intent(inout), target :: state
    type(quantum_numbers_t), dimension(:), intent(in) :: qn
    complex(default), intent(in) :: value
    logical, intent(in), optional :: match_only_flavor
    type(state_iterator_t) :: it
    call it%init (state)
    call it%go_to_qn (qn, match_only_flavor)
    if (it%is_valid ()) then
       call it%add_to_matrix_element (value)
    else
       call msg_fatal ("Cannot add to matrix element - it%node not allocated")
    end if
  end subroutine state_matrix_add_to_matrix_element
```

11.1.3 State iterators

Accessing the quantum state from outside is best done using a specialized iterator, i.e., a pointer to a particular branch of the quantum state trie. Technically, the iterator contains a pointer to a leaf node, but via parent pointers it allows

to access the whole branch where the leaf is attached. For quick access, we also keep the branch depth (which is assumed to be universal for a quantum state).

⟨*State matrices: public*⟩+≡
```
public :: state_iterator_t
```

⟨*State matrices: types*⟩+≡
```
type :: state_iterator_t
   private
   integer :: depth = 0
   type(state_matrix_t), pointer :: state => null ()
   type(node_t), pointer :: node => null ()
 contains
 ⟨State matrices: state iterator: TBP⟩
end type state_iterator_t
```

The initializer: Point at the first branch. Note that this cannot be pure, thus not be elemental, because the iterator can be used to manipulate data in the state matrix.

⟨*State matrices: state iterator: TBP*⟩≡
```
procedure :: init => state_iterator_init
```

⟨*State matrices: procedures*⟩+≡
```
subroutine state_iterator_init (it, state)
  class(state_iterator_t), intent(out) :: it
  type(state_matrix_t), intent(in), target :: state
  it%state => state
  it%depth = state%depth
  if (state%is_defined ()) then
     it%node => state%root
     do while (associated (it%node%child_first))
        it%node => it%node%child_first
     end do
  else
     it%node => null ()
  end if
end subroutine state_iterator_init
```

Go forward. Recursively programmed: if the next node does not exist, go back to the parent node and look at its successor (if present), etc.

There is a possible pitfall in the implementation: If the dummy pointer argument to the `find_next` routine is used directly, we still get the correct result for the iterator, but calling the recursion on `node%parent` means that we manipulate a parent pointer in the original state in addition to the iterator. Making a local copy of the pointer avoids this. Using pointer intent would be helpful, but we do not yet rely on this F2003 feature.

⟨*State matrices: state iterator: TBP*⟩+≡
```
procedure :: advance => state_iterator_advance
```

⟨*State matrices: procedures*⟩+≡
```
subroutine state_iterator_advance (it)
  class(state_iterator_t), intent(inout) :: it
  call find_next (it%node)
contains
```

```
    recursive subroutine find_next (node_in)
      type(node_t), intent(in), target :: node_in
      type(node_t), pointer :: node
      node => node_in
      if (associated (node%next)) then
         node => node%next
         do while (associated (node%child_first))
            node => node%child_first
         end do
         it%node => node
      else if (associated (node%parent)) then
         call find_next (node%parent)
      else
         it%node => null ()
      end if
    end subroutine find_next
  end subroutine state_iterator_advance
```

If all has been scanned, the iterator is at an undefined state. Check for this:

⟨*State matrices: state iterator: TBP*⟩+≡

```
  procedure :: is_valid => state_iterator_is_valid
```

⟨*State matrices: procedures*⟩+≡

```
  function state_iterator_is_valid (it) result (defined)
    logical :: defined
    class(state_iterator_t), intent(in) :: it
    defined = associated (it%node)
  end function state_iterator_is_valid
```

Return the matrix-element index that corresponds to the current node

⟨*State matrices: state iterator: TBP*⟩+≡

```
  procedure :: get_me_index => state_iterator_get_me_index
```

⟨*State matrices: procedures*⟩+≡

```
  function state_iterator_get_me_index (it) result (n)
    integer :: n
    class(state_iterator_t), intent(in) :: it
    n = it%node%me_index
  end function state_iterator_get_me_index
```

Return the number of times this quantum-number state has been added (noting that it is physically inserted only the first time). Note that for each state, there is an array of counters.

⟨*State matrices: state iterator: TBP*⟩+≡

```
  procedure :: get_me_count => state_iterator_get_me_count
```

⟨*State matrices: procedures*⟩+≡

```
  function state_iterator_get_me_count (it) result (n)
    integer, dimension(:), allocatable :: n
    class(state_iterator_t), intent(in) :: it
    if (allocated (it%node%me_count)) then
       allocate (n (size (it%node%me_count)))
       n = it%node%me_count
```

```
    else
       allocate (n (0))
    end if
  end function state_iterator_get_me_count
```

⟨*State matrices: state iterator: TBP*⟩+≡

```
  procedure :: get_depth => state_iterator_get_depth
```

⟨*State matrices: procedures*⟩+≡

```
  pure function state_iterator_get_depth (state_iterator) result (depth)
    integer :: depth
    class(state_iterator_t), intent(in) :: state_iterator
    depth = state_iterator%depth
  end function state_iterator_get_depth
```

Proceed to the state associated with the quantum numbers qn.

⟨*State matrices: state iterator: TBP*⟩+≡

```
  procedure :: go_to_qn => state_iterator_go_to_qn
```

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_iterator_go_to_qn (it, qn, match_only_flavor)
    class(state_iterator_t), intent(inout) :: it
    type(quantum_numbers_t), dimension(:), intent(in) :: qn
    logical, intent(in), optional :: match_only_flavor
    logical :: match_flv
    match_flv = .false.; if (present (match_only_flavor)) match_flv = .true.
    do while (it%is_valid ())
       if (match_flv) then
          if (all (qn .fmatch. it%get_quantum_numbers ())) then
             return
          else
             call it%advance ()
          end if
       else
          if (all (qn == it%get_quantum_numbers ())) then
             return
          else
             call it%advance ()
          end if
       end if
    end do
  end subroutine state_iterator_go_to_qn
```

Use the iterator to retrieve quantum-number information:

⟨*State matrices: state iterator: TBP*⟩+≡

```
  generic :: get_quantum_numbers => get_qn_multi, get_qn_slice, &
       get_qn_range, get_qn_single
  generic :: get_flavor => get_flv_multi, get_flv_slice, &
       get_flv_range, get_flv_single
  generic :: get_color => get_col_multi, get_col_slice, &
       get_col_range, get_col_single
  generic :: get_helicity => get_hel_multi, get_hel_slice, &
       get_hel_range, get_hel_single
```

⟨*State matrices: state iterator: TBP*⟩+≡

```
procedure :: get_qn_multi => state_iterator_get_qn_multi
procedure :: get_qn_slice => state_iterator_get_qn_slice
procedure :: get_qn_range => state_iterator_get_qn_range
procedure :: get_qn_single => state_iterator_get_qn_single
procedure :: get_flv_multi => state_iterator_get_flv_multi
procedure :: get_flv_slice => state_iterator_get_flv_slice
procedure :: get_flv_range => state_iterator_get_flv_range
procedure :: get_flv_single => state_iterator_get_flv_single
procedure :: get_col_multi => state_iterator_get_col_multi
procedure :: get_col_slice => state_iterator_get_col_slice
procedure :: get_col_range => state_iterator_get_col_range
procedure :: get_col_single => state_iterator_get_col_single
procedure :: get_hel_multi => state_iterator_get_hel_multi
procedure :: get_hel_slice => state_iterator_get_hel_slice
procedure :: get_hel_range => state_iterator_get_hel_range
procedure :: get_hel_single => state_iterator_get_hel_single
```

These versions return the whole quantum number array

⟨*State matrices: procedures*⟩+≡

```
function state_iterator_get_qn_multi (it) result (qn)
  class(state_iterator_t), intent(in) :: it
  type(quantum_numbers_t), dimension(it%depth) :: qn
  type(node_t), pointer :: node
  integer :: i
  node => it%node
  do i = it%depth, 1, -1
     qn(i) = node%qn
     node => node%parent
  end do
end function state_iterator_get_qn_multi

function state_iterator_get_flv_multi (it) result (flv)
  class(state_iterator_t), intent(in) :: it
  type(flavor_t), dimension(it%depth) :: flv
  flv = quantum_numbers_get_flavor &
       (it%get_quantum_numbers ())
end function state_iterator_get_flv_multi

function state_iterator_get_col_multi (it) result (col)
  class(state_iterator_t), intent(in) :: it
  type(color_t), dimension(it%depth) :: col
  col = quantum_numbers_get_color &
       (it%get_quantum_numbers ())
end function state_iterator_get_col_multi

function state_iterator_get_hel_multi (it) result (hel)
  class(state_iterator_t), intent(in) :: it
  type(helicity_t), dimension(it%depth) :: hel
  hel = quantum_numbers_get_helicity &
       (it%get_quantum_numbers ())
end function state_iterator_get_hel_multi
```

An array slice (derived from the above).

⟨*State matrices: procedures*⟩+≡

```
function state_iterator_get_qn_slice (it, index) result (qn)
  class(state_iterator_t), intent(in) :: it
  integer, dimension(:), intent(in) :: index
  type(quantum_numbers_t), dimension(size(index)) :: qn
  type(quantum_numbers_t), dimension(it%depth) :: qn_tmp
  qn_tmp = state_iterator_get_qn_multi (it)
  qn = qn_tmp(index)
end function state_iterator_get_qn_slice

function state_iterator_get_flv_slice (it, index) result (flv)
  class(state_iterator_t), intent(in) :: it
  integer, dimension(:), intent(in) :: index
  type(flavor_t), dimension(size(index)) :: flv
  flv = quantum_numbers_get_flavor &
       (it%get_quantum_numbers (index))
end function state_iterator_get_flv_slice

function state_iterator_get_col_slice (it, index) result (col)
  class(state_iterator_t), intent(in) :: it
  integer, dimension(:), intent(in) :: index
  type(color_t), dimension(size(index)) :: col
  col = quantum_numbers_get_color &
       (it%get_quantum_numbers (index))
end function state_iterator_get_col_slice

function state_iterator_get_hel_slice (it, index) result (hel)
  class(state_iterator_t), intent(in) :: it
  integer, dimension(:), intent(in) :: index
  type(helicity_t), dimension(size(index)) :: hel
  hel = quantum_numbers_get_helicity &
       (it%get_quantum_numbers (index))
end function state_iterator_get_hel_slice
```

An array range (implemented directly).

⟨*State matrices: procedures*⟩+≡

```
function state_iterator_get_qn_range (it, k1, k2) result (qn)
  class(state_iterator_t), intent(in) :: it
  integer, intent(in) :: k1, k2
  type(quantum_numbers_t), dimension(k2-k1+1) :: qn
  type(node_t), pointer :: node
  integer :: i
  node => it%node
  SCAN: do i = it%depth, 1, -1
     if (k1 <= i .and. i <= k2) then
        qn(i-k1+1) = node%qn
     else
        node => node%parent
     end if
  end do SCAN
end function state_iterator_get_qn_range

function state_iterator_get_flv_range (it, k1, k2) result (flv)
  class(state_iterator_t), intent(in) :: it
```

```
    integer, intent(in) :: k1, k2
    type(flavor_t), dimension(k2-k1+1) :: flv
    flv = quantum_numbers_get_flavor &
         (it%get_quantum_numbers (k1, k2))
  end function state_iterator_get_flv_range

  function state_iterator_get_col_range (it, k1, k2) result (col)
    class(state_iterator_t), intent(in) :: it
    integer, intent(in) :: k1, k2
    type(color_t), dimension(k2-k1+1) :: col
    col = quantum_numbers_get_color &
         (it%get_quantum_numbers (k1, k2))
  end function state_iterator_get_col_range

  function state_iterator_get_hel_range (it, k1, k2) result (hel)
    class(state_iterator_t), intent(in) :: it
    integer, intent(in) :: k1, k2
    type(helicity_t), dimension(k2-k1+1) :: hel
    hel = quantum_numbers_get_helicity &
         (it%get_quantum_numbers (k1, k2))
  end function state_iterator_get_hel_range
```

Just a specific single element

⟨*State matrices: procedures*⟩+≡

```
  function state_iterator_get_qn_single (it, k) result (qn)
    class(state_iterator_t), intent(in) :: it
    integer, intent(in) :: k
    type(quantum_numbers_t) :: qn
    type(node_t), pointer :: node
    integer :: i
    node => it%node
    SCAN: do i = it%depth, 1, -1
       if (i == k) then
          qn = node%qn
          exit SCAN
       else
          node => node%parent
       end if
    end do SCAN
  end function state_iterator_get_qn_single

  function state_iterator_get_flv_single (it, k) result (flv)
    class(state_iterator_t), intent(in) :: it
    integer, intent(in) :: k
    type(flavor_t) :: flv
    flv = quantum_numbers_get_flavor &
         (it%get_quantum_numbers (k))
  end function state_iterator_get_flv_single

  function state_iterator_get_col_single (it, k) result (col)
    class(state_iterator_t), intent(in) :: it
    integer, intent(in) :: k
    type(color_t) :: col
    col = quantum_numbers_get_color &
```

```
        (it%get_quantum_numbers (k))
end function state_iterator_get_col_single

function state_iterator_get_hel_single (it, k) result (hel)
  class(state_iterator_t), intent(in) :: it
  integer, intent(in) :: k
  type(helicity_t) :: hel
  hel = quantum_numbers_get_helicity &
        (it%get_quantum_numbers (k))
end function state_iterator_get_hel_single
```

Assign a model pointer to the current flavor entries.

⟨*State matrices: state iterator: TBP*⟩+≡

```
procedure :: set_model => state_iterator_set_model
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_iterator_set_model (it, model)
  class(state_iterator_t), intent(inout) :: it
  class(model_data_t), intent(in), target :: model
  type(node_t), pointer :: node
  integer :: i
  node => it%node
  do i = it%depth, 1, -1
     call node%qn%set_model (model)
     node => node%parent
  end do
end subroutine state_iterator_set_model
```

Retrieve the matrix element value associated with the current node.

⟨*State matrices: state iterator: TBP*⟩+≡

```
procedure :: get_matrix_element => state_iterator_get_matrix_element
```

⟨*State matrices: procedures*⟩+≡

```
function state_iterator_get_matrix_element (it) result (me)
  complex(default) :: me
  class(state_iterator_t), intent(in) :: it
  if (it%state%leaf_nodes_store_values) then
     me = it%node%me
  else if (it%node%me_index /= 0) then
     me = it%state%me(it%node%me_index)
  else
     me = 0
  end if
end function state_iterator_get_matrix_element
```

Set the matrix element value using the state iterator.

⟨*State matrices: state iterator: TBP*⟩+≡

```
procedure :: set_matrix_element => state_iterator_set_matrix_element
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_iterator_set_matrix_element (it, value)
  class(state_iterator_t), intent(inout) :: it
  complex(default), intent(in) :: value
  if (it%node%me_index /= 0) it%state%me(it%node%me_index) = value
```

```
end subroutine state_iterator_set_matrix_element
```

⟨*State matrices: state iterator: TBP*⟩+≡

```
procedure :: add_to_matrix_element => state_iterator_add_to_matrix_element
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_iterator_add_to_matrix_element (it, value)
  class(state_iterator_t), intent(inout) :: it
  complex(default), intent(in) :: value
  if (it%node%me_index /= 0) &
       it%state%me(it%node%me_index) = it%state%me(it%node%me_index) + value
end subroutine state_iterator_add_to_matrix_element
```

11.1.4 Operations on quantum states

Return a deep copy of a state matrix.

⟨*State matrices: public*⟩+≡

```
public :: assignment(=)
```

⟨*State matrices: interfaces*⟩≡

```
interface assignment(=)
   module procedure state_matrix_assign
end interface
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_assign (state_out, state_in)
  type(state_matrix_t), intent(out) :: state_out
  type(state_matrix_t), intent(in), target :: state_in
  type(state_iterator_t) :: it
  if (.not. state_in%is_defined ())  return
  call state_out%init ()
  call it%init (state_in)
  do while (it%is_valid ())
     call state_out%add_state (it%get_quantum_numbers (), &
          it%get_me_index ())
     call it%advance ()
  end do
  if (allocated (state_in%me)) then
     allocate (state_out%me (size (state_in%me)))
     state_out%me = state_in%me
  end if
  state_out%n_sub = state_in%n_sub
end subroutine state_matrix_assign
```

Determine the indices of all diagonal matrix elements.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: get_diagonal_entries => state_matrix_get_diagonal_entries
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_get_diagonal_entries (state, i)
  class(state_matrix_t), intent(in) :: state
  integer, dimension(:), allocatable, intent(out) :: i
```

```
    integer, dimension(state%n_matrix_elements) :: tmp
    integer :: n
    type(state_iterator_t) :: it
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    n = 0
    call it%init (state)
    allocate (qn (it%depth))
    do while (it%is_valid ())
       qn = it%get_quantum_numbers ()
       if (all (qn%are_diagonal ())) then
          n = n + 1
          tmp(n) = it%get_me_index ()
       end if
       call it%advance ()
    end do
    allocate (i(n))
    if (n > 0) i = tmp(:n)
  end subroutine state_matrix_get_diagonal_entries
```

Normalize all matrix elements, i.e., multiply by a common factor. Assuming that the factor is nonzero, of course.

⟨*State matrices: state matrix: TBP*⟩+≡

```
  procedure :: renormalize => state_matrix_renormalize
```

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_matrix_renormalize (state, factor)
    class(state_matrix_t), intent(inout) :: state
    complex(default), intent(in) :: factor
    state%me = state%me * factor
  end subroutine state_matrix_renormalize
```

Renormalize the state matrix by its trace, if nonzero. The renormalization is reflected in the state-matrix norm.

⟨*State matrices: state matrix: TBP*⟩+≡

```
  procedure :: normalize_by_trace => state_matrix_normalize_by_trace
```

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_matrix_normalize_by_trace (state)
    class(state_matrix_t), intent(inout) :: state
    real(default) :: trace
    trace = state%trace ()
    if (trace /= 0) then
       state%me = state%me / trace
       state%norm = state%norm * trace
    end if
  end subroutine state_matrix_normalize_by_trace
```

Analogous, but renormalize by maximal (absolute) value.

⟨*State matrices: state matrix: TBP*⟩+≡

```
  procedure :: normalize_by_max => state_matrix_normalize_by_max
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_normalize_by_max (state)
  class(state_matrix_t), intent(inout) :: state
  real(default) :: m
  m = maxval (abs (state%me))
  if (m /= 0) then
     state%me = state%me / m
     state%norm = state%norm * m
  end if
end subroutine state_matrix_normalize_by_max
```

Explicitly set the norm of a state matrix.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: set_norm => state_matrix_set_norm
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_set_norm (state, norm)
  class(state_matrix_t), intent(inout) :: state
  real(default), intent(in) :: norm
  state%norm = norm
end subroutine state_matrix_set_norm
```

Return the sum of all matrix element values.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: sum => state_matrix_sum
```

⟨*State matrices: procedures*⟩+≡

```
function state_matrix_sum (state) result (value)
  complex(default) :: value
  class(state_matrix_t), intent(in) :: state
  value = sum (state%me)
end function state_matrix_sum
```

Return the trace of a state matrix, i.e., the sum over all diagonal values. If `qn_in` is provided, only branches that match this quantum-numbers array are considered.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: trace => state_matrix_trace
```

⟨*State matrices: procedures*⟩+≡

```
function state_matrix_trace (state, qn_in) result (trace)
  complex(default) :: trace
  class(state_matrix_t), intent(in), target :: state
  type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_in
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  type(state_iterator_t) :: it
  allocate (qn (state%get_depth ()))
  trace = 0
  call it%init (state)
  do while (it%is_valid ())
     qn = it%get_quantum_numbers ()
     if (present (qn_in)) then
        if (.not. all (qn .match. qn_in)) then
```

```
            call it%advance ();  cycle
         end if
      end if
      if (all (qn%are_diagonal ())) then
         trace = trace + it%get_matrix_element ()
      end if
      call it%advance ()
   end do
end function state_matrix_trace
```

Append new states which are color-contracted versions of the existing states. The matrix element index of each color contraction coincides with the index of its origin, so no new matrix elements are generated. After this operation, no `freeze` must be performed anymore.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: add_color_contractions => state_matrix_add_color_contractions
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_add_color_contractions (state)
  class(state_matrix_t), intent(inout), target :: state
  type(state_iterator_t) :: it
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn_con
  integer, dimension(:), allocatable :: me_index
  integer :: depth, n_me, i, j
  depth = state%get_depth ()
  n_me = state%get_n_matrix_elements ()
  allocate (qn (depth, n_me))
  allocate (me_index (n_me))
  i = 0
  call it%init (state)
  do while (it%is_valid ())
     i = i + 1
     qn(:,i) = it%get_quantum_numbers ()
     me_index(i) = it%get_me_index ()
     call it%advance ()
  end do
  do i = 1, n_me
     call quantum_number_array_make_color_contractions (qn(:,i), qn_con)
     do j = 1, size (qn_con, 2)
        call state%add_state (qn_con(:,j), index = me_index(i))
     end do
  end do
end subroutine state_matrix_add_color_contractions
```

This procedure merges two state matrices of equal depth. For each quantum number (flavor, color, helicity), we take the entry from the first argument where defined, otherwise the second one. (If both are defined, we get an off-diagonal matrix.) The resulting trie combines the information of the input tries in all possible ways. Note that values are ignored, all values in the result are zero.

⟨*State matrices: public*⟩+≡

```
public :: merge_state_matrices
```

⟨*State matrices: procedures*⟩+≡

```
subroutine merge_state_matrices (state1, state2, state3)
  type(state_matrix_t), intent(in), target :: state1, state2
  type(state_matrix_t), intent(out) :: state3
  type(state_iterator_t) :: it1, it2
  type(quantum_numbers_t), dimension(state1%depth) :: qn1, qn2
  if (state1%depth /= state2%depth) then
     call state1%write ()
     call state2%write ()
     call msg_bug ("State matrices merge impossible: incompatible depths")
  end if
  call state3%init ()
  call it1%init (state1)
  do while (it1%is_valid ())
     qn1 = it1%get_quantum_numbers ()
     call it2%init (state2)
     do while (it2%is_valid ())
        qn2 = it2%get_quantum_numbers ()
        call state3%add_state (qn1 .merge. qn2)
        call it2%advance ()
     end do
     call it1%advance ()
  end do
  call state3%freeze ()
end subroutine merge_state_matrices
```

Multiply matrix elements from two state matrices. Choose the elements as given by the integer index arrays, multiply them and store the sum of products in the indicated matrix element. The suffixes mean: c=conjugate first factor; f=include weighting factor.

Note that the `dot_product` intrinsic function conjugates its first complex argument. This is intended for the `c` suffix case, but must be reverted for the plain-product case.

We provide analogous subroutines for just summing over state matrix entries. The `evaluate_sum` variant includes the state-matrix norm in the evaluation, the `evaluate_me_sum` takes into account just the matrix elements proper.

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: evaluate_product => state_matrix_evaluate_product
procedure :: evaluate_product_cf => state_matrix_evaluate_product_cf
procedure :: evaluate_square_c => state_matrix_evaluate_square_c
procedure :: evaluate_sum => state_matrix_evaluate_sum
procedure :: evaluate_me_sum => state_matrix_evaluate_me_sum
```

⟨*State matrices: procedures*⟩+≡

```
pure subroutine state_matrix_evaluate_product &
     (state, i, state1, state2, index1, index2)
  class(state_matrix_t), intent(inout) :: state
  integer, intent(in) :: i
  type(state_matrix_t), intent(in) :: state1, state2
  integer, dimension(:), intent(in) :: index1, index2
  state%me(i) = &
       dot_product (conjg (state1%me(index1)), state2%me(index2))
  state%norm = state1%norm * state2%norm
```

```
end subroutine state_matrix_evaluate_product

pure subroutine state_matrix_evaluate_product_cf &
     (state, i, state1, state2, index1, index2, factor)
  class(state_matrix_t), intent(inout) :: state
  integer, intent(in) :: i
  type(state_matrix_t), intent(in) :: state1, state2
  integer, dimension(:), intent(in) :: index1, index2
  complex(default), dimension(:), intent(in) :: factor
  state%me(i) = &
       dot_product (state1%me(index1), factor * state2%me(index2))
  state%norm = state1%norm * state2%norm
end subroutine state_matrix_evaluate_product_cf

pure subroutine state_matrix_evaluate_square_c (state, i, state1, index1)
  class(state_matrix_t), intent(inout) :: state
  integer, intent(in) :: i
  type(state_matrix_t), intent(in) :: state1
  integer, dimension(:), intent(in) :: index1
  state%me(i) = &
       dot_product (state1%me(index1), state1%me(index1))
  state%norm = abs (state1%norm) ** 2
end subroutine state_matrix_evaluate_square_c

pure subroutine state_matrix_evaluate_sum (state, i, state1, index1)
  class(state_matrix_t), intent(inout) :: state
  integer, intent(in) :: i
  type(state_matrix_t), intent(in) :: state1
  integer, dimension(:), intent(in) :: index1
  state%me(i) = &
       sum (state1%me(index1)) * state1%norm
end subroutine state_matrix_evaluate_sum

pure subroutine state_matrix_evaluate_me_sum (state, i, state1, index1)
  class(state_matrix_t), intent(inout) :: state
  integer, intent(in) :: i
  type(state_matrix_t), intent(in) :: state1
  integer, dimension(:), intent(in) :: index1
  state%me(i) = sum (state1%me(index1))
end subroutine state_matrix_evaluate_me_sum
```

Outer product (of states and matrix elements):

⟨*State matrices: public*⟩+≡

```
public :: outer_multiply
```

⟨*State matrices: interfaces*⟩+≡

```
interface outer_multiply
   module procedure outer_multiply_pair
   module procedure outer_multiply_array
end interface
```

This procedure constructs the outer product of two state matrices.

⟨*State matrices: procedures*⟩+≡

```
subroutine outer_multiply_pair (state1, state2, state3)
  type(state_matrix_t), intent(in), target :: state1, state2
  type(state_matrix_t), intent(out) :: state3
  type(state_iterator_t) :: it1, it2
  type(quantum_numbers_t), dimension(state1%depth) :: qn1
  type(quantum_numbers_t), dimension(state2%depth) :: qn2
  type(quantum_numbers_t), dimension(state1%depth+state2%depth) :: qn3
  complex(default) :: val1, val2
  call state3%init (store_values=.true.)
  call it1%init (state1)
  do while (it1%is_valid ())
     qn1 = it1%get_quantum_numbers ()
     val1 = it1%get_matrix_element ()
     call it2%init (state2)
     do while (it2%is_valid ())
        qn2 = it2%get_quantum_numbers ()
        val2 = it2%get_matrix_element ()
        qn3(:state1%depth) = qn1
        qn3(state1%depth+1:) = qn2
        call state3%add_state (qn3, value=val1 * val2)
        call it2%advance ()
     end do
     call it1%advance ()
  end do
  call state3%freeze ()
end subroutine outer_multiply_pair
```

This executes the above routine iteratively for an arbitrary number of state matrices.

⟨*State matrices: procedures*⟩+≡

```
subroutine outer_multiply_array (state_in, state_out)
  type(state_matrix_t), dimension(:), intent(in), target :: state_in
  type(state_matrix_t), intent(out) :: state_out
  type(state_matrix_t), dimension(:), allocatable, target :: state_tmp
  integer :: i, n
  n = size (state_in)
  select case (n)
  case (0)
     call state_out%init ()
  case (1)
     state_out = state_in(1)
  case (2)
     call outer_multiply_pair (state_in(1), state_in(2), state_out)
  case default
     allocate (state_tmp (n-2))
     call outer_multiply_pair (state_in(1), state_in(2), state_tmp(1))
     do i = 2, n - 2
        call outer_multiply_pair (state_tmp(i-1), state_in(i+1), state_tmp(i))
     end do
     call outer_multiply_pair (state_tmp(n-2), state_in(n), state_out)
     do i = 1, size(state_tmp)
        call state_tmp(i)%final ()
     end do
```

```
    end select
  end subroutine outer_multiply_array
```

11.1.5 Factorization

In physical events, the state matrix is factorized into single-particle state matrices. This is essentially a measurement.

In a simulation, we select one particular branch of the state matrix with a probability that is determined by the matrix elements at the leaves. (This makes sense only if the state matrix represents a squared amplitude.) The selection is based on a (random) value `x` between 0 and one that is provided as the third argument.

For flavor and color, we select a unique value for each particle. For polarization, we have three options (modes). Option 1 is to drop helicity information altogether and sum over all diagonal helicities. Option 2 is to select a unique diagonal helicity in the same way as flavor and color. Option 3 is, for each particle, to trace over all remaining helicities in order to obtain an array of independent single-particle helicity matrices.

Only branches that match the given quantum-number array `qn_in`, if present, are considered. For this array, color is ignored.

If the optional `correlated_state` is provided, it is assigned the correlated density matrix for the selected flavor-color branch, so multi-particle spin correlations remain available even if they are dropped in the single-particle density matrices. This should be done by the caller for the choice `FM_CORRELATED_HELICITY`, which otherwise is handled as `FM_IGNORE_HELICITY`.

The algorithm is as follows: First, we determine the normalization by summing over all diagonal matrix elements. In a second scan, we select one of the diagonal matrix elements by a cumulative comparison with the normalized random number. In the corresponding quantum number array, we undefine the helicity entries. Then, we scan the third time. For each branch that matches the selected quantum number array (i.e., definite flavor and color, arbitrary helicity), we determine its contribution to any of the single-particle state matrices. The matrix-element value is added if all other quantum numbers are diagonal, while the helicity of the chosen particle may be arbitrary; this helicity determines the branch in the single-particle state.

As a result, flavor and color quantum numbers are selected with the correct probability. Within this subset of states, each single-particle state matrix results from tracing over all other particles. Note that the single-particle state matrices are not normalized.

The flag `ok` is set to false if the matrix element sum is zero, so factorization is not possible. This can happen if an event did not pass cuts.

⟨*State matrices: parameters*⟩≡

```
integer, parameter, public :: FM_IGNORE_HELICITY = 1
integer, parameter, public :: FM_SELECT_HELICITY = 2
integer, parameter, public :: FM_FACTOR_HELICITY = 3
integer, parameter, public :: FM_CORRELATED_HELICITY = 4
```

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: factorize => state_matrix_factorize
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_matrix_factorize &
     (state, mode, x, ok, single_state, correlated_state, qn_in)
  class(state_matrix_t), intent(in), target :: state
  integer, intent(in) :: mode
  real(default), intent(in) :: x
  logical, intent(out) :: ok
  type(state_matrix_t), &
       dimension(:), allocatable, intent(out) :: single_state
  type(state_matrix_t), intent(out), optional :: correlated_state
  type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_in
  type(state_iterator_t) :: it
  real(default) :: s, xt
  complex(default) :: value
  integer :: i, depth
  type(quantum_numbers_t), dimension(:), allocatable :: qn, qn1
  type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
  logical, dimension(:), allocatable :: diagonal
  logical, dimension(:,:), allocatable :: mask
  ok = .true.
  if (x /= 0) then
     xt = x * abs (state%trace (qn_in))
  else
     xt = 0
  end if
  s = 0
  depth = state%get_depth ()
  allocate (qn (depth), qn1 (depth), diagonal (depth))
  call it%init (state)
  do while (it%is_valid ())
     qn = it%get_quantum_numbers ()
     if (present (qn_in)) then
        if (.not. all (qn .fhmatch. qn_in)) then
           call it%advance (); cycle
        end if
     end if
     if (all (qn%are_diagonal ())) then
        value = abs (it%get_matrix_element ())
        s = s + value
        if (s > xt)  exit
     end if
     call it%advance ()
  end do
  if (.not. it%is_valid ()) then
     if (s == 0)  ok = .false.
     call it%init (state)
  end if
  allocate (single_state (depth))
  do i = 1, depth
     call single_state(i)%init (store_values=.true.)
  end do
  if (present (correlated_state)) &
       call correlated_state%init (store_values=.true.)
  qn = it%get_quantum_numbers ()
```

```
select case (mode)
case (FM_SELECT_HELICITY)  ! single branch selected; shortcut
   do i = 1, depth
      call single_state(i)%add_state ([qn(i)], value=value)
   end do
   if (.not. present (correlated_state)) then
      do i = 1, size(single_state)
         call single_state(i)%freeze ()
      end do
      return
   end if
end select
allocate (qn_mask (depth))
call qn_mask%init (.false., .false., .false., .true.)
call qn%undefine (qn_mask)
select case (mode)
case (FM_FACTOR_HELICITY)
   allocate (mask (depth, depth))
   mask = .false.
   forall (i = 1:depth)  mask(i,i) = .true.
end select
call it%init (state)
do while (it%is_valid ())
   qn1 = it%get_quantum_numbers ()
   if (all (qn .match. qn1)) then
      !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
      !!! diagonal = qn1%are_diagonal ()
      do i = 1, depth
         diagonal(i) = qn1(i)%are_diagonal ()
      end do
      value = it%get_matrix_element ()
      select case (mode)
      case (FM_IGNORE_HELICITY, FM_CORRELATED_HELICITY)
         ! trace over diagonal states that match qn
         if (all (diagonal)) then
            do i = 1, depth
               call single_state(i)%add_state &
                    ([qn(i)], value=value, sum_values=.true.)
            end do
         end if
      case (FM_FACTOR_HELICITY)  ! trace over all other particles
         do i = 1, depth
            if (all (diagonal .or. mask(:,i))) then
               call single_state(i)%add_state &
                    ([qn1(i)], value=value, sum_values=.true.)
            end if
         end do
      end select
      if (present (correlated_state)) &
           call correlated_state%add_state (qn1, value=value)
   end if
   call it%advance ()
end do
do i = 1, depth
```

```
      call single_state(i)%freeze ()
    end do
    if (present (correlated_state)) &
         call correlated_state%freeze ()
end subroutine state_matrix_factorize
```

Auxiliary functions

⟨*State matrices: state matrix: TBP*⟩+≡

```
procedure :: get_polarization_density_matrix &
   => state_matrix_get_polarization_density_matrix
```

⟨*State matrices: procedures*⟩+≡

```
function state_matrix_get_polarization_density_matrix (state) result (pol_matrix)
  real(default), dimension(:,:), allocatable :: pol_matrix
  class(state_matrix_t), intent(in) :: state
  type(node_t), pointer :: current => null ()
  !!! What's the generic way to allocate the matrix?
  allocate (pol_matrix (4,4)); pol_matrix = 0
  if (associated (state%root%child_first)) then
     current => state%root%child_first
     do while (associated (current))
       call current%qn%write ()
       current => current%next
     end do
  else
     call msg_fatal ("Polarization state not allocated!")
  end if
end function state_matrix_get_polarization_density_matrix
```

Quantum-number matching

This feature allows us to check whether a given string of PDG values matches, in any ordering, any of the flavor combinations that the state matrix provides. We will also request the permutation of the successful match.

This type provides an account of the state's flavor content. We store all flavor combinations, as `pdg` values, in an array, assuming that the length is uniform.

We check only the entries selected by `mask_match`. Among those, only the entries selected by `mask_sort` are sorted and thus matched without respecting array element order. The entries that correspond to a true value in the associated `mask` are sorted. The mapping from the original state to the sorted state is given by the index array `map`.

⟨*State matrices: public*⟩+≡

```
public :: state_flv_content_t
```

⟨*State matrices: types*⟩+≡

```
type :: state_flv_content_t
   private
   integer, dimension(:,:), allocatable :: pdg
   integer, dimension(:,:), allocatable :: map
```

```
    logical, dimension(:), allocatable :: mask
  contains
  ⟨State matrices: state flv content: TBP⟩
end type state_flv_content_t
```

Output (debugging aid).

⟨*State matrices: state flv content: TBP*⟩≡

```
procedure :: write => state_flv_content_write
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_flv_content_write (state_flv, unit)
  class(state_flv_content_t), intent(in), target :: state_flv
  integer, intent(in), optional :: unit
  integer :: u, n, d, i, j
  u = given_output_unit (unit)
  d = size (state_flv%pdg, 1)
  n = size (state_flv%pdg, 2)
  do i = 1, n
     write (u, "(2x,'PDG =')", advance="no")
     do j = 1, d
        write (u, "(1x,I0)", advance="no")  state_flv%pdg(j,i)
     end do
     write (u, "(' :: map = (')", advance="no")
     do j = 1, d
        write (u, "(1x,I0)", advance="no")  state_flv%map(j,i)
     end do
     write (u, "(' )')")
  end do
end subroutine state_flv_content_write
```

Initialize with table length and mask. Each row of the `map` array, of length d, is initialized with $(0, 1, \ldots, d)$.

⟨*State matrices: state flv content: TBP*⟩+≡

```
procedure :: init => state_flv_content_init
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_flv_content_init (state_flv, n, mask)
  class(state_flv_content_t), intent(out) :: state_flv
  integer, intent(in) :: n
  logical, dimension(:), intent(in) :: mask
  integer :: d, i
  d = size (mask)
  allocate (state_flv%pdg (d, n), source = 0)
  allocate (state_flv%map (d, n), source = spread ([(i, i = 1, d)], 2, n))
  allocate (state_flv%mask (d), source = mask)
end subroutine state_flv_content_init
```

Manually fill the entries, one flavor set and mapping at a time.

⟨*State matrices: state flv content: TBP*⟩+≡

```
procedure :: set_entry => state_flv_content_set_entry
```

⟨*State matrices: procedures*⟩+≡

```
subroutine state_flv_content_set_entry (state_flv, i, pdg, map)
```

```
    class(state_flv_content_t), intent(inout) :: state_flv
    integer, intent(in) :: i
    integer, dimension(:), intent(in) :: pdg, map
    state_flv%pdg(:,i) = pdg
    where (map /= 0)
       state_flv%map(:,i) = map
    end where
  end subroutine state_flv_content_set_entry
```

Given a state matrix, determine the flavor content. That is, scan the state matrix and extract flavor only, build a new state matrix from that.

⟨*State matrices: state flv content: TBP*⟩+≡

```
  procedure :: fill => state_flv_content_fill
```

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_flv_content_fill &
       (state_flv, state_full, mask)
    class(state_flv_content_t), intent(out) :: state_flv
    type(state_matrix_t), intent(in), target :: state_full
    logical, dimension(:), intent(in) :: mask
    type(state_matrix_t), target :: state_tmp
    type(state_iterator_t) :: it
    type(flavor_t), dimension(:), allocatable :: flv
    integer, dimension(:), allocatable :: pdg, pdg_subset
    integer, dimension(:), allocatable :: idx, map_subset, idx_subset, map
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    integer :: n, d, c, i, j
    call state_tmp%init ()
    d = state_full%get_depth ()
    allocate (flv (d), qn (d), pdg (d), idx (d), map (d))
    idx = [(i, i = 1, d)]
    c = count (mask)
    allocate (pdg_subset (c), map_subset (c), idx_subset (c))
    call it%init (state_full)
    do while (it%is_valid ())
       flv = it%get_flavor ()
       call qn%init (flv)
       call state_tmp%add_state (qn)
       call it%advance ()
    end do
    n = state_tmp%get_n_leaves ()
    call state_flv%init (n, mask)
    i = 0
    call it%init (state_tmp)
    do while (it%is_valid ())
       i = i + 1
       flv = it%get_flavor ()
       !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
       !!! pdg = flv%get_pdg ()
       do j = 1, d
          pdg(j) = flv(j)%get_pdg ()
       end do
       idx_subset = pack (idx, mask)
       pdg_subset = pack (pdg, mask)
```

```
      map_subset = order_abs (pdg_subset)
      map = unpack (idx_subset (map_subset), mask, idx)
      call state_flv%set_entry (i, &
           unpack (pdg_subset(map_subset), mask, pdg), &
           order (map))
      call it%advance ()
    end do
    call state_tmp%final ()
  end subroutine state_flv_content_fill
```

Match a given flavor string against the flavor content. We sort the input string and check whether it matches any of the stored strings. If yes, return the mapping.

Only PDG entries under the preset mask are sorted before matching. The other entries must match exactly (i.e., without reordering). A zero entry matches anything. In any case, the length of the PDG string must be equal to the length d of the individual flavor-state entries.

⟨*State matrices: state flv content: TBP*⟩+≡

```
  procedure :: match => state_flv_content_match
```

⟨*State matrices: procedures*⟩+≡

```
  subroutine state_flv_content_match (state_flv, pdg, success, map)
    class(state_flv_content_t), intent(in) :: state_flv
    integer, dimension(:), intent(in) :: pdg
    logical, intent(out) :: success
    integer, dimension(:), intent(out) :: map
    integer, dimension(:), allocatable :: pdg_subset, pdg_sorted, map1, map2
    integer, dimension(:), allocatable :: idx, map_subset, idx_subset
    integer :: i, n, c, d
    c = count (state_flv%mask)
    d = size (state_flv%pdg, 1)
    n = size (state_flv%pdg, 2)
    allocate (idx (d), source = [(i, i = 1, d)])
    allocate (idx_subset (c), pdg_subset (c), map_subset (c))
    allocate (pdg_sorted (d), map1 (d), map2 (d))
    idx_subset = pack (idx, state_flv%mask)
    pdg_subset = pack (pdg, state_flv%mask)
    map_subset = order_abs (pdg_subset)
    pdg_sorted = unpack (pdg_subset(map_subset), state_flv%mask, pdg)
    success = .false.
    do i = 1, n
      if (all (pdg_sorted == state_flv%pdg(:,i) &
           .or. pdg_sorted == 0)) then
        success = .true.
        exit
      end if
    end do
    if (success) then
      map1 = state_flv%map(:,i)
      map2 = unpack (idx_subset(map_subset), state_flv%mask, idx)
      map = map2(map1)
      where (pdg == 0)  map = 0
    end if
```

```
    end subroutine state_flv_content_match
```

⟨*State matrices: procedures*⟩+≡

```
    elemental function pacify_complex (c_in) result (c_pac)
      complex(default), intent(in) :: c_in
      complex(default) :: c_pac
      c_pac = c_in
      if (real(c_pac) == -real(c_pac)) then
         c_pac = &
              cmplx (0._default, aimag(c_pac), kind=default)
      end if
      if (aimag(c_pac) == -aimag(c_pac)) then
         c_pac = &
              cmplx (real(c_pac), 0._default, kind=default)
      end if
    end function pacify_complex
```

11.1.6 Unit tests

Test module, followed by the corresponding implementation module.

⟨state_matrices_ut.f90⟩≡

```
  ⟨File header⟩

  module state_matrices_ut
    use unit_tests
    use state_matrices_uti

  ⟨Standard module head⟩

  ⟨State matrices: public test⟩

  contains

  ⟨State matrices: test driver⟩

  end module state_matrices_ut
```

⟨state_matrices_uti.f90⟩≡

```
  ⟨File header⟩

  module state_matrices_uti

  ⟨Use kinds⟩
    use io_units
    use format_defs, only: FMT_19
    use flavors
    use colors
    use helicities
    use quantum_numbers

    use state_matrices
```

```
⟨Standard module head⟩

⟨State matrices: test declarations⟩

contains

⟨State matrices: tests⟩

end module state_matrices_uti
```

API: driver for the unit tests below.

⟨*State matrices: public test*⟩≡

```
public :: state_matrix_test
```

⟨*State matrices: test driver*⟩≡

```
subroutine state_matrix_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨State matrices: execute tests⟩
end subroutine state_matrix_test
```

Create two quantum states of equal depth and merge them.

⟨*State matrices: execute tests*⟩≡

```
call test (state_matrix_1, "state_matrix_1", &
     "check merge of quantum states of equal depth", &
     u, results)
```

⟨*State matrices: test declarations*⟩≡

```
public :: state_matrix_1
```

⟨*State matrices: tests*⟩≡

```
subroutine state_matrix_1 (u)
  integer, intent(in) :: u
  type(state_matrix_t) :: state1, state2, state3
  type(flavor_t), dimension(3) :: flv
  type(color_t), dimension(3) :: col
  type(quantum_numbers_t), dimension(3) :: qn

  write (u, "(A)")  "* Test output: state_matrix_1"
  write (u, "(A)")  "*   Purpose: create and merge two quantum states"
  write (u, "(A)")

  write (u, "(A)")  "*  Initialization"
  write (u, "(A)")

  write (u, "(A)")  "*  State matrix 1"
  write (u, "(A)")

  call state1%init ()
  call flv%init ([1, 2, 11])
  call qn%init (flv, helicity ([ 1, 1, 1]))
  call state1%add_state (qn)
  call qn%init (flv, helicity ([ 1, 1, 1], [-1, 1, -1]))
  call state1%add_state (qn)
  call state1%freeze ()
  call state1%write (u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "*  State matrix 2"
    write (u, "(A)")

    call state2%init ()
    call col(1)%init ([501])
    call col(2)%init ([-501])
    call col(3)%init ([0])
    call qn%init (col, helicity ([-1, -1, 0]))
    call state2%add_state (qn)
    call col(3)%init ([99])
    call qn%init (col, helicity ([-1, -1, 0]))
    call state2%add_state (qn)
    call state2%freeze ()
    call state2%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Merge the state matrices"
    write (u, "(A)")

    call merge_state_matrices (state1, state2, state3)
    call state3%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Collapse the state matrix"
    write (u, "(A)")

    call state3%collapse (quantum_numbers_mask (.false., .false., &
         [.true.,.false.,.false.]))
    call state3%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"
    write (u, "(A)")

    call state1%final ()
    call state2%final ()
    call state3%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: state_matrix_1"
    write (u, "(A)")

  end subroutine state_matrix_1
```

Create a correlated three-particle state matrix and factorize it.

⟨*State matrices: execute tests*⟩+≡

```
  call test (state_matrix_2, "state_matrix_2", &
       "check factorizing 3-particle state matrix", &
       u, results)
```

⟨*State matrices: test declarations*⟩+≡

```
  public :: state_matrix_2
```

⟨*State matrices: tests*⟩+≡

```
subroutine state_matrix_2 (u)
  integer, intent(in) :: u
  type(state_matrix_t) :: state
  type(state_matrix_t), dimension(:), allocatable :: single_state
  type(state_matrix_t) :: correlated_state
  integer :: f, h11, h12, h21, h22, i, mode
  type(flavor_t), dimension(2) :: flv
  type(color_t), dimension(2) :: col
  type(helicity_t), dimension(2) :: hel
  type(quantum_numbers_t), dimension(2) :: qn
  logical :: ok

  write (u, "(A)")
  write (u, "(A)")  "* Test output: state_matrix_2"
  write (u, "(A)")  "*   Purpose: factorize correlated 3-particle state"
  write (u, "(A)")

  write (u, "(A)")  "*  Initialization"
  write (u, "(A)")

  call state%init ()
  do f = 1, 2
     do h11 = -1, 1, 2
        do h12 = -1, 1, 2
           do h21 = -1, 1, 2
              do h22 = -1, 1, 2
                 call flv%init ([f, -f])
                 call col(1)%init ([1])
                 call col(2)%init ([-1])
                 call hel%init ([h11,h12], [h21, h22])
                 call qn%init (flv, col, hel)
                 call state%add_state (qn)
              end do
           end do
        end do
     end do
  end do
  call state%freeze ()
  call state%write (u)

  write (u, "(A)")
  write (u, "(A,'(')," // FMT_19 // ",',')," // FMT_19 // ",')')") &
       "* Trace = ", state%trace ()
  write (u, "(A)")

  do mode = 1, 3
     write (u, "(A)")
     write (u, "(A,I1)")  "* Mode = ", mode
     call state%factorize &
          (mode, 0.15_default, ok, single_state, correlated_state)
     do i = 1, size (single_state)
        write (u, "(A)")
        call single_state(i)%write (u)
```

```
         write (u, "(A,'(',' // FMT_19 // ",',',' // FMT_19 // ",')')") &
              "Trace = ", single_state(i)%trace ()
      end do
      write (u, "(A)")
      call correlated_state%write (u)
      write (u, "(A,'(',' // FMT_19 // ",',',' // FMT_19 // ",')')")  &
           "Trace = ", correlated_state%trace ()
      do i = 1, size(single_state)
         call single_state(i)%final ()
      end do
      call correlated_state%final ()
   end do

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call state%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: state_matrix_2"

end subroutine state_matrix_2
```

Create a colored state matrix and add color contractions.

⟨*State matrices: execute tests*⟩+≡

```
call test (state_matrix_3, "state_matrix_3", &
     "check factorizing 3-particle state matrix", &
     u, results)
```

⟨*State matrices: test declarations*⟩+≡

```
public :: state_matrix_3
```

⟨*State matrices: tests*⟩+≡

```
subroutine state_matrix_3 (u)
  use physics_defs, only: HADRON_REMNANT_TRIPLET, HADRON_REMNANT_OCTET
  integer, intent(in) :: u
  type(state_matrix_t) :: state
  type(flavor_t), dimension(4) :: flv
  type(color_t), dimension(4) :: col
  type(quantum_numbers_t), dimension(4) :: qn

  write (u, "(A)")  "* Test output: state_matrix_3"
  write (u, "(A)")  "*   Purpose: add color connections to colored state"
  write (u, "(A)")

  write (u, "(A)")  "*  Initialization"
  write (u, "(A)")

  call state%init ()
  call flv%init ([ 1, -HADRON_REMNANT_TRIPLET, -1, HADRON_REMNANT_TRIPLET ])
  call col(1)%init ([17])
  call col(2)%init ([-17])
  call col(3)%init ([-19])
  call col(4)%init ([19])
  call qn%init (flv, col)
```

```
    call state%add_state (qn)
    call flv%init ([ 1, -HADRON_REMNANT_TRIPLET, 21, HADRON_REMNANT_OCTET ])
    call col(1)%init ([17])
    call col(2)%init ([-17])
    call col(3)%init ([3, -5])
    call col(4)%init ([5, -3])
    call qn%init (flv, col)
    call state%add_state (qn)
    call state%freeze ()

    write (u, "(A)") "* State:"
    write (u, "(A)")

    call state%write (u)
    call state%add_color_contractions ()

    write (u, "(A)") "* State with contractions:"
    write (u, "(A)")

    call state%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call state%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: state_matrx_3"

  end subroutine state_matrix_3
```

Create a correlated three-particle state matrix, write it to file and read again.

⟨*State matrices: execute tests*⟩+≡

```
  call test (state_matrix_4, "state_matrix_4", &
       "check raw I/O", &
       u, results)
```

⟨*State matrices: test declarations*⟩+≡

```
  public :: state_matrix_4
```

⟨*State matrices: tests*⟩+≡

```
  subroutine state_matrix_4 (u)
    integer, intent(in) :: u
    type(state_matrix_t), allocatable :: state
    integer :: f, h11, h12, h21, h22, i
    type(flavor_t), dimension(2) :: flv
    type(color_t), dimension(2) :: col
    type(helicity_t), dimension(2) :: hel
    type(quantum_numbers_t), dimension(2) :: qn
    integer :: unit, iostat

    write (u, "(A)")
    write (u, "(A)")  "* Test output: state_matrix_4"
    write (u, "(A)")  "*   Purpose: raw I/O for correlated 3-particle state"
    write (u, "(A)")
```

```
write (u, "(A)")  "*  Initialization"
write (u, "(A)")

allocate (state)

call state%init ()
do f = 1, 2
   do h11 = -1, 1, 2
      do h12 = -1, 1, 2
         do h21 = -1, 1, 2
            do h22 = -1, 1, 2
               call flv%init ([f, -f])
               call col(1)%init ([1])
               call col(2)%init ([-1])
               call hel%init ([h11, h12], [h21, h22])
               call qn%init (flv, col, hel)
               call state%add_state (qn)
            end do
         end do
      end do
   end do
end do
call state%freeze ()

call state%set_norm (3._default)
do i = 1, state%get_n_leaves ()
   call state%set_matrix_element (i, cmplx (2 * i, 2 * i + 1, default))
end do

call state%write (u)

write (u, "(A)")
write (u, "(A)")  "* Write to file and read again "
write (u, "(A)")

unit = free_unit ()
open (unit, action="readwrite", form="unformatted", status="scratch")
call state%write_raw (unit)
call state%final ()
deallocate (state)

allocate(state)
rewind (unit)
call state%read_raw (unit, iostat=iostat)
close (unit)

call state%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call state%final ()
deallocate (state)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: state_matrix_4"

  end subroutine state_matrix_4
```

Create a flavor-content object for a given state matrix and match it against trial flavor (i.e., PDG) strings.

⟨*State matrices: execute tests*⟩+≡

```
  call test (state_matrix_5, "state_matrix_5", &
       "check flavor content", &
       u, results)
```

⟨*State matrices: test declarations*⟩+≡

```
  public :: state_matrix_5
```

⟨*State matrices: tests*⟩+≡

```
  subroutine state_matrix_5 (u)
    integer, intent(in) :: u
    type(state_matrix_t), allocatable, target :: state
    type(state_iterator_t) :: it
    type(state_flv_content_t), allocatable :: state_flv
    type(flavor_t), dimension(4) :: flv1, flv2, flv3, flv4
    type(color_t), dimension(4) :: col1, col2
    type(helicity_t), dimension(4) :: hel1, hel2, hel3
    type(quantum_numbers_t), dimension(4) :: qn
    logical, dimension(4) :: mask

    write (u, "(A)")  "* Test output: state_matrix_5"
    write (u, "(A)")  "*   Purpose: check flavor-content state"
    write (u, "(A)")

    write (u, "(A)")  "* Set up arbitrary state matrix"
    write (u, "(A)")

    call flv1%init ([1, 4, 2, 7])
    call flv2%init ([1, 3,-3, 8])
    call flv3%init ([5, 6, 3, 7])
    call flv4%init ([6, 3, 5, 8])
    call hel1%init ([0, 1, -1, 0])
    call hel2%init ([0, 1, 1, 1])
    call hel3%init ([1, 0, 0, 0])
    call col1(1)%init ([0])
    call col1(2)%init ([0])
    call col1(3)%init ([0])
    call col1(4)%init ([0])
    call col2(1)%init ([5, -6])
    call col2(2)%init ([0])
    call col2(3)%init ([6, -5])
    call col2(4)%init ([0])

    allocate (state)
    call state%init ()
    call qn%init (flv1, col1, hel1)
    call state%add_state (qn)
```

```
call qn%init (flv1, col1, hel2)
call state%add_state (qn)
call qn%init (flv3, col1, hel3)
call state%add_state (qn)
call qn%init (flv4, col1, hel3)
call state%add_state (qn)
call qn%init (flv1, col2, hel3)
call state%add_state (qn)
call qn%init (flv2, col2, hel2)
call state%add_state (qn)
call qn%init (flv2, col2, hel1)
call state%add_state (qn)
call qn%init (flv2, col1, hel1)
call state%add_state (qn)
call qn%init (flv3, col1, hel1)
call state%add_state (qn)
call qn%init (flv3, col2, hel3)
call state%add_state (qn)
call qn%init (flv1, col1, hel1)
call state%add_state (qn)

write (u, "(A)")  "* Quantum number content"
write (u, "(A)")

call it%init (state)
do while (it%is_valid ())
   call quantum_numbers_write (it%get_quantum_numbers (), u)
   write (u, *)
   call it%advance ()
end do

write (u, "(A)")
write (u, "(A)")  "* Extract the flavor content"
write (u, "(A)")

mask = [.true., .true., .true., .false.]

allocate (state_flv)
call state_flv%fill (state, mask)
call state_flv%write (u)

write (u, "(A)")
write (u, "(A)")  "* Match trial sets"
write (u, "(A)")

call check ([1, 2, 3, 0])
call check ([1, 4, 2, 0])
call check ([4, 2, 1, 0])
call check ([1, 3, -3, 0])
call check ([1, -3, 3, 0])
call check ([6, 3, 5, 0])

write (u, "(A)")
write (u, "(A)")  "* Determine the flavor content with mask"
```

```
  write (u, "(A)")

  mask = [.false., .true., .true., .false.]

  call state_flv%fill (state, mask)
  call state_flv%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Match trial sets"
  write (u, "(A)")

  call check ([1, 2, 3, 0])
  call check ([1, 4, 2, 0])
  call check ([4, 2, 1, 0])
  call check ([1, 3, -3, 0])
  call check ([1, -3, 3, 0])
  call check ([6, 3, 5, 0])

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  deallocate (state_flv)

  call state%final ()
  deallocate (state)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: state_matrix_5"

contains

  subroutine check (pdg)
    integer, dimension(4), intent(in) :: pdg
    integer, dimension(4) :: map
    logical :: success
    call state_flv%match (pdg, success, map)
    write (u, "(2x,4(1x,I0),':',1x,L1)", advance="no")  pdg, success
    if (success) then
       write (u, "(2x,'map = (',4(1x,I0),' )')")  map
    else
       write (u, *)
    end if
  end subroutine check

end subroutine state_matrix_5
```

Create a state matrix with full flavor, color and helicity information. Afterwards, reduce such that it is only differential in flavor and initial-state helicities. This is used when preparing states for beam- polarized computations with external matrix element providers.

⟨*State matrices: execute tests*⟩+≡

```
call test (state_matrix_6, "state_matrix_6", &
           "check state matrix reduction", &
```

```
            u, results)
```

⟨*State matrices: test declarations*⟩+≡

```
    public :: state_matrix_6
```

⟨*State matrices: tests*⟩+≡

```
    subroutine state_matrix_6 (u)
      integer, intent(in) :: u
      type(state_matrix_t), allocatable :: state_orig, state_reduced
      type(flavor_t), dimension(4) :: flv
      type(helicity_t), dimension(4) :: hel
      type(color_t), dimension(4) :: col
      type(quantum_numbers_t), dimension(4) :: qn
      type(quantum_numbers_mask_t), dimension(4) :: qn_mask
      integer :: h1, h2, h3 , h4
      integer :: n_states = 0

      write (u, "(A)") "* Test output: state_matrix_6"
      write (u, "(A)") "* Purpose: Check state matrix reduction"
      write (u, "(A)")

      write (u, "(A)") "* Set up helicity-diagonal state matrix"
      write (u, "(A)")

      allocate (state_orig)
      call state_orig%init ()

      call flv%init ([11, -11, 1, -1])
      call col(3)%init ([1])
      call col(4)%init ([-1])
      do h1 = -1, 1, 2
         do h2 = -1, 1, 2
            do h3 = -1, 1, 2
               do h4 = -1, 1, 2
                  n_states = n_states + 1
                  call hel%init ([h1, h2, h3, h4], [h1, h2, h3, h4])
                  call qn%init (flv, col, hel)
                  call state_orig%add_state (qn)
               end do
            end do
         end do
      end do
      call state_orig%freeze ()

      write (u, "(A)") "* Original state: "
      write (u, "(A)")
      call state_orig%write (u)

      write (u, "(A)")
      write (u, "(A)") "* Setup quantum mask: "

      call qn_mask%init ([.false., .false., .false., .false.], &
                         [.true., .true., .true., .true.], &
                         [.false., .false., .true., .true.])
      call quantum_numbers_mask_write (qn_mask, u)
```

```
    write (u, "(A)")
    write (u, "(A)") "* Reducing the state matrix using above mask"
    write (u, "(A)")
    allocate (state_reduced)
    call state_orig%reduce (qn_mask, state_reduced)

    write (u, "(A)") "* Reduced state matrix: "
    call state_reduced%write (u)

    write (u, "(A)")
    write (u, "(A)") "* Test output end: state_matrix_6"


  end subroutine state_matrix_6
```

11.2 Interactions

This module defines the `interaction_t` type. It is an extension of the `state_matrix_t` type.

The state matrix is a representation of a multi-particle density matrix. It implements all possible flavor, color, and quantum-number assignments of the entries in a generic density matrix, and it can hold a complex matrix element for each entry. (Note that this matrix can hold non-diagonal entries in color and helicity space.) The `interaction_t` object associates this with a list of momenta, such that the whole object represents a multi-particle state.

The `interaction_t` holds information about which particles are incoming, virtual (i.e., kept for the records), or outgoing. Each particle can be associated to a source within another interaction. This allows us to automatically fill those interaction momenta which have been computed or defined elsewhere. It also contains internal parent-child relations and flags for (virtual) particles which are to be treated as resonances.

A quantum-number mask array summarizes, for each particle within the interaction, the treatment of flavor, color, or helicity (expose or ignore). A list of locks states which particles are bound to have an identical quantum-number mask. This is useful when the mask is changed at one place.

⟨`interactions.f90`⟩≡

```
⟨File header⟩

module interactions

⟨Use kinds⟩
  use io_units
  use diagnostics
  use sorting
  use lorentz
  use flavors
  use colors
  use helicities
  use quantum_numbers
  use state_matrices
```

⟨*Standard module head*⟩

⟨*Interactions: public*⟩

⟨*Interactions: types*⟩

⟨*Interactions: interfaces*⟩

```
contains
```

⟨*Interactions: procedures*⟩

```
end module interactions
```

11.2.1 External interaction links

Each particle in an interaction can have a link to a corresponding particle in another interaction. This allows to fetch the momenta of incoming or virtual particles from the interaction where they are defined. The link object consists of a pointer to the interaction and an index.

⟨*Interactions: types*⟩≡

```
type :: external_link_t
   private
   type(interaction_t), pointer :: int => null ()
   integer :: i
end type external_link_t
```

Set an external link.

⟨*Interactions: procedures*⟩≡

```
subroutine external_link_set (link, int, i)
  type(external_link_t), intent(out) :: link
  type(interaction_t), target, intent(in) :: int
  integer, intent(in) :: i
  if (i /= 0) then
     link%int => int
     link%i = i
  end if
end subroutine external_link_set
```

Reassign an external link to a new interaction (which should be an image of the original target).

⟨*Interactions: procedures*⟩+≡

```
subroutine external_link_reassign (link, int_src, int_target)
  type(external_link_t), intent(inout) :: link
  type(interaction_t), intent(in) :: int_src
  type(interaction_t), intent(in), target :: int_target
  if (associated (link%int)) then
     if (link%int%tag == int_src%tag)  link%int => int_target
  end if
end subroutine external_link_reassign
```

Return true if the link is set

⟨*Interactions: procedures*⟩+≡
```
function external_link_is_set (link) result (flag)
  logical :: flag
  type(external_link_t), intent(in) :: link
  flag = associated (link%int)
end function external_link_is_set
```

Return the interaction pointer.

⟨*Interactions: public*⟩≡
```
public :: external_link_get_ptr
```

⟨*Interactions: procedures*⟩+≡
```
function external_link_get_ptr (link) result (int)
  type(interaction_t), pointer :: int
  type(external_link_t), intent(in) :: link
  int => link%int
end function external_link_get_ptr
```

Return the index within that interaction

⟨*Interactions: public*⟩+≡
```
public :: external_link_get_index
```

⟨*Interactions: procedures*⟩+≡
```
function external_link_get_index (link) result (i)
  integer :: i
  type(external_link_t), intent(in) :: link
  i = link%i
end function external_link_get_index
```

Return a pointer to the momentum of the corresponding particle. If there is no association, return a null pointer.

⟨*Interactions: procedures*⟩+≡
```
function external_link_get_momentum_ptr (link) result (p)
  type(vector4_t), pointer :: p
  type(external_link_t), intent(in) :: link
  if (associated (link%int)) then
     p => link%int%p(link%i)
  else
     p => null ()
  end if
end function external_link_get_momentum_ptr
```

11.2.2 Internal relations

In addition to the external links, particles within the interaction have parent-child relations. Here, more than one link is possible, and we set up an array.

⟨*Interactions: types*⟩+≡
```
type :: internal_link_list_t
   private
   integer :: length = 0
```

```
    integer, dimension(:), allocatable :: link
  contains
  ⟨Interactions: internal link list: TBP⟩
end type internal_link_list_t
```

Output, non-advancing.

⟨*Interactions: internal link list: TBP*⟩≡

```
procedure :: write => internal_link_list_write
```

⟨*Interactions: procedures*⟩+≡

```
subroutine internal_link_list_write (object, unit)
  class(internal_link_list_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  do i = 1, object%length
     write (u, "(1x,I0)", advance="no")  object%link(i)
  end do
end subroutine internal_link_list_write
```

Append an item. Start with an array size of 2 and double the size if necessary.

Make sure that the indices are stored in ascending order. To this end, shift the existing entries right, starting from the end, as long as they are larger than the new entry.

⟨*Interactions: internal link list: TBP*⟩+≡

```
procedure :: append => internal_link_list_append
```

⟨*Interactions: procedures*⟩+≡

```
subroutine internal_link_list_append (link_list, link)
  class(internal_link_list_t), intent(inout) :: link_list
  integer, intent(in) :: link
  integer :: l, j
  integer, dimension(:), allocatable :: tmp
  l = link_list%length
  if (allocated (link_list%link)) then
     if (l == size (link_list%link)) then
        allocate (tmp (2 * l))
        tmp(:l) = link_list%link
        call move_alloc (from = tmp, to = link_list%link)
     end if
  else
     allocate (link_list%link (2))
  end if
  link_list%link(l+1) = link
  SHIFT_LINK_IN_PLACE: do j = l, 1, -1
     if (link >= link_list%link(j)) then
        exit SHIFT_LINK_IN_PLACE
     else
        link_list%link(j+1) = link_list%link(j)
        link_list%link(j) = link
     end if
  end do SHIFT_LINK_IN_PLACE
  link_list%length = l + 1
end subroutine internal_link_list_append
```

Return true if the link list is nonempty:

⟨*Interactions: internal link list: TBP*⟩+≡

```
procedure :: has_entries => internal_link_list_has_entries
```

⟨*Interactions: procedures*⟩+≡

```
function internal_link_list_has_entries (link_list) result (flag)
  class(internal_link_list_t), intent(in) :: link_list
  logical :: flag
  flag = link_list%length > 0
end function internal_link_list_has_entries
```

Return the list length

⟨*Interactions: internal link list: TBP*⟩+≡

```
procedure :: get_length => internal_link_list_get_length
```

⟨*Interactions: procedures*⟩+≡

```
function internal_link_list_get_length (link_list) result (length)
  class(internal_link_list_t), intent(in) :: link_list
  integer :: length
  length = link_list%length
end function internal_link_list_get_length
```

Return an entry.

⟨*Interactions: internal link list: TBP*⟩+≡

```
procedure :: get_link => internal_link_list_get_link
```

⟨*Interactions: procedures*⟩+≡

```
function internal_link_list_get_link (link_list, i) result (link)
  class(internal_link_list_t), intent(in) :: link_list
  integer, intent(in) :: i
  integer :: link
  if (i <= link_list%length) then
     link = link_list%link(i)
  else
     call msg_bug ("Internal link list: out of bounds")
  end if
end function internal_link_list_get_link
```

11.2.3 The interaction type

An interaction is an entangled system of particles. Thus, the interaction object consists of two parts: the subevent, and the quantum state which technically is a trie. The subnode levels beyond the trie root node are in correspondence to the subevent, so both should be traversed in parallel.

The subevent is implemented as an allocatable array of four-momenta. The first `n_in` particles are incoming, `n_vir` particles in-between can be kept for bookkeeping, and the last `n_out` particles are outgoing.

Distinct interactions are linked by their particles: for each particle, we have the possibility of links to corresponding particles in other interactions. Furthermore, for bookkeeping purposes we have a self-link array `relations` where the

parent-child relations are kept, and a flag array `resonant` which is set for an intermediate resonance.

Each momentum is associated with masks for flavor, color, and helicity. If a mask entry is set, the associated quantum number is to be ignored for that particle. If any mask has changed, the flag `update` is set.

We can have particle pairs locked together. If this is the case, the corresponding mask entries are bound to be equal. This is useful for particles that go through the interaction.

The interaction tag serves bookkeeping purposes. In particular, it identifies links in printout.

⟨*Interactions: public*⟩+≡
```
public :: interaction_t
```

⟨*Interactions: types*⟩+≡
```
type :: interaction_t
   private
   integer :: tag = 0
   type(state_matrix_t) :: state_matrix
   integer :: n_in = 0
   integer :: n_vir = 0
   integer :: n_out = 0
   integer :: n_tot = 0
   logical, dimension(:), allocatable :: p_is_known
   type(vector4_t), dimension(:), allocatable :: p
   type(external_link_t), dimension(:), allocatable :: source
   type(internal_link_list_t), dimension(:), allocatable :: parents
   type(internal_link_list_t), dimension(:), allocatable :: children
   logical, dimension(:), allocatable :: resonant
   type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
   integer, dimension(:), allocatable :: hel_lock
   logical :: update_state_matrix = .false.
   logical :: update_values = .false.
 contains
 ⟨Interactions: interaction: TBP⟩
end type interaction_t
```

Initialize the particle array with a fixed size. The first `n_in` particles are incoming, the rest outgoing. Masks are optional. There is also an optional tag. The interaction still needs fixing the values, but that is to be done after all branches have been added.

Interaction tags are assigned consecutively, using a `saved` variable local to this procedure. If desired, we can provide a seed for the interaction tags. Such a seed should be positive. The default seed is one. `tag=0` indicates an empty interaction.

If `set_relations` is set and true, we establish parent-child relations for all incoming and outgoing particles. Virtual particles are skipped; this option is normally used only for interations without virtual particles.

⟨*Interactions: interaction: TBP*⟩≡
```
procedure :: basic_init => interaction_init
```

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_init &
     (int, n_in, n_vir, n_out, &
```

```
        tag, resonant, mask, hel_lock, set_relations, store_values)
    class(interaction_t), intent(out) :: int
    integer, intent(in) :: n_in, n_vir, n_out
    integer, intent(in), optional :: tag
    logical, dimension(:), intent(in), optional :: resonant
    type(quantum_numbers_mask_t), dimension(:), intent(in), optional :: mask
    integer, dimension(:), intent(in), optional :: hel_lock
    logical, intent(in), optional :: set_relations, store_values
    logical :: set_rel
    integer :: i, j
    set_rel = .false.;  if (present (set_relations))  set_rel = set_relations
    call interaction_set_tag (int, tag)
    call int%state_matrix%init (store_values)
    int%n_in = n_in
    int%n_vir = n_vir
    int%n_out = n_out
    int%n_tot = n_in + n_vir + n_out
    allocate (int%p_is_known (int%n_tot))
    int%p_is_known = .false.
    allocate (int%p (int%n_tot))
    allocate (int%source (int%n_tot))
    allocate (int%parents (int%n_tot))
    allocate (int%children (int%n_tot))
    allocate (int%resonant (int%n_tot))
    if (present (resonant)) then
       int%resonant = resonant
    else
       int%resonant = .false.
    end if
    allocate (int%mask (int%n_tot))
    allocate (int%hel_lock (int%n_tot))
    if (present (mask)) then
       int%mask = mask
    end if
    if (present (hel_lock)) then
       int%hel_lock = hel_lock
    else
       int%hel_lock = 0
    end if
    int%update_state_matrix = .false.
    int%update_values = .true.
    if (set_rel) then
       do i = 1, n_in
          do j = 1, n_out
             call int%relate (i, n_in + j)
          end do
       end do
    end if
  end subroutine interaction_init
```

Set or create a unique tag for the interaction. Without interaction, reset the tag counter.

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_set_tag (int, tag)
  type(interaction_t), intent(inout), optional :: int
  integer, intent(in), optional :: tag
  integer, save :: stored_tag = 1
  if (present (int)) then
     if (present (tag)) then
        int%tag = tag
     else
        int%tag = stored_tag
        stored_tag = stored_tag + 1
     end if
  else if (present (tag)) then
     stored_tag = tag
  else
     stored_tag = 1
  end if
end subroutine interaction_set_tag
```

The public interface for the previous procedure only covers the reset functionality.

⟨*Interactions: public*⟩+≡

```
public :: reset_interaction_counter
```

⟨*Interactions: procedures*⟩+≡

```
subroutine reset_interaction_counter (tag)
  integer, intent(in), optional :: tag
  call interaction_set_tag (tag=tag)
end subroutine reset_interaction_counter
```

Finalizer: The state-matrix object contains pointers.

⟨*Interactions: interaction: TBP*⟩+≡

```
procedure :: final => interaction_final
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_final (object)
  class(interaction_t), intent(inout) :: object
  call object%state_matrix%final ()
end subroutine interaction_final
```

Output. The `verbose` option refers to the state matrix output.

⟨*Interactions: interaction: TBP*⟩+≡

```
procedure :: basic_write => interaction_write
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_write &
     (int, unit, verbose, show_momentum_sum, show_mass, show_state, &
     col_verbose, testflag)
  class(interaction_t), intent(in) :: int
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
  logical, intent(in), optional :: show_state, col_verbose, testflag
  integer :: u
  integer :: i, index_link
  type(interaction_t), pointer :: int_link
```

```
logical :: show_st
u = given_output_unit (unit);  if (u < 0)  return
show_st = .true.;  if (present (show_state))  show_st = show_state
if (int%tag /= 0) then
   write (u, "(1x,A,I0)")  "Interaction: ", int%tag
   do i = 1, int%n_tot
      if (i == 1 .and. int%n_in > 0) then
         write (u, "(1x,A)") "Incoming:"
      else if (i == int%n_in + 1 .and. int%n_vir > 0) then
         write (u, "(1x,A)") "Virtual:"
      else if (i == int%n_in + int%n_vir + 1 .and. int%n_out > 0) then
         write (u, "(1x,A)") "Outgoing:"
      end if
      write (u, "(1x,A,1x,I0)", advance="no") "Particle", i
      if (allocated (int%resonant)) then
         if (int%resonant(i)) then
            write (u, "(A)") "[r]"
         else
            write (u, *)
         end if
      else
         write (u, *)
      end if
      if (allocated (int%p)) then
         if (int%p_is_known(i)) then
            call vector4_write (int%p(i), u, show_mass, testflag)
         else
            write (u, "(A)")  "  [momentum undefined]"
         end if
      else
         write (u, "(A)") "  [momentum not allocated]"
      end if
      if (allocated (int%mask)) then
         write (u, "(1x,A)", advance="no")  "mask [fch] = "
         call int%mask(i)%write (u)
         write (u, *)
      end if
      if (int%parents(i)%has_entries () &
           .or. int%children(i)%has_entries ()) then
         write (u, "(1x,A)", advance="no") "internal links:"
         call int%parents(i)%write (u)
         if (int%parents(i)%has_entries ()) &
              write (u, "(1x,A)", advance="no") "=>"
         write (u, "(1x,A)", advance="no") "X"
         if (int%children(i)%has_entries ()) &
              write (u, "(1x,A)", advance="no") "=>"
         call int%children(i)%write (u)
         write (u, *)
      end if
      if (allocated (int%hel_lock)) then
         if (int%hel_lock(i) /= 0) then
            write (u, "(1x,A,1x,I0)")  "helicity lock:", int%hel_lock(i)
         end if
      end if
```

```
            if (external_link_is_set (int%source(i))) then
               write (u, "(1x,A)", advance="no") "source:"
               int_link => external_link_get_ptr (int%source(i))
               index_link = external_link_get_index (int%source(i))
               write (u, "(1x,'(',I0,')',I0)", advance="no") &
                    int_link%tag, index_link
               write (u, *)
            end if
         end do
         if (present (show_momentum_sum)) then
            if (allocated (int%p) .and. show_momentum_sum) then
               write (u, "(1x,A)") "Incoming particles (sum):"
               call vector4_write &
                    (sum (int%p(1 : int%n_in)), u, show_mass = show_mass)
               write (u, "(1x,A)") "Outgoing particles (sum):"
               call vector4_write &
                    (sum (int%p(int%n_in + int%n_vir + 1 : )), &
                     u, show_mass = show_mass)
               write (u, *)
            end if
         end if
         if (show_st) then
            call int%write_state_matrix (write_value_list = verbose, &
               verbose = verbose, unit = unit, col_verbose = col_verbose, &
               testflag = testflag)
         end if
      else
         write (u, "(1x,A)") "Interaction: [empty]"
      end if
   end subroutine interaction_write
```

⟨*Interactions: interaction: TBP*⟩+≡

```
   procedure :: write_state_matrix => interaction_write_state_matrix
```

⟨*Interactions: procedures*⟩+≡

```
   subroutine interaction_write_state_matrix (int, unit, write_value_list, &
      verbose, col_verbose, testflag)
      class(interaction_t), intent(in) :: int
      logical, intent(in), optional :: write_value_list, verbose, col_verbose
      logical, intent(in), optional :: testflag
      integer, intent(in), optional :: unit
      call int%state_matrix%write (write_value_list = verbose, &
         verbose = verbose, unit = unit, col_verbose = col_verbose, &
         testflag = testflag)
   end subroutine interaction_write_state_matrix
```

⟨*Interactions: interaction: TBP*⟩+≡

```
   procedure :: reduce_state_matrix => interaction_reduce_state_matrix
```

⟨*Interactions: procedures*⟩+≡

```
   subroutine interaction_reduce_state_matrix (int, qn_mask)
      class(interaction_t), intent(inout) :: int
      type(quantum_numbers_mask_t), intent(in), dimension(:) :: qn_mask
      type(state_matrix_t) :: state
```

```
    call int%state_matrix%reduce (qn_mask, state)
    int%state_matrix = state
  end subroutine interaction_reduce_state_matrix
```

Assignment: We implement this as a deep copy. This applies, in particular, to the state-matrix and internal-link components. Furthermore, the new interaction acquires a new tag.

⟨*Interactions: public*⟩+≡

```
  public :: assignment(=)
```

⟨*Interactions: interfaces*⟩≡

```
  interface assignment(=)
     module procedure interaction_assign
  end interface
```

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_assign (int_out, int_in)
    type(interaction_t), intent(out) :: int_out
    type(interaction_t), intent(in), target :: int_in
    call interaction_set_tag (int_out)
    int_out%state_matrix = int_in%state_matrix
    int_out%n_in  = int_in%n_in
    int_out%n_out = int_in%n_out
    int_out%n_vir = int_in%n_vir
    int_out%n_tot = int_in%n_tot
    if (allocated (int_in%p_is_known)) then
       allocate (int_out%p_is_known (size (int_in%p_is_known)))
       int_out%p_is_known = int_in%p_is_known
    end if
    if (allocated (int_in%p)) then
       allocate (int_out%p (size (int_in%p)))
       int_out%p = int_in%p
    end if
    if (allocated (int_in%source)) then
       allocate (int_out%source (size (int_in%source)))
       int_out%source = int_in%source
    end if
    if (allocated (int_in%parents)) then
       allocate (int_out%parents (size (int_in%parents)))
       int_out%parents = int_in%parents
    end if
    if (allocated (int_in%children)) then
       allocate (int_out%children (size (int_in%children)))
       int_out%children = int_in%children
    end if
    if (allocated (int_in%resonant)) then
       allocate (int_out%resonant (size (int_in%resonant)))
       int_out%resonant = int_in%resonant
    end if
    if (allocated (int_in%mask)) then
       allocate (int_out%mask (size (int_in%mask)))
       int_out%mask = int_in%mask
    end if
```

```
    if (allocated (int_in%hel_lock)) then
       allocate (int_out%hel_lock (size (int_in%hel_lock)))
       int_out%hel_lock = int_in%hel_lock
    end if
    int_out%update_state_matrix = int_in%update_state_matrix
    int_out%update_values = int_in%update_values
  end subroutine interaction_assign
```

11.2.4 Methods inherited from the state matrix member

Until F2003 is standard, we cannot implement inheritance directly. Therefore, we need wrappers for "inherited" methods.

Make a new branch in the state matrix if it does not yet exist. This is not just a wrapper but it introduces the interaction mask: where a quantum number is masked, it is not transferred but set undefined. After this, the value array has to be updated.

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: add_state => interaction_add_state
```

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_add_state &
       (int, qn, index, value, sum_values, counter_index, me_index)
    class(interaction_t), intent(inout) :: int
    type(quantum_numbers_t), dimension(:), intent(in) :: qn
    integer, intent(in), optional :: index
    complex(default), intent(in), optional :: value
    logical, intent(in), optional :: sum_values
    integer, intent(in), optional :: counter_index
    integer, intent(out), optional :: me_index
    type(quantum_numbers_t), dimension(size(qn)) :: qn_tmp
    qn_tmp = qn
    call qn_tmp%undefine (int%mask)
    call int%state_matrix%add_state (qn_tmp, index, value, sum_values, &
         counter_index, me_index)
    int%update_values = .true.
  end subroutine interaction_add_state
```

Freeze the quantum state: First collapse the quantum state, i.e., remove quantum numbers if any mask has changed, then fix the array of value pointers.

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: freeze => interaction_freeze
```

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_freeze (int)
    class(interaction_t), intent(inout) :: int
    if (int%update_state_matrix) then
       call int%state_matrix%collapse (int%mask)
       int%update_state_matrix = .false.
       int%update_values = .true.
    end if
    if (int%update_values) then
       call int%state_matrix%freeze ()
```

```
      int%update_values = .false.
    end if
  end subroutine interaction_freeze
```

Return true if the state matrix is empty.

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: is_empty => interaction_is_empty
```

⟨*Interactions: procedures*⟩+≡

```
  pure function interaction_is_empty (int) result (flag)
    logical :: flag
    class(interaction_t), intent(in) :: int
    flag = int%state_matrix%is_empty ()
  end function interaction_is_empty
```

Get the number of values stored in the state matrix:

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: get_n_matrix_elements => &
       interaction_get_n_matrix_elements
```

⟨*Interactions: procedures*⟩+≡

```
  pure function interaction_get_n_matrix_elements (int) result (n)
    integer :: n
    class(interaction_t), intent(in) :: int
    n = int%state_matrix%get_n_matrix_elements ()
  end function interaction_get_n_matrix_elements
```

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: get_n_in_helicities => interaction_get_n_in_helicities
```

⟨*Interactions: procedures*⟩+≡

```
  function interaction_get_n_in_helicities (int) result (n_hel)
    integer :: n_hel
    class(interaction_t), intent(in) :: int
    type(interaction_t) :: int_copy
    type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
    type(quantum_numbers_t), dimension(:,:), allocatable :: qn
    integer :: i
    allocate (qn_mask (int%n_tot))
    do i = 1, int%n_tot
       if (i <= int%n_in) then
          call qn_mask(i)%init (.true., .true., .false.)
       else
          call qn_mask(i)%init (.true., .true., .true.)
       end if
    end do
    int_copy = int
    call int_copy%set_mask (qn_mask)
    call int_copy%freeze ()
    allocate (qn (int_copy%state_matrix%get_n_matrix_elements (), &
         int_copy%state_matrix%get_depth ()))
    qn = int_copy%get_quantum_numbers ()
    n_hel = 0
    do i = 1, size (qn, dim=1)
```

```
      if (all (qn(i,:)%get_subtraction_index () == 0)) n_hel = n_hel + 1
    end do
    call int_copy%final ()
    deallocate (qn_mask)
    deallocate (qn)
  end function interaction_get_n_in_helicities
```

Get the size of the me-array of the associated state matrix for debugging purposes

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: get_me_size => interaction_get_me_size
```

⟨*Interactions: procedures*⟩+≡

```
  pure function interaction_get_me_size (int) result (n)
    integer :: n
    class(interaction_t), intent(in) :: int
    n = int%state_matrix%get_me_size ()
  end function interaction_get_me_size
```

Get the norm of the state matrix (if the norm has been taken out, otherwise this would be unity).

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: get_norm => interaction_get_norm
```

⟨*Interactions: procedures*⟩+≡

```
  pure function interaction_get_norm (int) result (norm)
    real(default) :: norm
    class(interaction_t), intent(in) :: int
    norm = int%state_matrix%get_norm ()
  end function interaction_get_norm
```

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: get_n_sub => interaction_get_n_sub
```

⟨*Interactions: procedures*⟩+≡

```
  pure function interaction_get_n_sub (int) result (n_sub)
    integer :: n_sub
    class(interaction_t), intent(in) :: int
    n_sub = int%state_matrix%get_n_sub ()
  end function interaction_get_n_sub
```

Get the quantum number array that corresponds to a given index.

⟨*Interactions: interaction: TBP*⟩+≡

```
  generic :: get_quantum_numbers => get_quantum_numbers_single, &
                                    get_quantum_numbers_all, &
                                    get_quantum_numbers_all_qn_mask
  procedure :: get_quantum_numbers_single => &
     interaction_get_quantum_numbers_single
  procedure :: get_quantum_numbers_all => &
     interaction_get_quantum_numbers_all
  procedure :: get_quantum_numbers_all_qn_mask => &
     interaction_get_quantum_numbers_all_qn_mask
```

⟨*Interactions: procedures*⟩+≡

```
function interaction_get_quantum_numbers_single (int, i) result (qn)
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  class(interaction_t), intent(in), target :: int
  integer, intent(in) :: i
  allocate (qn (int%state_matrix%get_depth ()))
  qn = int%state_matrix%get_quantum_number (i)
end function interaction_get_quantum_numbers_single

function interaction_get_quantum_numbers_all (int) result (qn)
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn
  class(interaction_t), intent(in), target :: int
  integer :: i
```

⟨*Interactions: get quantum numbers all*⟩

⟨*Interactions: get quantum numbers all*⟩≡

```
  allocate (qn (int%state_matrix%get_n_matrix_elements (), &
     int%state_matrix%get_depth()))
  do i = 1, int%state_matrix%get_n_matrix_elements ()
     qn (i, :) = int%state_matrix%get_quantum_number (i)
  end do
```

⟨*Interactions: procedures*⟩+≡

```
end function interaction_get_quantum_numbers_all

function interaction_get_quantum_numbers_all_qn_mask (int, qn_mask) &
   result (qn)
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn
  class(interaction_t), intent(in) :: int
  type(quantum_numbers_mask_t), intent(in) :: qn_mask
  integer :: n_redundant, n_all, n_me
  integer :: i
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn_all
```

⟨*Interactions: get quantum numbers all qn mask*⟩

⟨*Interactions: get quantum numbers all qn mask*⟩≡

```
  call int%state_matrix%get_quantum_numbers (qn_all)
  n_redundant = count (qn_all%are_redundant (qn_mask))
  n_all = size (qn_all)
  !!! Number of matrix elements = survivors / n_particles
  n_me = (n_all - n_redundant) / int%state_matrix%get_depth ()
  allocate (qn (n_me, int%state_matrix%get_depth()))
  do i = 1, n_me
     if (.not. any (qn_all(i, :)%are_redundant (qn_mask))) &
        qn (i, :) = qn_all (i, :)
  end do
```

⟨*Interactions: procedures*⟩+≡

```
end function interaction_get_quantum_numbers_all_qn_mask
```

⟨*Interactions: interaction: TBP*⟩+≡

```
procedure :: get_quantum_numbers_all_sub => interaction_get_quantum_numbers_all_sub
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_get_quantum_numbers_all_sub (int, qn)
  class(interaction_t), intent(in) :: int
```

```
    type(quantum_numbers_t), dimension(:,:), allocatable, intent(out) :: qn
    integer :: i
  ⟨Interactions: get quantum numbers all⟩
  end subroutine interaction_get_quantum_numbers_all_sub
```

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: get_quantum_numbers_mask => interaction_get_quantum_numbers_mask
```

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_get_quantum_numbers_mask (int, qn_mask, qn)
    class(interaction_t), intent(in) :: int
    type(quantum_numbers_mask_t), intent(in) :: qn_mask
    type(quantum_numbers_t), dimension(:,:), allocatable, intent(out) :: qn
    integer :: n_redundant, n_all, n_me
    integer :: i
    type(quantum_numbers_t), dimension(:,:), allocatable :: qn_all
  ⟨Interactions: get quantum numbers all qn mask⟩
  end subroutine interaction_get_quantum_numbers_mask
```

Get the matrix element that corresponds to a set of quantum numbers, a given index, or return the whole array.

⟨*Interactions: interaction: TBP*⟩+≡

```
  generic :: get_matrix_element => get_matrix_element_single
  generic :: get_matrix_element => get_matrix_element_array
  procedure :: get_matrix_element_single => &
     interaction_get_matrix_element_single
  procedure :: get_matrix_element_array => &
     interaction_get_matrix_element_array
```

⟨*Interactions: procedures*⟩+≡

```
  elemental function interaction_get_matrix_element_single (int, i) result (me)
    complex(default) :: me
    class(interaction_t), intent(in) :: int
    integer, intent(in) :: i
    me = int%state_matrix%get_matrix_element (i)
  end function interaction_get_matrix_element_single
```

⟨*Interactions: procedures*⟩+≡

```
  function interaction_get_matrix_element_array (int) result (me)
    complex(default), dimension(:), allocatable :: me
    class(interaction_t), intent(in) :: int
    allocate (me (int%get_n_matrix_elements ()))
    me = int%state_matrix%get_matrix_element ()
  end function interaction_get_matrix_element_array
```

Set the complex value(s) stored in the quantum state.

⟨*Interactions: interaction: TBP*⟩+≡

```
  generic :: set_matrix_element => interaction_set_matrix_element_qn, &
       interaction_set_matrix_element_all, &
       interaction_set_matrix_element_array, &
       interaction_set_matrix_element_single, &
       interaction_set_matrix_element_clone
  procedure :: interaction_set_matrix_element_qn
```

```
procedure :: interaction_set_matrix_element_all
procedure :: interaction_set_matrix_element_array
procedure :: interaction_set_matrix_element_single
procedure :: interaction_set_matrix_element_clone
```

Indirect access via the quantum number array:

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_set_matrix_element_qn (int, qn, val)
  class(interaction_t), intent(inout) :: int
  type(quantum_numbers_t), dimension(:), intent(in) :: qn
  complex(default), intent(in) :: val
  call int%state_matrix%set_matrix_element (qn, val)
end subroutine interaction_set_matrix_element_qn
```

Set all entries of the matrix-element array to a given value.

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_set_matrix_element_all (int, value)
  class(interaction_t), intent(inout) :: int
  complex(default), intent(in) :: value
  call int%state_matrix%set_matrix_element (value)
end subroutine interaction_set_matrix_element_all
```

Set the matrix-element array directly.

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_set_matrix_element_array (int, value)
  class(interaction_t), intent(inout) :: int
  complex(default), dimension(:), intent(in) :: value
  call int%state_matrix%set_matrix_element (value)
end subroutine interaction_set_matrix_element_array

pure subroutine interaction_set_matrix_element_single (int, i, value)
  class(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  complex(default), intent(in) :: value
  call int%state_matrix%set_matrix_element (i, value)
end subroutine interaction_set_matrix_element_single
```

Clone from another (matching) interaction.

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_set_matrix_element_clone (int, int1)
  class(interaction_t), intent(inout) :: int
  class(interaction_t), intent(in) :: int1
  call int%state_matrix%set_matrix_element (int1%state_matrix)
end subroutine interaction_set_matrix_element_clone
```

⟨*Interactions: interaction: TBP*⟩+≡

```
procedure :: set_only_matrix_element => interaction_set_only_matrix_element
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_set_only_matrix_element (int, i, value)
  class(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  complex(default), intent(in) :: value
```

```
    call int%set_matrix_element (cmplx (0, 0, default))
    call int%set_matrix_element (i, value)
  end subroutine interaction_set_only_matrix_element
```

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: add_to_matrix_element => interaction_add_to_matrix_element
```

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_add_to_matrix_element (int, qn, value, match_only_flavor)
    class(interaction_t), intent(inout) :: int
    type(quantum_numbers_t), dimension(:), intent(in) :: qn
    complex(default), intent(in) :: value
    logical, intent(in), optional :: match_only_flavor
    call int%state_matrix%add_to_matrix_element (qn, value, match_only_flavor)
  end subroutine interaction_add_to_matrix_element
```

Get the indices of any diagonal matrix elements.

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: get_diagonal_entries => interaction_get_diagonal_entries
```

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_get_diagonal_entries (int, i)
    class(interaction_t), intent(in) :: int
    integer, dimension(:), allocatable, intent(out) :: i
    call int%state_matrix%get_diagonal_entries (i)
  end subroutine interaction_get_diagonal_entries
```

Renormalize the state matrix by its trace, if nonzero. The renormalization is reflected in the state-matrix norm.

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: normalize_by_trace => interaction_normalize_by_trace
```

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_normalize_by_trace (int)
    class(interaction_t), intent(inout) :: int
    call int%state_matrix%normalize_by_trace ()
  end subroutine interaction_normalize_by_trace
```

Analogous, but renormalize by maximal (absolute) value.

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: normalize_by_max => interaction_normalize_by_max
```

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_normalize_by_max (int)
    class(interaction_t), intent(inout) :: int
    call int%state_matrix%normalize_by_max ()
  end subroutine interaction_normalize_by_max
```

Explicitly set the norm value (of the state matrix).

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: set_norm => interaction_set_norm
```

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_set_norm (int, norm)
  class(interaction_t), intent(inout) :: int
  real(default), intent(in) :: norm
  call int%state_matrix%set_norm (norm)
end subroutine interaction_set_norm
```

⟨*Interactions: interaction: TBP*⟩+≡
```
procedure :: set_state_matrix => interaction_set_state_matrix
```

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_set_state_matrix (int, state)
  class(interaction_t), intent(inout) :: int
  type(state_matrix_t), intent(in) :: state
  int%state_matrix = state
end subroutine interaction_set_state_matrix
```

Return the maximum absolute value of color indices.

⟨*Interactions: interaction: TBP*⟩+≡
```
procedure :: get_max_color_value => &
     interaction_get_max_color_value
```

⟨*Interactions: procedures*⟩+≡
```
function interaction_get_max_color_value (int) result (cmax)
  class(interaction_t), intent(in) :: int
  integer :: cmax
  cmax = int%state_matrix%get_max_color_value ()
end function interaction_get_max_color_value
```

Factorize the state matrix into single-particle state matrices, the branch selection depending on a (random) value between 0 and 1; optionally also return a correlated state matrix.

⟨*Interactions: interaction: TBP*⟩+≡
```
procedure :: factorize => interaction_factorize
```

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_factorize &
     (int, mode, x, ok, single_state, correlated_state, qn_in)
  class(interaction_t), intent(in), target :: int
  integer, intent(in) :: mode
  real(default), intent(in) :: x
  logical, intent(out) :: ok
  type(state_matrix_t), &
       dimension(:), allocatable, intent(out) :: single_state
  type(state_matrix_t), intent(out), optional :: correlated_state
  type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_in
  call int%state_matrix%factorize &
       (mode, x, ok, single_state, correlated_state, qn_in)
end subroutine interaction_factorize
```

Sum all matrix element values

⟨*Interactions: interaction: TBP*⟩+≡
```
procedure :: sum => interaction_sum
```

⟨*Interactions: procedures*⟩+≡
```
function interaction_sum (int) result (value)
  class(interaction_t), intent(in) :: int
  complex(default) :: value
  value = int%state_matrix%sum ()
end function interaction_sum
```

Append new states which are color-contracted versions of the existing states. The matrix element index of each color contraction coincides with the index of its origin, so no new matrix elements are generated. After this operation, no `freeze` must be performed anymore.

⟨*Interactions: interaction: TBP*⟩+≡
```
procedure :: add_color_contractions => &
     interaction_add_color_contractions
```

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_add_color_contractions (int)
  class(interaction_t), intent(inout) :: int
  call int%state_matrix%add_color_contractions ()
end subroutine interaction_add_color_contractions
```

Multiply matrix elements from two interactions. Choose the elements as given by the integer index arrays, multiply them and store the sum of products in the indicated matrix element. The suffixes mean: c=conjugate first factor; f=include weighting factor.

⟨*Interactions: interaction: TBP*⟩+≡
```
procedure :: evaluate_product => interaction_evaluate_product
procedure :: evaluate_product_cf => interaction_evaluate_product_cf
procedure :: evaluate_square_c => interaction_evaluate_square_c
procedure :: evaluate_sum => interaction_evaluate_sum
procedure :: evaluate_me_sum => interaction_evaluate_me_sum
```

⟨*Interactions: procedures*⟩+≡
```
pure subroutine interaction_evaluate_product &
     (int, i, int1, int2, index1, index2)
  class(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  type(interaction_t), intent(in) :: int1, int2
  integer, dimension(:), intent(in) :: index1, index2
  call int%state_matrix%evaluate_product &
       (i, int1%state_matrix, int2%state_matrix, &
        index1, index2)
end subroutine interaction_evaluate_product

pure subroutine interaction_evaluate_product_cf &
     (int, i, int1, int2, index1, index2, factor)
  class(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  type(interaction_t), intent(in) :: int1, int2
  integer, dimension(:), intent(in) :: index1, index2
  complex(default), dimension(:), intent(in) :: factor
  call int%state_matrix%evaluate_product_cf &
       (i, int1%state_matrix, int2%state_matrix, &
```

```
        index1, index2, factor)
end subroutine interaction_evaluate_product_cf

pure subroutine interaction_evaluate_square_c (int, i, int1, index1)
  class(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  type(interaction_t), intent(in) :: int1
  integer, dimension(:), intent(in) :: index1
  call int%state_matrix%evaluate_square_c (i, int1%state_matrix, index1)
end subroutine interaction_evaluate_square_c

pure subroutine interaction_evaluate_sum (int, i, int1, index1)
  class(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  type(interaction_t), intent(in) :: int1
  integer, dimension(:), intent(in) :: index1
  call int%state_matrix%evaluate_sum (i, int1%state_matrix, index1)
end subroutine interaction_evaluate_sum

pure subroutine interaction_evaluate_me_sum (int, i, int1, index1)
  class(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  type(interaction_t), intent(in) :: int1
  integer, dimension(:), intent(in) :: index1
  call int%state_matrix%evaluate_me_sum (i, int1%state_matrix, index1)
end subroutine interaction_evaluate_me_sum
```

11.2.5 Accessing contents

Return the integer tag.

⟨*Interactions: interaction: TBP*⟩+≡

```
procedure :: get_tag => interaction_get_tag
```

⟨*Interactions: procedures*⟩+≡

```
function interaction_get_tag (int) result (tag)
  class(interaction_t), intent(in) :: int
  integer :: tag
  tag = int%tag
end function interaction_get_tag
```

Return the number of particles.

⟨*Interactions: interaction: TBP*⟩+≡

```
procedure :: get_n_tot => interaction_get_n_tot
procedure :: get_n_in => interaction_get_n_in
procedure :: get_n_vir => interaction_get_n_vir
procedure :: get_n_out => interaction_get_n_out
```

⟨*Interactions: procedures*⟩+≡

```
pure function interaction_get_n_tot (object) result (n_tot)
  class(interaction_t), intent(in) :: object
  integer :: n_tot
  n_tot = object%n_tot
end function interaction_get_n_tot
```

```
pure function interaction_get_n_in (object) result (n_in)
  class(interaction_t), intent(in) :: object
  integer :: n_in
  n_in = object%n_in
end function interaction_get_n_in

pure function interaction_get_n_vir (object) result (n_vir)
  class(interaction_t), intent(in) :: object
  integer :: n_vir
  n_vir = object%n_vir
end function interaction_get_n_vir

pure function interaction_get_n_out (object) result (n_out)
  class(interaction_t), intent(in) :: object
  integer :: n_out
  n_out = object%n_out
end function interaction_get_n_out
```

Return a momentum index. The flags specify whether to keep/drop incoming, virtual, or outgoing momenta. Check for illegal values.

⟨*Interactions: procedures*⟩+≡

```
function idx (int, i, outgoing)
  integer :: idx
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  logical, intent(in), optional :: outgoing
  logical :: in, vir, out
  if (present (outgoing)) then
     in  = .not. outgoing
     vir = .false.
     out = outgoing
  else
     in = .true.
     vir = .true.
     out = .true.
  end if
  idx = 0
  if (in) then
     if (vir) then
        if (out) then
           if (i <= int%n_tot)  idx = i
        else
           if (i <= int%n_in + int%n_vir)  idx = i
        end if
     else if (out) then
        if (i <= int%n_in) then
           idx = i
        else if (i <= int%n_in + int%n_out) then
           idx = int%n_vir + i
        end if
     else
        if (i <= int%n_in)  idx = i
```

```
      end if
    else if (vir) then
      if (out) then
        if (i <= int%n_vir + int%n_out)  idx = int%n_in + i
      else
        if (i <= int%n_vir)  idx = int%n_in + i
      end if
    else if (out) then
      if (i <= int%n_out)  idx = int%n_in + int%n_vir + i
    end if
    if (idx == 0) then
      call int%basic_write ()
      print *, i, in, vir, out
      call msg_bug (" Momentum index is out of range for this interaction")
    end if
  end function idx
```

Return all or just a specific four-momentum.

⟨*Interactions: interaction: TBP*⟩+≡

```
  generic :: get_momenta => get_momenta_all, get_momenta_idx
  procedure :: get_momentum => interaction_get_momentum
  procedure :: get_momenta_all => interaction_get_momenta_all
  procedure :: get_momenta_idx => interaction_get_momenta_idx
```

⟨*Interactions: procedures*⟩+≡

```
  function interaction_get_momenta_all (int, outgoing) result (p)
    class(interaction_t), intent(in) :: int
    type(vector4_t), dimension(:), allocatable :: p
    logical, intent(in), optional :: outgoing
    integer :: i
    if (present (outgoing)) then
      if (outgoing) then
        allocate (p (int%n_out))
      else
        allocate (p (int%n_in))
      end if
    else
      allocate (p (int%n_tot))
    end if
    do i = 1, size (p)
      p(i) = int%p(idx (int, i, outgoing))
    end do
  end function interaction_get_momenta_all

  function interaction_get_momenta_idx (int, jj) result (p)
    class(interaction_t), intent(in) :: int
    type(vector4_t), dimension(:), allocatable :: p
    integer, dimension(:), intent(in) :: jj
    allocate (p (size (jj)))
    p = int%p(jj)
  end function interaction_get_momenta_idx

  function interaction_get_momentum (int, i, outgoing) result (p)
    class(interaction_t), intent(in) :: int
```

```
    type(vector4_t) :: p
    integer, intent(in) :: i
    logical, intent(in), optional :: outgoing
    p = int%p(idx (int, i, outgoing))
  end function interaction_get_momentum
```

This is a variant as a subroutine. Redundant, but the function above fails at times for gfortran 4.5.0 (double allocation, compiler bug).

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: get_momenta_sub => interaction_get_momenta_sub
```

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_get_momenta_sub (int, p, outgoing)
    class(interaction_t), intent(in) :: int
    type(vector4_t), dimension(:), intent(out) :: p
    logical, intent(in), optional :: outgoing
    integer :: i
    do i = 1, size (p)
       p(i) = int%p(idx (int, i, outgoing))
    end do
  end subroutine interaction_get_momenta_sub
```

Return a shallow copy of the state matrix:

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: get_state_matrix_ptr => &
       interaction_get_state_matrix_ptr
```

⟨*Interactions: procedures*⟩+≡

```
  function interaction_get_state_matrix_ptr (int) result (state)
    class(interaction_t), intent(in), target :: int
    type(state_matrix_t), pointer :: state
    state => int%state_matrix
  end function interaction_get_state_matrix_ptr
```

Return the array of resonance flags

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: get_resonance_flags => interaction_get_resonance_flags
```

⟨*Interactions: procedures*⟩+≡

```
  function interaction_get_resonance_flags (int) result (resonant)
    class(interaction_t), intent(in) :: int
    logical, dimension(size(int%resonant)) :: resonant
    resonant = int%resonant
  end function interaction_get_resonance_flags
```

Return the quantum-numbers mask (or part of it)

⟨*Interactions: interaction: TBP*⟩+≡

```
  generic :: get_mask => get_mask_all, get_mask_slice
  procedure :: get_mask_all => interaction_get_mask_all
  procedure :: get_mask_slice => interaction_get_mask_slice
```

⟨*Interactions: procedures*⟩+≡

```
function interaction_get_mask_all (int) result (mask)
  class(interaction_t), intent(in) :: int
  type(quantum_numbers_mask_t), dimension(size(int%mask)) :: mask
  mask = int%mask
end function interaction_get_mask_all

function interaction_get_mask_slice (int, index) result (mask)
  class(interaction_t), intent(in) :: int
  integer, dimension(:), intent(in) :: index
  type(quantum_numbers_mask_t), dimension(size(index)) :: mask
  mask = int%mask(index)
end function interaction_get_mask_slice
```

Compute the invariant mass squared of the incoming particles (if any, otherwise outgoing).

⟨*Interactions: public*⟩+≡

```
public :: interaction_get_s
```

⟨*Interactions: procedures*⟩+≡

```
function interaction_get_s (int) result (s)
  real(default) :: s
  type(interaction_t), intent(in) :: int
  if (int%n_in /= 0) then
     s = sum (int%p(:int%n_in)) ** 2
  else
     s = sum (int%p(int%n_vir + 1 : )) ** 2
  end if
end function interaction_get_s
```

Compute the Lorentz transformation that transforms the incoming particles from the center-of-mass frame to the lab frame where they are given. If the c.m. mass squared is negative or zero, return the identity.

⟨*Interactions: public*⟩+≡

```
public :: interaction_get_cm_transformation
```

⟨*Interactions: procedures*⟩+≡

```
function interaction_get_cm_transformation (int) result (lt)
  type(lorentz_transformation_t) :: lt
  type(interaction_t), intent(in) :: int
  type(vector4_t) :: p_cm
  real(default) :: s
  if (int%n_in /= 0) then
     p_cm = sum (int%p(:int%n_in))
  else
     p_cm = sum (int%p(int%n_vir+1:))
  end if
  s = p_cm ** 2
  if (s > 0) then
     lt = boost (p_cm, sqrt (s))
  else
     lt = identity
  end if
end function interaction_get_cm_transformation
```

Return flavor, momentum, and position of the first outgoing unstable particle present in the interaction. Note that we need not iterate through the state matrix; if there is an unstable particle, it will be present in all state-matrix entries.

⟨*Interactions: public*⟩+≡
```
public :: interaction_get_unstable_particle
```

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_get_unstable_particle (int, flv, p, i)
  type(interaction_t), intent(in), target :: int
  type(flavor_t), intent(out) :: flv
  type(vector4_t), intent(out) :: p
  integer, intent(out) :: i
  type(state_iterator_t) :: it
  type(flavor_t), dimension(int%n_tot) :: flv_array
  call it%init (int%state_matrix)
  flv_array = it%get_flavor ()
  do i = int%n_in + int%n_vir + 1, int%n_tot
     if (.not. flv_array(i)%is_stable ()) then
        flv = flv_array(i)
        p = int%p(i)
        return
     end if
  end do
end subroutine interaction_get_unstable_particle
```

Return the complete set of *outgoing* flavors, assuming that the flavor quantum number is not suppressed.

⟨*Interactions: public*⟩+≡
```
public :: interaction_get_flv_out
```

⟨*Interactions: procedures*⟩+≡
```
subroutine interaction_get_flv_out (int, flv)
  type(interaction_t), intent(in), target :: int
  type(flavor_t), dimension(:,:), allocatable, intent(out) :: flv
  type(state_iterator_t) :: it
  type(flavor_t), dimension(:), allocatable :: flv_state
  integer :: n_in, n_vir, n_out, n_tot, n_state, i
  n_in = int%get_n_in ()
  n_vir = int%get_n_vir ()
  n_out = int%get_n_out ()
  n_tot = int%get_n_tot ()
  n_state = int%get_n_matrix_elements ()
  allocate (flv (n_out, n_state))
  allocate (flv_state (n_tot))
  i = 1
  call it%init (int%get_state_matrix_ptr ())
  do while (it%is_valid ())
     flv_state = it%get_flavor ()
     flv(:,i) = flv_state(n_in + n_vir + 1 : )
     i = i + 1
     call it%advance ()
```

```
    end do
  end subroutine interaction_get_flv_out
```

Determine the flavor content of the interaction. We analyze the state matrix for this, and we select the outgoing particles of the hard process only for the required mask, which indicates the particles that can appear in any order in a matching event record.

We have to assume that any radiated particles (beam remnants) appear at the beginning of the particles marked as outgoing.

⟨*Interactions: public*⟩+≡

```
  public :: interaction_get_flv_content
```

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_get_flv_content (int, state_flv, n_out_hard)
    type(interaction_t), intent(in), target :: int
    type(state_flv_content_t), intent(out) :: state_flv
    integer, intent(in) :: n_out_hard
    logical, dimension(:), allocatable :: mask
    integer :: n_tot
    n_tot = int%get_n_tot ()
    allocate (mask (n_tot), source = .false.)
    mask(n_tot-n_out_hard + 1 : ) = .true.
    call state_flv%fill (int%get_state_matrix_ptr (), mask)
  end subroutine interaction_get_flv_content
```

11.2.6 Modifying contents

Set the quantum numbers mask.

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: set_mask => interaction_set_mask
```

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_set_mask (int, mask)
    class(interaction_t), intent(inout) :: int
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: mask
    if (size (int%mask) /= size (mask)) &
       call msg_fatal ("Attempting to set mask with unfitting size!")
    int%mask = mask
    int%update_state_matrix = .true.
  end subroutine interaction_set_mask
```

Merge a particular mask entry, respecting a possible helicity lock for this entry. We apply an OR relation, which means that quantum numbers are summed over if either of the two masks requires it.

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_merge_mask_entry (int, i, mask)
    type(interaction_t), intent(inout) :: int
    integer, intent(in) :: i
    type(quantum_numbers_mask_t), intent(in) :: mask
    type(quantum_numbers_mask_t) :: mask_tmp
    integer :: ii
```

```
    ii = idx (int, i)
    if (int%mask(ii) .neqv. mask) then
       int%mask(ii) = int%mask(ii) .or. mask
       if (int%hel_lock(ii) /= 0) then
          call mask_tmp%assign (mask, helicity=.true.)
          int%mask(int%hel_lock(ii)) = int%mask(int%hel_lock(ii)) .or. mask_tmp
       end if
    end if
    int%update_state_matrix = .true.
  end subroutine interaction_merge_mask_entry
```

Fill the momenta array, do not care about the quantum numbers of particles.

⟨*Interactions: interaction: TBP*⟩+≡

```
  procedure :: reset_momenta => interaction_reset_momenta
  procedure :: set_momenta => interaction_set_momenta
  procedure :: set_momentum => interaction_set_momentum
```

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_reset_momenta (int)
    class(interaction_t), intent(inout) :: int
    int%p = vector4_null
    int%p_is_known = .true.
  end subroutine interaction_reset_momenta

  subroutine interaction_set_momenta (int, p, outgoing)
    class(interaction_t), intent(inout) :: int
    type(vector4_t), dimension(:), intent(in) :: p
    logical, intent(in), optional :: outgoing
    integer :: i, index
    do i = 1, size (p)
       index = idx (int, i, outgoing)
       int%p(index) = p(i)
       int%p_is_known(index) = .true.
    end do
  end subroutine interaction_set_momenta

  subroutine interaction_set_momentum (int, p, i, outgoing)
    class(interaction_t), intent(inout) :: int
    type(vector4_t), intent(in) :: p
    integer, intent(in) :: i
    logical, intent(in), optional :: outgoing
    integer :: index
    index = idx (int, i, outgoing)
    int%p(index) = p
    int%p_is_known(index) = .true.
  end subroutine interaction_set_momentum
```

This more sophisticated version of setting values is used for structure functions, in particular if nontrivial flavor, color, and helicity may be present: set values selectively for the given flavors. If there is more than one flavor, scan the interaction and check for a matching flavor at the specified particle location. If it matches, insert the value that corresponds to this flavor.

⟨*Interactions: public*⟩+≡

```
public :: interaction_set_flavored_values
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_set_flavored_values (int, value, flv_in, pos)
  type(interaction_t), intent(inout) :: int
  complex(default), dimension(:), intent(in) :: value
  type(flavor_t), dimension(:), intent(in) :: flv_in
  integer, intent(in) :: pos
  type(state_iterator_t) :: it
  type(flavor_t) :: flv
  integer :: i
  if (size (value) == 1) then
     call int%set_matrix_element (value(1))
  else
     call it%init (int%state_matrix)
     do while (it%is_valid ())
        flv = it%get_flavor (pos)
        SCAN_FLV: do i = 1, size (value)
           if (flv == flv_in(i)) then
              call it%set_matrix_element (value(i))
              exit SCAN_FLV
           end if
        end do SCAN_FLV
        call it%advance ()
     end do
  end if
end subroutine interaction_set_flavored_values
```

11.2.7 Handling Linked interactions

Store relations between corresponding particles within one interaction. The first particle is the parent, the second one the child. Links are established in both directions.

These relations have no effect on the propagation of momenta etc., they are rather used for mother-daughter relations in event output.

⟨*Interactions: interaction: TBP*⟩+≡

```
procedure :: relate => interaction_relate
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_relate (int, i1, i2)
  class(interaction_t), intent(inout), target :: int
  integer, intent(in) :: i1, i2
  if (i1 /= 0 .and. i2 /= 0) then
     call int%children(i1)%append (i2)
     call int%parents(i2)%append (i1)
  end if
end subroutine interaction_relate
```

Transfer internal parent-child relations defined within interaction `int1` to a new interaction `int` where the particle indices are mapped to. Some particles in `int1` may have no image in `int`. In that case, a child entry maps to zero, and we skip this relation.

Also transfer resonance flags.

⟨*Interactions: interaction: TBP*⟩+≡

```
procedure :: transfer_relations => interaction_transfer_relations
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_transfer_relations (int1, int2, map)
  class(interaction_t), intent(in) :: int1
  class(interaction_t), intent(inout), target :: int2
  integer, dimension(:), intent(in) :: map
  integer :: i, j, k
  do i = 1, size (map)
     do j = 1, int1%parents(i)%get_length ()
        k = int1%parents(i)%get_link (j)
        call int2%relate (map(k), map(i))
     end do
     if (map(i) /= 0) then
        int2%resonant(map(i)) = int1%resonant(i)
     end if
  end do
end subroutine interaction_transfer_relations
```

Make up internal parent-child relations for the particle(s) that are connected to a new interaction `int`.

If `resonant` is defined and true, the connections are marked as resonant in the result interaction

⟨*Interactions: interaction: TBP*⟩+≡

```
procedure :: relate_connections => interaction_relate_connections
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_relate_connections &
     (int, int_in, connection_index, &
      map, map_connections, resonant)
  class(interaction_t), intent(inout), target :: int
  class(interaction_t), intent(in) :: int_in
  integer, dimension(:), intent(in) :: connection_index
  integer, dimension(:), intent(in) :: map, map_connections
  logical, intent(in), optional :: resonant
  logical :: reson
  integer :: i, j, i2, k2
  reson = .false.;  if (present (resonant))  reson = resonant
  do i = 1, size (map_connections)
     k2 = connection_index(i)
     do j = 1, int_in%children(k2)%get_length ()
        i2 = int_in%children(k2)%get_link (j)
        call int%relate (map_connections(i), map(i2))
     end do
     int%resonant(map_connections(i)) = reson
  end do
end subroutine interaction_relate_connections
```

Return the number of source/target links of the internal connections of particle `i`.

⟨*Interactions: public*⟩+≡

```
public :: interaction_get_n_children
public :: interaction_get_n_parents
```

⟨*Interactions: procedures*⟩+≡

```
function interaction_get_n_children (int, i) result (n)
  integer :: n
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  n = int%children(i)%get_length ()
end function interaction_get_n_children

function interaction_get_n_parents (int, i) result (n)
  integer :: n
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  n = int%parents(i)%get_length ()
end function interaction_get_n_parents
```

Return the source/target links of the internal connections of particle `i` as an array.

⟨*Interactions: public*⟩+≡

```
public :: interaction_get_children
public :: interaction_get_parents
```

⟨*Interactions: procedures*⟩+≡

```
function interaction_get_children (int, i) result (idx)
  integer, dimension(:), allocatable :: idx
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  integer :: k, l
  l = int%children(i)%get_length ()
  allocate (idx (l))
  do k = 1, l
     idx(k) = int%children(i)%get_link (k)
  end do
end function interaction_get_children

function interaction_get_parents (int, i) result (idx)
  integer, dimension(:), allocatable :: idx
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  integer :: k, l
  l = int%parents(i)%get_length ()
  allocate (idx (l))
  do k = 1, l
     idx(k) = int%parents(i)%get_link (k)
  end do
end function interaction_get_parents
```

Add a source link from an interaction to a corresponding particle within another interaction. These links affect the propagation of particles: the two linked particles are considered as the same particle, outgoing and incoming.

⟨*Interactions: interaction: TBP*⟩+≡

```
procedure :: set_source_link => interaction_set_source_link
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_set_source_link (int, i, int1, i1)
  class(interaction_t), intent(inout) :: int
  integer, intent(in) :: i
  class(interaction_t), intent(in), target :: int1
  integer, intent(in) :: i1
  if (i /= 0)  call external_link_set (int%source(i), int1, i1)
end subroutine interaction_set_source_link
```

Reassign links to a new interaction (which is an image of the current interaction).

⟨*Interactions: public*⟩+≡

```
public :: interaction_reassign_links
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_reassign_links (int, int_src, int_target)
  type(interaction_t), intent(inout) :: int
  type(interaction_t), intent(in) :: int_src
  type(interaction_t), intent(in), target :: int_target
  integer :: i
  if (allocated (int%source)) then
     do i = 1, size (int%source)
        call external_link_reassign (int%source(i), int_src, int_target)
     end do
  end if
end subroutine interaction_reassign_links
```

Since links are one-directional, if we want to follow them backwards we have to scan all possibilities. This procedure returns the index of the particle within `int` which points to the particle `i1` within interaction `int1`. If unsuccessful, return zero.

⟨*Interactions: public*⟩+≡

```
public :: interaction_find_link
```

⟨*Interactions: procedures*⟩+≡

```
function interaction_find_link (int, int1, i1) result (i)
  integer :: i
  type(interaction_t), intent(in) :: int, int1
  integer, intent(in) :: i1
  type(interaction_t), pointer :: int_tmp
  do i = 1, int%n_tot
     int_tmp => external_link_get_ptr (int%source(i))
     if (int_tmp%tag == int1%tag) then
        if (external_link_get_index (int%source(i)) == i1)  return
     end if
  end do
  i = 0
end function interaction_find_link
```

The inverse: return interaction pointer and index for the ultimate source of `i` within `int`.

⟨*Interactions: interaction: TBP*⟩+≡

```
procedure :: find_source => interaction_find_source
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_find_source (int, i, int1, i1)
  class(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  type(interaction_t), intent(out), pointer :: int1
  integer, intent(out) :: i1
  type(external_link_t) :: link
  link = interaction_get_ultimate_source (int, i)
  int1 => external_link_get_ptr (link)
  i1 = external_link_get_index (link)
end subroutine interaction_find_source
```

Follow source links recursively to return the ultimate source of a particle.

⟨*Interactions: procedures*⟩+≡

```
function interaction_get_ultimate_source (int, i) result (link)
  type(external_link_t) :: link
  type(interaction_t), intent(in) :: int
  integer, intent(in) :: i
  type(interaction_t), pointer :: int_src
  integer :: i_src
  link = int%source(i)
  if (external_link_is_set (link)) then
     do
        int_src => external_link_get_ptr (link)
        i_src = external_link_get_index (link)
        if (external_link_is_set (int_src%source(i_src))) then
           link = int_src%source(i_src)
        else
           exit
        end if
     end do
  end if
end function interaction_get_ultimate_source
```

Update mask entries by merging them with corresponding masks in interactions linked to the current one. The mask determines quantum numbers which are summed over.

Note that both the mask of the current interaction and the mask of the linked interaction are updated (side effect!). This ensures that both agree for the linked particle.

⟨*Interactions: public*⟩+≡

```
public :: interaction_exchange_mask
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_exchange_mask (int)
  type(interaction_t), intent(inout) :: int
  integer :: i, index_link
  type(interaction_t), pointer :: int_link
  do i = 1, int%n_tot
     if (external_link_is_set (int%source(i))) then
        int_link => external_link_get_ptr (int%source(i))
        index_link = external_link_get_index (int%source(i))
        call interaction_merge_mask_entry &
```

```
            (int, i, int_link%mask(index_link))
         call interaction_merge_mask_entry &
            (int_link, index_link, int%mask(i))
      end if
   end do
   call int%freeze ()
end subroutine interaction_exchange_mask
```

Copy momenta from interactions linked to the current one.

⟨*Interactions: interaction: TBP*⟩+≡

```
procedure :: receive_momenta => interaction_receive_momenta
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_receive_momenta (int)
   class(interaction_t), intent(inout) :: int
   integer :: i, index_link
   type(interaction_t), pointer :: int_link
   do i = 1, int%n_tot
      if (external_link_is_set (int%source(i))) then
         int_link => external_link_get_ptr (int%source(i))
         index_link = external_link_get_index (int%source(i))
         call int%set_momentum (int_link%p(index_link), i)
      end if
   end do
end subroutine interaction_receive_momenta
```

The inverse operation: Copy momenta back to the interactions linked to the current one.

⟨*Interactions: public*⟩+≡

```
public :: interaction_send_momenta
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_send_momenta (int)
   type(interaction_t), intent(in) :: int
   integer :: i, index_link
   type(interaction_t), pointer :: int_link
   do i = 1, int%n_tot
      if (external_link_is_set (int%source(i))) then
         int_link => external_link_get_ptr (int%source(i))
         index_link = external_link_get_index (int%source(i))
         call int_link%set_momentum (int%p(i), index_link)
      end if
   end do
end subroutine interaction_send_momenta
```

For numerical comparisons: pacify all momenta in an interaction.

⟨*Interactions: public*⟩+≡

```
public :: interaction_pacify_momenta
```

⟨*Interactions: procedures*⟩+≡

```
subroutine interaction_pacify_momenta (int, acc)
   type(interaction_t), intent(inout) :: int
   real(default), intent(in) :: acc
```

```
    integer :: i
    do i = 1, int%n_tot
       call pacify (int%p(i), acc)
    end do
  end subroutine interaction_pacify_momenta
```

⟨*Interactions: public*⟩+≡

```
  public :: interaction_declare_subtraction
```

⟨*Interactions: procedures*⟩+≡

```
  subroutine interaction_declare_subtraction (int, n_sub)
    type(interaction_t), intent(inout), target :: int
    integer, intent(in) :: n_sub
    type(state_iterator_t) :: it
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    integer :: i, s
    integer :: n_me_orig
    complex(default), dimension(:), allocatable :: me_orig
    call it%init (int%state_matrix)
    i = 1; n_me_orig = int%state_matrix%get_n_matrix_elements ()
    allocate (me_orig (n_me_orig))
    allocate (qn (it%get_depth ()))
    do while (it%is_valid () .and. i <= n_me_orig)
       qn = it%get_quantum_numbers ()
       me_orig (i) = it%get_matrix_element ()
       do s = 1, n_sub
          call qn%set_subtraction_index (s)
          call int%state_matrix%add_state (qn)
       end do
       call it%advance ()
       i = i + 1
    end do
    call int%state_matrix%freeze()
    do i = 1, n_me_orig
       call int%state_matrix%set_matrix_element (i, me_orig(i))
       call int%state_matrix%set_matrix_element (i + n_me_orig, me_orig(i))
    end do
  end subroutine interaction_declare_subtraction
```

11.2.8 Recovering connections

When creating an evaluator for two interactions, we have to know by which particles they are connected. The connection indices can be determined if we have two linked interactions. We assume that `int1` is the source and `int2` the target, so the connections of interest are stored within `int2`. A connection is found if either the source is `int1`, or the (ultimate) source of a particle within `int2` coincides with the (ultimate) source of a aparticle within `int1`. The result is an array of index pairs.

To make things simple, we scan the interaction twice, once for counting hits, then allocate the array, then scan again and store the connections.

The connections are scanned for `int2`, which has sources in `int1`. It may happen that the order of connections is interchanged (crossed). We require the

indices in int1 to be sorted, so we reorder both index arrays correspondingly before returning them. (After this, the indices in int2 may be out of order.)

⟨*Interactions: public*⟩+≡

```
public :: find_connections
```

⟨*Interactions: procedures*⟩+≡

```
subroutine find_connections (int1, int2, n, connection_index)
  class(interaction_t), intent(in) :: int1, int2
  integer, intent(out) :: n
  integer, dimension(:,:), intent(out), allocatable :: connection_index
  integer, dimension(:,:), allocatable :: conn_index_tmp
  integer, dimension(:), allocatable :: ordering
  integer :: i, j, k
  type(external_link_t) :: link2, link1
  type(interaction_t), pointer :: int_link, int_link1
  n = 0
  do i = 1, size (int2%source)
     link2 = interaction_get_ultimate_source (int2, i)
     if (external_link_is_set (link2)) then
        int_link => external_link_get_ptr (link2)
        if (int_link%tag == int1%tag) then
           n = n + 1
        else
           k = external_link_get_index (link2)
           do j = 1, size (int1%source)
              link1 = interaction_get_ultimate_source (int1, j)
              if (external_link_is_set (link1)) then
                 int_link1 => external_link_get_ptr (link1)
                 if (int_link1%tag == int_link%tag) then
                    if (external_link_get_index (link1) == k) then
                       n = n + 1
                    end if
                 end if
              end if
           end do
        end if
     end if
  end do
  allocate (conn_index_tmp (n, 2))
  n = 0
  do i = 1, size (int2%source)
     link2 = interaction_get_ultimate_source (int2, i)
     if (external_link_is_set (link2)) then
        int_link => external_link_get_ptr (link2)
        if (int_link%tag == int1%tag) then
           n = n + 1
           conn_index_tmp(n,1) = external_link_get_index (int2%source(i))
           conn_index_tmp(n,2) = i
        else
           k = external_link_get_index (link2)
           do j = 1, size (int1%source)
              link1 = interaction_get_ultimate_source (int1, j)
              if (external_link_is_set (link1)) then
                 int_link1 => external_link_get_ptr (link1)
```

```
                  if (int_link1%tag == int_link%tag) then
                     if (external_link_get_index (link1) == k) then
                        n = n + 1
                        conn_index_tmp(n,1) = j
                        conn_index_tmp(n,2) = i
                     end if
                  end if
               end if
            end do
         end if
      end if
   end do
   allocate (connection_index (n, 2))
   if (n > 1) then
      allocate (ordering (n))
      ordering = order (conn_index_tmp(:,1))
      connection_index = conn_index_tmp(ordering,:)
   else
      connection_index = conn_index_tmp
   end if
end subroutine find_connections
```

11.2.9 Unit tests

Test module, followed by the corresponding implementation module.

⟨interactions_ut.f90⟩≡

```
⟨File header⟩

module interactions_ut
  use unit_tests
  use interactions_uti

⟨Standard module head⟩

⟨Interactions: public test⟩

contains

⟨Interactions: test driver⟩

end module interactions_ut
```

⟨interactions_uti.f90⟩≡

```
⟨File header⟩

module interactions_uti

⟨Use kinds⟩
  use lorentz
  use flavors
  use colors
  use helicities
  use quantum_numbers
```

```
  use state_matrices

  use interactions
```

⟨*Standard module head*⟩

⟨*Interactions: test declarations*⟩

```
contains
```

⟨*Interactions: tests*⟩

```
end module interactions_uti
```

API: driver for the unit tests below.

⟨*Interactions: public test*⟩≡

```
  public :: interaction_test
```

⟨*Interactions: test driver*⟩≡

```
  subroutine interaction_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```

⟨*Interactions: execute tests*⟩

```
  end subroutine interaction_test
```

Generate an interaction of a polarized virtual photon and a colored quark which may be either up or down. Remove the quark polarization. Generate another interaction for the quark radiating a photon and link this to the first interation. The radiation ignores polarization; transfer this information to the first interaction to simplify it. Then, transfer the momentum to the radiating quark and perform a splitting.

⟨*Interactions: execute tests*⟩≡

```
  call test (interaction_1, "interaction_1", &
       "check interaction setup", &
       u, results)
```

⟨*Interactions: test declarations*⟩≡

```
  public :: interaction_1
```

⟨*Interactions: tests*⟩≡

```
  subroutine interaction_1 (u)
    integer, intent(in) :: u
    type(interaction_t), target :: int, rad
    type(vector4_t), dimension(3) :: p
    type(quantum_numbers_mask_t), dimension(3) :: mask
    p(2) = vector4_moving (500._default, 500._default, 1)
    p(3) = vector4_moving (500._default,-500._default, 1)
    p(1) = p(2) + p(3)

    write (u, "(A)")  "* Test output: interaction"
    write (u, "(A)")  "*   Purpose: check routines for interactions"
    write (u, "(A)")

    call int%basic_init (1, 0, 2, set_relations=.true., &
         store_values = .true. )
```

```
  call int_set (int, 1, -1, 1, 1, &
       cmplx (0.3_default, 0.1_default, kind=default))
  call int_set (int, 1, -1,-1, 1, &
       cmplx (0.5_default,-0.7_default, kind=default))
  call int_set (int, 1, 1, 1, 1, &
       cmplx (0.1_default, 0._default, kind=default))
  call int_set (int, -1, 1, -1, 2, &
       cmplx (0.4_default, -0.1_default, kind=default))
  call int_set (int, 1, 1, 1, 2, &
       cmplx (0.2_default, 0._default, kind=default))
  call int%freeze ()
  call int%set_momenta (p)
  mask = quantum_numbers_mask (.false.,.false., [.true.,.true.,.true.])
  call rad%basic_init (1, 0, 2, &
       mask=mask, set_relations=.true., store_values = .true.)
  call rad_set (1)
  call rad_set (2)
  call rad%set_source_link (1, int, 2)
  call interaction_exchange_mask (rad)
  call rad%receive_momenta ()
  p(1) = rad%get_momentum (1)
  p(2) = 0.4_default * p(1)
  p(3) = p(1) - p(2)
  call rad%set_momenta (p(2:3), outgoing=.true.)
  call int%freeze ()
  call rad%freeze ()
  call rad%set_matrix_element &
       (cmplx (0._default, 0._default, kind=default))
  call int%basic_write (u)
  write (u, "(A)")
  call rad%basic_write (u)
  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"
  call int%final ()
  call rad%final ()
  write (u, "(A)")
  write (u, "(A)")  "* Test interaction_1: successful."
contains
  subroutine int_set (int, h1, h2, hq, q, val)
    type(interaction_t), target, intent(inout) :: int
    integer, intent(in) :: h1, h2, hq, q
    type(flavor_t), dimension(3) :: flv
    type(color_t), dimension(3) :: col
    type(helicity_t), dimension(3) :: hel
    type(quantum_numbers_t), dimension(3) :: qn
    complex(default), intent(in) :: val
    call flv%init ([21, q, -q])
    call col(2)%init_col_acl (5, 0)
    call col(3)%init_col_acl (0, 5)
    call hel%init ([h1, hq, -hq], [h2, hq, -hq])
    call qn%init (flv, col, hel)
    call int%add_state (qn)
    call int%set_matrix_element (val)
  end subroutine int_set
```

```
    subroutine rad_set (q)
      integer, intent(in) :: q
      type(flavor_t), dimension(3) :: flv
      type(quantum_numbers_t), dimension(3) :: qn
      call flv%init ([ q, q, 21 ])
      call qn%init (flv)
      call rad%add_state (qn)
    end subroutine rad_set
  end subroutine interaction_1
```

11.3 Matrix element evaluation

The `evaluator_t` type is an extension of the `interaction_t` type. It represents either a density matrix as the square of a transition matrix element, or the product of two density matrices. Usually, some quantum numbers are summed over in the result.

The `interaction_t` subobject represents a multi-particle interaction with incoming, virtual, and outgoing particles and the associated (not necessarily diagonal) density matrix of quantum state. When the evaluator is initialized, this interaction is constructed from the input interaction(s).

In addition, the initialization process sets up a multiplication table. For each matrix element of the result, it states which matrix elements are to be taken from the input interaction(s), multiplied (optionally, with an additional weight factor) and summed over.

Eventually, to a processes we associate a chain of evaluators which are to be evaluated sequentially. The physical event and its matrix element value(s) can be extracted from the last evaluator in such a chain.

Evaluators are constructed only once (as long as this is possible) during an initialization step. Then, for each event, momenta are computed and transferred among evaluators using the links within the interaction subobject. The multiplication tables enable fast evaluation of the result without looking at quantum numbers anymore.

⟨`evaluators.f90`⟩≡
  ⟨*File header*⟩

```
module evaluators
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩

```
  use io_units
  use format_defs, only: FMT_19
  use diagnostics
  use lorentz
  use model_data
  use flavors
  use colors
  use helicities
  use quantum_numbers
  use state_matrices
  use interactions
```

⟨*Standard module head*⟩

⟨*Evaluators: public*⟩

⟨*Evaluators: parameters*⟩

⟨*Evaluators: types*⟩

⟨*Evaluators: interfaces*⟩

```
contains
```

⟨*Evaluators: procedures*⟩

```
end module evaluators
```

11.3.1 Array of pairings

The evaluator contains an array of `pairing_array` objects. This makes up the multiplication table.

Each pairing array contains two list of matrix element indices and a list of numerical factors. The matrix element indices correspond to the input interactions. The corresponding matrix elements are to be multiplied and optionally multiplied by a factor. The results are summed over to yield one specific matrix element of the result evaluator.

⟨*Evaluators: types*⟩≡

```
type :: pairing_array_t
   integer, dimension(:), allocatable :: i1, i2
   complex(default), dimension(:), allocatable :: factor
end type pairing_array_t
```

⟨*Evaluators: procedures*⟩≡

```
elemental subroutine pairing_array_init (pa, n, has_i2, has_factor)
  type(pairing_array_t), intent(out) :: pa
  integer, intent(in) :: n
  logical, intent(in) :: has_i2, has_factor
  allocate (pa%i1 (n))
  if (has_i2)  allocate (pa%i2 (n))
  if (has_factor)  allocate (pa%factor (n))
end subroutine pairing_array_init
```

11.3.2 The evaluator type

Possible variants of evaluators:

⟨*Evaluators: parameters*⟩≡

```
integer, parameter :: &
     EVAL_UNDEFINED = 0, &
     EVAL_PRODUCT = 1, &
     EVAL_SQUARED_FLOWS = 2, &
```

```
      EVAL_SQUARE_WITH_COLOR_FACTORS = 3, &
      EVAL_COLOR_CONTRACTION = 4, &
      EVAL_IDENTITY = 5, &
      EVAL_QN_SUM = 6
```

The evaluator type contains the result interaction and an array of pairing lists, one for each matrix element in the result interaction.

⟨*Evaluators: public*⟩≡

```
public :: evaluator_t
```

⟨*Evaluators: types*⟩+≡

```
type, extends (interaction_t) :: evaluator_t
   private
   integer :: type = EVAL_UNDEFINED
   class(interaction_t), pointer :: int_in1 => null ()
   class(interaction_t), pointer :: int_in2 => null ()
   type(pairing_array_t), dimension(:), allocatable :: pairing_array
 contains
 ⟨Evaluators: evaluator: TBP⟩
end type evaluator_t
```

Output.

⟨*Evaluators: evaluator: TBP*⟩≡

```
procedure :: write => evaluator_write
```

⟨*Evaluators: procedures*⟩+≡

```
subroutine evaluator_write (eval, unit, &
     verbose, show_momentum_sum, show_mass, show_state, show_table, &
     col_verbose, testflag)
  class(evaluator_t), intent(in) :: eval
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
  logical, intent(in), optional :: show_state, show_table, col_verbose
  logical, intent(in), optional :: testflag
  logical :: conjugate, square, show_tab
  integer :: u, i, j
  u = given_output_unit (unit);  if (u < 0)  return
  show_tab = .true.;  if (present (show_table))  show_tab = .false.
  call eval%basic_write &
       (unit, verbose, show_momentum_sum, show_mass, &
          show_state, col_verbose, testflag)
  if (show_tab) then
     write (u, "(1x,A)")  "Matrix-element multiplication"
     write (u, "(2x,A)", advance="no")  "Input interaction 1:"
     if (associated (eval%int_in1)) then
        write (u, "(1x,I0)")  eval%int_in1%get_tag ()
     else
        write (u, "(A)")  " [undefined]"
     end if
     write (u, "(2x,A)", advance="no")  "Input interaction 2:"
     if (associated (eval%int_in2)) then
        write (u, "(1x,I0)")  eval%int_in2%get_tag ()
     else
        write (u, "(A)")  " [undefined]"
     end if
```

```
      select case (eval%type)
      case (EVAL_SQUARED_FLOWS, EVAL_SQUARE_WITH_COLOR_FACTORS)
         conjugate = .true.
         square = .true.
      case default
         conjugate = .false.
         square = .false.
      end select
      if (eval%type == EVAL_IDENTITY) then
         write (u, "(1X,A)") "Identity evaluator, pairing array unused"
         return
      end if
      if (allocated (eval%pairing_array)) then
         do i = 1, size (eval%pairing_array)
            write (u, "(2x,A,I0,A)")  "ME(", i, ") = "
            do j = 1, size (eval%pairing_array(i)%i1)
               write (u, "(4x,A)", advance="no")  "+"
               if (allocated (eval%pairing_array(i)%i2)) then
                  write (u, "(1x,A,I0,A)", advance="no")  &
                       "ME1(", eval%pairing_array(i)%i1(j), ")"
                  if (conjugate) then
                     write (u, "(A)", advance="no")  "* x"
                  else
                     write (u, "(A)", advance="no")  " x"
                  end if
                  write (u, "(1x,A,I0,A)", advance="no")  &
                       "ME2(", eval%pairing_array(i)%i2(j), ")"
               else if (square) then
                  write (u, "(1x,A)", advance="no")  "|"
                  write (u, "(A,I0,A)", advance="no")  &
                       "ME1(", eval%pairing_array(i)%i1(j), ")"
                  write (u, "(A)", advance="no")  "|^2"
               else
                  write (u, "(1x,A,I0,A)", advance="no")  &
                       "ME1(", eval%pairing_array(i)%i1(j), ")"
               end if
               if (allocated (eval%pairing_array(i)%factor)) then
                  write (u, "(1x,A)", advance="no")  "x"
                  write (u, "(1x,'(',," // FMT_19 // ",',',," // FMT_19 // &
                       ",')')") eval%pairing_array(i)%factor(j)
               else
                  write (u, *)
               end if
            end do
         end do
      end if
   end if
end subroutine evaluator_write
```

Assignment: Deep copy of the interaction component.

⟨*Evaluators: public*⟩+≡

```
public :: assignment(=)
```

⟨*Evaluators: interfaces*⟩≡

```
interface assignment(=)
   module procedure evaluator_assign
end interface
```

⟨*Evaluators: procedures*⟩+≡

```
subroutine evaluator_assign (eval_out, eval_in)
  type(evaluator_t), intent(out) :: eval_out
  type(evaluator_t), intent(in) :: eval_in
  eval_out%type = eval_in%type
  eval_out%int_in1 => eval_in%int_in1
  eval_out%int_in2 => eval_in%int_in2
  eval_out%interaction_t = eval_in%interaction_t
  if (allocated (eval_in%pairing_array)) then
     allocate (eval_out%pairing_array (size (eval_in%pairing_array)))
     eval_out%pairing_array = eval_in%pairing_array
  end if
end subroutine evaluator_assign
```

11.3.3 Auxiliary structures for evaluator creation

Creating an evaluator that properly handles all quantum numbers requires some bookkeeping. In this section, we define several auxiliary types and methods that organize and simplify this task. More structures are defined within the specific initializers (as local types and internal subroutines).

These types are currently implemented in a partial object-oriented way: We define some basic methods for initialization etc. here, but the evaluator routines below do access their internals as well. This simplifies some things such as index addressing using array slices, at the expense of losing some clarity.

Index mapping

Index mapping are abundant when constructing an evaluator. To have arrays of index mappings, we define this:

⟨*Evaluators: types*⟩+≡

```
type :: index_map_t
   integer, dimension(:), allocatable :: entry
end type index_map_t
```

⟨*Evaluators: procedures*⟩+≡

```
elemental subroutine index_map_init (map, n)
  type(index_map_t), intent(out) :: map
  integer, intent(in) :: n
  allocate (map%entry (n))
  map%entry = 0
end subroutine index_map_init
```

⟨*Evaluators: procedures*⟩+≡

```
function index_map_exists (map) result (flag)
  logical :: flag
  type(index_map_t), intent(in) :: map
```

```
    flag = allocated (map%entry)
  end function index_map_exists
```

⟨*Evaluators: interfaces*⟩+≡

```
  interface size
     module procedure index_map_size
  end interface
```

⟨*Evaluators: procedures*⟩+≡

```
  function index_map_size (map) result (s)
    integer :: s
    type(index_map_t), intent(in) :: map
    if (allocated (map%entry)) then
       s = size (map%entry)
    else
       s = 0
    end if
  end function index_map_size
```

⟨*Evaluators: interfaces*⟩+≡

```
  interface assignment(=)
     module procedure index_map_assign_int
     module procedure index_map_assign_array
  end interface
```

⟨*Evaluators: procedures*⟩+≡

```
  elemental subroutine index_map_assign_int (map, ival)
    type(index_map_t), intent(inout) :: map
    integer, intent(in) :: ival
    map%entry = ival
  end subroutine index_map_assign_int

  subroutine index_map_assign_array (map, array)
    type(index_map_t), intent(inout) :: map
    integer, dimension(:), intent(in) :: array
    map%entry = array
  end subroutine index_map_assign_array
```

⟨*Evaluators: procedures*⟩+≡

```
  elemental subroutine index_map_set_entry (map, i, ival)
    type(index_map_t), intent(inout) :: map
    integer, intent(in) :: i
    integer, intent(in) :: ival
    map%entry(i) = ival
  end subroutine index_map_set_entry
```

⟨*Evaluators: procedures*⟩+≡

```
  elemental function index_map_get_entry (map, i) result (ival)
    integer :: ival
    type(index_map_t), intent(in) :: map
    integer, intent(in) :: i
    ival = map%entry(i)
```

```
end function index_map_get_entry
```

Index mapping (two-dimensional)

This is a variant with a square matrix instead of an array.

⟨*Evaluators: types*⟩+≡

```
type :: index_map2_t
   integer :: s = 0
   integer, dimension(:,:), allocatable :: entry
end type index_map2_t
```

⟨*Evaluators: procedures*⟩+≡

```
elemental subroutine index_map2_init (map, n)
  type(index_map2_t), intent(out) :: map
  integer, intent(in) :: n
  map%s = n
  allocate (map%entry (n, n))
end subroutine index_map2_init
```

⟨*Evaluators: procedures*⟩+≡

```
function index_map2_exists (map) result (flag)
  logical :: flag
  type(index_map2_t), intent(in) :: map
  flag = allocated (map%entry)
end function index_map2_exists
```

⟨*Evaluators: interfaces*⟩+≡

```
interface size
   module procedure index_map2_size
end interface
```

⟨*Evaluators: procedures*⟩+≡

```
function index_map2_size (map) result (s)
  integer :: s
  type(index_map2_t), intent(in) :: map
  s = map%s
end function index_map2_size
```

⟨*Evaluators: interfaces*⟩+≡

```
interface assignment(=)
   module procedure index_map2_assign_int
end interface
```

⟨*Evaluators: procedures*⟩+≡

```
elemental subroutine index_map2_assign_int (map, ival)
  type(index_map2_t), intent(inout) :: map
  integer, intent(in) :: ival
  map%entry = ival
end subroutine index_map2_assign_int
```

⟨*Evaluators: procedures*⟩+≡
```
elemental subroutine index_map2_set_entry (map, i, j, ival)
  type(index_map2_t), intent(inout) :: map
  integer, intent(in) :: i, j
  integer, intent(in) :: ival
  map%entry(i,j) = ival
end subroutine index_map2_set_entry
```

⟨*Evaluators: procedures*⟩+≡
```
elemental function index_map2_get_entry (map, i, j) result (ival)
  integer :: ival
  type(index_map2_t), intent(in) :: map
  integer, intent(in) :: i, j
  ival = map%entry(i,j)
end function index_map2_get_entry
```

Auxiliary structures: particle mask

This is a simple container of a logical array.

⟨*Evaluators: types*⟩+≡
```
type :: prt_mask_t
   logical, dimension(:), allocatable :: entry
end type prt_mask_t
```

⟨*Evaluators: procedures*⟩+≡
```
subroutine prt_mask_init (mask, n)
  type(prt_mask_t), intent(out) :: mask
  integer, intent(in) :: n
  allocate (mask%entry (n))
end subroutine prt_mask_init
```

⟨*Evaluators: interfaces*⟩+≡
```
interface size
   module procedure prt_mask_size
end interface
```

⟨*Evaluators: procedures*⟩+≡
```
function prt_mask_size (mask) result (s)
  integer :: s
  type(prt_mask_t), intent(in) :: mask
  s = size (mask%entry)
end function prt_mask_size
```

Quantum number containers

Trivial transparent containers:

⟨*Evaluators: types*⟩+≡
```
type :: qn_list_t
   type(quantum_numbers_t), dimension(:,:), allocatable :: qn
```

```
end type qn_list_t

type :: qn_mask_array_t
   type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
end type qn_mask_array_t
```

Auxiliary structures: connection entries

This type is used as intermediate storage when computing the product of two evaluators or the square of an evaluator. The quantum-number array `qn` corresponds to the particles common to both interactions, but irrelevant quantum numbers (color) masked out. The index arrays `index_in` determine the entries in the input interactions that contribute to this connection. `n_index` is the size of these arrays, and `count` is used while filling the entries. Finally, the quantum-number arrays `qn_in_list` are the actual entries in the input interaction that contribute. In the product case, they exclude the connected quantum numbers.

Each evaluator has its own `connection_table` which contains an array of `connection_entry` objects, but also has stuff that specifically applies to the evaluator type. Hence, we do not generalize the `connection_table_t` type.

The filling procedure `connection_entry_add_state` is specific to the various evaluator types.

⟨*Evaluators: types*⟩+≡

```
type :: connection_entry_t
   type(quantum_numbers_t), dimension(:), allocatable :: qn_conn
   integer, dimension(:), allocatable :: n_index
   integer, dimension(:), allocatable :: count
   type(index_map_t), dimension(:), allocatable :: index_in
   type(qn_list_t), dimension(:), allocatable :: qn_in_list
end type connection_entry_t
```

⟨*Evaluators: procedures*⟩+≡

```
subroutine connection_entry_init &
    (entry, n_count, n_map, qn_conn, count, n_rest)
  type(connection_entry_t), intent(out) :: entry
  integer, intent(in) :: n_count, n_map
  type(quantum_numbers_t), dimension(:), intent(in) :: qn_conn
  integer, dimension(n_count), intent(in) :: count
  integer, dimension(n_count), intent(in) :: n_rest
  integer :: i
  allocate (entry%qn_conn (size (qn_conn)))
  allocate (entry%n_index (n_count))
  allocate (entry%count (n_count))
  allocate (entry%index_in (n_map))
  allocate (entry%qn_in_list (n_count))
  entry%qn_conn = qn_conn
  entry%n_index = count
  entry%count = 0
  if (size (entry%index_in) == size (count)) then
     call index_map_init (entry%index_in, count)
  else
     call index_map_init (entry%index_in, count(1))
```

```
    end if
    do i = 1, n_count
      allocate (entry%qn_in_list(i)%qn (n_rest(i), count(i)))
    end do
  end subroutine connection_entry_init
```

⟨*Evaluators: procedures*⟩+≡

```
  subroutine connection_entry_write (entry, unit)
    type(connection_entry_t), intent(in) :: entry
    integer, intent(in), optional :: unit
    integer :: i, j
    integer :: u
    u = given_output_unit (unit)
    call quantum_numbers_write (entry%qn_conn, unit)
    write (u, *)
    do i = 1, size (entry%n_index)
       write (u, *)  "Input interaction", i
       do j = 1, entry%n_index(i)
          if (size (entry%n_index) == size (entry%index_in)) then
             write (u, "(2x,I0,4x,I0,2x)", advance = "no") &
                  j, index_map_get_entry (entry%index_in(i), j)
          else
             write (u, "(2x,I0,4x,I0,2x,I0,2x)", advance = "no") &
                  j, index_map_get_entry (entry%index_in(1), j), &
                     index_map_get_entry (entry%index_in(2), j)
          end if
          call quantum_numbers_write (entry%qn_in_list(i)%qn(:,j), unit)
          write (u, *)
       end do
    end do
  end subroutine connection_entry_write
```

Color handling

For managing color-factor computation, we introduce this local type. The `index` is the index in the color table that corresponds to a given matrix element index in the input interaction. The `col` array stores the color assignments in rows. The `factor` array associates a complex number with each pair of arrays in the color table. The `factor_is_known` array reveals whether a given factor is known already or still has to be computed.

⟨*Evaluators: types*⟩+≡

```
  type :: color_table_t
     integer, dimension(:), allocatable :: index
     type(color_t), dimension(:,:), allocatable :: col
     logical, dimension(:,:), allocatable :: factor_is_known
     complex(default), dimension(:,:), allocatable :: factor
  end type color_table_t
```

This is the initializer. We extract the color states from the given state matrices, establish index mappings between the two states (implemented by the array

me_index), make an array of color states, and initialize the color-factor table. The latter is two-dimensional (includes interference) and not yet filled.

⟨*Evaluators: procedures*⟩+≡

```
subroutine color_table_init (color_table, state, n_tot)
  type(color_table_t), intent(out) :: color_table
  type(state_matrix_t), intent(in) :: state
  integer, intent(in) :: n_tot
  type(state_iterator_t) :: it
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  type(state_matrix_t) :: state_col
  integer :: index, n_col_state
  allocate (color_table%index (state%get_n_matrix_elements ()))
  color_table%index = 0
  allocate (qn (n_tot))
  call state_col%init ()
  call it%init (state)
  do while (it%is_valid ())
     index = it%get_me_index ()
     call qn%init (col = it%get_color ())
     call state_col%add_state (qn, me_index = color_table%index(index))
     call it%advance ()
  end do
  n_col_state = state_col%get_n_matrix_elements ()
  allocate (color_table%col (n_tot, n_col_state))
  call it%init (state_col)
  do while (it%is_valid ())
     index = it%get_me_index ()
     color_table%col(:,index) = it%get_color ()
     call it%advance ()
  end do
  call state_col%final ()
  allocate (color_table%factor_is_known (n_col_state, n_col_state))
  allocate (color_table%factor (n_col_state, n_col_state))
  color_table%factor_is_known = .false.
end subroutine color_table_init
```

Output (debugging use):

⟨*Evaluators: procedures*⟩+≡

```
subroutine color_table_write (color_table, unit)
  type(color_table_t), intent(in) :: color_table
  integer, intent(in), optional :: unit
  integer :: i, j
  integer :: u
  u = given_output_unit (unit)
  write (u, *) "Color table:"
  if (allocated (color_table%index)) then
     write (u, *) "  Index mapping state => color table:"
     do i = 1, size (color_table%index)
        write (u, "(3x,I0,2x,I0,2x)")  i, color_table%index(i)
     end do
     write (u, *) "  Color table:"
     do i = 1, size (color_table%col, 2)
        write (u, "(3x,I0,2x)", advance = "no")  i
```

```
         call color_write (color_table%col(:,i), unit)
         write (u, *)
      end do
      write (u, *) "  Defined color factors:"
      do i = 1, size (color_table%factor, 1)
         do j = 1, size (color_table%factor, 2)
            if (color_table%factor_is_known(i,j)) then
               write (u, *)  i, j, color_table%factor(i,j)
            end if
         end do
      end do
   end if
end subroutine color_table_write
```

This subroutine sets color factors, based on information from the hard matrix element: the list of pairs of color-flow indices (in the basis of the matrix element code), the list of corresponding factors, and the list of mappings from the matrix element index in the input interaction to the color-flow index in the hard matrix element object.

We first determine the mapping of color-flow indices from the hard matrix element code to the current color table. The mapping could be nontrivial because the latter is derived from iterating over a state matrix, which may return states in non-canonical order. The translation table can be determined because we have, for the complete state matrix, both the mapping to the hard interaction (the input `col_index_hi`) and the mapping to the current color table (the component `color_table%index`).

Once this mapping is known, we scan the list of index pairs `color_flow_index` and translate them to valid color-table index pairs. For this pair, the color factor is set using the corresponding entry in the list `col_factor`.

⟨*Evaluators: procedures*⟩+≡

```
subroutine color_table_set_color_factors (color_table, &
     col_flow_index, col_factor, col_index_hi)
  type(color_table_t), intent(inout) :: color_table
  integer, dimension(:,:), intent(in) :: col_flow_index
  complex(default), dimension(:), intent(in) :: col_factor
  integer, dimension(:), intent(in) :: col_index_hi
  integer, dimension(:), allocatable :: hi_to_ct
  integer :: n_cflow
  integer :: hi_index, me_index, ct_index, cf_index
  integer, dimension(2) :: hi_index_pair, ct_index_pair
  n_cflow = size (col_index_hi)
  if (size (color_table%index) /= n_cflow) &
       call msg_bug ("Mismatch between hard matrix element and color table")
  allocate (hi_to_ct (n_cflow))
  do me_index = 1, size (color_table%index)
     ct_index = color_table%index(me_index)
     hi_index = col_index_hi(me_index)
     hi_to_ct(hi_index) = ct_index
  end do
  do cf_index = 1, size (col_flow_index, 2)
     hi_index_pair = col_flow_index(:,cf_index)
     ct_index_pair = hi_to_ct(hi_index_pair)
```

```
      color_table%factor(ct_index_pair(1), ct_index_pair(2)) = &
           col_factor(cf_index)
      color_table%factor_is_known(ct_index_pair(1), ct_index_pair(2)) = .true.
   end do
end subroutine color_table_set_color_factors
```

This function returns a color factor, given two indices which point to the matrix elements of the initial state matrix. Internally, we can map them to the corresponding indices in the color table. As a side effect, we store the color factor in the color table for later lookup. (I.e., this function is impure.)

⟨*Evaluators: procedures*⟩+≡

```
function color_table_get_color_factor (color_table, index1, index2, nc) &
    result (factor)
  real(default) :: factor
  type(color_table_t), intent(inout) :: color_table
  integer, intent(in) :: index1, index2
  integer, intent(in), optional :: nc
  integer :: i1, i2
  i1 = color_table%index(index1)
  i2 = color_table%index(index2)
  if (color_table%factor_is_known(i1,i2)) then
     factor = real(color_table%factor(i1,i2), kind=default)
  else
     factor = compute_color_factor &
          (color_table%col(:,i1), color_table%col(:,i2), nc)
     color_table%factor(i1,i2) = factor
     color_table%factor_is_known(i1,i2) = .true.
  end if
end function color_table_get_color_factor
```

11.3.4 Creating an evaluator: Matrix multiplication

The evaluator for matrix multiplication is the most complicated variant.

The initializer takes two input interactions and constructs the result evaluator, which consists of the interaction and the multiplication table for the product (or convolution) of the two. Normally, the input interactions are connected by one or more common particles (e.g., decay, structure function convolution).

In the result interaction, quantum numbers of the connections can be summed over. This is determined by the `qn_mask_conn` argument. The `qn_mask_rest` argument is its analog for the other particles within the result interaction. (E.g., for the trace of the state matrix, all quantum numbers are summed over.) Finally, the `connections_are_resonant` argument tells whether the connecting particles should be marked as resonant in the final event record. This is useful for decays.

The algorithm consists of the following steps:

1. `find_connections`: Find the particles which are connected, i.e., common to both input interactions. Either they are directly linked, or both are linked to a common source.

2. `compute_index_bounds_and_mappings`: Compute the mappings of particle indices from the input interactions to the result interaction. There is a separate mapping for the connected particles.

3. `accumulate_connected_states`: Create an auxiliary state matrix which lists the possible quantum numbers for the connected particles. When building this matrix, count the number of times each assignment is contained in any of the input states and, for each of the input states, record the index of the matrix element within the new state matrix. For the connected particles, reassign color indices such that no color state is present twice in different color-index assignment. Note that helicity assignments of the connected state can be (and will be) off-diagonal, so no spin correlations are lost in decays.

   Do this for both input interactions.

4. `allocate_connection_entries`: Allocate a table of connections. Each table row corresponds to one state in the auxiliary matrix, and to multiple states of the input interactions. It collects all states of the unconnected particles in the two input interactions that are associated with the particular state (quantum-number assignment) of the connected particles.

5. `fill_connection_table`: Fill the table of connections by scanning both input interactions. When copying states, reassign color indices for the unconnected particles such that they match between all involved particle sets (interaction 1, interaction 2, and connected particles).

6. `make_product_interaction`: Scan the table of connections we have just built. For each entry, construct all possible pairs of states of the unconnected particles and combine them with the specific connected-particle state. This is a possible quantum-number assignment of the result interaction. Now mask all quantum numbers that should be summed over, and append this to the result state matrix. Record the matrix element index of the result. We now have the result interaction.

7. `make_pairing_array`: First allocate the pairing array with the number of entries of the result interaction. Then scan the table of connections again. For each entry, record the indices of the matrix elements which have to be multiplied and summed over in order to compute this particular matrix element. This makes up the multiplication table.

8. `record_links`: Transfer all source pointers from the input interactions to the result interaction. Do the same for the internal parent-child relations and resonance assignments. For the connected particles, make up appropriate additional parent-child relations. This allows for fetching momenta from other interactions when a new event is filled, and to reconstruct the event history when the event is analyzed.

After all this is done, for each event, we just have to evaluate the pairing arrays (multiplication tables) in order to compute the result matrix elements in their proper positions. The quantum-number assignments remain fixed from now on.

⟨*Evaluators: evaluator: TBP*⟩+≡

```
procedure :: init_product => evaluator_init_product
```

⟨*Evaluators: procedures*⟩+≡

```
subroutine evaluator_init_product &
     (eval, int_in1, int_in2, qn_mask_conn, qn_filter_conn, qn_mask_rest, &
      connections_are_resonant)

  class(evaluator_t), intent(out), target :: eval
  class(interaction_t), intent(in), target :: int_in1, int_in2
  type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
  type(quantum_numbers_t), intent(in), optional :: qn_filter_conn
  type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
  logical, intent(in), optional :: connections_are_resonant

  type(qn_mask_array_t), dimension(2) :: qn_mask_in
  type(state_matrix_t), pointer :: state_in1, state_in2

  type :: connection_table_t
     integer :: n_conn = 0
     integer, dimension(2) :: n_rest = 0
     integer :: n_tot = 0
     integer :: n_me_conn = 0
     type(state_matrix_t) :: state
     type(index_map_t), dimension(:), allocatable :: index_conn
     type(connection_entry_t), dimension(:), allocatable :: entry
     type(index_map_t) :: index_result
  end type connection_table_t
  type(connection_table_t) :: connection_table

  integer :: n_in, n_vir, n_out, n_tot
  integer, dimension(2) :: n_rest
  integer :: n_conn

  integer, dimension(:,:), allocatable :: connection_index
  type(index_map_t), dimension(2) :: prt_map_in
  type(index_map_t) :: prt_map_conn
  type(prt_mask_t), dimension(2) :: prt_is_connected
  !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
  type(quantum_numbers_mask_t), dimension(:), allocatable :: &
       qn_mask_conn_initial, int_in1_mask, int_in2_mask

  integer :: i

  eval%type = EVAL_PRODUCT
  eval%int_in1 => int_in1
  eval%int_in2 => int_in2

  state_in1 => int_in1%get_state_matrix_ptr ()
  state_in2 => int_in2%get_state_matrix_ptr ()

  call find_connections (int_in1, int_in2, n_conn, connection_index)
  if (n_conn == 0) then
     call msg_message ("First interaction:")
     call int_in1%basic_write (col_verbose=.true.)
     call msg_message ("Second interaction:")
     call int_in2%basic_write (col_verbose=.true.)
```

```
   call msg_fatal ("Evaluator product: no connections found between factors")
end if
call compute_index_bounds_and_mappings &
     (int_in1, int_in2, n_conn, &
      n_in, n_vir, n_out, n_tot, &
      n_rest, prt_map_in, prt_map_conn)

call prt_mask_init (prt_is_connected(1), int_in1%get_n_tot ())
call prt_mask_init (prt_is_connected(2), int_in2%get_n_tot ())
do i = 1, 2
   prt_is_connected(i)%entry = .true.
   prt_is_connected(i)%entry(connection_index(:,i)) = .false.
end do
!!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
allocate (qn_mask_conn_initial (n_conn), &
     int_in1_mask (n_conn), int_in2_mask (n_conn))
int_in1_mask = int_in1%get_mask (connection_index(:,1))
int_in2_mask = int_in2%get_mask (connection_index(:,2))
do i = 1, n_conn
   qn_mask_conn_initial(i) = int_in1_mask(i) .or. int_in2_mask(i)
end do
allocate (qn_mask_in(1)%mask (int_in1%get_n_tot ()))
allocate (qn_mask_in(2)%mask (int_in2%get_n_tot ()))
qn_mask_in(1)%mask = int_in1%get_mask ()
qn_mask_in(2)%mask = int_in2%get_mask ()

call connection_table_init (connection_table, &
     state_in1, state_in2, &
     qn_mask_conn_initial,  &
     n_conn, connection_index, n_rest, &
     qn_filter_conn)
call connection_table_fill (connection_table, &
     state_in1, state_in2, &
     connection_index, prt_is_connected)
call make_product_interaction (eval%interaction_t, &
     n_in, n_vir, n_out, &
     connection_table, &
     prt_map_in, prt_is_connected, &
     qn_mask_in, qn_mask_conn_initial, &
     qn_mask_conn, qn_filter_conn, qn_mask_rest)
call make_pairing_array (eval%pairing_array, &
     eval%get_n_matrix_elements (), &
     connection_table)
call record_links (eval%interaction_t, &
     int_in1, int_in2, connection_index, prt_map_in, prt_map_conn, &
     prt_is_connected, connections_are_resonant)
call connection_table_final (connection_table)


if (eval%get_n_matrix_elements () == 0) then
   print *, "Evaluator product"
   print *, "First interaction"
   call int_in1%basic_write (col_verbose=.true.)
   print *
```

```
      print *, "Second interaction"
      call int_in2%basic_write (col_verbose=.true.)
      print *
      call msg_fatal ("Product of density matrices is empty", &
          [var_str ("   ------------------------------------------"), &
           var_str ("This happens when two density matrices are convoluted "), &
           var_str ("but the processes they belong to (e.g., production "), &
           var_str ("and decay) do not match. This could happen if the "), &
           var_str ("beam specification does not match the hard "), &
           var_str ("process. Or it may indicate a WHIZARD bug.")])
  end if

contains

  subroutine compute_index_bounds_and_mappings &
       (int1, int2, n_conn, &
        n_in, n_vir, n_out, n_tot, &
        n_rest, prt_map_in, prt_map_conn)
    class(interaction_t), intent(in) :: int1, int2
    integer, intent(in) :: n_conn
    integer, intent(out) :: n_in, n_vir, n_out, n_tot
    integer, dimension(2), intent(out) :: n_rest
    type(index_map_t), dimension(2), intent(out) :: prt_map_in
    type(index_map_t), intent(out) :: prt_map_conn
    integer, dimension(:), allocatable :: index
    integer :: n_in1, n_vir1, n_out1
    integer :: n_in2, n_vir2, n_out2
    integer :: k
    n_in1  = int1%get_n_in  ()
    n_vir1 = int1%get_n_vir ()
    n_out1 = int1%get_n_out () - n_conn
    n_rest(1) = n_in1 + n_vir1 + n_out1
    n_in2  = int2%get_n_in  () - n_conn
    n_vir2 = int2%get_n_vir ()
    n_out2 = int2%get_n_out ()
    n_rest(2) = n_in2 + n_vir2 + n_out2
    n_in  = n_in1  + n_in2
    n_vir = n_vir1 + n_vir2 + n_conn
    n_out = n_out1 + n_out2
    n_tot = n_in + n_vir + n_out
    call index_map_init (prt_map_in, n_rest)
    call index_map_init (prt_map_conn, n_conn)
    allocate (index (n_tot))
    index = [ (i, i = 1, n_tot) ]
    prt_map_in(1)%entry(1 : n_in1) = index(  1 :   n_in1)
    k =     n_in1
    prt_map_in(2)%entry(1 : n_in2) = index(k + 1 : k + n_in2)
    k = k + n_in2
    prt_map_in(1)%entry(n_in1 + 1 : n_in1 + n_vir1) = index(k + 1 : k + n_vir1)
    k = k + n_vir1
    prt_map_in(2)%entry(n_in2 + 1 : n_in2 + n_vir2) = index(k + 1 : k + n_vir2)
    k = k + n_vir2
    prt_map_conn%entry = index(k + 1 : k + n_conn)
    k = k + n_conn
```

```
  prt_map_in(1)%entry(n_in1 + n_vir1 + 1 : n_rest(1)) = index(k + 1 : k + n_out1)
  k = k + n_out1
  prt_map_in(2)%entry(n_in2 + n_vir2 + 1 : n_rest(2)) = index(k + 1 : k + n_out2)
end subroutine compute_index_bounds_and_mappings

subroutine connection_table_init &
    (connection_table, state_in1, state_in2, qn_mask_conn, &
     n_conn, connection_index, n_rest, &
     qn_filter_conn)
  type(connection_table_t), intent(out) :: connection_table
  type(state_matrix_t), intent(in), target :: state_in1, state_in2
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_conn
  integer, intent(in) :: n_conn
  integer, dimension(:,:), intent(in) :: connection_index
  integer, dimension(2), intent(in) :: n_rest
  type(quantum_numbers_t), intent(in), optional :: qn_filter_conn
  integer, dimension(2) :: n_me_in
  type(state_iterator_t) :: it
  type(quantum_numbers_t), dimension(n_conn) :: qn
  integer :: i, me_index_in, me_index_conn, n_me_conn
  integer, dimension(2) :: me_count
  connection_table%n_conn = n_conn
  connection_table%n_rest = n_rest
  n_me_in(1) = state_in1%get_n_matrix_elements ()
  n_me_in(2) = state_in2%get_n_matrix_elements ()
  allocate (connection_table%index_conn (2))
  call index_map_init (connection_table%index_conn, n_me_in)
  connection_table%index_conn = 0
  call connection_table%state%init (n_counters = 2)
  do i = 1, 2
     select case (i)
     case (1);  call it%init (state_in1)
     case (2);  call it%init (state_in2)
     end select
     do while (it%is_valid ())
        qn = it%get_quantum_numbers (connection_index(:,i))
        call qn%undefine (qn_mask_conn)
        if (present (qn_filter_conn)) then
           if (.not. all (qn .match. qn_filter_conn)) then
              call it%advance ();  cycle
           end if
        end if
        call quantum_numbers_canonicalize_color (qn)
        me_index_in = it%get_me_index ()
        call connection_table%state%add_state (qn, &
            counter_index = i, me_index = me_index_conn)
        call index_map_set_entry (connection_table%index_conn(i), &
             me_index_in, me_index_conn)
        call it%advance ()
     end do
  end do
  n_me_conn = connection_table%state%get_n_matrix_elements ()
  connection_table%n_me_conn = n_me_conn
  allocate (connection_table%entry (n_me_conn))
```

```
  call it%init (connection_table%state)
  do while (it%is_valid ())
     i = it%get_me_index ()
     me_count = it%get_me_count ()
     call connection_entry_init (connection_table%entry(i), 2, 2, &
          it%get_quantum_numbers (), me_count, n_rest)
     call it%advance ()
  end do
end subroutine connection_table_init

subroutine connection_table_final (connection_table)
  type(connection_table_t), intent(inout) :: connection_table
  call connection_table%state%final ()
end subroutine connection_table_final

subroutine connection_table_write (connection_table, unit)
  type(connection_table_t), intent(in) :: connection_table
  integer, intent(in), optional :: unit
  integer :: i, j
  integer :: u
  u = given_output_unit (unit)
  write (u, *) "Connection table:"
  call connection_table%state%write (unit)
  if (allocated (connection_table%index_conn)) then
     write (u, *) "  Index mapping input => connection table:"
     do i = 1, size (connection_table%index_conn)
        write (u, *) "  Input state", i
        do j = 1, size (connection_table%index_conn(i))
           write (u, *)  j, &
                 index_map_get_entry (connection_table%index_conn(i), j)
        end do
     end do
  end if
  if (allocated (connection_table%entry)) then
     write (u, *) "  Connection table contents:"
     do i = 1, size (connection_table%entry)
        call connection_entry_write (connection_table%entry(i), unit)
     end do
  end if
  if (index_map_exists (connection_table%index_result)) then
     write (u, *) "  Index mapping connection table => output:"
     do i = 1, size (connection_table%index_result)
        write (u, *)  i, &
             index_map_get_entry (connection_table%index_result, i)
     end do
  end if
end subroutine connection_table_write

subroutine connection_table_fill &
    (connection_table, state_in1, state_in2, &
     connection_index, prt_is_connected)
  type(connection_table_t), intent(inout) :: connection_table
  type(state_matrix_t), intent(in), target :: state_in1, state_in2
  integer, dimension(:,:), intent(in) :: connection_index
```

```
  type(prt_mask_t), dimension(2), intent(in) :: prt_is_connected
  type(state_iterator_t) :: it
  integer :: index_in, index_conn
  integer :: color_offset
  integer :: n_result_entries
  integer :: i, k
  color_offset = connection_table%state%get_max_color_value ()
  do i = 1, 2
     select case (i)
     case (1);  call it%init (state_in1)
     case (2);  call it%init (state_in2)
     end select
     do while (it%is_valid ())
        index_in = it%get_me_index ()
        index_conn = index_map_get_entry &
                          (connection_table%index_conn(i), index_in)
        if (index_conn /= 0) then
           call connection_entry_add_state &
                (connection_table%entry(index_conn), i, &
                 index_in, it%get_quantum_numbers (), &
                 connection_index(:,i), prt_is_connected(i), &
                 color_offset)
        end if
        call it%advance ()
     end do
     color_offset = color_offset + state_in1%get_max_color_value ()
  end do
  n_result_entries = 0
  do k = 1, size (connection_table%entry)
     n_result_entries = &
          n_result_entries + product (connection_table%entry(k)%n_index)
  end do
  call index_map_init (connection_table%index_result, n_result_entries)
end subroutine connection_table_fill

subroutine connection_entry_add_state &
    (entry, i, index_in, qn_in, connection_index, prt_is_connected, &
     color_offset)
  type(connection_entry_t), intent(inout) :: entry
  integer, intent(in) :: i
  integer, intent(in) :: index_in
  type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
  integer, dimension(:), intent(in) :: connection_index
  type(prt_mask_t), intent(in) :: prt_is_connected
  integer, intent(in) :: color_offset
  integer :: c
  integer, dimension(:,:), allocatable :: color_map
  entry%count(i) = entry%count(i) + 1
  c = entry%count(i)
  call make_color_map (color_map, &
       qn_in(connection_index), entry%qn_conn)
  call index_map_set_entry (entry%index_in(i), c, index_in)
  entry%qn_in_list(i)%qn(:,c) = pack (qn_in, prt_is_connected%entry)
  call quantum_numbers_translate_color &
```

```
      (entry%qn_in_list(i)%qn(:,c), color_map, color_offset)
end subroutine connection_entry_add_state

subroutine make_product_interaction (int, &
     n_in, n_vir, n_out, &
     connection_table, &
     prt_map_in, prt_is_connected, &
     qn_mask_in, qn_mask_conn_initial, &
     qn_mask_conn, qn_filter_conn, qn_mask_rest)
  type(interaction_t), intent(out), target :: int
  integer, intent(in) :: n_in, n_vir, n_out
  type(connection_table_t), intent(inout), target :: connection_table
  type(index_map_t), dimension(2), intent(in) :: prt_map_in
  type(prt_mask_t), dimension(2), intent(in) :: prt_is_connected
  type(qn_mask_array_t), dimension(2), intent(in) :: qn_mask_in
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: &
       qn_mask_conn_initial
  type(quantum_numbers_mask_t), intent(in) :: qn_mask_conn
  type(quantum_numbers_t), intent(in), optional :: qn_filter_conn
  type(quantum_numbers_mask_t), intent(in), optional :: qn_mask_rest
  type(index_map_t), dimension(2) :: prt_index_in
  type(index_map_t) :: prt_index_conn
  integer :: n_tot, n_conn
  integer, dimension(2) :: n_rest
  integer :: i, j, k, m
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
  type(connection_entry_t), pointer :: entry
  integer :: result_index
  n_conn = connection_table%n_conn
  n_rest = connection_table%n_rest
  n_tot = sum (n_rest) + n_conn
  allocate (qn (n_tot), qn_mask (n_tot))
  do i = 1, 2
     call index_map_init (prt_index_in(i), n_rest(i))
     prt_index_in(i) = &
          prt_map_in(i)%entry ([ (j, j = 1, n_rest(i)) ])
  end do
  call index_map_init (prt_index_conn, n_conn)
  prt_index_conn = prt_map_conn%entry ([ (j, j = 1, n_conn) ])
  do i = 1, 2
     if (present (qn_mask_rest)) then
        qn_mask(prt_index_in(i)%entry) = &
             pack (qn_mask_in(i)%mask, prt_is_connected(i)%entry) &
             .or. qn_mask_rest
     else
        qn_mask(prt_index_in(i)%entry) = &
             pack (qn_mask_in(i)%mask, prt_is_connected(i)%entry)
     end if
  end do
  qn_mask(prt_index_conn%entry) = qn_mask_conn_initial .or. qn_mask_conn
  call eval%interaction_t%basic_init (n_in, n_vir, n_out, mask = qn_mask)
  m = 1
  do i = 1, connection_table%n_me_conn
```

```
      entry => connection_table%entry(i)
      qn(prt_index_conn%entry) = &
           quantum_numbers_undefined (entry%qn_conn, qn_mask_conn)
      if (present (qn_filter_conn)) then
         if (.not. all (qn(prt_index_conn%entry) .match. qn_filter_conn)) &
              cycle
      end if
      do j = 1, entry%n_index(1)
         qn(prt_index_in(1)%entry) = entry%qn_in_list(1)%qn(:,j)
         do k = 1, entry%n_index(2)
            qn(prt_index_in(2)%entry) = entry%qn_in_list(2)%qn(:,k)
            call int%add_state (qn, me_index = result_index)
            call index_map_set_entry &
                 (connection_table%index_result, m, result_index)
            m = m + 1
         end do
      end do
   end do
   call int%freeze ()
end subroutine make_product_interaction

subroutine make_pairing_array (pa, n_matrix_elements, connection_table)
   type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
   integer, intent(in) :: n_matrix_elements
   type(connection_table_t), intent(in), target :: connection_table
   type(connection_entry_t), pointer :: entry
   integer, dimension(:), allocatable :: n_entries
   integer :: i, j, k, m, r
   allocate (pa (n_matrix_elements))
   allocate (n_entries (n_matrix_elements))
   n_entries = 0
   do m = 1, size (connection_table%index_result)
      r = index_map_get_entry (connection_table%index_result, m)
      n_entries(r) = n_entries(r) + 1
   end do
   call pairing_array_init &
        (pa, n_entries, has_i2=.true., has_factor=.false.)
   m = 1
   n_entries = 0
   do i = 1, connection_table%n_me_conn
      entry => connection_table%entry(i)
      do j = 1, entry%n_index(1)
         do k = 1, entry%n_index(2)
            r = index_map_get_entry (connection_table%index_result, m)
            n_entries(r) = n_entries(r) + 1
            pa(r)%i1(n_entries(r)) = &
                 index_map_get_entry (entry%index_in(1), j)
            pa(r)%i2(n_entries(r)) = &
                 index_map_get_entry (entry%index_in(2), k)
            m = m + 1
         end do
      end do
   end do
end subroutine make_pairing_array
```

```
  subroutine record_links (int, &
       int_in1, int_in2, connection_index, prt_map_in, prt_map_conn, &
       prt_is_connected, connections_are_resonant)
    class(interaction_t), intent(inout) :: int
    class(interaction_t), intent(in), target :: int_in1, int_in2
    integer, dimension(:,:), intent(in) :: connection_index
    type(index_map_t), dimension(2), intent(in) :: prt_map_in
    type(index_map_t), intent(in) :: prt_map_conn
    type(prt_mask_t), dimension(2), intent(in) :: prt_is_connected
    logical, intent(in), optional :: connections_are_resonant
    type(index_map_t), dimension(2) :: prt_map_all
    integer :: i, j, k, ival
    call index_map_init (prt_map_all(1), size (prt_is_connected(1)))
    k = 0
    j = 0
    do i = 1, size (prt_is_connected(1))
       if (prt_is_connected(1)%entry(i)) then
          j = j + 1
          ival = index_map_get_entry (prt_map_in(1), j)
          call index_map_set_entry (prt_map_all(1), i, ival)
       else
          k = k + 1
          ival = index_map_get_entry (prt_map_conn, k)
          call index_map_set_entry (prt_map_all(1), i, ival)
       end if
       call int%set_source_link (ival, int_in1, i)
    end do
    call int_in1%transfer_relations (int, prt_map_all(1)%entry)
    call index_map_init (prt_map_all(2), size (prt_is_connected(2)))
    j = 0
    do i = 1, size (prt_is_connected(2))
       if (prt_is_connected(2)%entry(i)) then
          j = j + 1
          ival = index_map_get_entry (prt_map_in(2), j)
          call index_map_set_entry (prt_map_all(2), i, ival)
          call int%set_source_link (ival, int_in2, i)
       else
          call index_map_set_entry (prt_map_all(2), i, 0)
       end if
    end do
    call int_in2%transfer_relations (int, prt_map_all(2)%entry)
    call int%relate_connections &
         (int_in2, connection_index(:,2), prt_map_all(2)%entry, &
         prt_map_conn%entry, connections_are_resonant)
  end subroutine record_links

end subroutine evaluator_init_product
```

11.3.5 Creating an evaluator: square

The generic initializer for an evaluator that squares a matrix element. Depending on the provided mask, we select the appropriate specific initializer for either diagonal or non-diagonal helicity density matrices.

⟨*Evaluators: evaluator: TBP*⟩+≡

```
procedure :: init_square => evaluator_init_square
```

⟨*Evaluators: procedures*⟩+≡

```
subroutine evaluator_init_square (eval, int_in, qn_mask, &
     col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
  class(evaluator_t), intent(out), target :: eval
  class(interaction_t), intent(in), target :: int_in
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
  integer, dimension(:,:), intent(in), optional :: col_flow_index
  complex(default), dimension(:), intent(in), optional :: col_factor
  integer, dimension(:), intent(in), optional :: col_index_hi
  logical, intent(in), optional :: expand_color_flows
  integer, intent(in), optional :: nc
  if (all (qn_mask%diagonal_helicity ())) then
     call eval%init_square_diag (int_in, qn_mask, &
          col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
  else
     call eval%init_square_nondiag (int_in, qn_mask, &
          col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
  end if
end subroutine evaluator_init_square
```

Color-summed squared matrix (diagonal helicities)

The initializer for an evaluator that squares a matrix element, including color factors. The mask must be such that off-diagonal matrix elements are excluded.

If `color_flows` is set, the evaluator keeps color-flow entries separate and drops all interfering color structures. The color factors are set to unity in this case.

There is only one input interaction. The quantum-number mask is an array, one entry for each particle, so they can be treated individually. For academic purposes, we allow for the number of colors being different from three (but 3 is the default).

The algorithm is analogous to multiplication, with a few notable differences:

1. The connected particles are known, the correspondence is one-to-one. All particles are connected, and the mapping of indices is trivial, which simplifies the following steps.

2. `accumulate_connected_states`: The matrix of connected states encompasses all particles, but color indices are removed. However, ghost states are still kept separate from physical color states. No color-index reassignment is necessary.

3. The table of connections contains single index and quantum-number arrays instead of pairs of them. They are paired with themselves in all possible ways.

4. `make_squared_interaction`: Now apply the predefined quantum-numbers mask, which usually collects all color states (physical and ghosts), and possibly a helicity sum.

5. `make_pairing_array`: For each pair of input states, compute the color factor (including a potential ghost-parity sign) and store this in the pairing array together with the matrix-element indices for multiplication.

6. `record_links`: This is again trivial due to the one-to-one correspondence.

⟨*Evaluators: evaluator: TBP*⟩+≡
```
procedure :: init_square_diag => evaluator_init_square_diag
```

⟨*Evaluators: procedures*⟩+≡
```
subroutine evaluator_init_square_diag (eval, int_in, qn_mask, &
     col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)

  class(evaluator_t), intent(out), target :: eval
  class(interaction_t), intent(in), target :: int_in
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
  integer, dimension(:,:), intent(in), optional :: col_flow_index
  complex(default), dimension(:), intent(in), optional :: col_factor
  integer, dimension(:), intent(in), optional :: col_index_hi
  logical, intent(in), optional :: expand_color_flows
  integer, intent(in), optional :: nc

  integer :: n_in, n_vir, n_out, n_tot
  type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask_initial
  type(state_matrix_t), pointer :: state_in

  type :: connection_table_t
    integer :: n_tot = 0
    integer :: n_me_conn = 0
    type(state_matrix_t) :: state
    type(index_map_t) :: index_conn
    type(connection_entry_t), dimension(:), allocatable :: entry
    type(index_map_t) :: index_result
  end type connection_table_t
  type(connection_table_t) :: connection_table

  logical :: sum_colors
  type(color_table_t) :: color_table

  if (present (expand_color_flows)) then
     sum_colors = .not. expand_color_flows
  else
     sum_colors = .true.
  end if

  if (sum_colors) then
     eval%type = EVAL_SQUARE_WITH_COLOR_FACTORS
  else
     eval%type = EVAL_SQUARED_FLOWS
  end if
  eval%int_in1 => int_in
```

```
  n_in  = int_in%get_n_in  ()
  n_vir = int_in%get_n_vir ()
  n_out = int_in%get_n_out ()
  n_tot = int_in%get_n_tot ()

  state_in => int_in%get_state_matrix_ptr ()

  allocate (qn_mask_initial (n_tot))
  qn_mask_initial = int_in%get_mask ()
  call qn_mask_initial%set_color (sum_colors, mask_cg=.false.)
  if (sum_colors) then
     call color_table_init (color_table, state_in, n_tot)
     if (present (col_flow_index) .and. present (col_factor) &
         .and. present (col_index_hi)) then
        call color_table_set_color_factors &
             (color_table, col_flow_index, col_factor, col_index_hi)
     end if
  end if

  call connection_table_init (connection_table, state_in, &
       qn_mask_initial, qn_mask, n_tot)
  call connection_table_fill (connection_table, state_in)
  call make_squared_interaction (eval%interaction_t, &
       n_in, n_vir, n_out, n_tot, &
       connection_table, sum_colors, qn_mask_initial .or. qn_mask)
  call make_pairing_array (eval%pairing_array, &
       eval%get_n_matrix_elements (), &
       connection_table, sum_colors, color_table, n_in, n_tot, nc)
  call record_links (eval, int_in, n_tot)
  call connection_table_final (connection_table)

contains

  subroutine connection_table_init &
       (connection_table, state_in, qn_mask_in, qn_mask, n_tot)
    type(connection_table_t), intent(out) :: connection_table
    type(state_matrix_t), intent(in), target :: state_in
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
    integer, intent(in) :: n_tot
    type(quantum_numbers_t), dimension(n_tot) :: qn
    type(state_iterator_t) :: it
    integer :: i, n_me_in, me_index_in
    integer :: me_index_conn, n_me_conn
    integer, dimension(1) :: me_count
    logical :: qn_passed
    connection_table%n_tot = n_tot
    n_me_in = state_in%get_n_matrix_elements ()
    call index_map_init (connection_table%index_conn, n_me_in)
    connection_table%index_conn = 0
    call connection_table%state%init (n_counters=1)
    call it%init (state_in)
    do while (it%is_valid ())
```

```
      qn = it%get_quantum_numbers ()
      if (all (quantum_numbers_are_physical (qn, qn_mask))) then
         call qn%undefine (qn_mask_in)
         qn_passed = .true.
         if (qn_passed) then
            me_index_in = it%get_me_index ()
            call connection_table%state%add_state (qn, &
                 counter_index = 1, me_index = me_index_conn)
            call index_map_set_entry (connection_table%index_conn, &
                 me_index_in, me_index_conn)
         end if
      end if
      call it%advance ()
   end do
   n_me_conn = connection_table%state%get_n_matrix_elements ()
   connection_table%n_me_conn = n_me_conn
   allocate (connection_table%entry (n_me_conn))
   call it%init (connection_table%state)
   do while (it%is_valid ())
      i = it%get_me_index ()
      me_count = it%get_me_count ()
      call connection_entry_init (connection_table%entry(i), 1, 2, &
           it%get_quantum_numbers (), me_count, [n_tot])
      call it%advance ()
   end do
end subroutine connection_table_init

subroutine connection_table_final (connection_table)
   type(connection_table_t), intent(inout) :: connection_table
   call connection_table%state%final ()
end subroutine connection_table_final

subroutine connection_table_write (connection_table, unit)
   type(connection_table_t), intent(in) :: connection_table
   integer, intent(in), optional :: unit
   integer :: i
   integer :: u
   u = given_output_unit (unit)
   write (u, *) "Connection table:"
   call connection_table%state%write (unit)
   if (index_map_exists (connection_table%index_conn)) then
      write (u, *) "  Index mapping input => connection table:"
      do i = 1, size (connection_table%index_conn)
         write (u, *)  i, &
                index_map_get_entry (connection_table%index_conn, i)
      end do
   end if
   if (allocated (connection_table%entry)) then
      write (u, *) "  Connection table contents"
      do i = 1, size (connection_table%entry)
         call connection_entry_write (connection_table%entry(i), unit)
      end do
   end if
   if (index_map_exists (connection_table%index_result)) then
```

```
      write (u, *) "  Index mapping connection table => output"
      do i = 1, size (connection_table%index_result)
         write (u, *)  i, &
               index_map_get_entry (connection_table%index_result, i)
      end do
   end if
end subroutine connection_table_write

subroutine connection_table_fill (connection_table, state)
   type(connection_table_t), intent(inout) :: connection_table
   type(state_matrix_t), intent(in), target :: state
   integer :: index_in, index_conn, n_result_entries
   type(state_iterator_t) :: it
   integer :: k
   call it%init (state)
   do while (it%is_valid ())
      index_in = it%get_me_index ()
      index_conn = &
           index_map_get_entry (connection_table%index_conn, index_in)
      if (index_conn /= 0) then
         call connection_entry_add_state &
              (connection_table%entry(index_conn), &
               index_in, it%get_quantum_numbers ())
      end if
      call it%advance ()
   end do
   n_result_entries = 0
   do k = 1, size (connection_table%entry)
      n_result_entries = &
           n_result_entries + connection_table%entry(k)%n_index(1) ** 2
   end do
   call index_map_init (connection_table%index_result, n_result_entries)
   connection_table%index_result = 0
end subroutine connection_table_fill

subroutine connection_entry_add_state (entry, index_in, qn_in)
   type(connection_entry_t), intent(inout) :: entry
   integer, intent(in) :: index_in
   type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
   integer :: c
   entry%count = entry%count + 1
   c = entry%count(1)
   call index_map_set_entry (entry%index_in(1), c, index_in)
   entry%qn_in_list(1)%qn(:,c) = qn_in
end subroutine connection_entry_add_state

subroutine make_squared_interaction (int, &
      n_in, n_vir, n_out, n_tot, &
      connection_table, sum_colors, qn_mask)
   type(interaction_t), intent(out), target :: int
   integer, intent(in) :: n_in, n_vir, n_out, n_tot
   type(connection_table_t), intent(inout), target :: connection_table
   logical, intent(in) :: sum_colors
   type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
```

```
  type(connection_entry_t), pointer :: entry
  integer :: result_index, n_contrib
  integer :: i, m
  type(quantum_numbers_t), dimension(n_tot) :: qn
  call eval%interaction_t%basic_init (n_in, n_vir, n_out, mask=qn_mask)
  m = 0
  do i = 1, connection_table%n_me_conn
     entry => connection_table%entry(i)
     qn = quantum_numbers_undefined (entry%qn_conn, qn_mask)
     if (.not. sum_colors)   call qn(1:n_in)%invert_color ()
     call int%add_state (qn, me_index = result_index)
     n_contrib = entry%n_index(1) ** 2
     connection_table%index_result%entry(m+1:m+n_contrib) = result_index
     m = m + n_contrib
  end do
  call int%freeze ()
end subroutine make_squared_interaction

subroutine make_pairing_array (pa, &
     n_matrix_elements, connection_table, sum_colors, color_table, &
     n_in, n_tot, nc)
  type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
  integer, intent(in) :: n_matrix_elements
  type(connection_table_t), intent(in), target :: connection_table
  logical, intent(in) :: sum_colors
  type(color_table_t), intent(inout) :: color_table
  type(connection_entry_t), pointer :: entry
  integer, intent(in) :: n_in, n_tot
  integer, intent(in), optional :: nc
  integer, dimension(:), allocatable :: n_entries
  integer :: i, k, l, ks, ls, m, r
  integer :: color_multiplicity_in
  allocate (pa (n_matrix_elements))
  allocate (n_entries (n_matrix_elements))
  n_entries = 0
  do m = 1, size (connection_table%index_result)
     r = index_map_get_entry (connection_table%index_result, m)
     n_entries(r) = n_entries(r) + 1
  end do
  call pairing_array_init &
       (pa, n_entries, has_i2 = sum_colors, has_factor = sum_colors)
  m = 1
  n_entries = 0
  do i = 1, connection_table%n_me_conn
     entry => connection_table%entry(i)
     do k = 1, entry%n_index(1)
        if (sum_colors) then
           color_multiplicity_in = &
                 product (abs (quantum_numbers_get_color_type &
                                 (entry%qn_in_list(1)%qn(:n_in, k))))
           do l = 1, entry%n_index(1)
              r = index_map_get_entry (connection_table%index_result, m)
              n_entries(r) = n_entries(r) + 1
              ks = index_map_get_entry (entry%index_in(1), k)
```

```
            ls = index_map_get_entry (entry%index_in(1), l)
            pa(r)%i1(n_entries(r)) = ks
            pa(r)%i2(n_entries(r)) = ls
            pa(r)%factor(n_entries(r)) = &
                 color_table_get_color_factor (color_table, ks, ls, nc) &
                 / color_multiplicity_in
            m = m + 1
         end do
      else
         r = index_map_get_entry (connection_table%index_result, m)
         n_entries(r) = n_entries(r) + 1
         ks = index_map_get_entry (entry%index_in(1), k)
         pa(r)%i1(n_entries(r)) = ks
         m = m + 1
      end if
   end do
 end do
end subroutine make_pairing_array

subroutine record_links (int, int_in, n_tot)
  class(interaction_t), intent(inout) :: int
  class(interaction_t), intent(in), target :: int_in
  integer, intent(in) :: n_tot
  integer, dimension(n_tot) :: map
  integer :: i
  do i = 1, n_tot
     call int%set_source_link (i, int_in, i)
  end do
  map = [ (i, i = 1, n_tot) ]
  call int_in%transfer_relations (int, map)
end subroutine record_links

end subroutine evaluator_init_square_diag
```

Color-summed squared matrix (support nodiagonal helicities)

The initializer for an evaluator that squares a matrix element, including color factors. Unless requested otherwise by the quantum-number mask, the result contains off-diagonal matrix elements. (The input interaction must be diagonal since it represents an amplitude, not a density matrix.)

There is only one input interaction. The quantum-number mask is an array, one entry for each particle, so they can be treated individually. For academic purposes, we allow for the number of colors being different from three (but 3 is the default).

The algorithm is analogous to the previous one, with some additional complications due to the necessity to loop over two helicity indices.

⟨*Evaluators: evaluator: TBP*⟩+≡

```
procedure :: init_square_nondiag => evaluator_init_square_nondiag
```

⟨*Evaluators: procedures*⟩+≡

```
subroutine evaluator_init_square_nondiag (eval, int_in, qn_mask, &
     col_flow_index, col_factor, col_index_hi, expand_color_flows, nc)
```

```
class(evaluator_t), intent(out), target :: eval
class(interaction_t), intent(in), target :: int_in
type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
integer, dimension(:,:), intent(in), optional :: col_flow_index
complex(default), dimension(:), intent(in), optional :: col_factor
integer, dimension(:), intent(in), optional :: col_index_hi
logical, intent(in), optional :: expand_color_flows
integer, intent(in), optional :: nc

integer :: n_in, n_vir, n_out, n_tot
type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask_initial
type(state_matrix_t), pointer :: state_in

type :: connection_table_t
  integer :: n_tot = 0
  integer :: n_me_conn = 0
  type(state_matrix_t) :: state
  type(index_map2_t) :: index_conn
  type(connection_entry_t), dimension(:), allocatable :: entry
  type(index_map_t) :: index_result
end type connection_table_t
type(connection_table_t) :: connection_table

logical :: sum_colors
type(color_table_t) :: color_table

if (present (expand_color_flows)) then
   sum_colors = .not. expand_color_flows
else
   sum_colors = .true.
end if

if (sum_colors) then
   eval%type = EVAL_SQUARE_WITH_COLOR_FACTORS
else
   eval%type = EVAL_SQUARED_FLOWS
end if
eval%int_in1 => int_in

n_in  = int_in%get_n_in  ()
n_vir = int_in%get_n_vir ()
n_out = int_in%get_n_out ()
n_tot = int_in%get_n_tot ()

state_in => int_in%get_state_matrix_ptr ()

allocate (qn_mask_initial (n_tot))
qn_mask_initial = int_in%get_mask ()
call qn_mask_initial%set_color (sum_colors, mask_cg=.false.)
if (sum_colors) then
   call color_table_init (color_table, state_in, n_tot)
   if (present (col_flow_index) .and. present (col_factor) &
       .and. present (col_index_hi)) then
```

```
        call color_table_set_color_factors &
             (color_table, col_flow_index, col_factor, col_index_hi)
     end if
  end if

  call connection_table_init (connection_table, state_in, &
       qn_mask_initial, qn_mask, n_tot)
  call connection_table_fill (connection_table, state_in)
  call make_squared_interaction (eval%interaction_t, &
       n_in, n_vir, n_out, n_tot, &
       connection_table, sum_colors, qn_mask_initial .or. qn_mask)
  call make_pairing_array (eval%pairing_array, &
       eval%get_n_matrix_elements (), &
       connection_table, sum_colors, color_table, n_in, n_tot, nc)
  call record_links (eval, int_in, n_tot)
  call connection_table_final (connection_table)

contains

  subroutine connection_table_init &
       (connection_table, state_in, qn_mask_in, qn_mask, n_tot)
    type(connection_table_t), intent(out) :: connection_table
    type(state_matrix_t), intent(in), target :: state_in
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
    integer, intent(in) :: n_tot
    type(quantum_numbers_t), dimension(n_tot) :: qn1, qn2, qn
    type(state_iterator_t) :: it1, it2, it
    integer :: i, n_me_in, me_index_in1, me_index_in2
    integer :: me_index_conn, n_me_conn
    integer, dimension(1) :: me_count
    logical :: qn_passed
    connection_table%n_tot = n_tot
    n_me_in = state_in%get_n_matrix_elements ()
    call index_map2_init (connection_table%index_conn, n_me_in)
    connection_table%index_conn = 0
    call connection_table%state%init (n_counters=1)
    call it1%init (state_in)
    do while (it1%is_valid ())
       qn1 = it1%get_quantum_numbers ()
       me_index_in1 = it1%get_me_index ()
       call it2%init (state_in)
       do while (it2%is_valid ())
          qn2 = it2%get_quantum_numbers ()
          if (all (quantum_numbers_are_compatible (qn1, qn2, qn_mask))) then
             qn = qn1 .merge. qn2
             call qn%undefine (qn_mask_in)
             qn_passed = .true.
             if (qn_passed) then
                me_index_in2 = it2%get_me_index ()
                call connection_table%state%add_state (qn, &
                     counter_index = 1, me_index = me_index_conn)
                call index_map2_set_entry (connection_table%index_conn, &
                     me_index_in1, me_index_in2, me_index_conn)
```

```
            end if
         end if
         call it2%advance ()
      end do
      call it1%advance ()
   end do
   n_me_conn = connection_table%state%get_n_matrix_elements ()
   connection_table%n_me_conn = n_me_conn
   allocate (connection_table%entry (n_me_conn))
   call it%init (connection_table%state)
   do while (it%is_valid ())
      i = it%get_me_index ()
      me_count = it%get_me_count ()
      call connection_entry_init (connection_table%entry(i), 1, 2, &
           it%get_quantum_numbers (), me_count, [n_tot])
      call it%advance ()
   end do
end subroutine connection_table_init

subroutine connection_table_final (connection_table)
   type(connection_table_t), intent(inout) :: connection_table
   call connection_table%state%final ()
end subroutine connection_table_final

subroutine connection_table_write (connection_table, unit)
   type(connection_table_t), intent(in) :: connection_table
   integer, intent(in), optional :: unit
   integer :: i, j
   integer :: u
   u = given_output_unit (unit)
   write (u, *) "Connection table:"
   call connection_table%state%write (unit)
   if (index_map2_exists (connection_table%index_conn)) then
      write (u, *) "  Index mapping input => connection table:"
      do i = 1, size (connection_table%index_conn)
         do j = 1, size (connection_table%index_conn)
            write (u, *)  i, j, &
                 index_map2_get_entry (connection_table%index_conn, i, j)
         end do
      end do
   end if
   if (allocated (connection_table%entry)) then
      write (u, *) "  Connection table contents"
      do i = 1, size (connection_table%entry)
         call connection_entry_write (connection_table%entry(i), unit)
      end do
   end if
   if (index_map_exists (connection_table%index_result)) then
      write (u, *) "  Index mapping connection table => output"
      do i = 1, size (connection_table%index_result)
         write (u, *)  i, &
              index_map_get_entry (connection_table%index_result, i)
      end do
   end if
```

```
end subroutine connection_table_write

subroutine connection_table_fill (connection_table, state)
  type(connection_table_t), intent(inout), target :: connection_table
  type(state_matrix_t), intent(in), target :: state
  integer :: index1_in, index2_in, index_conn, n_result_entries
  type(state_iterator_t) :: it1, it2
  integer :: k
  call it1%init (state)
  do while (it1%is_valid ())
     index1_in = it1%get_me_index ()
     call it2%init (state)
     do while (it2%is_valid ())
        index2_in = it2%get_me_index ()
        index_conn = index_map2_get_entry &
                        (connection_table%index_conn, index1_in, index2_in)
        if (index_conn /= 0) then
           call connection_entry_add_state &
                 (connection_table%entry(index_conn), &
                  index1_in, index2_in, &
                  it1%get_quantum_numbers () &
                  .merge. &
                  it2%get_quantum_numbers ())
        end if
        call it2%advance ()
     end do
     call it1%advance ()
  end do
  n_result_entries = 0
  do k = 1, size (connection_table%entry)
     n_result_entries = &
          n_result_entries + connection_table%entry(k)%n_index(1)
  end do
  call index_map_init (connection_table%index_result, n_result_entries)
  connection_table%index_result = 0
end subroutine connection_table_fill

subroutine connection_entry_add_state (entry, index1_in, index2_in, qn_in)
  type(connection_entry_t), intent(inout) :: entry
  integer, intent(in) :: index1_in, index2_in
  type(quantum_numbers_t), dimension(:), intent(in) :: qn_in
  integer :: c
  entry%count = entry%count + 1
  c = entry%count(1)
  call index_map_set_entry (entry%index_in(1), c, index1_in)
  call index_map_set_entry (entry%index_in(2), c, index2_in)
  entry%qn_in_list(1)%qn(:,c) = qn_in
end subroutine connection_entry_add_state

subroutine make_squared_interaction (int, &
     n_in, n_vir, n_out, n_tot, &
     connection_table, sum_colors, qn_mask)
  type(interaction_t), intent(out), target :: int
  integer, intent(in) :: n_in, n_vir, n_out, n_tot
```

```
  type(connection_table_t), intent(inout), target :: connection_table
  logical, intent(in) :: sum_colors
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
  type(connection_entry_t), pointer :: entry
  integer :: result_index
  integer :: i, k, m
  type(quantum_numbers_t), dimension(n_tot) :: qn
  call eval%interaction_t%basic_init (n_in, n_vir, n_out, mask=qn_mask)
  m = 0
  do i = 1, connection_table%n_me_conn
     entry => connection_table%entry(i)
     do k = 1, size (entry%qn_in_list(1)%qn, 2)
        qn = quantum_numbers_undefined &
                (entry%qn_in_list(1)%qn(:,k), qn_mask)
        if (.not. sum_colors)  call qn(1:n_in)%invert_color ()
        call int%add_state (qn, me_index = result_index)
        call index_map_set_entry (connection_table%index_result, m + 1, &
             result_index)
        m = m + 1
     end do
  end do
  call int%freeze ()
end subroutine make_squared_interaction

subroutine make_pairing_array (pa, &
     n_matrix_elements, connection_table, sum_colors, color_table, &
     n_in, n_tot, nc)
  type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
  integer, intent(in) :: n_matrix_elements
  type(connection_table_t), intent(in), target :: connection_table
  logical, intent(in) :: sum_colors
  type(color_table_t), intent(inout) :: color_table
  type(connection_entry_t), pointer :: entry
  integer, intent(in) :: n_in, n_tot
  integer, intent(in), optional :: nc
  integer, dimension(:), allocatable :: n_entries
  integer :: i, k, k1s, k2s, m, r
  integer :: color_multiplicity_in
  allocate (pa (n_matrix_elements))
  allocate (n_entries (n_matrix_elements))
  n_entries = 0
  do m = 1, size (connection_table%index_result)
     r = index_map_get_entry (connection_table%index_result, m)
     n_entries(r) = n_entries(r) + 1
  end do
  call pairing_array_init &
       (pa, n_entries, has_i2 = sum_colors, has_factor = sum_colors)
  m = 1
  n_entries = 0
  do i = 1, connection_table%n_me_conn
     entry => connection_table%entry(i)
     do k = 1, entry%n_index(1)
        r = index_map_get_entry (connection_table%index_result, m)
        n_entries(r) = n_entries(r) + 1
```

```
            if (sum_colors) then
               k1s = index_map_get_entry (entry%index_in(1), k)
               k2s = index_map_get_entry (entry%index_in(2), k)
               pa(r)%i1(n_entries(r)) = k1s
               pa(r)%i2(n_entries(r)) = k2s
               color_multiplicity_in = &
                     product (abs (quantum_numbers_get_color_type &
                                   (entry%qn_in_list(1)%qn(:n_in, k))))
               pa(r)%factor(n_entries(r)) = &
                    color_table_get_color_factor (color_table, k1s, k2s, nc) &
                    / color_multiplicity_in
            else
               k1s = index_map_get_entry (entry%index_in(1), k)
               pa(r)%i1(n_entries(r)) = k1s
            end if
            m = m + 1
         end do
      end do
   end subroutine make_pairing_array

   subroutine record_links (int, int_in, n_tot)
      class(interaction_t), intent(inout) :: int
      class(interaction_t), intent(in), target :: int_in
      integer, intent(in) :: n_tot
      integer, dimension(n_tot) :: map
      integer :: i
      do i = 1, n_tot
         call int%set_source_link (i, int_in, i)
      end do
      map = [ (i, i = 1, n_tot) ]
      call int_in%transfer_relations (int, map)
   end subroutine record_links

end subroutine evaluator_init_square_nondiag
```

Copy with additional contracted color states

This evaluator involves no square or multiplication, its matrix elements are just copies of the (single) input interaction. However, the state matrix of the interaction contains additional states that have color indices contracted. This is used for copies of the beam or structure-function interactions that need to match the hard interaction also in the case where its color indices coincide.

⟨*Evaluators: evaluator: TBP*⟩+≡

```
procedure :: init_color_contractions => evaluator_init_color_contractions
```

⟨*Evaluators: procedures*⟩+≡

```
subroutine evaluator_init_color_contractions (eval, int_in)
   class(evaluator_t), intent(out), target :: eval
   type(interaction_t), intent(in), target :: int_in
   integer :: n_in, n_vir, n_out, n_tot
   type(state_matrix_t) :: state_with_contractions
   integer, dimension(:), allocatable :: me_index
   integer, dimension(:), allocatable :: result_index
```

```
  eval%type = EVAL_COLOR_CONTRACTION
  eval%int_in1 => int_in
  n_in  = int_in%get_n_in  ()
  n_vir = int_in%get_n_vir ()
  n_out = int_in%get_n_out ()
  n_tot = int_in%get_n_tot ()
  state_with_contractions = int_in%get_state_matrix_ptr ()
  call state_with_contractions%add_color_contractions ()
  call make_contracted_interaction (eval%interaction_t, &
       me_index, result_index, &
       n_in, n_vir, n_out, n_tot, &
       state_with_contractions, int_in%get_mask ())
  call make_pairing_array (eval%pairing_array, me_index, result_index)
  call record_links (eval, int_in, n_tot)
  call state_with_contractions%final ()

contains

  subroutine make_contracted_interaction (int, &
       me_index, result_index, &
       n_in, n_vir, n_out, n_tot, state, qn_mask)
    type(interaction_t), intent(out), target :: int
    integer, dimension(:), intent(out), allocatable :: me_index
    integer, dimension(:), intent(out), allocatable :: result_index
    integer, intent(in) :: n_in, n_vir, n_out, n_tot
    type(state_matrix_t), intent(in) :: state
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
    type(state_iterator_t) :: it
    integer :: n_me, i
    type(quantum_numbers_t), dimension(n_tot) :: qn
    call int%basic_init (n_in, n_vir, n_out, mask=qn_mask)
    n_me = state%get_n_leaves ()
    allocate (me_index (n_me))
    allocate (result_index (n_me))
    call it%init (state)
    i = 0
    do while (it%is_valid ())
       i = i + 1
       me_index(i) = it%get_me_index ()
       qn = it%get_quantum_numbers ()
       call int%add_state (qn, me_index = result_index(i))
       call it%advance ()
    end do
    call int%freeze ()
  end subroutine make_contracted_interaction

  subroutine make_pairing_array (pa, me_index, result_index)
    type(pairing_array_t), dimension(:), intent(out), allocatable :: pa
    integer, dimension(:), intent(in) :: me_index, result_index
    integer, dimension(:), allocatable :: n_entries
    integer :: n_matrix_elements, r, i
    n_matrix_elements = size (me_index)
    allocate (pa (n_matrix_elements))
    allocate (n_entries (n_matrix_elements))
```

```
    n_entries = 1
    call pairing_array_init &
         (pa, n_entries, has_i2=.false., has_factor=.false.)
    do i = 1, n_matrix_elements
       r = result_index(i)
       pa(r)%i1(1) = me_index(i)
    end do
  end subroutine make_pairing_array

  subroutine record_links (int, int_in, n_tot)
    class(interaction_t), intent(inout) :: int
    class(interaction_t), intent(in), target :: int_in
    integer, intent(in) :: n_tot
    integer, dimension(n_tot) :: map
    integer :: i
    do i = 1, n_tot
       call int%set_source_link (i, int_in, i)
    end do
    map = [ (i, i = 1, n_tot) ]
    call int_in%transfer_relations (int, map)
  end subroutine record_links

end subroutine evaluator_init_color_contractions
```

Auxiliary procedure for initialization

This will become a standard procedure in F2008. The result is true if the number of true values in the mask is odd. We use the function for determining the ghost parity of a quantum-number array.

[tho:] It's not used anymore and `mod (count (mask), 2) == 1` is a cooler implementation anyway.

⟨*(UNUSED) Evaluators: procedures*⟩≡

```
function parity (mask)
  logical :: parity
  logical, dimension(:) :: mask
  integer :: i
  parity = .false.
  do i = 1, size (mask)
     if (mask(i))  parity = .not. parity
  end do
end function parity
```

Reassign external source links from one to another.

⟨*Evaluators: public*⟩+≡

```
public :: evaluator_reassign_links
```

⟨*Evaluators: interfaces*⟩+≡

```
interface evaluator_reassign_links
   module procedure evaluator_reassign_links_eval
   module procedure evaluator_reassign_links_int
end interface
```

⟨*Evaluators: procedures*⟩+≡

```
subroutine evaluator_reassign_links_eval (eval, eval_src, eval_target)
  type(evaluator_t), intent(inout) :: eval
  type(evaluator_t), intent(in) :: eval_src
  type(evaluator_t), intent(in), target :: eval_target
  if (associated (eval%int_in1)) then
     if (eval%int_in1%get_tag () &
          == eval_src%get_tag ()) then
        eval%int_in1 => eval_target%interaction_t
     end if
  end if
  if (associated (eval%int_in2)) then
     if (eval%int_in2%get_tag () &
          == eval_src%get_tag ()) then
        eval%int_in2 => eval_target%interaction_t
     end if
  end if
  call interaction_reassign_links &
       (eval%interaction_t, eval_src%interaction_t, &
       eval_target%interaction_t)
end subroutine evaluator_reassign_links_eval

subroutine evaluator_reassign_links_int (eval, int_src, int_target)
  type(evaluator_t), intent(inout) :: eval
  type(interaction_t), intent(in) :: int_src
  type(interaction_t), intent(in), target :: int_target
  if (associated (eval%int_in1)) then
     if (eval%int_in1%get_tag () &
          == int_src%get_tag ()) then
        eval%int_in1 => int_target
     end if
  end if
  if (associated (eval%int_in2)) then
     if (eval%int_in2%get_tag () &
          == int_src%get_tag ()) then
        eval%int_in2 => int_target
     end if
  end if
  call interaction_reassign_links (eval%interaction_t, int_src, int_target)
end subroutine evaluator_reassign_links_int
```

Return flavor, momentum, and position of the first unstable particle present in the interaction.

⟨*Evaluators: public*⟩+≡

```
public :: evaluator_get_unstable_particle
```

⟨*Evaluators: procedures*⟩+≡

```
subroutine evaluator_get_unstable_particle (eval, flv, p, i)
  type(evaluator_t), intent(in) :: eval
  type(flavor_t), intent(out) :: flv
  type(vector4_t), intent(out) :: p
  integer, intent(out) :: i
  call interaction_get_unstable_particle (eval%interaction_t, flv, p, i)
end subroutine evaluator_get_unstable_particle
```

11.3.6 Creating an evaluator: identity

The identity evaluator creates a copy of the first input evaluator; the second input is not used.

All particles link back to the input evaluatorand the internal relations are copied. As evaluation does take a shortcut by cloning the matrix elements, the pairing array is not used and does not have to be set up.

⟨*Evaluators: evaluator: TBP*⟩+≡

```
procedure :: init_identity => evaluator_init_identity
```

⟨*Evaluators: procedures*⟩+≡

```
subroutine evaluator_init_identity (eval, int)
  class(evaluator_t), intent(out), target :: eval
  class(interaction_t), intent(in), target :: int
  integer :: n_in, n_out, n_vir, n_tot
  integer :: i
  integer, dimension(:), allocatable :: map
  type(state_matrix_t), pointer :: state
  type(state_iterator_t) :: it

  eval%type = EVAL_IDENTITY
  eval%int_in1 => int
  nullify (eval%int_in2)
  n_in = int%get_n_in ()
  n_out = int%get_n_out ()
  n_vir = int%get_n_vir ()
  n_tot = int%get_n_tot ()
  call eval%interaction_t%basic_init (n_in, n_vir, n_out, &
     mask = int%get_mask (), &
     resonant = int%get_resonance_flags ())
  do i = 1, n_tot
     call eval%set_source_link (i, int, i)
  end do
  allocate (map(n_tot))
  map = [(i, i = 1, n_tot)]
  call int%transfer_relations (eval, map)
  state => int%get_state_matrix_ptr ()
  call it%init (state)
  do while (it%is_valid ())
     call eval%add_state (it%get_quantum_numbers (), &
          it%get_me_index ())
     call it%advance ()
  end do
  call eval%freeze ()

end subroutine evaluator_init_identity
```

11.3.7 Creating an evaluator: quantum number sum

This evaluator operates on the diagonal of a density matrix and sums over the quantum numbers specified by the mask. The optional argument `drop` allows to drop a particle from the resulting density matrix. The handling of virtuals is not completely sane, especially in connection with dropping particles.

When summing over matrix element entries, we keep the separation into entries and normalization (in the corresponding evaluation routine below).

⟨*Evaluators: evaluator: TBP*⟩+≡

```
procedure :: init_qn_sum => evaluator_init_qn_sum
```

⟨*Evaluators: procedures*⟩+≡

```
subroutine evaluator_init_qn_sum (eval, int, qn_mask, drop)
  class(evaluator_t), intent(out), target :: eval
  class(interaction_t), target, intent(in) :: int
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask
  logical, intent(in), optional, dimension(:) :: drop
  type(state_iterator_t) :: it_old, it_new
  integer, dimension(:), allocatable :: pairing_size, pairing_target, i_new
  integer, dimension(:), allocatable :: map
  integer :: n_in, n_out, n_vir, n_tot, n_me_old, n_me_new
  integer :: i, j
  type(state_matrix_t), pointer :: state_new, state_old
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  logical :: matched
  logical, dimension(size (qn_mask)) :: dropped
  integer :: ndropped
  integer, dimension(:), allocatable :: inotdropped
  type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
  logical, dimension(:), allocatable :: resonant

  eval%type = EVAL_QN_SUM
  eval%int_in1 => int
  nullify (eval%int_in2)
  if (present (drop)) then
     dropped = drop
  else
     dropped = .false.
  end if
  ndropped = count (dropped)

  n_in = int%get_n_in ()
  n_out = int%get_n_out () - ndropped
  n_vir = int%get_n_vir ()
  n_tot = int%get_n_tot () - ndropped

  allocate (inotdropped (n_tot))
  i = 1
  do j = 1, n_tot + ndropped
     if (dropped (j)) cycle
     inotdropped(i) = j
     i = i + 1
  end do
```

```
allocate (mask(n_tot + ndropped))
mask = int%get_mask ()
allocate (resonant(n_tot + ndropped))
resonant = int%get_resonance_flags ()
call eval%interaction_t%basic_init (n_in, n_vir, n_out, &
   mask = mask(inotdropped) .or. qn_mask(inotdropped), &
   resonant = resonant(inotdropped))
i = 1
do j = 1, n_tot + ndropped
   if (dropped(j)) cycle
   call eval%set_source_link (i, int, j)
   i = i + 1
end do
allocate (map(n_tot + ndropped))
i = 1
do j = 1, n_tot + ndropped
   if (dropped (j)) then
      map(j) = 0
   else
      map(j) = i
      i = i + 1
   end if
end do
call int%transfer_relations (eval, map)

n_me_old = int%get_n_matrix_elements ()
allocate (pairing_size (n_me_old), source = 0)
allocate (pairing_target (n_me_old), source = 0)
pairing_size = 0
state_old => int%get_state_matrix_ptr ()
state_new => eval%get_state_matrix_ptr ()
call it_old%init (state_old)
allocate (qn(n_tot + ndropped))
do while (it_old%is_valid ())
   qn = it_old%get_quantum_numbers ()
   if (.not. all (qn%are_diagonal ())) then
      call it_old%advance ()
      cycle
   end if
   matched = .false.
   call it_new%init (state_new)
   if (eval%get_n_matrix_elements () > 0) then
      do while (it_new%is_valid ())
         if (all (qn(inotdropped) .match. &
            it_new%get_quantum_numbers ())) &
         then
            matched = .true.
            i = it_new%get_me_index ()
            exit
         end if
         call it_new%advance ()
      end do
   end if
   if (.not. matched) then
```

```
         call eval%add_state (qn(inotdropped))
         i = eval%get_n_matrix_elements ()
      end if
      pairing_size(i) = pairing_size(i) + 1
      pairing_target(it_old%get_me_index ()) = i
      call it_old%advance ()
   end do
   call eval%freeze ()

   n_me_new = eval%get_n_matrix_elements ()
   allocate (eval%pairing_array (n_me_new))
   do i = 1, n_me_new
      call pairing_array_init (eval%pairing_array(i), &
           pairing_size(i), .false., .false.)
   end do

   allocate (i_new (n_me_new), source = 0)
   do i = 1, n_me_old
      j = pairing_target(i)
      if (j > 0) then
         i_new(j) = i_new(j) + 1
         eval%pairing_array(j)%i1(i_new(j)) = i
      end if
   end do

end subroutine evaluator_init_qn_sum
```

11.3.8 Evaluation

When the input interactions (which are pointed to in the pairings stored within the evaluator) are filled with values, we can activate the evaluator, i.e., calculate the result values which are stored in the interaction.

The evaluation of matrix elements can be done in parallel. A `forall` construct is not appropriate, however. We would need `do concurrent` here. Nevertheless, the evaluation functions are marked as `pure`.

⟨*Evaluators: evaluator: TBP*⟩+≡

```
procedure :: evaluate => evaluator_evaluate
```

⟨*Evaluators: procedures*⟩+≡

```
subroutine evaluator_evaluate (eval)
  class(evaluator_t), intent(inout), target :: eval
  integer :: i
  select case (eval%type)
  case (EVAL_PRODUCT)
     do i = 1, size(eval%pairing_array)
        call eval%evaluate_product (i, &
             eval%int_in1, eval%int_in2, &
             eval%pairing_array(i)%i1, eval%pairing_array(i)%i2)
     end do
  case (EVAL_SQUARE_WITH_COLOR_FACTORS)
     do i = 1, size(eval%pairing_array)
        call eval%evaluate_product_cf (i, &
```

```
                eval%int_in1, eval%int_in1, &
                eval%pairing_array(i)%i1, eval%pairing_array(i)%i2, &
                eval%pairing_array(i)%factor)
         end do
      case (EVAL_SQUARED_FLOWS)
         do i = 1, size(eval%pairing_array)
            call eval%evaluate_square_c (i, &
                 eval%int_in1, &
                 eval%pairing_array(i)%i1)
         end do
      case (EVAL_COLOR_CONTRACTION)
         do i = 1, size(eval%pairing_array)
            call eval%evaluate_sum (i, &
                 eval%int_in1, &
                 eval%pairing_array(i)%i1)
         end do
      case (EVAL_IDENTITY)
         call eval%set_matrix_element (eval%int_in1)
      case (EVAL_QN_SUM)
         do i = 1, size (eval%pairing_array)
            call eval%evaluate_me_sum (i, &
               eval%int_in1, eval%pairing_array(i)%i1)
            call eval%set_norm (eval%int_in1%get_norm ())
         end do
      end select
   end subroutine evaluator_evaluate
```

11.3.9 Unit tests

Test module, followed by the corresponding implementation module.

⟨evaluators_ut.f90⟩≡

```
⟨File header⟩

module evaluators_ut
  use unit_tests
  use evaluators_uti

⟨Standard module head⟩

⟨Evaluators: public test⟩

contains

⟨Evaluators: test driver⟩

end module evaluators_ut
```

⟨evaluators_uti.f90⟩≡

```
⟨File header⟩

module evaluators_uti

⟨Use kinds⟩
```

```
  use lorentz
  use flavors
  use colors
  use helicities
  use quantum_numbers
  use interactions
  use model_data

  use evaluators
```

⟨*Standard module head*⟩

⟨*Evaluators: test declarations*⟩

```
contains
```

⟨*Evaluators: tests*⟩

```
end module evaluators_uti
```

API: driver for the unit tests below.

⟨*Evaluators: public test*⟩≡

```
  public :: evaluator_test
```

⟨*Evaluators: test driver*⟩≡

```
  subroutine evaluator_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```
⟨*Evaluators: execute tests*⟩
```
  end subroutine evaluator_test
```

Test: Create two interactions. The interactions are twofold connected. The first connection has a helicity index that is kept, the second connection has a helicity index that is summed over. Concatenate the interactions in an evaluator, which thus contains a result interaction. Fill the input interactions with values, activate the evaluator and print the result.

⟨*Evaluators: execute tests*⟩≡

```
  call test (evaluator_1, "evaluator_1", &
       "check evaluators (1)", &
       u, results)
```

⟨*Evaluators: test declarations*⟩≡

```
  public :: evaluator_1
```

⟨*Evaluators: tests*⟩≡

```
  subroutine evaluator_1 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(interaction_t), target :: int_qqtt, int_tbw, int1, int2
    type(flavor_t), dimension(:), allocatable :: flv
    type(color_t), dimension(:), allocatable :: col
    type(helicity_t), dimension(:), allocatable :: hel
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    integer :: f, c, h1, h2, h3
    type(vector4_t), dimension(4) :: p
```

```
type(vector4_t), dimension(2) :: q
type(quantum_numbers_mask_t) :: qn_mask_conn
type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask2
type(evaluator_t), target :: eval, eval2, eval3

call model%init_sm_test ()

write (u, "(A)")   "*** Evaluator for matrix product"
write (u, "(A)")   "***   Construct interaction for qq -> tt"
write (u, "(A)")
call int_qqtt%basic_init (2, 0, 2, set_relations=.true.)
allocate (flv (4), col (4), hel (4), qn (4))
allocate (qn_mask2 (4))
do c = 1, 2
   select case (c)
   case (1)
      call col%init_col_acl ([1, 0, 1, 0], [0, 2, 0, 2])
   case (2)
      call col%init_col_acl ([1, 0, 2, 0], [0, 1, 0, 2])
   end select
   do f = 1, 2
      call flv%init ([f, -f, 6, -6], model)
      do h1 = -1, 1, 2
         call hel(3)%init (h1)
         do h2 = -1, 1, 2
            call hel(4)%init (h2)
            call qn%init (flv, col, hel)
            call int_qqtt%add_state (qn)
         end do
      end do
   end do
end do
call int_qqtt%freeze ()
deallocate (flv, col, hel, qn)
write (u, "(A)")  "***   Construct interaction for t -> bW"
call int_tbw%basic_init (1, 0, 2, set_relations=.true.)
allocate (flv (3), col (3), hel (3), qn (3))
call flv%init ([6, 5, 24], model)
call col%init_col_acl ([1, 1, 0], [0, 0, 0])
do h1 = -1, 1, 2
   call hel(1)%init (h1)
   do h2 = -1, 1, 2
      call hel(2)%init (h2)
      do h3 = -1, 1
         call hel(3)%init (h3)
         call qn%init (flv, col, hel)
         call int_tbw%add_state (qn)
      end do
   end do
end do
call int_tbw%freeze ()
deallocate (flv, col, hel, qn)
write (u, "(A)")  "***   Link interactions"
call int_tbw%set_source_link (1, int_qqtt, 3)
```

```
qn_mask_conn = quantum_numbers_mask (.false.,.false.,.true.)
write (u, "(A)")
write (u, "(A)")  "***   Show input"
call int_qqtt%basic_write (unit = u)
write (u, "(A)")
call int_tbw%basic_write (unit = u)
write (u, "(A)")
write (u, "(A)")  "***   Evaluate product"
call eval%init_product (int_qqtt, int_tbw, qn_mask_conn)
call eval%write (unit = u)

 call int1%basic_init (2, 0, 2, set_relations=.true.)
 call int2%basic_init (1, 0, 2, set_relations=.true.)
 p(1) = vector4_moving (1000._default, 1000._default, 3)
 p(2) = vector4_moving (200._default, 200._default, 2)
 p(3) = vector4_moving (100._default, 200._default, 1)
 p(4) = p(1) - p(2) - p(3)
 call int1%set_momenta (p)
 q(1) = vector4_moving (50._default,-50._default, 3)
 q(2) = p(2) + p(4) - q(1)
 call int2%set_momenta (q, outgoing=.true.)
 call int1%set_matrix_element ([(2._default,0._default), &
      (4._default,1._default), (-3._default,0._default)])
 call int2%set_matrix_element ([(-3._default,0._default), &
      (0._default,1._default), (1._default,2._default)])
 call eval%receive_momenta ()
 call eval%evaluate ()
 call int1%basic_write (unit = u)
 write (u, "(A)")
 call int2%basic_write (unit = u)
 write (u, "(A)")
 call eval%write (unit = u)
 write (u, "(A)")
 call int1%final ()
 call int2%final ()
 call eval%final ()

 write (u, "(A)")
 write (u, "(A)")   "*** Evaluator for matrix square"
 allocate (flv(4), col(4), qn(4))
 call int1%basic_init (2, 0, 2, set_relations=.true.)
 call flv%init ([1, -1, 21, 21], model)
 call col(1)%init ([1])
 call col(2)%init ([-2])
 call col(3)%init ([2, -3])
 call col(4)%init ([3, -1])
 call qn%init (flv, col)
 call int1%add_state (qn)
 call col(3)%init ([3, -1])
 call col(4)%init ([2, -3])
 call qn%init (flv, col)
 call int1%add_state (qn)
 call col(3)%init ([2, -1])
 call col(4)%init (.true.)
```

```
      call qn%init (flv, col)
      call int1%add_state (qn)
      call int1%freeze ()
      ! [qn_mask2 not set since default is false]
      call eval%init_square (int1, qn_mask2, nc=3)
      call eval2%init_square_nondiag (int1, qn_mask2)
      qn_mask2 = quantum_numbers_mask (.false., .true., .true.)
      call eval3%init_square_diag (eval, qn_mask2)
      call int1%set_matrix_element &
           ([(2._default,0._default), &
           (4._default,1._default), (-3._default,0._default)])
      call int1%set_momenta (p)
      call int1%basic_write (unit = u)
      write (u, "(A)")
      call eval%receive_momenta ()
      call eval%evaluate ()
      call eval%write (unit = u)
      write (u, "(A)")
      call eval2%receive_momenta ()
      call eval2%evaluate ()
      call eval2%write (unit = u)
      write (u, "(A)")
      call eval3%receive_momenta ()
      call eval3%evaluate ()
      call eval3%write (unit = u)
      call int1%final ()
      call eval%final ()
      call eval2%final ()
      call eval3%final ()

      call model%final ()
   end subroutine evaluator_1
```

⟨*Evaluators: execute tests*⟩+≡

```
   call test (evaluator_2, "evaluator_2", &
        "check evaluators (2)", &
        u, results)
```

⟨*Evaluators: test declarations*⟩+≡

```
   public :: evaluator_2
```

⟨*Evaluators: tests*⟩+≡

```
   subroutine evaluator_2 (u)
     integer, intent(in) :: u
     type(model_data_t), target :: model
     type(interaction_t), target :: int
     integer :: h1, h2, h3, h4
     type(helicity_t), dimension(4) :: hel
     type(color_t), dimension(4) :: col
     type(flavor_t), dimension(4) :: flv
     type(quantum_numbers_t), dimension(4) :: qn
     type(vector4_t), dimension(4) :: p
     type(evaluator_t) :: eval
     integer :: i
```

```
call model%init_sm_test ()

write (u, "(A)") "*** Creating interaction for e+ e- -> W+ W-"
write (u, "(A)")

call flv%init ([11, -11, 24, -24], model)
do i = 1, 4
   call col(i)%init ()
end do
call int%basic_init (2, 0, 2, set_relations=.true.)
do h1 = -1, 1, 2
   call hel(1)%init (h1)
   do h2 = -1, 1, 2
      call hel(2)%init (h2)
      do h3 = -1, 1
         call hel(3)%init (h3)
         do h4 = -1, 1
            call hel(4)%init (h4)
            call qn%init (flv, col, hel)
            call int%add_state (qn)
         end do
      end do
   end do
end do
call int%freeze ()
call int%set_matrix_element &
   ([(cmplx (i, kind=default), i = 1, 36)])
p(1) = vector4_moving (1000._default, 1000._default, 3)
p(2) = vector4_moving (1000._default, -1000._default, 3)
p(3) = vector4_moving (1000._default, &
   sqrt (1E6_default - 80._default**2), 3)
p(4) = p(1) + p(2) - p(3)
call int%set_momenta (p)
write (u, "(A)") "*** Setting up evaluator"
write (u, "(A)")

call eval%init_identity (int)
write (u, "(A)") "*** Transferring momenta and evaluating"
write (u, "(A)")

call eval%receive_momenta ()
call eval%evaluate ()
write (u, "(A)")  "*******************************************************"
write (u, "(A)")  "   Interaction dump"
write (u, "(A)")  "*******************************************************"
call int%basic_write (unit = u)
write (u, "(A)")
write (u, "(A)")  "*******************************************************"
write (u, "(A)")  "   Evaluator dump"
write (u, "(A)")  "*******************************************************"
call eval%write (unit = u)
write (u, "(A)")
write (u, "(A)")   "*** cleaning up"
call int%final ()
```

```
    call eval%final ()

    call model%final ()
  end subroutine evaluator_2
```

⟨*Evaluators: execute tests*⟩+≡

```
  call test (evaluator_3, "evaluator_3", &
      "check evaluators (3)", &
      u, results)
```

⟨*Evaluators: test declarations*⟩+≡

```
  public :: evaluator_3
```

⟨*Evaluators: tests*⟩+≡

```
  subroutine evaluator_3 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(interaction_t), target :: int
    integer :: h1, h2, h3, h4
    type(helicity_t), dimension(4) :: hel
    type(color_t), dimension(4) :: col
    type(flavor_t), dimension(4) :: flv1, flv2
    type(quantum_numbers_t), dimension(4) :: qn
    type(vector4_t), dimension(4) :: p
    type(evaluator_t) :: eval1, eval2, eval3
    type(quantum_numbers_mask_t), dimension(4) :: qn_mask
    integer :: i

    call model%init_sm_test ()

    write (u, "(A)")  "*** Creating interaction for e+/mu+ e-/mu- -> W+ W-"
    call flv1%init ([11, -11, 24, -24], model)
    call flv2%init ([13, -13, 24, -24], model)
    do i = 1, 4
       call col (i)%init ()
    end do
    call int%basic_init (2, 0, 2, set_relations=.true.)
    do h1 = -1, 1, 2
       call hel(1)%init (h1)
       do h2 = -1, 1, 2
          call hel(2)%init (h2)
          do h3 = -1, 1
             call hel(3)%init (h3)
             do h4 = -1, 1
                call hel(4)%init (h4)
                call qn%init (flv1, col, hel)
                call int%add_state (qn)
                call qn%init (flv2, col, hel)
                call int%add_state (qn)
             end do
          end do
       end do
    end do
    call int%freeze ()
    call int%set_matrix_element &
```

```
      ([(cmplx (1, kind=default), i = 1, 72)])
    p(1) = vector4_moving (1000._default, 1000._default, 3)
    p(2) = vector4_moving (1000._default, -1000._default, 3)
    p(3) = vector4_moving (1000._default, &
       sqrt (1E6_default - 80._default**2), 3)
    p(4) = p(1) + p(2) - p(3)
    call int%set_momenta (p)
    write (u, "(A)")  "*** Setting up evaluators"
    call qn_mask%init (.false., .true., .true.)
    call eval1%init_qn_sum (int, qn_mask)
    call qn_mask%init (.true., .true., .true.)
    call eval2%init_qn_sum (int, qn_mask)
    call qn_mask%init (.false., .true., .false.)
    call eval3%init_qn_sum (int, qn_mask, &
      [.false., .false., .false., .true.])
    write (u, "(A)")  "*** Transferring momenta and evaluating"
    call eval1%receive_momenta ()
    call eval1%evaluate ()
    call eval2%receive_momenta ()
    call eval2%evaluate ()
    call eval3%receive_momenta ()
    call eval3%evaluate ()
    write (u, "(A)")  "*******************************************************"
    write (u, "(A)")  "   Interaction dump"
    write (u, "(A)")  "*******************************************************"
    call int%basic_write (unit = u)
    write (u, "(A)")
    write (u, "(A)")  "*******************************************************"
    write (u, "(A)")  "   Evaluator dump --- spin sum"
    write (u, "(A)")  "*******************************************************"
    call eval1%write (unit = u)
    call eval1%basic_write (unit = u)
    write (u, "(A)")  "*******************************************************"
    write (u, "(A)")  "   Evaluator dump --- spin / flavor sum"
    write (u, "(A)")  "*******************************************************"
    call eval2%write (unit = u)
    call eval2%basic_write (unit = u)
    write (u, "(A)")  "*******************************************************"
    write (u, "(A)")  "   Evaluator dump --- flavor sum, drop last W"
    write (u, "(A)")  "*******************************************************"
    call eval3%write (unit = u)
    call eval3%basic_write (unit = u)
    write (u, "(A)")
    write (u, "(A)")  "*** cleaning up"
    call int%final ()
    call eval1%final ()
    call eval2%final ()
    call eval3%final ()

    call model%final ()
  end subroutine evaluator_3
```

This test evaluates a product with different quantum-number masks and filters for the linked entry.

⟨*Evaluators: execute tests*⟩+≡

```
call test (evaluator_4, "evaluator_4", &
     "check evaluator product with filter", &
     u, results)
```

⟨*Evaluators: test declarations*⟩+≡

```
public :: evaluator_4
```

⟨*Evaluators: tests*⟩+≡

```
subroutine evaluator_4 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(interaction_t), target :: int1, int2
  integer :: h1, h2, h3
  type(helicity_t), dimension(3) :: hel
  type(color_t), dimension(3) :: col
  type(flavor_t), dimension(2) :: flv1, flv2
  type(flavor_t), dimension(3) :: flv3, flv4
  type(quantum_numbers_t), dimension(3) :: qn
  type(evaluator_t) :: eval1, eval2, eval3, eval4
  type(quantum_numbers_mask_t) :: qn_mask
  type(flavor_t) :: flv_filter
  type(helicity_t) :: hel_filter
  type(color_t) :: col_filter
  type(quantum_numbers_t) :: qn_filter
  integer :: i

  write (u, "(A)")  "* Test output: evaluator_4"
  write (u, "(A)")  "*   Purpose: test evaluator products &
       &with mask and filter"
  write (u, "(A)")

  call model%init_sm_test ()

  write (u, "(A)")  "* Creating interaction for e- -> W+/Z"
  write (u, "(A)")

  call flv1%init ([11, 24], model)
  call flv2%init ([11, 23], model)
  do i = 1, 3
     call col(i)%init ()
  end do
  call int1%basic_init (1, 0, 1, set_relations=.true.)
  do h1 = -1, 1, 2
     call hel(1)%init (h1)
     do h2 = -1, 1
        call hel(2)%init (h2)
        call qn(:2)%init (flv1, col(:2), hel(:2))
        call int1%add_state (qn(:2))
        call qn(:2)%init (flv2, col(:2), hel(:2))
        call int1%add_state (qn(:2))
     end do
  end do
  call int1%freeze ()
  call int1%basic_write (u)
```

```
write (u, "(A)")
write (u, "(A)")  "* Creating interaction for W+/Z -> u ubar/dbar"
write (u, "(A)")

call flv3%init ([24, 2, -1], model)
call flv4%init ([23, 2, -2], model)

call int2%basic_init (1, 0, 2, set_relations=.true.)
do h1 = -1, 1
   call hel(1)%init (h1)
   do h2 = -1, 1, 2
      call hel(2)%init (h2)
      do h3 = -1, 1, 2
         call hel(3)%init (h3)
         call qn(:3)%init (flv3, col(:3), hel(:3))
         call int2%add_state (qn(:3))
         call qn(:3)%init (flv4, col(:3), hel(:3))
         call int2%add_state (qn(:3))
      end do
   end do
end do
call int2%freeze ()

call int2%set_source_link (1, int1, 2)
call int2%basic_write (u)

write (u, "(A)")
write (u, "(A)")  "* Product evaluator"
write (u, "(A)")

call qn_mask%init (.false., .false., .false.)
call eval1%init_product (int1, int2, qn_mask_conn = qn_mask)
call eval1%write (u)

write (u, "(A)")
write (u, "(A)")  "* Product evaluator with helicity mask"
write (u, "(A)")

call qn_mask%init (.false., .false., .true.)
call eval2%init_product (int1, int2, qn_mask_conn = qn_mask)
call eval2%write (u)

write (u, "(A)")
write (u, "(A)")  "* Product with flavor filter and helicity mask"
write (u, "(A)")

call qn_mask%init (.false., .false., .true.)
call flv_filter%init (24, model)
call hel_filter%init ()
call col_filter%init ()
call qn_filter%init (flv_filter, col_filter, hel_filter)
call eval3%init_product (int1, int2, &
     qn_mask_conn = qn_mask, qn_filter_conn = qn_filter)
```

```
  call eval3%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Product with helicity filter and mask"
  write (u, "(A)")

  call qn_mask%init (.false., .false., .true.)
  call flv_filter%init ()
  call hel_filter%init (0)
  call col_filter%init ()
  call qn_filter%init (flv_filter, col_filter, hel_filter)
  call eval4%init_product (int1, int2, &
       qn_mask_conn = qn_mask, qn_filter_conn = qn_filter)
  call eval4%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call eval1%final ()
  call eval2%final ()
  call eval3%final ()
  call eval4%final ()

  call int1%final ()
  call int2%final ()

  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: evaluator_4"

end subroutine evaluator_4
```

Chapter 12

Sindarin Built-In Types

Here, we define a couple of types and objects which are useful both internally for `WHIZARD`, and visible to the user, so they correspond to Sindarin types.

particle_specifiers Expressions for particles and particle alternatives, involving particle names.

pdg_arrays Integer (PDG) codes for particles. Useful for particle aliases (e.g., 'quark' for u, d, s etc.).

jets Define (pseudo)jets as objects. Functional only if the `fastjet` library is linked. (This may change in the future.)

subevents Particle collections built from event records, for use in analysis and other Sindarin expressions

analysis Observables, histograms, and plots.

12.1 Particle Specifiers

In this module we introduce a type for specifying a particle or particle alternative. In addition to the particle specifiers (strings separated by colons), the type contains an optional flag `polarized` and a string `decay`. If the `polarized` flag is set, particle polarization information should be kept when generating events for this process. If the `decay` string is set, it is the ID of a decay process which should be applied to this particle when generating events.

In input/output form, the `polarized` flag is indicated by an asterisk `(*)` in brackets, and the `decay` is indicated by its ID in brackets.

The `read` and `write` procedures in this module are not type-bound but generic procedures which handle scalar and array arguments.

⟨`particle_specifiers.f90`⟩≡

```
  ⟨File header⟩

  module particle_specifiers

  ⟨Use strings⟩
    use io_units
    use diagnostics
```

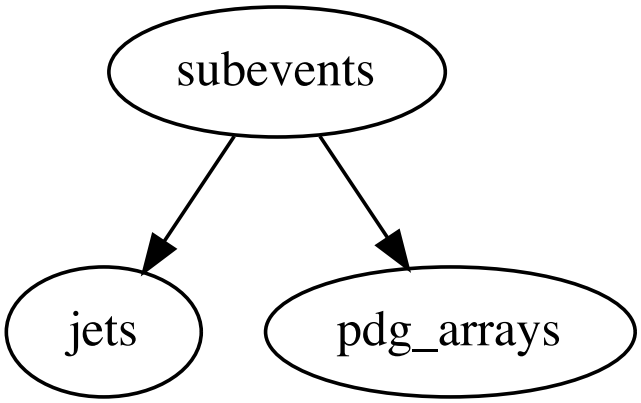

Figure 12.1: Module dependencies in `src/types`.

```
⟨Standard module head⟩

⟨Particle specifiers: public⟩

⟨Particle specifiers: types⟩

⟨Particle specifiers: interfaces⟩

contains

⟨Particle specifiers: procedures⟩

end module particle_specifiers
```

12.1.1 Base type

This is an abstract type which can hold a single particle or an expression.

⟨*Particle specifiers: types*⟩≡

```
type, abstract :: prt_spec_expr_t
 contains
 ⟨Particle specifiers: prt spec expr: TBP⟩
end type prt_spec_expr_t
```

Output, as a string.

⟨*Particle specifiers: prt spec expr: TBP*⟩≡

```
procedure (prt_spec_expr_to_string), deferred :: to_string
```

⟨*Particle specifiers: interfaces*⟩≡

```
abstract interface
   function prt_spec_expr_to_string (object) result (string)
     import
     class(prt_spec_expr_t), intent(in) :: object
     type(string_t) :: string
   end function prt_spec_expr_to_string
end interface
```

Call an `expand` method for all enclosed subexpressions (before handling the current expression).

⟨*Particle specifiers: prt spec expr: TBP*⟩+≡

```
procedure (prt_spec_expr_expand_sub), deferred :: expand_sub
```

⟨*Particle specifiers: interfaces*⟩+≡

```
abstract interface
   subroutine prt_spec_expr_expand_sub (object)
     import
     class(prt_spec_expr_t), intent(inout) :: object
   end subroutine prt_spec_expr_expand_sub
end interface
```

12.1.2 Wrapper type

This wrapper can hold a particle expression of any kind. We need it so we can make variadic arrays.

⟨*Particle specifiers: public*⟩≡

```
public :: prt_expr_t
```

⟨*Particle specifiers: types*⟩+≡

```
type :: prt_expr_t
   class(prt_spec_expr_t), allocatable :: x
 contains
 ⟨Particle specifiers: prt expr: TBP⟩
end type prt_expr_t
```

Output as a string: delegate.

⟨*Particle specifiers: prt expr: TBP*⟩≡

```
procedure :: to_string => prt_expr_to_string
```

⟨*Particle specifiers: procedures*⟩≡

```
recursive function prt_expr_to_string (object) result (string)
  class(prt_expr_t), intent(in) :: object
  type(string_t) :: string
  if (allocated (object%x)) then
     string = object%x%to_string ()
  else
     string = ""
  end if
end function prt_expr_to_string
```

Allocate the expression as a particle specifier and copy the value.

⟨*Particle specifiers: prt expr: TBP*⟩+≡

```
procedure :: init_spec => prt_expr_init_spec
```

⟨*Particle specifiers: procedures*⟩+≡

```
subroutine prt_expr_init_spec (object, spec)
  class(prt_expr_t), intent(out) :: object
  type(prt_spec_t), intent(in) :: spec
  allocate (prt_spec_t :: object%x)
  select type (x => object%x)
  type is (prt_spec_t)
     x = spec
  end select
end subroutine prt_expr_init_spec
```

Allocate as a list/sum and allocate for a given length

⟨*Particle specifiers: prt expr: TBP*⟩+≡

```
procedure :: init_list => prt_expr_init_list
procedure :: init_sum => prt_expr_init_sum
```

⟨*Particle specifiers: procedures*⟩+≡

```
subroutine prt_expr_init_list (object, n)
  class(prt_expr_t), intent(out) :: object
  integer, intent(in) :: n
  allocate (prt_spec_list_t :: object%x)
  select type (x => object%x)
  type is (prt_spec_list_t)
     allocate (x%expr (n))
  end select
end subroutine prt_expr_init_list

subroutine prt_expr_init_sum (object, n)
  class(prt_expr_t), intent(out) :: object
  integer, intent(in) :: n
  allocate (prt_spec_sum_t :: object%x)
  select type (x => object%x)
  type is (prt_spec_sum_t)
     allocate (x%expr (n))
  end select
end subroutine prt_expr_init_sum
```

Return the number of terms. This is unity, except if the expression is a sum.

⟨*Particle specifiers: prt expr: TBP*⟩+≡

```
procedure :: get_n_terms => prt_expr_get_n_terms
```

⟨*Particle specifiers: procedures*⟩+≡

```
function prt_expr_get_n_terms (object) result (n)
  class(prt_expr_t), intent(in) :: object
  integer :: n
  if (allocated (object%x)) then
     select type (x => object%x)
     type is (prt_spec_sum_t)
        n = size (x%expr)
     class default
```

```
         n = 1
      end select
   else
      n = 0
   end if
end function prt_expr_get_n_terms
```

Transform one of the terms, as returned by the previous method, to an array of particle specifiers. The array has more than one entry if the selected term is a list. This makes sense only if the expression has been completely expanded, so the list contains only atoms.

⟨*Particle specifiers: prt expr: TBP*⟩+≡

```
procedure :: term_to_array => prt_expr_term_to_array
```

⟨*Particle specifiers: procedures*⟩+≡

```
recursive subroutine prt_expr_term_to_array (object, array, i)
  class(prt_expr_t), intent(in) :: object
  type(prt_spec_t), dimension(:), intent(inout), allocatable :: array
  integer, intent(in) :: i
  integer :: j
  if (allocated (array))  deallocate (array)
  select type (x => object%x)
  type is (prt_spec_t)
     allocate (array (1))
     array(1) = x
  type is (prt_spec_list_t)
     allocate (array (size (x%expr)))
     do j = 1, size (array)
        select type (y => x%expr(j)%x)
        type is (prt_spec_t)
           array(j) = y
        end select
     end do
  type is (prt_spec_sum_t)
     call x%expr(i)%term_to_array (array, 1)
  end select
end subroutine prt_expr_term_to_array
```

12.1.3 The atomic type

The trivial case is a single particle, including optional decay and polarization attributes.

Definition

The particle is unstable if the `decay` array is allocated. The `polarized` flag and decays may not be set simultaneously.

⟨*Particle specifiers: public*⟩+≡

```
public :: prt_spec_t
```

⟨*Particle specifiers: types*⟩+≡

```
type, extends (prt_spec_expr_t) :: prt_spec_t
   private
   type(string_t) :: name
   logical :: polarized = .false.
   type(string_t), dimension(:), allocatable :: decay
 contains
 ⟨Particle specifiers: prt spec: TBP⟩
end type prt_spec_t
```

I/O

Output. Old-style subroutines.

⟨*Particle specifiers: public*⟩+≡

```
public :: prt_spec_write
```

⟨*Particle specifiers: interfaces*⟩+≡

```
interface prt_spec_write
   module procedure prt_spec_write1
   module procedure prt_spec_write2
end interface prt_spec_write
```

⟨*Particle specifiers: procedures*⟩+≡

```
subroutine prt_spec_write1 (object, unit, advance)
  type(prt_spec_t), intent(in) :: object
  integer, intent(in), optional :: unit
  character(len=*), intent(in), optional :: advance
  character(3) :: adv
  integer :: u
  u = given_output_unit (unit)
  adv = "yes";  if (present (advance))  adv = advance
  write (u, "(A)", advance = adv)  char (object%to_string ())
end subroutine prt_spec_write1
```

Write an array as a list of particle specifiers.

⟨*Particle specifiers: procedures*⟩+≡

```
subroutine prt_spec_write2 (prt_spec, unit, advance)
  type(prt_spec_t), dimension(:), intent(in) :: prt_spec
  integer, intent(in), optional :: unit
  character(len=*), intent(in), optional :: advance
  character(3) :: adv
  integer :: u, i
  u = given_output_unit (unit)
  adv = "yes";  if (present (advance))  adv = advance
  do i = 1, size (prt_spec)
     if (i > 1)  write (u, "(A)", advance="no")  ", "
     call prt_spec_write (prt_spec(i), u, advance="no")
  end do
  write (u, "(A)", advance = adv)
end subroutine prt_spec_write2
```

Read. Input may be string or array of strings.

⟨*Particle specifiers: public*⟩+≡

```
public :: prt_spec_read
```

⟨*Particle specifiers: interfaces*⟩+≡

```
interface prt_spec_read
   module procedure prt_spec_read1
   module procedure prt_spec_read2
end interface prt_spec_read
```

Read a single particle specifier

⟨*Particle specifiers: procedures*⟩+≡

```
pure subroutine prt_spec_read1 (prt_spec, string)
  type(prt_spec_t), intent(out) :: prt_spec
  type(string_t), intent(in) :: string
  type(string_t) :: arg, buffer
  integer :: b1, b2, c, n, i
  b1 = scan (string, "(")
  b2 = scan (string, ")")
  if (b1 == 0) then
     prt_spec%name = trim (adjustl (string))
  else
     prt_spec%name = trim (adjustl (extract (string, 1, b1-1)))
     arg = trim (adjustl (extract (string, b1+1, b2-1)))
     if (arg == "*") then
        prt_spec%polarized = .true.
     else
        n = 0
        buffer = arg
        do
           if (verify (buffer, " ") == 0)  exit
           n = n + 1
           c = scan (buffer, "+")
           if (c == 0)  exit
           buffer = extract (buffer, c+1)
        end do
        allocate (prt_spec%decay (n))
        buffer = arg
        do i = 1, n
           c = scan (buffer, "+")
           if (c == 0)  c = len (buffer) + 1
           prt_spec%decay(i) = trim (adjustl (extract (buffer, 1, c-1)))
           buffer = extract (buffer, c+1)
        end do
     end if
  end if
end subroutine prt_spec_read1
```

Read a particle specifier array, given as a single string. The array is allocated to the correct size.

⟨*Particle specifiers: procedures*⟩+≡

```
pure subroutine prt_spec_read2 (prt_spec, string)
  type(prt_spec_t), dimension(:), intent(out), allocatable :: prt_spec
  type(string_t), intent(in) :: string
```

```
    type(string_t) :: buffer
    integer :: c, i, n
    n = 0
    buffer = string
    do
       n = n + 1
       c = scan (buffer, ",")
       if (c == 0)  exit
       buffer = extract (buffer, c+1)
    end do
    allocate (prt_spec (n))
    buffer = string
    do i = 1, size (prt_spec)
       c = scan (buffer, ",")
       if (c == 0)  c = len (buffer) + 1
       call prt_spec_read (prt_spec(i), &
            trim (adjustl (extract (buffer, 1, c-1))))
       buffer = extract (buffer, c+1)
    end do
  end subroutine prt_spec_read2
```

Constructor

Initialize a particle specifier.

⟨*Particle specifiers: public*⟩+≡

```
  public :: new_prt_spec
```

⟨*Particle specifiers: interfaces*⟩+≡

```
  interface new_prt_spec
     module procedure new_prt_spec
     module procedure new_prt_spec_polarized
     module procedure new_prt_spec_unstable
  end interface new_prt_spec
```

⟨*Particle specifiers: procedures*⟩+≡

```
  elemental function new_prt_spec (name) result (prt_spec)
    type(string_t), intent(in) :: name
    type(prt_spec_t) :: prt_spec
    prt_spec%name = name
  end function new_prt_spec

  elemental function new_prt_spec_polarized (name, polarized) result (prt_spec)
    type(string_t), intent(in) :: name
    logical, intent(in) :: polarized
    type(prt_spec_t) :: prt_spec
    prt_spec%name = name
    prt_spec%polarized = polarized
  end function new_prt_spec_polarized

  pure function new_prt_spec_unstable (name, decay) result (prt_spec)
    type(string_t), intent(in) :: name
    type(string_t), dimension(:), intent(in) :: decay
    type(prt_spec_t) :: prt_spec
    prt_spec%name = name
```

```
    allocate (prt_spec%decay (size (decay)))
    prt_spec%decay = decay
  end function new_prt_spec_unstable
```

Access Methods

Return the particle name without qualifiers

⟨*Particle specifiers: prt spec: TBP*⟩≡

```
  procedure :: get_name => prt_spec_get_name
```

⟨*Particle specifiers: procedures*⟩+≡

```
  elemental function prt_spec_get_name (prt_spec) result (name)
    class(prt_spec_t), intent(in) :: prt_spec
    type(string_t) :: name
    name = prt_spec%name
  end function prt_spec_get_name
```

Return the name with qualifiers

⟨*Particle specifiers: prt spec: TBP*⟩+≡

```
  procedure :: to_string => prt_spec_to_string
```

⟨*Particle specifiers: procedures*⟩+≡

```
  function prt_spec_to_string (object) result (string)
    class(prt_spec_t), intent(in) :: object
    type(string_t) :: string
    integer :: i
    string = object%name
    if (allocated (object%decay)) then
       string = string // "("
       do i = 1, size (object%decay)
          if (i > 1)  string = string // " + "
          string = string // object%decay(i)
       end do
       string = string // ")"
    else if (object%polarized) then
       string = string // "(*)"
    end if
  end function prt_spec_to_string
```

Return the polarization flag

⟨*Particle specifiers: prt spec: TBP*⟩+≡

```
  procedure :: is_polarized => prt_spec_is_polarized
```

⟨*Particle specifiers: procedures*⟩+≡

```
  elemental function prt_spec_is_polarized (prt_spec) result (flag)
    class(prt_spec_t), intent(in) :: prt_spec
    logical :: flag
    flag = prt_spec%polarized
  end function prt_spec_is_polarized
```

The particle is unstable if there is a decay array.

⟨*Particle specifiers: prt spec: TBP*⟩+≡

```
  procedure :: is_unstable => prt_spec_is_unstable
```

⟨*Particle specifiers: procedures*⟩+≡

```
elemental function prt_spec_is_unstable (prt_spec) result (flag)
  class(prt_spec_t), intent(in) :: prt_spec
  logical :: flag
  flag = allocated (prt_spec%decay)
end function prt_spec_is_unstable
```

Return the number of decay channels

⟨*Particle specifiers: prt spec: TBP*⟩+≡

```
procedure :: get_n_decays => prt_spec_get_n_decays
```

⟨*Particle specifiers: procedures*⟩+≡

```
elemental function prt_spec_get_n_decays (prt_spec) result (n)
  class(prt_spec_t), intent(in) :: prt_spec
  integer :: n
  if (allocated (prt_spec%decay)) then
     n = size (prt_spec%decay)
  else
     n = 0
  end if
end function prt_spec_get_n_decays
```

Return the decay channels

⟨*Particle specifiers: prt spec: TBP*⟩+≡

```
procedure :: get_decays => prt_spec_get_decays
```

⟨*Particle specifiers: procedures*⟩+≡

```
subroutine prt_spec_get_decays (prt_spec, decay)
  class(prt_spec_t), intent(in) :: prt_spec
  type(string_t), dimension(:), allocatable, intent(out) :: decay
  if (allocated (prt_spec%decay)) then
     allocate (decay (size (prt_spec%decay)))
     decay = prt_spec%decay
  else
     allocate (decay (0))
  end if
end subroutine prt_spec_get_decays
```

Miscellaneous

There is nothing to expand here:

⟨*Particle specifiers: prt spec: TBP*⟩+≡

```
procedure :: expand_sub => prt_spec_expand_sub
```

⟨*Particle specifiers: procedures*⟩+≡

```
subroutine prt_spec_expand_sub (object)
  class(prt_spec_t), intent(inout) :: object
end subroutine prt_spec_expand_sub
```

12.1.4 List

A list of particle specifiers, indicating, e.g., the final state of a process.

⟨*Particle specifiers: public*⟩+≡
```
public :: prt_spec_list_t
```

⟨*Particle specifiers: types*⟩+≡
```
type, extends (prt_spec_expr_t) :: prt_spec_list_t
   type(prt_expr_t), dimension(:), allocatable :: expr
 contains
 ⟨Particle specifiers: prt spec list: TBP⟩
end type prt_spec_list_t
```

Output: Concatenate the components. Insert brackets if the component is also a list. The components of the `expr` array, if any, should all be filled.

⟨*Particle specifiers: prt spec list: TBP*⟩≡
```
procedure :: to_string => prt_spec_list_to_string
```

⟨*Particle specifiers: procedures*⟩+≡
```
recursive function prt_spec_list_to_string (object) result (string)
  class(prt_spec_list_t), intent(in) :: object
  type(string_t) :: string
  integer :: i
  string = ""
  if (allocated (object%expr)) then
     do i = 1, size (object%expr)
        if (i > 1)  string = string // ", "
        select type (x => object%expr(i)%x)
        type is (prt_spec_list_t)
           string = string // "(" // x%to_string () // ")"
        class default
           string = string // x%to_string ()
        end select
     end do
  end if
end function prt_spec_list_to_string
```

Flatten: if there is a subexpression which is also a list, include the components as direct members of the current list.

⟨*Particle specifiers: prt spec list: TBP*⟩+≡
```
procedure :: flatten => prt_spec_list_flatten
```

⟨*Particle specifiers: procedures*⟩+≡
```
subroutine prt_spec_list_flatten (object)
  class(prt_spec_list_t), intent(inout) :: object
  type(prt_expr_t), dimension(:), allocatable :: tmp_expr
  integer :: i, n_flat, i_flat
  n_flat = 0
  do i = 1, size (object%expr)
     select type (y => object%expr(i)%x)
     type is (prt_spec_list_t)
        n_flat = n_flat + size (y%expr)
     class default
        n_flat = n_flat + 1
```

```
      end select
   end do
   if (n_flat > size (object%expr)) then
      allocate (tmp_expr (n_flat))
      i_flat = 0
      do i = 1, size (object%expr)
         select type (y => object%expr(i)%x)
         type is (prt_spec_list_t)
            tmp_expr (i_flat + 1 : i_flat + size (y%expr)) = y%expr
            i_flat = i_flat + size (y%expr)
         class default
            tmp_expr (i_flat + 1) = object%expr(i)
            i_flat = i_flat + 1
         end select
      end do
   end if
   if (allocated (tmp_expr)) &
        call move_alloc (from = tmp_expr, to = object%expr)
end subroutine prt_spec_list_flatten
```

Convert a list of sums into a sum of lists. (Subexpressions which are not sums are left untouched.)

⟨*Particle specifiers: procedures*⟩+≡

```
subroutine distribute_prt_spec_list (object)
   class(prt_spec_expr_t), intent(inout), allocatable :: object
   class(prt_spec_expr_t), allocatable :: new_object
   integer, dimension(:), allocatable :: n, ii
   integer :: k, n_expr, n_terms, i_term
   select type (object)
   type is (prt_spec_list_t)
      n_expr = size (object%expr)
      allocate (n (n_expr), source = 1)
      allocate (ii (n_expr), source = 1)
      do k = 1, size (object%expr)
         select type (y => object%expr(k)%x)
         type is (prt_spec_sum_t)
            n(k) = size (y%expr)
         end select
      end do
      n_terms = product (n)
      if (n_terms > 1) then
         allocate (prt_spec_sum_t :: new_object)
         select type (new_object)
         type is (prt_spec_sum_t)
            allocate (new_object%expr (n_terms))
            do i_term = 1, n_terms
               allocate (prt_spec_list_t :: new_object%expr(i_term)%x)
               select type (x => new_object%expr(i_term)%x)
               type is (prt_spec_list_t)
                  allocate (x%expr (n_expr))
                  do k = 1, n_expr
                     select type (y => object%expr(k)%x)
                     type is (prt_spec_sum_t)
```

```
                  x%expr(k) = y%expr(ii(k))
               class default
                  x%expr(k) = object%expr(k)
               end select
            end do
         end select
         INCR_INDEX: do k = n_expr, 1, -1
            if (ii(k) < n(k)) then
               ii(k) = ii(k) + 1
               exit INCR_INDEX
            else
               ii(k) = 1
            end if
         end do INCR_INDEX
      end do
   end select
   end if
end select
if (allocated (new_object)) call move_alloc (from = new_object, to = object)
end subroutine distribute_prt_spec_list
```

Apply `expand` to all components of the list.

⟨*Particle specifiers: prt spec list: TBP*⟩+≡

```
procedure :: expand_sub => prt_spec_list_expand_sub
```

⟨*Particle specifiers: procedures*⟩+≡

```
recursive subroutine prt_spec_list_expand_sub (object)
  class(prt_spec_list_t), intent(inout) :: object
  integer :: i
  if (allocated (object%expr)) then
     do i = 1, size (object%expr)
        call object%expr(i)%expand ()
     end do
  end if
end subroutine prt_spec_list_expand_sub
```

12.1.5 Sum

A sum of particle specifiers, indicating, e.g., a sum of final states.

⟨*Particle specifiers: public*⟩+≡

```
public :: prt_spec_sum_t
```

⟨*Particle specifiers: types*⟩+≡

```
type, extends (prt_spec_expr_t) :: prt_spec_sum_t
   type(prt_expr_t), dimension(:), allocatable :: expr
 contains
 ⟨Particle specifiers: prt spec sum: TBP⟩
end type prt_spec_sum_t
```

Output: Concatenate the components. Insert brackets if the component is a list or also a sum. The components of the `expr` array, if any, should all be filled.

⟨*Particle specifiers: prt spec sum: TBP*⟩≡

```
procedure :: to_string => prt_spec_sum_to_string
```

⟨*Particle specifiers: procedures*⟩+≡

```
recursive function prt_spec_sum_to_string (object) result (string)
  class(prt_spec_sum_t), intent(in) :: object
  type(string_t) :: string
  integer :: i
  string = ""
  if (allocated (object%expr)) then
     do i = 1, size (object%expr)
        if (i > 1)  string = string // " + "
        select type (x => object%expr(i)%x)
        type is (prt_spec_list_t)
           string = string // "(" // x%to_string () // ")"
        type is (prt_spec_sum_t)
           string = string // "(" // x%to_string () // ")"
        class default
           string = string // x%to_string ()
        end select
     end do
  end if
end function prt_spec_sum_to_string
```

Flatten: if there is a subexpression which is also a sum, include the components as direct members of the current sum.

This is identical to `prt_spec_list_flatten` above, except for the type.

⟨*Particle specifiers: prt spec sum: TBP*⟩+≡

```
procedure :: flatten => prt_spec_sum_flatten
```

⟨*Particle specifiers: procedures*⟩+≡

```
subroutine prt_spec_sum_flatten (object)
  class(prt_spec_sum_t), intent(inout) :: object
  type(prt_expr_t), dimension(:), allocatable :: tmp_expr
  integer :: i, n_flat, i_flat
  n_flat = 0
  do i = 1, size (object%expr)
     select type (y => object%expr(i)%x)
     type is (prt_spec_sum_t)
        n_flat = n_flat + size (y%expr)
        class default
        n_flat = n_flat + 1
     end select
  end do
  if (n_flat > size (object%expr)) then
     allocate (tmp_expr (n_flat))
     i_flat = 0
     do i = 1, size (object%expr)
        select type (y => object%expr(i)%x)
        type is (prt_spec_sum_t)
           tmp_expr (i_flat + 1 : i_flat + size (y%expr)) = y%expr
           i_flat = i_flat + size (y%expr)
        class default
           tmp_expr (i_flat + 1) = object%expr(i)
           i_flat = i_flat + 1
        end select
     end do
```

```
    end if
    if (allocated (tmp_expr)) &
         call move_alloc (from = tmp_expr, to = object%expr)
  end subroutine prt_spec_sum_flatten
```

Apply `expand` to all terms in the sum.

⟨*Particle specifiers: prt spec sum: TBP*⟩+≡

```
  procedure :: expand_sub => prt_spec_sum_expand_sub
```

⟨*Particle specifiers: procedures*⟩+≡

```
  recursive subroutine prt_spec_sum_expand_sub (object)
    class(prt_spec_sum_t), intent(inout) :: object
    integer :: i
    if (allocated (object%expr)) then
       do i = 1, size (object%expr)
          call object%expr(i)%expand ()
       end do
    end if
  end subroutine prt_spec_sum_expand_sub
```

12.1.6 Expression Expansion

The `expand` method transforms each particle specifier expression into a sum of lists, according to the rules

$$a, (b, c) \to a, b, c \tag{12.1}$$

$$a + (b + c) \to a + b + c \tag{12.2}$$

$$a, b + c \to (a, b) + (a, c) \tag{12.3}$$

Note that the precedence of comma and plus are opposite to this expansion, so the parentheses in the final expression are necessary.

We assume that subexpressions are filled, i.e., arrays are allocated.

⟨*Particle specifiers: prt expr: TBP*⟩+≡

```
  procedure :: expand => prt_expr_expand
```

⟨*Particle specifiers: procedures*⟩+≡

```
  recursive subroutine prt_expr_expand (expr)
    class(prt_expr_t), intent(inout) :: expr
    if (allocated (expr%x)) then
       call distribute_prt_spec_list (expr%x)
       call expr%x%expand_sub ()
       select type (x => expr%x)
       type is (prt_spec_list_t)
          call x%flatten ()
       type is (prt_spec_sum_t)
          call x%flatten ()
       end select
    end if
  end subroutine prt_expr_expand
```

12.1.7 Unit Tests

Test module, followed by the corresponding implementation module.

⟨particle_specifiers_ut.f90⟩≡

```
⟨File header⟩

module particle_specifiers_ut
  use unit_tests
  use particle_specifiers_uti

⟨Standard module head⟩

⟨Particle specifiers: public test⟩

contains

⟨Particle specifiers: test driver⟩

end module particle_specifiers_ut
```

⟨particle_specifiers_uti.f90⟩≡

```
⟨File header⟩

module particle_specifiers_uti

⟨Use strings⟩

  use particle_specifiers

⟨Standard module head⟩

⟨Particle specifiers: test declarations⟩

contains

⟨Particle specifiers: tests⟩

end module particle_specifiers_uti
```

API: driver for the unit tests below.

⟨*Particle specifiers: public test*⟩≡

```
public :: particle_specifiers_test
```

⟨*Particle specifiers: test driver*⟩≡

```
subroutine particle_specifiers_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨Particle specifiers: execute tests⟩
end subroutine particle_specifiers_test
```

Particle specifier array

Define, read and write an array of particle specifiers.

⟨*Particle specifiers: execute tests*⟩≡

```
call test (particle_specifiers_1, "particle_specifiers_1", &
     "Handle particle specifiers", &
     u, results)
```

⟨*Particle specifiers: test declarations*⟩≡

```
public :: particle_specifiers_1
```

⟨*Particle specifiers: tests*⟩≡

```
subroutine particle_specifiers_1 (u)
  integer, intent(in) :: u
  type(prt_spec_t), dimension(:), allocatable :: prt_spec
  type(string_t), dimension(:), allocatable :: decay
  type(string_t), dimension(0) :: no_decay
  integer :: i, j

  write (u, "(A)")  "* Test output: particle_specifiers_1"
  write (u, "(A)")  "*   Purpose: Read and write a particle specifier array"
  write (u, "(A)")

  allocate (prt_spec (5))
  prt_spec = [ &
       new_prt_spec (var_str ("a")), &
       new_prt_spec (var_str ("b"), .true.), &
       new_prt_spec (var_str ("c"), [var_str ("dec1")]), &
       new_prt_spec (var_str ("d"), [var_str ("dec1"), var_str ("dec2")]), &
       new_prt_spec (var_str ("e"), no_decay) &
       ]
  do i = 1, size (prt_spec)
     write (u, "(A)")  char (prt_spec(i)%to_string ())
  end do
  write (u, "(A)")

  call prt_spec_read (prt_spec, &
       var_str (" a, b( *), c( dec1), d (dec1 + dec2 ), e()"))
  call prt_spec_write (prt_spec, u)

  do i = 1, size (prt_spec)
     write (u, "(A)")
     write (u, "(A,A)")  char (prt_spec(i)%get_name ()), ":"
     write (u, "(A,L1)")  "polarized = ", prt_spec(i)%is_polarized ()
     write (u, "(A,L1)")  "unstable  = ", prt_spec(i)%is_unstable ()
     write (u, "(A,I0)")  "n_decays  = ", prt_spec(i)%get_n_decays ()
     call prt_spec(i)%get_decays (decay)
     write (u, "(A)", advance="no") "decays    ="
     do j = 1, size (decay)
        write (u, "(1x,A)", advance="no") char (decay(j))
     end do
     write (u, "(A)")
  end do

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: particle_specifiers_1"
end subroutine particle_specifiers_1
```

Particle specifier expressions

Nested expressions (only basic particles, no decay specs).

⟨*Particle specifiers: execute tests*⟩+≡

```
call test (particle_specifiers_2, "particle_specifiers_2", &
     "Particle specifier expressions", &
     u, results)
```

⟨*Particle specifiers: test declarations*⟩+≡

```
public :: particle_specifiers_2
```

⟨*Particle specifiers: tests*⟩+≡

```
subroutine particle_specifiers_2 (u)
  integer, intent(in) :: u
  type(prt_spec_t) :: a, b, c, d, e, f
  type(prt_expr_t) :: pe1, pe2, pe3
  type(prt_expr_t) :: pe4, pe5, pe6, pe7, pe8, pe9
  integer :: i
  type(prt_spec_t), dimension(:), allocatable :: pa

  write (u, "(A)")  "* Test output: particle_specifiers_2"
  write (u, "(A)")  "*   Purpose: Create and display particle expressions"
  write (u, "(A)")

  write (u, "(A)")  "* Basic expressions"
  write (u, *)

  a = new_prt_spec (var_str ("a"))
  b = new_prt_spec (var_str ("b"))
  c = new_prt_spec (var_str ("c"))
  d = new_prt_spec (var_str ("d"))
  e = new_prt_spec (var_str ("e"))
  f = new_prt_spec (var_str ("f"))

  call pe1%init_spec (a)
  write (u, "(A)")  char (pe1%to_string ())

  call pe2%init_sum (2)
  select type (x => pe2%x)
  type is (prt_spec_sum_t)
     call x%expr(1)%init_spec (a)
     call x%expr(2)%init_spec (b)
  end select
  write (u, "(A)")  char (pe2%to_string ())

  call pe3%init_list (2)
  select type (x => pe3%x)
  type is (prt_spec_list_t)
     call x%expr(1)%init_spec (a)
     call x%expr(2)%init_spec (b)
  end select
  write (u, "(A)")  char (pe3%to_string ())

  write (u, *)
  write (u, "(A)")  "* Nested expressions"
```

```
write (u, *)

call pe4%init_list (2)
select type (x => pe4%x)
type is (prt_spec_list_t)
   call x%expr(1)%init_sum (2)
   select type (y => x%expr(1)%x)
   type is (prt_spec_sum_t)
      call y%expr(1)%init_spec (a)
      call y%expr(2)%init_spec (b)
   end select
   call x%expr(2)%init_spec (c)
end select
write (u, "(A)")  char (pe4%to_string ())

call pe5%init_list (2)
select type (x => pe5%x)
type is (prt_spec_list_t)
   call x%expr(1)%init_list (2)
   select type (y => x%expr(1)%x)
   type is (prt_spec_list_t)
      call y%expr(1)%init_spec (a)
      call y%expr(2)%init_spec (b)
   end select
   call x%expr(2)%init_spec (c)
end select
write (u, "(A)")  char (pe5%to_string ())

call pe6%init_sum (2)
select type (x => pe6%x)
type is (prt_spec_sum_t)
   call x%expr(1)%init_spec (a)
   call x%expr(2)%init_sum (2)
   select type (y => x%expr(2)%x)
   type is (prt_spec_sum_t)
      call y%expr(1)%init_spec (b)
      call y%expr(2)%init_spec (c)
   end select
end select
write (u, "(A)")  char (pe6%to_string ())

call pe7%init_list (2)
select type (x => pe7%x)
type is (prt_spec_list_t)
   call x%expr(1)%init_sum (2)
   select type (y => x%expr(1)%x)
   type is (prt_spec_sum_t)
      call y%expr(1)%init_spec (a)
      call y%expr(2)%init_list (2)
      select type (z => y%expr(2)%x)
      type is (prt_spec_list_t)
         call z%expr(1)%init_spec (b)
         call z%expr(2)%init_spec (c)
      end select
```

```
   end select
   call x%expr(2)%init_spec (d)
end select
write (u, "(A)")  char (pe7%to_string ())

call pe8%init_sum (2)
select type (x => pe8%x)
type is (prt_spec_sum_t)
   call x%expr(1)%init_list (2)
   select type (y => x%expr(1)%x)
   type is (prt_spec_list_t)
      call y%expr(1)%init_spec (a)
      call y%expr(2)%init_spec (b)
   end select
   call x%expr(2)%init_list (2)
   select type (y => x%expr(2)%x)
   type is (prt_spec_list_t)
      call y%expr(1)%init_spec (c)
      call y%expr(2)%init_spec (d)
   end select
end select
write (u, "(A)")  char (pe8%to_string ())

call pe9%init_list (3)
select type (x => pe9%x)
type is (prt_spec_list_t)
   call x%expr(1)%init_sum (2)
   select type (y => x%expr(1)%x)
   type is (prt_spec_sum_t)
      call y%expr(1)%init_spec (a)
      call y%expr(2)%init_spec (b)
   end select
   call x%expr(2)%init_spec (c)
   call x%expr(3)%init_sum (3)
   select type (y => x%expr(3)%x)
   type is (prt_spec_sum_t)
      call y%expr(1)%init_spec (d)
      call y%expr(2)%init_spec (e)
      call y%expr(3)%init_spec (f)
   end select
end select
write (u, "(A)")  char (pe9%to_string ())

write (u, *)
write (u, "(A)")  "* Expand as sum"
write (u, *)

call pe1%expand ()
write (u, "(A)")  char (pe1%to_string ())

call pe4%expand ()
write (u, "(A)")  char (pe4%to_string ())

call pe5%expand ()
```

```
  write (u, "(A)")  char (pe5%to_string ())

  call pe6%expand ()
  write (u, "(A)")  char (pe6%to_string ())

  call pe7%expand ()
  write (u, "(A)")  char (pe7%to_string ())

  call pe8%expand ()
  write (u, "(A)")  char (pe8%to_string ())

  call pe9%expand ()
  write (u, "(A)")  char (pe9%to_string ())

  write (u, *)
  write (u, "(A)")  "* Transform to arrays:"

  write (u, "(A)")  "* Atomic specifier"
  do i = 1, pe1%get_n_terms ()
     call pe1%term_to_array (pa, i)
     call prt_spec_write (pa, u)
  end do

  write (u, *)
  write (u, "(A)")  "* List"
  do i = 1, pe5%get_n_terms ()
     call pe5%term_to_array (pa, i)
     call prt_spec_write (pa, u)
  end do

  write (u, *)
  write (u, "(A)")  "* Sum of atoms"
  do i = 1, pe6%get_n_terms ()
     call pe6%term_to_array (pa, i)
     call prt_spec_write (pa, u)
  end do

  write (u, *)
  write (u, "(A)")  "* Sum of lists"
  do i = 1, pe9%get_n_terms ()
     call pe9%term_to_array (pa, i)
     call prt_spec_write (pa, u)
  end do

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: particle_specifiers_2"
end subroutine particle_specifiers_2
```

12.2 PDG arrays

For defining aliases, we introduce a special type which holds a set of (integer) PDG codes.

⟨pdg_arrays.f90⟩≡
⟨*File header*⟩

```
module pdg_arrays

  use io_units
  use sorting
  use physics_defs, only: UNDEFINED
```

⟨*Standard module head*⟩

⟨*PDG arrays: public*⟩

⟨*PDG arrays: types*⟩

⟨*PDG arrays: interfaces*⟩

```
contains
```

⟨*PDG arrays: procedures*⟩

```
end module pdg_arrays
```

12.2.1 Type definition

Using an allocatable array eliminates the need for initializer and/or finalizer.

⟨*PDG arrays: public*⟩≡

```
public :: pdg_array_t
```

⟨*PDG arrays: types*⟩≡

```
type :: pdg_array_t
   private
   integer, dimension(:), allocatable :: pdg
 contains
 ⟨PDG arrays: pdg array: TBP⟩
end type pdg_array_t
```

Output

⟨*PDG arrays: public*⟩+≡

```
public :: pdg_array_write
```

⟨*PDG arrays: pdg array: TBP*⟩≡

```
procedure :: write => pdg_array_write
```

⟨*PDG arrays: procedures*⟩≡

```
subroutine pdg_array_write (aval, unit)
  class(pdg_array_t), intent(in) :: aval
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(A)", advance="no")  "PDG("
  if (allocated (aval%pdg)) then
     do i = 1, size (aval%pdg)
        if (i > 1)  write (u, "(A)", advance="no")  ", "
```

```
        write (u, "(I0)", advance="no")  aval%pdg(i)
      end do
    end if
    write (u, "(A)", advance="no")  ")"
  end subroutine pdg_array_write
```

⟨*PDG arrays: public*⟩+≡

```
  public :: pdg_array_write_set
```

⟨*PDG arrays: procedures*⟩+≡

```
  subroutine pdg_array_write_set (aval, unit)
    type(pdg_array_t), intent(in), dimension(:) :: aval
    integer, intent(in), optional :: unit
    integer :: i
    do i = 1, size (aval)
       call aval(i)%write (unit)
       print *, ''
    end do
  end subroutine pdg_array_write_set
```

12.2.2 Basic operations

Assignment. We define assignment from and to an integer array. Note that the integer array, if it is the l.h.s., must be declared allocatable by the caller.

⟨*PDG arrays: public*⟩+≡

```
  public :: assignment(=)
```

⟨*PDG arrays: interfaces*⟩≡

```
  interface assignment(=)
     module procedure pdg_array_from_int_array
     module procedure pdg_array_from_int
     module procedure int_array_from_pdg_array
  end interface
```

⟨*PDG arrays: procedures*⟩+≡

```
  subroutine pdg_array_from_int_array (aval, iarray)
    type(pdg_array_t), intent(out) :: aval
    integer, dimension(:), intent(in) :: iarray
    allocate (aval%pdg (size (iarray)))
    aval%pdg = iarray
  end subroutine pdg_array_from_int_array

  elemental subroutine pdg_array_from_int (aval, int)
    type(pdg_array_t), intent(out) :: aval
    integer, intent(in) :: int
    allocate (aval%pdg (1))
    aval%pdg = int
  end subroutine pdg_array_from_int

  subroutine int_array_from_pdg_array (iarray, aval)
    integer, dimension(:), allocatable, intent(out) :: iarray
    type(pdg_array_t), intent(in) :: aval
```

```
    if (allocated (aval%pdg)) then
       allocate (iarray (size (aval%pdg)))
       iarray = aval%pdg
    else
       allocate (iarray (0))
    end if
  end subroutine int_array_from_pdg_array
```

Allocate space for a PDG array

⟨*PDG arrays: public*⟩+≡

```
  public :: pdg_array_init
```

⟨*PDG arrays: procedures*⟩+≡

```
   subroutine pdg_array_init (aval, n_elements)
     type(pdg_array_t), intent(inout) :: aval
     integer, intent(in) :: n_elements
     allocate(aval%pdg(n_elements))
   end subroutine pdg_array_init
```

Deallocate a previously allocated pdg array

⟨*PDG arrays: public*⟩+≡

```
  public :: pdg_array_delete
```

⟨*PDG arrays: procedures*⟩+≡

```
  subroutine pdg_array_delete (aval)
    type(pdg_array_t), intent(inout) :: aval
    if (allocated (aval%pdg)) deallocate (aval%pdg)
  end subroutine pdg_array_delete
```

Merge two pdg arrays, i.e. append a particle string to another leaving out doublettes

⟨*PDG arrays: public*⟩+≡

```
  public :: pdg_array_merge
```

⟨*PDG arrays: procedures*⟩+≡

```
  subroutine pdg_array_merge (aval1, aval2)
    type(pdg_array_t), intent(inout) :: aval1
    type(pdg_array_t), intent(in) :: aval2
    type(pdg_array_t) :: aval
    if (allocated (aval1%pdg) .and. allocated (aval2%pdg)) then
      if (.not. any (aval1%pdg == aval2%pdg)) aval = aval1 // aval2
    else if (allocated (aval1%pdg)) then
      aval = aval1
    else if (allocated (aval2%pdg)) then
      aval = aval2
    end if
    call pdg_array_delete (aval1)
    aval1 = aval%pdg
  end subroutine pdg_array_merge
```

Length of the array.

⟨*PDG arrays: public*⟩+≡

```
  public :: pdg_array_get_length
```

⟨*PDG arrays: pdg array: TBP*⟩+≡
```
procedure :: get_length => pdg_array_get_length
```

⟨*PDG arrays: procedures*⟩+≡
```
elemental function pdg_array_get_length (aval) result (n)
  class(pdg_array_t), intent(in) :: aval
  integer :: n
  if (allocated (aval%pdg)) then
     n = size (aval%pdg)
  else
     n = 0
  end if
end function pdg_array_get_length
```

Return the element with index i.

⟨*PDG arrays: public*⟩+≡
```
public :: pdg_array_get
```

⟨*PDG arrays: pdg array: TBP*⟩+≡
```
procedure :: get => pdg_array_get
```

⟨*PDG arrays: procedures*⟩+≡
```
elemental function pdg_array_get (aval, i) result (pdg)
  class(pdg_array_t), intent(in) :: aval
  integer, intent(in), optional :: i
  integer :: pdg
  if (present (i)) then
     pdg = aval%pdg(i)
  else
     pdg = aval%pdg(1)
  end if
end function pdg_array_get
```

Explicitly set the element with index i.

⟨*PDG arrays: pdg array: TBP*⟩+≡
```
procedure :: set => pdg_array_set
```

⟨*PDG arrays: procedures*⟩+≡
```
subroutine pdg_array_set (aval, i, pdg)
  class(pdg_array_t), intent(inout) :: aval
  integer, intent(in) :: i
  integer, intent(in) :: pdg
  aval%pdg(i) = pdg
end subroutine pdg_array_set
```

Replace element with index `i` by a new array of elements.

⟨*PDG arrays: public*⟩+≡
```
public :: pdg_array_replace
```

⟨*PDG arrays: pdg array: TBP*⟩+≡
```
procedure :: replace => pdg_array_replace
```

⟨*PDG arrays: procedures*⟩+≡

```
function pdg_array_replace (aval, i, pdg_new) result (aval_new)
  class(pdg_array_t), intent(in) :: aval
  integer, intent(in) :: i
  integer, dimension(:), intent(in) :: pdg_new
  type(pdg_array_t) :: aval_new
  integer :: n, l
  n = size (aval%pdg)
  l = size (pdg_new)
  allocate (aval_new%pdg (n + l - 1))
  aval_new%pdg(:i-1) = aval%pdg(:i-1)
  aval_new%pdg(i:i+l-1) = pdg_new
  aval_new%pdg(i+l:) = aval%pdg(i+1:)
end function pdg_array_replace
```

Concatenate two PDG arrays

⟨*PDG arrays: public*⟩+≡

```
public :: operator(//)
```

⟨*PDG arrays: interfaces*⟩+≡

```
interface operator(//)
   module procedure concat_pdg_arrays
end interface
```

⟨*PDG arrays: procedures*⟩+≡

```
function concat_pdg_arrays (aval1, aval2) result (aval)
  type(pdg_array_t) :: aval
  type(pdg_array_t), intent(in) :: aval1, aval2
  integer :: n1, n2
  if (allocated (aval1%pdg) .and. allocated (aval2%pdg)) then
     n1 = size (aval1%pdg)
     n2 = size (aval2%pdg)
     allocate (aval%pdg (n1 + n2))
     aval%pdg(:n1) = aval1%pdg
     aval%pdg(n1+1:) = aval2%pdg
  else if (allocated (aval1%pdg)) then
     aval = aval1
  else if (allocated (aval2%pdg)) then
     aval = aval2
  end if
end function concat_pdg_arrays
```

12.2.3 Matching

A PDG array matches a given PDG code if the code is present within the array. If either one is zero (UNDEFINED), the match also succeeds.

⟨*PDG arrays: public*⟩+≡

```
public :: operator(.match.)
```

⟨*PDG arrays: interfaces*⟩+≡

```
interface operator(.match.)
   module procedure pdg_array_match_integer
```

```
   module procedure pdg_array_match_pdg_array
end interface
```

Match a single code against the array.

⟨*PDG arrays: procedures*⟩+≡

```
elemental function pdg_array_match_integer (aval, pdg) result (flag)
  logical :: flag
  type(pdg_array_t), intent(in) :: aval
  integer, intent(in) :: pdg
  if (allocated (aval%pdg)) then
     flag = pdg == UNDEFINED &
          .or. any (aval%pdg == UNDEFINED) &
          .or. any (aval%pdg == pdg)
  else
     flag = .false.
  end if
end function pdg_array_match_integer
```

Check if the pdg-number corresponds to a quark

⟨*PDG arrays: public*⟩+≡

```
public :: is_quark
```

⟨*PDG arrays: procedures*⟩+≡

```
elemental function is_quark (pdg_nr)
  logical :: is_quark
  integer, intent(in) :: pdg_nr
  if (abs (pdg_nr) >= 1 .and. abs (pdg_nr) <= 6) then
     is_quark = .true.
  else
     is_quark = .false.
  end if
end function is_quark
```

Check if pdg-number corresponds to a gluon

⟨*PDG arrays: public*⟩+≡

```
public :: is_gluon
```

⟨*PDG arrays: procedures*⟩+≡

```
elemental function is_gluon (pdg_nr)
  logical :: is_gluon
  integer, intent(in) :: pdg_nr
  if (pdg_nr == 21) then
     is_gluon = .true.
  else
     is_gluon = .false.
  end if
end function is_gluon
```

Check if pdg-number corresponds to a colored particle

⟨*PDG arrays: public*⟩+≡

```
public :: is_colored
```

⟨*PDG arrays: procedures*⟩+≡

```
elemental function is_colored (pdg_nr)
  logical :: is_colored
  integer, intent(in) :: pdg_nr
  is_colored = is_quark (pdg_nr) .or. is_gluon (pdg_nr)
end function is_colored
```

⟨*PDG arrays: public*⟩+≡

```
public :: is_lepton
```

⟨*PDG arrays: procedures*⟩+≡

```
elemental function is_lepton (pdg_nr)
  logical :: is_lepton
  integer, intent(in) :: pdg_nr
  if (abs (pdg_nr) >= 11 .and. abs (pdg_nr) <= 16) then
    is_lepton = .true.
  else
    is_lepton = .false.
  end if
end function is_lepton
```

⟨*PDG arrays: public*⟩+≡

```
public :: is_fermion
```

⟨*PDG arrays: procedures*⟩+≡

```
elemental function is_fermion (pdg_nr)
  logical :: is_fermion
  integer, intent(in) :: pdg_nr
  is_fermion = is_lepton(pdg_nr) .or. is_quark(pdg_nr)
end function is_fermion
```

Check if the pdg-number corresponds to a massless vector boson

⟨*PDG arrays: public*⟩+≡

```
public :: is_massless_vector
```

⟨*PDG arrays: procedures*⟩+≡

```
function is_massless_vector (pdg_nr) result (res)
  integer, intent(in) :: pdg_nr
  logical :: res
  if (pdg_nr == 21 .or. pdg_nr == 22) then
    res = .true.
  else
    res = .false.
  end if
end function is_massless_vector
```

Check if pdg-number corresponds to a massive vector boson

⟨*PDG arrays: public*⟩+≡

```
public :: is_massive_vector
```

⟨*PDG arrays: procedures*⟩+≡

```
function is_massive_vector (pdg_nr) result (res)
  integer, intent(in) :: pdg_nr
  logical :: res
  if (pdg_nr == 23 .or. pdg_nr == 24) then
    res = .true.
  else
    res = .false.
  end if
end function is_massive_vector
```

Check if particle is strongly interacting

⟨*PDG arrays: pdg array: TBP*⟩+≡

```
procedure :: has_colored_particles => pdg_array_has_colored_particles
```

⟨*PDG arrays: procedures*⟩+≡

```
function pdg_array_has_colored_particles (pdg) result (colored)
  class(pdg_array_t), intent(in) :: pdg
  logical :: colored
  integer :: i, pdg_nr
  colored = .false.
  do i = 1, size (pdg%pdg)
    pdg_nr = pdg%pdg(i)
    if (is_quark (pdg_nr) .or. is_gluon (pdg_nr)) then
       colored = .true.
       exit
    end if
  end do
end function pdg_array_has_colored_particles
```

Match two arrays. Succeeds if any pair of entries matches.

⟨*PDG arrays: procedures*⟩+≡

```
function pdg_array_match_pdg_array (aval1, aval2) result (flag)
  logical :: flag
  type(pdg_array_t), intent(in) :: aval1, aval2
  if (allocated (aval1%pdg) .and. allocated (aval2%pdg)) then
     flag = any (aval1 .match. aval2%pdg)
  else
     flag = .false.
  end if
end function pdg_array_match_pdg_array
```

Comparison. Here, we take the PDG arrays as-is, assuming that they are sorted.

The ordering is a bit odd: first, we look only at the absolute values of the PDG codes. If they all match, the particle comes before the antiparticle, scanning from left to right.

⟨*PDG arrays: public*⟩+≡

```
public :: operator(<)
public :: operator(>)
public :: operator(<=)
public :: operator(>=)
public :: operator(==)
public :: operator(/=)
```

⟨*PDG arrays: interfaces*⟩+≡

```
interface operator(<)
   module procedure pdg_array_lt
end interface
interface operator(>)
   module procedure pdg_array_gt
end interface
interface operator(<=)
   module procedure pdg_array_le
end interface
interface operator(>=)
   module procedure pdg_array_ge
end interface
interface operator(==)
   module procedure pdg_array_eq
end interface
interface operator(/=)
   module procedure pdg_array_ne
end interface
```

⟨*PDG arrays: procedures*⟩+≡

```
elemental function pdg_array_lt (aval1, aval2) result (flag)
  type(pdg_array_t), intent(in) :: aval1, aval2
  logical :: flag
  integer :: i
  if (size (aval1%pdg) /= size (aval2%pdg)) then
     flag = size (aval1%pdg) < size (aval2%pdg)
  else
     do i = 1, size (aval1%pdg)
        if (abs (aval1%pdg(i)) /= abs (aval2%pdg(i))) then
           flag = abs (aval1%pdg(i)) < abs (aval2%pdg(i))
           return
        end if
     end do
     do i = 1, size (aval1%pdg)
        if (aval1%pdg(i) /= aval2%pdg(i)) then
           flag = aval1%pdg(i) > aval2%pdg(i)
           return
        end if
     end do
     flag = .false.
  end if
end function pdg_array_lt

elemental function pdg_array_gt (aval1, aval2) result (flag)
  type(pdg_array_t), intent(in) :: aval1, aval2
  logical :: flag
  flag = .not. (aval1 < aval2 .or. aval1 == aval2)
end function pdg_array_gt

elemental function pdg_array_le (aval1, aval2) result (flag)
  type(pdg_array_t), intent(in) :: aval1, aval2
  logical :: flag
  flag = aval1 < aval2 .or. aval1 == aval2
```

```
end function pdg_array_le

elemental function pdg_array_ge (aval1, aval2) result (flag)
  type(pdg_array_t), intent(in) :: aval1, aval2
  logical :: flag
  flag = .not. (aval1 < aval2)
end function pdg_array_ge

elemental function pdg_array_eq (aval1, aval2) result (flag)
  type(pdg_array_t), intent(in) :: aval1, aval2
  logical :: flag
  if (size (aval1%pdg) /= size (aval2%pdg)) then
     flag = .false.
  else
     flag = all (aval1%pdg == aval2%pdg)
  end if
end function pdg_array_eq

elemental function pdg_array_ne (aval1, aval2) result (flag)
  type(pdg_array_t), intent(in) :: aval1, aval2
  logical :: flag
  flag = .not. (aval1 == aval2)
end function pdg_array_ne
```

Equivalence. Two PDG arrays are equivalent if either one contains `UNDEFINED` or if each element of array 1 is present in array 2, and vice versa.

⟨*PDG arrays: public*⟩+≡

```
public :: operator(.eqv.)
public :: operator(.neqv.)
```

⟨*PDG arrays: interfaces*⟩+≡

```
interface operator(.eqv.)
   module procedure pdg_array_equivalent
end interface
interface operator(.neqv.)
   module procedure pdg_array_inequivalent
end interface
```

⟨*PDG arrays: procedures*⟩+≡

```
elemental function pdg_array_equivalent (aval1, aval2) result (eq)
  logical :: eq
  type(pdg_array_t), intent(in) :: aval1, aval2
  logical, dimension(:), allocatable :: match1, match2
  integer :: i
  if (allocated (aval1%pdg) .and. allocated (aval2%pdg)) then
     eq = any (aval1%pdg == UNDEFINED) &
          .or. any (aval2%pdg == UNDEFINED)
     if (.not. eq) then
        allocate (match1 (size (aval1%pdg)))
        allocate (match2 (size (aval2%pdg)))
        match1 = .false.
        match2 = .false.
        do i = 1, size (aval1%pdg)
           match2 = match2 .or. aval1%pdg(i) == aval2%pdg
```

```
        end do
        do i = 1, size (aval2%pdg)
           match1 = match1 .or. aval2%pdg(i) == aval1%pdg
        end do
        eq = all (match1) .and. all (match2)
     end if
  else
     eq = .false.
  end if
end function pdg_array_equivalent

elemental function pdg_array_inequivalent (aval1, aval2) result (neq)
  logical :: neq
  type(pdg_array_t), intent(in) :: aval1, aval2
  neq = .not. pdg_array_equivalent (aval1, aval2)
end function pdg_array_inequivalent
```

12.2.4 Sorting

Sort a PDG array by absolute value, particle before antiparticle. After sorting, we eliminate double entries.

⟨*PDG arrays: public*⟩+≡

```
public :: sort_abs
```

⟨*PDG arrays: interfaces*⟩+≡

```
interface sort_abs
   module procedure pdg_array_sort_abs
end interface
```

⟨*PDG arrays: pdg array: TBP*⟩+≡

```
procedure :: sort_abs => pdg_array_sort_abs
```

⟨*PDG arrays: procedures*⟩+≡

```
function pdg_array_sort_abs (aval1, unique) result (aval2)
  class(pdg_array_t), intent(in) :: aval1
  logical, intent(in), optional :: unique
  type(pdg_array_t) :: aval2
  integer, dimension(:), allocatable :: tmp
  logical, dimension(:), allocatable :: mask
  integer :: i, n
  logical :: uni
  uni = .false.;  if (present (unique))  uni = unique
  n = size (aval1%pdg)
  if (uni) then
     allocate (tmp (n), mask(n))
     tmp = sort_abs (aval1%pdg)
     mask(1) = .true.
     do i = 2, n
        mask(i) = tmp(i) /= tmp(i-1)
     end do
     allocate (aval2%pdg (count (mask)))
     aval2%pdg = pack (tmp, mask)
  else
```

```
      allocate (aval2%pdg (n))
      aval2%pdg = sort_abs (aval1%pdg)
    end if
  end function pdg_array_sort_abs
```

⟨*PDG arrays: pdg array: TBP*⟩+≡

```
  procedure :: intersect => pdg_array_intersect
```

⟨*PDG arrays: procedures*⟩+≡

```
  function pdg_array_intersect (aval1, match) result (aval2)
   class(pdg_array_t), intent(in) :: aval1
   integer, dimension(:) :: match
   type(pdg_array_t) :: aval2
   integer, dimension(:), allocatable :: isec
   integer :: i
   isec = pack (aval1%pdg, [(any(aval1%pdg(i) == match), i=1,size(aval1%pdg))])
   aval2 = isec
  end function pdg_array_intersect
```

⟨*PDG arrays: pdg array: TBP*⟩+≡

```
  procedure :: search_for_particle => pdg_array_search_for_particle
```

⟨*PDG arrays: procedures*⟩+≡

```
  function pdg_array_search_for_particle (pdg, i_part) result (found)
    class(pdg_array_t), intent(in) :: pdg
    integer, intent(in) :: i_part
    logical :: found
    found = any (pdg%pdg == i_part)
  end function pdg_array_search_for_particle
```

⟨*PDG arrays: pdg array: TBP*⟩+≡

```
  procedure :: invert => pdg_array_invert
```

⟨*PDG arrays: procedures*⟩+≡

```
  function pdg_array_invert (pdg) result (pdg_inverse)
    class(pdg_array_t), intent(in) :: pdg
    type(pdg_array_t) :: pdg_inverse
    integer :: i, n
    n = size (pdg%pdg)
    allocate (pdg_inverse%pdg (n))
    do i = 1, n
       select case (pdg%pdg(i))
       case (21, 22, 23, 25)
          pdg_inverse%pdg(i) = pdg%pdg(i)
       case default
          pdg_inverse%pdg(i) = -pdg%pdg(i)
       end select
    end do
  end function pdg_array_invert
```

12.2.5 PDG array list

A PDG array list, or PDG list, is an array of PDG-array objects with some convenience methods.

⟨*PDG arrays: public*⟩+≡
```
public :: pdg_list_t
```

⟨*PDG arrays: types*⟩+≡
```
type :: pdg_list_t
   type(pdg_array_t), dimension(:), allocatable :: a
 contains
 ⟨PDG arrays: pdg list: TBP⟩
end type pdg_list_t
```

Output, as a comma-separated list without advancing I/O.

⟨*PDG arrays: pdg list: TBP*⟩≡
```
procedure :: write => pdg_list_write
```

⟨*PDG arrays: procedures*⟩+≡
```
subroutine pdg_list_write (object, unit)
  class(pdg_list_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  if (allocated (object%a)) then
     do i = 1, size (object%a)
        if (i > 1)  write (u, "(A)", advance="no")  ", "
        call object%a(i)%write (u)
     end do
  end if
end subroutine pdg_list_write
```

Initialize for a certain size. The entries are initially empty PDG arrays.

⟨*PDG arrays: pdg list: TBP*⟩+≡
```
generic :: init => pdg_list_init_size
procedure, private :: pdg_list_init_size
```

⟨*PDG arrays: procedures*⟩+≡
```
subroutine pdg_list_init_size (pl, n)
  class(pdg_list_t), intent(out) :: pl
  integer, intent(in) :: n
  allocate (pl%a (n))
end subroutine pdg_list_init_size
```

Initialize with a definite array of PDG codes. That is, each entry in the list becomes a single-particle PDG array.

⟨*PDG arrays: pdg list: TBP*⟩+≡
```
generic :: init => pdg_list_init_int_array
procedure, private :: pdg_list_init_int_array
```

⟨*PDG arrays: procedures*⟩+≡
```
subroutine pdg_list_init_int_array (pl, pdg)
  class(pdg_list_t), intent(out) :: pl
  integer, dimension(:), intent(in) :: pdg
```

```
    integer :: i
    allocate (pl%a (size (pdg)))
    do i = 1, size (pdg)
       pl%a(i) = pdg(i)
    end do
  end subroutine pdg_list_init_int_array
```

Set one of the entries. No bounds-check.

⟨*PDG arrays: pdg list: TBP*⟩+≡

```
  generic :: set => pdg_list_set_int
  generic :: set => pdg_list_set_int_array
  generic :: set => pdg_list_set_pdg_array
  procedure, private :: pdg_list_set_int
  procedure, private :: pdg_list_set_int_array
  procedure, private :: pdg_list_set_pdg_array
```

⟨*PDG arrays: procedures*⟩+≡

```
  subroutine pdg_list_set_int (pl, i, pdg)
    class(pdg_list_t), intent(inout) :: pl
    integer, intent(in) :: i
    integer, intent(in) :: pdg
    pl%a(i) = pdg
  end subroutine pdg_list_set_int

  subroutine pdg_list_set_int_array (pl, i, pdg)
    class(pdg_list_t), intent(inout) :: pl
    integer, intent(in) :: i
    integer, dimension(:), intent(in) :: pdg
    pl%a(i) = pdg
  end subroutine pdg_list_set_int_array

  subroutine pdg_list_set_pdg_array (pl, i, pa)
    class(pdg_list_t), intent(inout) :: pl
    integer, intent(in) :: i
    type(pdg_array_t), intent(in) :: pa
    pl%a(i) = pa
  end subroutine pdg_list_set_pdg_array
```

Array size, not the length of individual entries

⟨*PDG arrays: pdg list: TBP*⟩+≡

```
  procedure :: get_size => pdg_list_get_size
```

⟨*PDG arrays: procedures*⟩+≡

```
  function pdg_list_get_size (pl) result (n)
    class(pdg_list_t), intent(in) :: pl
    integer :: n
    if (allocated (pl%a)) then
       n = size (pl%a)
    else
       n = 0
    end if
  end function pdg_list_get_size
```

Return an entry, as a PDG array.

⟨*PDG arrays: pdg list: TBP*⟩+≡
```
procedure :: get => pdg_list_get
```

⟨*PDG arrays: procedures*⟩+≡
```
function pdg_list_get (pl, i) result (pa)
  type(pdg_array_t) :: pa
  class(pdg_list_t), intent(in) :: pl
  integer, intent(in) :: i
  pa = pl%a(i)
end function pdg_list_get
```

Check if the list entries are all either mutually disjoint or identical. The individual entries (PDG arrays) should already be sorted, so we can test for equality.

⟨*PDG arrays: pdg list: TBP*⟩+≡
```
procedure :: is_regular => pdg_list_is_regular
```

⟨*PDG arrays: procedures*⟩+≡
```
function pdg_list_is_regular (pl) result (flag)
  class(pdg_list_t), intent(in) :: pl
  logical :: flag
  integer :: i, j, s
  s = pl%get_size ()
  flag = .true.
  do i = 1, s
     do j = i + 1, s
        if (pl%a(i) .match. pl%a(j)) then
           if (pl%a(i) /= pl%a(j)) then
              flag = .false.
              return
           end if
        end if
     end do
  end do
end function pdg_list_is_regular
```

Sort the list. First, each entry gets sorted, including elimination of doublers. Then, we sort the list, using the first member of each PDG array as the marker. No removal of doublers at this stage.

If `n_in` is supplied, we do not reorder the first `n_in` particle entries.

⟨*PDG arrays: pdg list: TBP*⟩+≡
```
procedure :: sort_abs => pdg_list_sort_abs
```

⟨*PDG arrays: procedures*⟩+≡
```
function pdg_list_sort_abs (pl, n_in) result (pl_sorted)
  class(pdg_list_t), intent(in) :: pl
  integer, intent(in), optional :: n_in
  type(pdg_list_t) :: pl_sorted
  type(pdg_array_t), dimension(:), allocatable :: pa
  integer, dimension(:), allocatable :: pdg, map
  integer :: i, n0
  call pl_sorted%init (pl%get_size ())
  if (allocated (pl%a)) then
     allocate (pa (size (pl%a)))
```

```
      do i = 1, size (pl%a)
         pa(i) = pl%a(i)%sort_abs (unique = .true.)
      end do
      allocate (pdg (size (pa)), source = 0)
      do i = 1, size (pa)
         if (allocated (pa(i)%pdg)) then
            if (size (pa(i)%pdg) > 0) then
               pdg(i) = pa(i)%pdg(1)
            end if
         end if
      end do
      if (present (n_in)) then
         n0 = n_in
      else
         n0 = 0
      end if
      allocate (map (size (pdg)))
      map(:n0) = [(i, i = 1, n0)]
      map(n0+1:) = n0 + order_abs (pdg(n0+1:))
      do i = 1, size (pa)
         call pl_sorted%set (i, pa(map(i)))
      end do
    end if
  end function pdg_list_sort_abs
```

Compare sorted lists: equality. The result is undefined if some entries are not allocated.

⟨*PDG arrays: pdg list: TBP*⟩+≡

```
  generic :: operator (==) => pdg_list_eq
  procedure, private :: pdg_list_eq
```

⟨*PDG arrays: procedures*⟩+≡

```
  function pdg_list_eq (pl1, pl2) result (flag)
    class(pdg_list_t), intent(in) :: pl1, pl2
    logical :: flag
    integer :: i
    flag = .false.
    if (allocated (pl1%a) .and. allocated (pl2%a)) then
       if (size (pl1%a) == size (pl2%a)) then
          do i = 1, size (pl1%a)
             associate (a1 => pl1%a(i), a2 => pl2%a(i))
               if (allocated (a1%pdg) .and. allocated (a2%pdg)) then
                  if (size (a1%pdg) == size (a2%pdg)) then
                     if (size (a1%pdg) > 0) then
                        if (a1%pdg(1) /= a2%pdg(1)) return
                     end if
                  else
                     return
                  end if
               else
                  return
               end if
             end associate
          end do
```

```
          flag = .true.
       end if
    end if
  end function pdg_list_eq
```

Compare sorted lists. The result is undefined if some entries are not allocated.

The ordering is quite complicated. First, a shorter list comes before a longer list. Comparing entry by entry, a shorter entry comes first. Next, we check the first PDG code within corresponding entries. This is compared by absolute value. If equal, particle comes before antiparticle. Finally, if all is equal, the result is false.

⟨*PDG arrays: pdg list: TBP*⟩+≡

```
  generic :: operator (<) => pdg_list_lt
  procedure, private :: pdg_list_lt
```

⟨*PDG arrays: procedures*⟩+≡

```
  function pdg_list_lt (pl1, pl2) result (flag)
    class(pdg_list_t), intent(in) :: pl1, pl2
    logical :: flag
    integer :: i
    flag = .false.
    if (allocated (pl1%a) .and. allocated (pl2%a)) then
       if (size (pl1%a) < size (pl2%a)) then
          flag = .true.;  return
       else if (size (pl1%a) > size (pl2%a)) then
          return
       else
          do i = 1, size (pl1%a)
             associate (a1 => pl1%a(i), a2 => pl2%a(i))
               if (allocated (a1%pdg) .and. allocated (a2%pdg)) then
                  if (size (a1%pdg) < size (a2%pdg)) then
                     flag = .true.;  return
                  else if (size (a1%pdg) > size (a2%pdg)) then
                     return
                  else
                     if (size (a1%pdg) > 0) then
                        if (abs (a1%pdg(1)) < abs (a2%pdg(1))) then
                           flag = .true.;  return
                        else if (abs (a1%pdg(1)) > abs (a2%pdg(1))) then
                           return
                        else if (a1%pdg(1) > 0 .and. a2%pdg(1) < 0) then
                           flag = .true.;  return
                        else if (a1%pdg(1) < 0 .and. a2%pdg(1) > 0) then
                           return
                        end if
                     end if
                  end if
               else
                  return
               end if
             end associate
          end do
          flag = .false.
       end if
```

```
    end if
  end function pdg_list_lt
```

Replace an entry. In the result, the entry `#i` is replaced by the contents of the second argument. The result is not sorted.

If `n_in` is also set and `i` is less or equal to `n_in`, replace `#i` only by the first entry of `pl_insert`, and insert the remainder after entry `n_in`.

⟨*PDG arrays: pdg list: TBP*⟩+≡

```
  procedure :: replace => pdg_list_replace
```

⟨*PDG arrays: procedures*⟩+≡

```
  function pdg_list_replace (pl, i, pl_insert, n_in) result (pl_out)
    class(pdg_list_t), intent(in) :: pl
    integer, intent(in) :: i
    class(pdg_list_t), intent(in) :: pl_insert
    integer, intent(in), optional :: n_in
    type(pdg_list_t) :: pl_out
    integer :: n, n_insert, n_out, k
    n = pl%get_size ()
    n_insert = pl_insert%get_size ()
    n_out = n + n_insert - 1
    call pl_out%init (n_out)
!    if (allocated (pl%a)) then
       do k = 1, i - 1
          pl_out%a(k) = pl%a(k)
       end do
!    end if
    if (present (n_in)) then
       pl_out%a(i) = pl_insert%a(1)
       do k = i + 1, n_in
          pl_out%a(k) = pl%a(k)
       end do
       do k = 1, n_insert - 1
          pl_out%a(n_in+k) = pl_insert%a(1+k)
       end do
       do k = 1, n - n_in
          pl_out%a(n_in+k+n_insert-1) = pl%a(n_in+k)
       end do
    else
!       if (allocated (pl_insert%a)) then
          do k = 1, n_insert
             pl_out%a(i-1+k) = pl_insert%a(k)
          end do
!       end if
!       if (allocated (pl%a)) then
          do k = 1, n - i
             pl_out%a(i+n_insert-1+k) = pl%a(i+k)
          end do
       end if
!    end if
  end function pdg_list_replace
```

Replace the entries of `pl` by the matching entries of `pl_match`, one by one. This

is done in-place. If there is no match, return failure.

⟨*PDG arrays: pdg list: TBP*⟩+≡

```
procedure :: match_replace => pdg_list_match_replace
```

⟨*PDG arrays: procedures*⟩+≡

```
subroutine pdg_list_match_replace (pl, pl_match, success)
  class(pdg_list_t), intent(inout) :: pl
  class(pdg_list_t), intent(in) :: pl_match
  logical, intent(out) :: success
  integer :: i, j
  success = .true.
  SCAN_ENTRIES: do i = 1, size (pl%a)
     do j = 1, size (pl_match%a)
        if (pl%a(i) .match. pl_match%a(j)) then
           pl%a(i) = pl_match%a(j)
           cycle SCAN_ENTRIES
        end if
     end do
     success = .false.
     return
  end do SCAN_ENTRIES
end subroutine pdg_list_match_replace
```

Just check if a PDG array matches any entry in the PDG list. The second version returns the position of the match within the list. An optional mask indicates the list elements that should be checked.

⟨*PDG arrays: pdg list: TBP*⟩+≡

```
generic :: operator (.match.) => pdg_list_match_pdg_array
procedure, private :: pdg_list_match_pdg_array
procedure :: find_match => pdg_list_find_match_pdg_array
```

⟨*PDG arrays: procedures*⟩+≡

```
function pdg_list_match_pdg_array (pl, pa) result (flag)
  class(pdg_list_t), intent(in) :: pl
  type(pdg_array_t), intent(in) :: pa
  logical :: flag
  flag = pl%find_match (pa) /= 0
end function pdg_list_match_pdg_array

function pdg_list_find_match_pdg_array (pl, pa, mask) result (i)
  class(pdg_list_t), intent(in) :: pl
  type(pdg_array_t), intent(in) :: pa
  logical, dimension(:), intent(in), optional :: mask
  integer :: i
  do i = 1, size (pl%a)
     if (present (mask)) then
        if (.not. mask(i))  cycle
     end if
     if (pl%a(i) .match. pa)  return
  end do
  i = 0
end function pdg_list_find_match_pdg_array
```

Some old compilers have problems with allocatable arrays as intent(out) or as function result, so be conservative here:

⟨*PDG arrays: pdg list: TBP*⟩+≡

```
procedure :: create_pdg_array => pdg_list_create_pdg_array
```

⟨*PDG arrays: procedures*⟩+≡

```
subroutine pdg_list_create_pdg_array (pl, pdg)
  class(pdg_list_t), intent(in) :: pl
  type(pdg_array_t), dimension(:), intent(inout), allocatable :: pdg
  integer :: n_elements
  integer :: i
  associate (a => pl%a)
    n_elements = size (a)
    if (allocated (pdg))  deallocate (pdg)
    allocate (pdg (n_elements))
    do i = 1, n_elements
       pdg(i) = a(i)
    end do
  end associate
end subroutine pdg_list_create_pdg_array
```

⟨*PDG arrays: pdg list: TBP*⟩+≡

```
procedure :: create_antiparticles => pdg_list_create_antiparticles
```

⟨*PDG arrays: procedures*⟩+≡

```
subroutine pdg_list_create_antiparticles (pl, pl_anti, n_new_particles)
  class(pdg_list_t), intent(in) :: pl
  type(pdg_list_t), intent(out) :: pl_anti
  integer, intent(out) :: n_new_particles
  type(pdg_list_t) :: pl_inverse
  integer :: i, n
  integer :: n_identical
  logical, dimension(:), allocatable :: collect
  n = pl%get_size (); n_identical = 0
  allocate (collect (n)); collect = .true.
  call pl_inverse%init (n)
  do i = 1, n
     pl_inverse%a(i) = pl%a(i)%invert()
  end do
  do i = 1, n
     if (any (pl_inverse%a(i) == pl%a)) then
        collect(i) = .false.
        n_identical = n_identical + 1
     end if
  end do
  n_new_particles = n - n_identical
  if (n_new_particles > 0) then
     call pl_anti%init (n_new_particles)
     do i = 1, n
        if (collect (i)) pl_anti%a(i) = pl_inverse%a(i)
     end do
 end if
end subroutine pdg_list_create_antiparticles
```

⟨*PDG arrays: pdg list: TBP*⟩+≡

```
procedure :: search_for_particle => pdg_list_search_for_particle
```

⟨*PDG arrays: procedures*⟩+≡

```
function pdg_list_search_for_particle (pl, i_part) result (found)
  class(pdg_list_t), intent(in) :: pl
  integer, intent(in) :: i_part
  logical :: found
  integer :: i_pl
  do i_pl = 1, size (pl%a)
     found = pl%a(i_pl)%search_for_particle (i_part)
     if (found) return
  end do
end function pdg_list_search_for_particle
```

⟨*PDG arrays: pdg list: TBP*⟩+≡

```
procedure :: contains_colored_particles => pdg_list_contains_colored_particles
```

⟨*PDG arrays: procedures*⟩+≡

```
function pdg_list_contains_colored_particles (pl) result (colored)
  class(pdg_list_t), intent(in) :: pl
  logical :: colored
  integer :: i
  colored = .false.
  do i = 1, size (pl%a)
     if (pl%a(i)%has_colored_particles()) then
        colored = .true.
        exit
     end if
  end do
end function pdg_list_contains_colored_particles
```

12.2.6 Unit tests

Test module, followed by the corresponding implementation module.

⟨pdg_arrays_ut.f90⟩≡

```
⟨File header⟩

module pdg_arrays_ut
  use unit_tests
  use pdg_arrays_uti

⟨Standard module head⟩

⟨PDG arrays: public test⟩

contains

⟨PDG arrays: test driver⟩

end module pdg_arrays_ut
```

⟨pdg_arrays_uti.f90⟩≡

```
⟨File header⟩

module pdg_arrays_uti

  use pdg_arrays

⟨Standard module head⟩

⟨PDG arrays: test declarations⟩

contains

⟨PDG arrays: tests⟩

end module pdg_arrays_uti
```

API: driver for the unit tests below.

⟨*PDG arrays: public test*⟩≡

```
  public :: pdg_arrays_test
```

⟨*PDG arrays: test driver*⟩≡

```
  subroutine pdg_arrays_test (u, results)
    integer, intent(in) :: u
    type (test_results_t), intent(inout) :: results
  ⟨PDG arrays: execute tests⟩
  end subroutine pdg_arrays_test
```

Basic functionality.

⟨*PDG arrays: execute tests*⟩≡

```
  call test (pdg_arrays_1, "pdg_arrays_1", &
       "create and sort PDG array", &
       u, results)
```

⟨*PDG arrays: test declarations*⟩≡

```
  public :: pdg_arrays_1
```

⟨*PDG arrays: tests*⟩≡

```
  subroutine pdg_arrays_1 (u)
    integer, intent(in) :: u

    type(pdg_array_t) :: pa, pa1, pa2, pa3, pa4, pa5, pa6
    integer, dimension(:), allocatable :: pdg

    write (u, "(A)")  "* Test output: pdg_arrays_1"
    write (u, "(A)")  "*   Purpose: create and sort PDG arrays"
    write (u, "(A)")

    write (u, "(A)")  "* Assignment"
    write (u, "(A)")

    call pa%write (u)
    write (u, *)
    write (u, "(A,I0)")  "length = ", pa%get_length ()
    pdg = pa
    write (u, "(A,3(1x,I0))")  "contents = ", pdg
```

```
write (u, *)
pa = 1
call pa%write (u)
write (u, *)
write (u, "(A,I0)")  "length = ", pa%get_length ()
pdg = pa
write (u, "(A,3(1x,I0))")  "contents = ", pdg

write (u, *)
pa = [1, 2, 3]
call pa%write (u)
write (u, *)
write (u, "(A,I0)")  "length = ", pa%get_length ()
pdg = pa
write (u, "(A,3(1x,I0))")  "contents = ", pdg
write (u, "(A,I0)")  "element #2 = ", pa%get (2)

write (u, *)
write (u, "(A)")  "* Replace"
write (u, *)

pa = pa%replace (2, [-5, 5, -7])
call pa%write (u)
write (u, *)

write (u, *)
write (u, "(A)")  "* Sort"
write (u, *)

pa = [1, -7, 3, -5, 5, 3]
call pa%write (u)
write (u, *)
pa1 = pa%sort_abs ()
pa2 = pa%sort_abs (unique = .true.)
call pa1%write (u)
write (u, *)
call pa2%write (u)
write (u, *)

write (u, *)
write (u, "(A)")  "* Compare"
write (u, *)

pa1 = [1, 3]
pa2 = [1, 2, -2]
pa3 = [1, 2, 4]
pa4 = [1, 2, 4]
pa5 = [1, 2, -4]
pa6 = [1, 2, -3]

write (u, "(A,6(1x,L1))")  "< ", &
     pa1 < pa2, pa2 < pa3, pa3 < pa4, pa4 < pa5, pa5 < pa6, pa6 < pa1
write (u, "(A,6(1x,L1))")  "> ", &
```

```
        pa1 > pa2, pa2 > pa3, pa3 > pa4, pa4 > pa5, pa5 > pa6, pa6 > pa1
    write (u, "(A,6(1x,L1))")  "<=", &
        pa1 <= pa2, pa2 <= pa3, pa3 <= pa4, pa4 <= pa5, pa5 <= pa6, pa6 <= pa1
    write (u, "(A,6(1x,L1))")  ">=", &
        pa1 >= pa2, pa2 >= pa3, pa3 >= pa4, pa4 >= pa5, pa5 >= pa6, pa6 >= pa1
    write (u, "(A,6(1x,L1))")  "==", &
        pa1 == pa2, pa2 == pa3, pa3 == pa4, pa4 == pa5, pa5 == pa6, pa6 == pa1
    write (u, "(A,6(1x,L1))")  "/=", &
        pa1 /= pa2, pa2 /= pa3, pa3 /= pa4, pa4 /= pa5, pa5 /= pa6, pa6 /= pa1

    write (u, *)
    pa1 = [0]
    pa2 = [1, 2]
    pa3 = [1, -2]

    write (u, "(A,6(1x,L1))")  "eqv ", &
        pa1 .eqv. pa1, pa1 .eqv. pa2, &
        pa2 .eqv. pa2, pa2 .eqv. pa3

    write (u, "(A,6(1x,L1))")  "neqv", &
        pa1 .neqv. pa1, pa1 .neqv. pa2, &
        pa2 .neqv. pa2, pa2 .neqv. pa3


    write (u, *)
    write (u, "(A,6(1x,L1))")  "match", &
        pa1 .match. 0, pa1 .match. 1, &
        pa2 .match. 0, pa2 .match. 1, pa2 .match. 3

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: pdg_arrays_1"

  end subroutine pdg_arrays_1
```

PDG array list, i.e., arrays of arrays.

⟨*PDG arrays: execute tests*⟩+≡

```
  call test (pdg_arrays_2, "pdg_arrays_2", &
      "create and sort PDG lists", &
      u, results)
```

⟨*PDG arrays: test declarations*⟩+≡

```
  public :: pdg_arrays_2
```

⟨*PDG arrays: tests*⟩+≡

```
  subroutine pdg_arrays_2 (u)
    integer, intent(in) :: u

    type(pdg_array_t) :: pa
    type(pdg_list_t) :: pl, pl1

    write (u, "(A)")  "* Test output: pdg_arrays_2"
    write (u, "(A)")  "*   Purpose: create and sort PDG lists"
    write (u, "(A)")

    write (u, "(A)")  "* Assignment"
```

```
    write (u, "(A)")

    call pl%init (3)
    call pl%set (1, 42)
    call pl%set (2, [3, 2])
    pa = [5, -5]
    call pl%set (3, pa)
    call pl%write (u)
    write (u, *)
    write (u, "(A,I0)")  "size = ", pl%get_size ()

    write (u, "(A)")
    write (u, "(A)")  "* Sort"
    write (u, "(A)")

    pl = pl%sort_abs ()
    call pl%write (u)
    write (u, *)

    write (u, "(A)")
    write (u, "(A)")  "* Extract item #3"
    write (u, "(A)")

    pa = pl%get (3)
    call pa%write (u)
    write (u, *)

    write (u, "(A)")
    write (u, "(A)")  "* Replace item #3"
    write (u, "(A)")

    call pl1%init (2)
    call pl1%set (1, [2, 4])
    call pl1%set (2, -7)

    pl = pl%replace (3, pl1)
    call pl%write (u)
    write (u, *)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: pdg_arrays_2"

  end subroutine pdg_arrays_2
```

Check if a (sorted) PDG array lists is regular. The entries (PDG arrays) must not overlap, unless they are identical.

⟨*PDG arrays: execute tests*⟩+≡

```
  call test (pdg_arrays_3, "pdg_arrays_3", &
      "check PDG lists", &
      u, results)
```

⟨*PDG arrays: test declarations*⟩+≡

```
  public :: pdg_arrays_3
```

⟨*PDG arrays: tests*⟩+≡

```
subroutine pdg_arrays_3 (u)
  integer, intent(in) :: u

  type(pdg_list_t) :: pl

  write (u, "(A)")  "* Test output: pdg_arrays_3"
  write (u, "(A)")  "*   Purpose: check for regular PDG lists"
  write (u, "(A)")

  write (u, "(A)")  "* Regular list"
  write (u, "(A)")

  call pl%init (4)
  call pl%set (1, [1, 2])
  call pl%set (2, [1, 2])
  call pl%set (3, [5, -5])
  call pl%set (4, 42)
  call pl%write (u)
  write (u, *)
  write (u, "(L1)") pl%is_regular ()

  write (u, "(A)")
  write (u, "(A)")  "* Irregular list"
  write (u, "(A)")

  call pl%init (4)
  call pl%set (1, [1, 2])
  call pl%set (2, [1, 2])
  call pl%set (3, [2, 5, -5])
  call pl%set (4, 42)
  call pl%write (u)
  write (u, *)
  write (u, "(L1)") pl%is_regular ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: pdg_arrays_3"

end subroutine pdg_arrays_3
```

Compare PDG array lists. The lists must be regular, i.e., sorted and with non-overlapping (or identical) entries.

⟨*PDG arrays: execute tests*⟩+≡

```
call test (pdg_arrays_4, "pdg_arrays_4", &
     "compare PDG lists", &
     u, results)
```

⟨*PDG arrays: test declarations*⟩+≡

```
public :: pdg_arrays_4
```

⟨*PDG arrays: tests*⟩+≡

```
subroutine pdg_arrays_4 (u)
  integer, intent(in) :: u

  type(pdg_list_t) :: pl1, pl2, pl3
```

```
write (u, "(A)")  "* Test output: pdg_arrays_4"
write (u, "(A)")  "*   Purpose: check for regular PDG lists"
write (u, "(A)")

write (u, "(A)")  "* Create lists"
write (u, "(A)")

call pl1%init (4)
call pl1%set (1, [1, 2])
call pl1%set (2, [1, 2])
call pl1%set (3, [5, -5])
call pl1%set (4, 42)
write (u, "(I1,1x)", advance = "no")  1
call pl1%write (u)
write (u, *)

call pl2%init (2)
call pl2%set (1, 3)
call pl2%set (2, [5, -5])
write (u, "(I1,1x)", advance = "no")  2
call pl2%write (u)
write (u, *)

call pl3%init (2)
call pl3%set (1, 4)
call pl3%set (2, [5, -5])
write (u, "(I1,1x)", advance = "no")  3
call pl3%write (u)
write (u, *)

write (u, "(A)")
write (u, "(A)")  "* a == b"
write (u, "(A)")

write (u, "(2x,A)")  "123"
write (u, *)
write (u, "(I1,1x,4L1)")  1, pl1 == pl1, pl1 == pl2, pl1 == pl3
write (u, "(I1,1x,4L1)")  2, pl2 == pl1, pl2 == pl2, pl2 == pl3
write (u, "(I1,1x,4L1)")  3, pl3 == pl1, pl3 == pl2, pl3 == pl3

write (u, "(A)")
write (u, "(A)")  "* a < b"
write (u, "(A)")

write (u, "(2x,A)")  "123"
write (u, *)
write (u, "(I1,1x,4L1)")  1, pl1 < pl1, pl1 < pl2, pl1 < pl3
write (u, "(I1,1x,4L1)")  2, pl2 < pl1, pl2 < pl2, pl2 < pl3
write (u, "(I1,1x,4L1)")  3, pl3 < pl1, pl3 < pl2, pl3 < pl3

write (u, "(A)")
write (u, "(A)")  "* Test output end: pdg_arrays_4"
```

```
end subroutine pdg_arrays_4
```

Match-replace: translate all entries in the first list into the matching entries of the second list, if there is a match.

⟨*PDG arrays: execute tests*⟩+≡

```
call test (pdg_arrays_5, "pdg_arrays_5", &
     "match PDG lists", &
     u, results)
```

⟨*PDG arrays: test declarations*⟩+≡

```
public :: pdg_arrays_5
```

⟨*PDG arrays: tests*⟩+≡

```
subroutine pdg_arrays_5 (u)
  integer, intent(in) :: u

  type(pdg_list_t) :: pl1, pl2, pl3
  logical :: success

  write (u, "(A)")  "* Test output: pdg_arrays_5"
  write (u, "(A)")  "*   Purpose: match-replace"
  write (u, "(A)")

  write (u, "(A)")  "* Create lists"
  write (u, "(A)")

  call pl1%init (2)
  call pl1%set (1, [1, 2])
  call pl1%set (2, 42)
  call pl1%write (u)
  write (u, *)
  call pl3%init (2)
  call pl3%set (1, [42, -42])
  call pl3%set (2, [1, 2, 3, 4])
  call pl1%match_replace (pl3, success)
  call pl3%write (u)
  write (u, "(1x,A,1x,L1,':',1x)", advance="no")  "=>", success
  call pl1%write (u)
  write (u, *)

  write (u, *)

  call pl2%init (2)
  call pl2%set (1, 9)
  call pl2%set (2, 42)
  call pl2%write (u)
  write (u, *)
  call pl2%match_replace (pl3, success)
  call pl3%write (u)
  write (u, "(1x,A,1x,L1,':',1x)", advance="no")  "=>", success
  call pl2%write (u)
  write (u, *)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: pdg_arrays_5"
```

```
end subroutine pdg_arrays_5
```

12.3 Jets

The FastJet library is linked externally, if available. The wrapper code is also in a separate directory. Here, we define WHIZARD-specific procedures and tests.

⟨jets.f90⟩≡

```
⟨File header⟩

module jets

  use fastjet !NODEP!

⟨Standard module head⟩

⟨Jets: public⟩

contains

⟨Jets: procedures⟩

end module jets
```

12.3.1 Re-exported symbols

We use this module as a proxy for the FastJet interface, therefore we re-export some symbols.

⟨*Jets: public*⟩≡

```
public :: fastjet_available
public :: fastjet_init
public :: jet_definition_t
public :: pseudojet_t
public :: pseudojet_vector_t
public :: cluster_sequence_t
public :: assignment (=)
```

The initialization routine prints the banner.

⟨*Jets: procedures*⟩≡

```
subroutine fastjet_init ()
  call print_banner ()
end subroutine fastjet_init
```

The jet algorithm codes (actually, integers)

⟨*Jets: public*⟩+≡

```
public :: kt_algorithm
public :: cambridge_algorithm
public :: antikt_algorithm
public :: genkt_algorithm
public :: cambridge_for_passive_algorithm
public :: genkt_for_passive_algorithm
public :: ee_kt_algorithm
public :: ee_genkt_algorithm
public :: plugin_algorithm
public :: undefined_jet_algorithm
```

12.3.2 Unit tests

Test module, followed by the corresponding implementation module.

⟨jets_ut.f90⟩≡

```
⟨File header⟩

module jets_ut
  use unit_tests
  use jets_uti

⟨Standard module head⟩

⟨Jets: public test⟩

contains

⟨Jets: test driver⟩

end module jets_ut
```

⟨jets_uti.f90⟩≡

```
⟨File header⟩

module jets_uti

⟨Use kinds⟩
  use fastjet !NODEP!

  use jets

⟨Standard module head⟩

⟨Jets: test declarations⟩

contains

⟨Jets: tests⟩

end module jets_uti
```

API: driver for the unit tests below.

⟨Jets: public test⟩≡

```
  public :: jets_test
```

⟨Jets: test driver⟩≡

```
  subroutine jets_test (u, results)
    integer, intent(in) :: u
    type (test_results_t), intent(inout) :: results
  ⟨Jets: execute tests⟩
  end subroutine jets_test
```

This test is actually the minimal example from the FastJet manual, translated to Fortran.

Note that FastJet creates pseudojet vectors, which we mirror in the `pseudojet_vector_t`, but immediately assign to pseudojet arrays. Without automatic finalization available in the compilers, we should avoid this in actual code and rather introduce intermediate variables for those objects, which we can finalize explicitly.

⟨*Jets: execute tests*⟩≡
```
call test (jets_1, "jets_1", &
     "basic FastJet functionality", &
     u, results)
```

⟨*Jets: test declarations*⟩≡
```
public :: jets_1
```

⟨*Jets: tests*⟩≡
```
subroutine jets_1 (u)
  integer, intent(in) :: u

  type(pseudojet_t), dimension(:), allocatable :: prt, jets, constituents
  type(jet_definition_t) :: jet_def
  type(cluster_sequence_t) :: cs

  integer, parameter :: dp = default
  integer :: i, j

  write (u, "(A)")  "* Test output: jets_1"
  write (u, "(A)")  "*   Purpose: test basic FastJet functionality"
  write (u, "(A)")

  write (u, "(A)")  "* Print banner"
  call print_banner ()

  write (u, *)
  write (u, "(A)")  "* Prepare input particles"
  allocate (prt (3))
  call prt(1)%init ( 99._dp, 0.1_dp, 0._dp, 100._dp)
  call prt(2)%init (  4._dp,-0.1_dp, 0._dp,   5._dp)
  call prt(3)%init (-99._dp, 0._dp,  0._dp,  99._dp)

  write (u, *)
  write (u, "(A)")  "* Define jet algorithm"
  call jet_def%init (antikt_algorithm, 0.7_dp)

  write (u, *)
  write (u, "(A)")  "* Cluster particles according to jet algorithm"

  write (u, *)
  write (u, "(A,A)")  "Clustering with ", jet_def%description ()
  call cs%init (pseudojet_vector (prt), jet_def)

  write (u, *)
  write (u, "(A)")  "* Sort output jets"
  jets = sorted_by_pt (cs%inclusive_jets ())

  write (u, *)
```

```
  write (u, "(A)")  "* Print jet observables and constituents"
  write (u, *)
  write (u, "(4x,3(7x,A3))") "pt", "y", "phi"
  do i = 1, size (jets)
     write (u, "(A,1x,I0,A,3(1x,F9.5))") &
          "jet", i, ":", jets(i)%perp (), jets(i)%rap (), jets(i)%phi ()
     constituents = jets(i)%constituents ()
     do j = 1, size (constituents)
        write (u, "(4x,A,1x,I0,A,F9.5)") &
             "constituent", j, "'s pt:", constituents(j)%perp ()
     end do
     do j = 1, size (constituents)
        call constituents(j)%final ()
     end do
  end do

  write (u, *)
  write (u, "(A)")  "* Cleanup"

  do i = 1, size (prt)
     call prt(i)%final ()
  end do
  do i = 1, size (jets)
     call jets(i)%final ()
  end do
  call jet_def%final ()
  call cs%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: jets_1"

end subroutine jets_1
```

12.4 Subevents

The purpose of subevents is to store the relevant part of the physical event (either partonic or hadronic), and to hold particle selections and combinations which are constructed in cut or analysis expressions.

⟨subevents.f90⟩≡

```
⟨File header⟩

module subevents

  use, intrinsic :: iso_c_binding !NODEP!

⟨Use kinds⟩
  use io_units
  use format_defs, only: FMT_14, FMT_19
  use format_utils, only: pac_fmt
  use sorting
  use c_particles
  use lorentz
  use pdg_arrays
  use jets

⟨Standard module head⟩

⟨Subevents: public⟩

⟨Subevents: parameters⟩

⟨Subevents: types⟩

⟨Subevents: interfaces⟩

contains

⟨Subevents: procedures⟩

end module subevents
```

12.4.1 Particles

For the purpose of this module, a particle has a type which can indicate a beam, incoming, outgoing, or composite particle, flavor and helicity codes (integer, undefined for composite), four-momentum and invariant mass squared. (Other particles types are used in extended event types, but also defined here.) Furthermore, each particle has an allocatable array of ancestors – particle indices which indicate the building blocks of a composite particle. For an incoming/outgoing particle, the array contains only the index of the particle itself.

For incoming particles, the momentum is inverted before storing it in the particle object.

⟨*Subevents: parameters*⟩≡

```
integer, parameter, public :: PRT_UNDEFINED = 0
integer, parameter, public :: PRT_BEAM = -9
```

```
integer, parameter, public :: PRT_INCOMING = 1
integer, parameter, public :: PRT_OUTGOING = 2
integer, parameter, public :: PRT_COMPOSITE = 3
integer, parameter, public :: PRT_VIRTUAL = 4
integer, parameter, public :: PRT_RESONANT = 5
integer, parameter, public :: PRT_BEAM_REMNANT = 9
```

The type

We initialize only the type here and mark as unpolarized. The initializers below do the rest.

⟨*Subevents: public*⟩≡

```
public :: prt_t
```

⟨*Subevents: types*⟩≡

```
type :: prt_t
   private
   integer :: type = PRT_UNDEFINED
   integer :: pdg
   logical :: polarized = .false.
   integer :: h
   type(vector4_t) :: p
   real(default) :: p2
   integer, dimension(:), allocatable :: src
end type prt_t
```

Initializers. Polarization is set separately. Finalizers are not needed.

⟨*Subevents: procedures*⟩≡

```
subroutine prt_init_beam (prt, pdg, p, p2, src)
  type(prt_t), intent(out) :: prt
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in) :: src
  prt%type = PRT_BEAM
  call prt_set (prt, pdg, - p, p2, src)
end subroutine prt_init_beam

subroutine prt_init_incoming (prt, pdg, p, p2, src)
  type(prt_t), intent(out) :: prt
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in) :: src
  prt%type = PRT_INCOMING
  call prt_set (prt, pdg, - p, p2, src)
end subroutine prt_init_incoming

subroutine prt_init_outgoing (prt, pdg, p, p2, src)
  type(prt_t), intent(out) :: prt
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
```

```
    real(default), intent(in) :: p2
    integer, dimension(:), intent(in) :: src
    prt%type = PRT_OUTGOING
    call prt_set (prt, pdg, p, p2, src)
  end subroutine prt_init_outgoing

  subroutine prt_init_composite (prt, p, src)
    type(prt_t), intent(out) :: prt
    type(vector4_t), intent(in) :: p
    integer, dimension(:), intent(in) :: src
    prt%type = PRT_COMPOSITE
    call prt_set (prt, 0, p, p**2, src)
  end subroutine prt_init_composite
```

This version is for temporary particle objects, so the `src` array is not set.

⟨*Subevents: public*⟩+≡

```
  public :: prt_init_combine
```

⟨*Subevents: procedures*⟩+≡

```
  subroutine prt_init_combine (prt, prt1, prt2)
    type(prt_t), intent(out) :: prt
    type(prt_t), intent(in) :: prt1, prt2
    type(vector4_t) :: p
    integer, dimension(0) :: src
    prt%type = PRT_COMPOSITE
    p = prt1%p + prt2%p
    call prt_set (prt, 0, p, p**2, src)
  end subroutine prt_init_combine
```

Init from a pseudojet object.

⟨*Subevents: procedures*⟩+≡

```
  subroutine prt_init_pseudojet (prt, jet, src, pdg)
    type(prt_t), intent(out) :: prt
    type(pseudojet_t), intent(in) :: jet
    integer, dimension(:), intent(in) :: src
    integer, intent(in) :: pdg
    type(vector4_t) :: p
    prt%type = PRT_COMPOSITE
    p = vector4_moving (jet%e(), &
         vector3_moving ([jet%px(), jet%py(), jet%pz()]))
    call prt_set (prt, pdg, p, p**2, src)
  end subroutine prt_init_pseudojet
```

Accessing contents

⟨*Subevents: public*⟩+≡

```
  public :: prt_get_pdg
```

⟨*Subevents: procedures*⟩+≡

```
  elemental function prt_get_pdg (prt) result (pdg)
    integer :: pdg
    type(prt_t), intent(in) :: prt
    pdg = prt%pdg
```

```
    end function prt_get_pdg
```

⟨*Subevents: public*⟩+≡

```
    public :: prt_get_momentum
```

⟨*Subevents: procedures*⟩+≡

```
    elemental function prt_get_momentum (prt) result (p)
      type(vector4_t) :: p
      type(prt_t), intent(in) :: prt
      p = prt%p
    end function prt_get_momentum
```

⟨*Subevents: public*⟩+≡

```
    public :: prt_get_msq
```

⟨*Subevents: procedures*⟩+≡

```
    elemental function prt_get_msq (prt) result (msq)
      real(default) :: msq
      type(prt_t), intent(in) :: prt
      msq = prt%p2
    end function prt_get_msq
```

⟨*Subevents: public*⟩+≡

```
    public :: prt_is_polarized
```

⟨*Subevents: procedures*⟩+≡

```
    elemental function prt_is_polarized (prt) result (flag)
      logical :: flag
      type(prt_t), intent(in) :: prt
      flag = prt%polarized
    end function prt_is_polarized
```

⟨*Subevents: public*⟩+≡

```
    public :: prt_get_helicity
```

⟨*Subevents: procedures*⟩+≡

```
    elemental function prt_get_helicity (prt) result (h)
      integer :: h
      type(prt_t), intent(in) :: prt
      h = prt%h
    end function prt_get_helicity
```

Setting data

Set the PDG, momentum and momentum squared, and ancestors. If allocate-on-assignment is available, this can be simplified.

⟨*Subevents: procedures*⟩+≡

```
    subroutine prt_set (prt, pdg, p, p2, src)
      type(prt_t), intent(inout) :: prt
      integer, intent(in) :: pdg
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: p2
      integer, dimension(:), intent(in) :: src
```

```
    prt%pdg = pdg
    prt%p = p
    prt%p2 = p2
    if (allocated (prt%src)) then
       if (size (src) /= size (prt%src)) then
          deallocate (prt%src)
          allocate (prt%src (size (src)))
       end if
    else
       allocate (prt%src (size (src)))
    end if
    prt%src = src
  end subroutine prt_set
```

Set the particle PDG code separately.

⟨*Subevents: procedures*⟩+≡

```
  elemental subroutine prt_set_pdg (prt, pdg)
    type(prt_t), intent(inout) :: prt
    integer, intent(in) :: pdg
    prt%pdg = pdg
  end subroutine prt_set_pdg
```

Set the momentum separately.

⟨*Subevents: procedures*⟩+≡

```
  elemental subroutine prt_set_p (prt, p)
    type(prt_t), intent(inout) :: prt
    type(vector4_t), intent(in) :: p
    prt%p = p
  end subroutine prt_set_p
```

Set the squared invariant mass separately.

⟨*Subevents: procedures*⟩+≡

```
  elemental subroutine prt_set_p2 (prt, p2)
    type(prt_t), intent(inout) :: prt
    real(default), intent(in) :: p2
    prt%p2 = p2
  end subroutine prt_set_p2
```

Set helicity (optional).

⟨*Subevents: procedures*⟩+≡

```
  subroutine prt_polarize (prt, h)
    type(prt_t), intent(inout) :: prt
    integer, intent(in) :: h
    prt%polarized = .true.
    prt%h = h
  end subroutine prt_polarize
```

Conversion

Transform a `prt_t` object into a `c_prt_t` object.

⟨*Subevents: public*⟩+≡

```
    public :: c_prt
```

⟨*Subevents: interfaces*⟩≡

```
    interface c_prt
       module procedure c_prt_from_prt
    end interface
```

⟨*Subevents: procedures*⟩+≡

```
    elemental function c_prt_from_prt (prt) result (c_prt)
      type(c_prt_t) :: c_prt
      type(prt_t), intent(in) :: prt
      c_prt = prt%p
      c_prt%type = prt%type
      c_prt%pdg = prt%pdg
      if (prt%polarized) then
         c_prt%polarized = 1
      else
         c_prt%polarized = 0
      end if
      c_prt%h = prt%h
    end function c_prt_from_prt
```

Output

⟨*Subevents: public*⟩+≡

```
    public :: prt_write
```

⟨*Subevents: procedures*⟩+≡

```
    subroutine prt_write (prt, unit, testflag)
      type(prt_t), intent(in) :: prt
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: testflag
      logical :: pacified
      type(prt_t) :: tmp
      character(len=7) :: fmt
      integer :: u, i
      call pac_fmt (fmt, FMT_19, FMT_14, testflag)
      u = given_output_unit (unit);  if (u < 0)  return
      pacified = .false. ; if (present (testflag))  pacified = testflag
      tmp = prt
      if (pacified) call pacify (tmp)
      write (u, "(1x,A)", advance="no")  "prt("
      select case (prt%type)
      case (PRT_UNDEFINED);    write (u, "('?')", advance="no")
      case (PRT_BEAM);         write (u, "('b:')", advance="no")
      case (PRT_INCOMING);     write (u, "('i:')", advance="no")
      case (PRT_OUTGOING);     write (u, "('o:')", advance="no")
      case (PRT_COMPOSITE);    write (u, "('c:')", advance="no")
      end select
      select case (prt%type)
      case (PRT_BEAM, PRT_INCOMING, PRT_OUTGOING)
         if (prt%polarized) then
            write (u, "(I0,'/',I0,'|')", advance="no")  prt%pdg, prt%h
         else
```

```
         write (u, "(I0,'|')", advance="no") prt%pdg
      end if
   end select
   select case (prt%type)
   case (PRT_BEAM, PRT_INCOMING, PRT_OUTGOING, PRT_COMPOSITE)
      write (u, "(" // FMT_14 // ",';'," // FMT_14 // ",',',"  // &
           FMT_14 // ",',',"  // FMT_14 // ")", advance="no") tmp%p
      write (u, "('|'," // fmt // ")", advance="no") tmp%p2
   end select
   if (allocated (prt%src)) then
      write (u, "('|')", advance="no")
      do i = 1, size (prt%src)
         write (u, "(1x,I0)", advance="no")  prt%src(i)
      end do
   end if
   write (u, "(A)")  ")"
end subroutine prt_write
```

Tools

Two particles match if their `src` arrays are the same.

⟨*Subevents: interfaces*⟩+≡

```
interface operator(.match.)
   module procedure prt_match
end interface
```

⟨*Subevents: procedures*⟩+≡

```
elemental function prt_match (prt1, prt2) result (match)
  logical :: match
  type(prt_t), intent(in) :: prt1, prt2
  if (size (prt1%src) == size (prt2%src)) then
     match = all (prt1%src == prt2%src)
  else
     match = .false.
  end if
end function prt_match
```

The combine operation makes a pseudoparticle whose momentum is the result of adding (the momenta of) the pair of input particles. We trace the particles from which a particle is built by storing a `src` array. Each particle entry in the `src` list contains a list of indices which indicates its building blocks. The indices refer to an original list of particles. Index lists are sorted, and they contain no element more than once.

We thus require that in a given pseudoparticle, each original particle occurs at most once.

The result is intent(inout), so it will not be initialized when the subroutine is entered.

⟨*Subevents: procedures*⟩+≡

```
subroutine prt_combine (prt, prt_in1, prt_in2, ok)
  type(prt_t), intent(inout) :: prt
```

```
    type(prt_t), intent(in) :: prt_in1, prt_in2
    logical :: ok
    integer, dimension(:), allocatable :: src
    call combine_index_lists (src, prt_in1%src, prt_in2%src)
    ok = allocated (src)
    if (ok)  call prt_init_composite (prt, prt_in1%p + prt_in2%p, src)
  end subroutine prt_combine
```

This variant does not produce the combined particle, it just checks whether the combination is valid (no common `src` entry).

⟨*Subevents: public*⟩+≡

```
  public :: are_disjoint
```

⟨*Subevents: procedures*⟩+≡

```
  function are_disjoint (prt_in1, prt_in2) result (flag)
    logical :: flag
    type(prt_t), intent(in) :: prt_in1, prt_in2
    flag = index_lists_are_disjoint (prt_in1%src, prt_in2%src)
  end function are_disjoint
```

`src` Lists with length > 1 are built by a `combine` operation which merges the lists in a sorted manner. If the result would have a duplicate entry, it is discarded, and the result is unallocated.

⟨*Subevents: procedures*⟩+≡

```
  subroutine combine_index_lists (res, src1, src2)
    integer, dimension(:), intent(in) :: src1, src2
    integer, dimension(:), allocatable :: res
    integer :: i1, i2, i
    allocate (res (size (src1) + size (src2)))
    if (size (src1) == 0) then
       res = src2
       return
    else if (size (src2) == 0) then
       res = src1
       return
    end if
    i1 = 1
    i2 = 1
    LOOP: do i = 1, size (res)
       if (src1(i1) < src2(i2)) then
          res(i) = src1(i1);  i1 = i1 + 1
          if (i1 > size (src1)) then
             res(i+1:) = src2(i2:)
             exit LOOP
          end if
       else if (src1(i1) > src2(i2)) then
          res(i) = src2(i2);  i2 = i2 + 1
          if (i2 > size (src2)) then
             res(i+1:) = src1(i1:)
             exit LOOP
          end if
       else
          deallocate (res)
```

```
            exit LOOP
         end if
      end do LOOP
   end subroutine combine_index_lists
```

This function is similar, but it does not actually merge the list, it just checks whether they are disjoint (no common `src` entry).

⟨*Subevents: procedures*⟩+≡

```
   function index_lists_are_disjoint (src1, src2) result (flag)
      logical :: flag
      integer, dimension(:), intent(in) :: src1, src2
      integer :: i1, i2, i
      flag = .true.
      i1 = 1
      i2 = 1
      LOOP: do i = 1, size (src1) + size (src2)
         if (src1(i1) < src2(i2)) then
            i1 = i1 + 1
            if (i1 > size (src1)) then
               exit LOOP
            end if
         else if (src1(i1) > src2(i2)) then
            i2 = i2 + 1
            if (i2 > size (src2)) then
               exit LOOP
            end if
         else
            flag = .false.
            exit LOOP
         end if
      end do LOOP
   end function index_lists_are_disjoint
```

12.4.2 subevents

Particles are collected in subevents. This type is implemented as a dynamically allocated array, which need not be completely filled. The value `n_active` determines the number of meaningful entries.

Type definition

⟨*Subevents: public*⟩+≡

```
   public :: subevt_t
```

⟨*Subevents: types*⟩+≡

```
   type :: subevt_t
      private
      integer :: n_active = 0
      type(prt_t), dimension(:), allocatable :: prt
    contains
    ⟨Subevents: subevt: TBP⟩
   end type subevt_t
```

Initialize, allocating with size zero (default) or given size. The number of contained particles is set equal to the size.

⟨*Subevents: public*⟩+≡
```
public :: subevt_init
```

⟨*Subevents: procedures*⟩+≡
```
subroutine subevt_init (subevt, n_active)
  type(subevt_t), intent(out) :: subevt
  integer, intent(in), optional :: n_active
  if (present (n_active))  subevt%n_active = n_active
  allocate (subevt%prt (subevt%n_active))
end subroutine subevt_init
```

(Re-)allocate the subevent with some given size. If the size is greater than the previous one, do a real reallocation. Otherwise, just reset the recorded size. Contents are untouched, but become invalid.

⟨*Subevents: public*⟩+≡
```
public :: subevt_reset
```

⟨*Subevents: procedures*⟩+≡
```
subroutine subevt_reset (subevt, n_active)
  type(subevt_t), intent(inout) :: subevt
  integer, intent(in) :: n_active
  subevt%n_active = n_active
  if (subevt%n_active > size (subevt%prt)) then
     deallocate (subevt%prt)
     allocate (subevt%prt (subevt%n_active))
  end if
end subroutine subevt_reset
```

Output. No prefix for the headline 'subevt', because this will usually be printed appending to a previous line.

⟨*Subevents: public*⟩+≡
```
public :: subevt_write
```

⟨*Subevents: subevt: TBP*⟩≡
```
procedure :: write => subevt_write
```

⟨*Subevents: procedures*⟩+≡
```
subroutine subevt_write (object, unit, prefix, pacified)
  class(subevt_t), intent(in) :: object
  integer, intent(in), optional :: unit
  character(*), intent(in), optional :: prefix
  logical, intent(in), optional :: pacified
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)") "subevent:"
  do i = 1, object%n_active
     if (present (prefix))  write (u, "(A)", advance="no") prefix
     write (u, "(1x,I0)", advance="no")  i
     call prt_write (object%prt(i), unit = unit, testflag = pacified)
  end do
end subroutine subevt_write
```

Defined assignment: transfer only meaningful entries. This is a deep copy (as would be default assignment).

⟨*Subevents: interfaces*⟩+≡

```
interface assignment(=)
   module procedure subevt_assign
end interface
```

⟨*Subevents: procedures*⟩+≡

```
subroutine subevt_assign (subevt, subevt_in)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: subevt_in
  if (.not. allocated (subevt%prt)) then
     call subevt_init (subevt, subevt_in%n_active)
  else
     call subevt_reset (subevt, subevt_in%n_active)
  end if
  subevt%prt(:subevt%n_active) = subevt_in%prt(:subevt%n_active)
end subroutine subevt_assign
```

Fill contents

Store incoming/outgoing particles which are completely defined.

⟨*Subevents: public*⟩+≡

```
public :: subevt_set_beam
public :: subevt_set_incoming
public :: subevt_set_outgoing
public :: subevt_set_composite
```

⟨*Subevents: procedures*⟩+≡

```
subroutine subevt_set_beam (subevt, i, pdg, p, p2, src)
  type(subevt_t), intent(inout) :: subevt
  integer, intent(in) :: i
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in), optional :: src
  if (present (src)) then
     call prt_init_beam (subevt%prt(i), pdg, p, p2, src)
  else
     call prt_init_beam (subevt%prt(i), pdg, p, p2, [i])
  end if
end subroutine subevt_set_beam

subroutine subevt_set_incoming (subevt, i, pdg, p, p2, src)
  type(subevt_t), intent(inout) :: subevt
  integer, intent(in) :: i
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in), optional :: src
  if (present (src)) then
     call prt_init_incoming (subevt%prt(i), pdg, p, p2, src)
```

```
  else
     call prt_init_incoming (subevt%prt(i), pdg, p, p2, [i])
  end if
end subroutine subevt_set_incoming

subroutine subevt_set_outgoing (subevt, i, pdg, p, p2, src)
  type(subevt_t), intent(inout) :: subevt
  integer, intent(in) :: i
  integer, intent(in) :: pdg
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: p2
  integer, dimension(:), intent(in), optional :: src
  if (present (src)) then
     call prt_init_outgoing (subevt%prt(i), pdg, p, p2, src)
  else
     call prt_init_outgoing (subevt%prt(i), pdg, p, p2, [i])
  end if
end subroutine subevt_set_outgoing

subroutine subevt_set_composite (subevt, i, p, src)
  type(subevt_t), intent(inout) :: subevt
  integer, intent(in) :: i
  type(vector4_t), intent(in) :: p
  integer, dimension(:), intent(in) :: src
  call prt_init_composite (subevt%prt(i), p, src)
end subroutine subevt_set_composite
```

Separately assign flavors, simultaneously for all incoming/outgoing particles.

⟨*Subevents: public*⟩+≡

```
public :: subevt_set_pdg_beam
public :: subevt_set_pdg_incoming
public :: subevt_set_pdg_outgoing
```

⟨*Subevents: procedures*⟩+≡

```
subroutine subevt_set_pdg_beam (subevt, pdg)
  type(subevt_t), intent(inout) :: subevt
  integer, dimension(:), intent(in) :: pdg
  integer :: i, j
  j = 1
  do i = 1, subevt%n_active
     if (subevt%prt(i)%type == PRT_BEAM) then
        call prt_set_pdg (subevt%prt(i), pdg(j))
        j = j + 1
        if (j > size (pdg))  exit
     end if
  end do
end subroutine subevt_set_pdg_beam

subroutine subevt_set_pdg_incoming (subevt, pdg)
  type(subevt_t), intent(inout) :: subevt
  integer, dimension(:), intent(in) :: pdg
  integer :: i, j
  j = 1
  do i = 1, subevt%n_active
```

```
      if (subevt%prt(i)%type == PRT_INCOMING) then
         call prt_set_pdg (subevt%prt(i), pdg(j))
         j = j + 1
         if (j > size (pdg))  exit
      end if
   end do
end subroutine subevt_set_pdg_incoming

subroutine subevt_set_pdg_outgoing (subevt, pdg)
   type(subevt_t), intent(inout) :: subevt
   integer, dimension(:), intent(in) :: pdg
   integer :: i, j
   j = 1
   do i = 1, subevt%n_active
      if (subevt%prt(i)%type == PRT_OUTGOING) then
         call prt_set_pdg (subevt%prt(i), pdg(j))
         j = j + 1
         if (j > size (pdg))  exit
      end if
   end do
end subroutine subevt_set_pdg_outgoing
```

Separately assign momenta, simultaneously for all incoming/outgoing particles.

⟨*Subevents: public*⟩+≡

```
public :: subevt_set_p_beam
public :: subevt_set_p_incoming
public :: subevt_set_p_outgoing
```

⟨*Subevents: procedures*⟩+≡

```
subroutine subevt_set_p_beam (subevt, p)
   type(subevt_t), intent(inout) :: subevt
   type(vector4_t), dimension(:), intent(in) :: p
   integer :: i, j
   j = 1
   do i = 1, subevt%n_active
      if (subevt%prt(i)%type == PRT_BEAM) then
         call prt_set_p (subevt%prt(i), p(j))
         j = j + 1
         if (j > size (p))  exit
      end if
   end do
end subroutine subevt_set_p_beam

subroutine subevt_set_p_incoming (subevt, p)
   type(subevt_t), intent(inout) :: subevt
   type(vector4_t), dimension(:), intent(in) :: p
   integer :: i, j
   j = 1
   do i = 1, subevt%n_active
      if (subevt%prt(i)%type == PRT_INCOMING) then
         call prt_set_p (subevt%prt(i), p(j))
         j = j + 1
         if (j > size (p))  exit
      end if
```

```
  end do
end subroutine subevt_set_p_incoming

subroutine subevt_set_p_outgoing (subevt, p)
  type(subevt_t), intent(inout) :: subevt
  type(vector4_t), dimension(:), intent(in) :: p
  integer :: i, j
  j = 1
  do i = 1, subevt%n_active
     if (subevt%prt(i)%type == PRT_OUTGOING) then
        call prt_set_p (subevt%prt(i), p(j))
        j = j + 1
        if (j > size (p))  exit
     end if
  end do
end subroutine subevt_set_p_outgoing
```

Separately assign the squared invariant mass, simultaneously for all incoming/outgoing particles.

⟨*Subevents: public*⟩+≡

```
public :: subevt_set_p2_beam
public :: subevt_set_p2_incoming
public :: subevt_set_p2_outgoing
```

⟨*Subevents: procedures*⟩+≡

```
subroutine subevt_set_p2_beam (subevt, p2)
  type(subevt_t), intent(inout) :: subevt
  real(default), dimension(:), intent(in) :: p2
  integer :: i, j
  j = 1
  do i = 1, subevt%n_active
     if (subevt%prt(i)%type == PRT_BEAM) then
        call prt_set_p2 (subevt%prt(i), p2(j))
        j = j + 1
        if (j > size (p2))  exit
     end if
  end do
end subroutine subevt_set_p2_beam

subroutine subevt_set_p2_incoming (subevt, p2)
  type(subevt_t), intent(inout) :: subevt
  real(default), dimension(:), intent(in) :: p2
  integer :: i, j
  j = 1
  do i = 1, subevt%n_active
     if (subevt%prt(i)%type == PRT_INCOMING) then
        call prt_set_p2 (subevt%prt(i), p2(j))
        j = j + 1
        if (j > size (p2))  exit
     end if
  end do
end subroutine subevt_set_p2_incoming

subroutine subevt_set_p2_outgoing (subevt, p2)
```

```
    type(subevt_t), intent(inout) :: subevt
    real(default), dimension(:), intent(in) :: p2
    integer :: i, j
    j = 1
    do i = 1, subevt%n_active
       if (subevt%prt(i)%type == PRT_OUTGOING) then
          call prt_set_p2 (subevt%prt(i), p2(j))
          j = j + 1
          if (j > size (p2))  exit
       end if
    end do
  end subroutine subevt_set_p2_outgoing
```

Set polarization for an entry

⟨*Subevents: public*⟩+≡

```
  public :: subevt_polarize
```

⟨*Subevents: procedures*⟩+≡

```
  subroutine subevt_polarize (subevt, i, h)
    type(subevt_t), intent(inout) :: subevt
    integer, intent(in) :: i, h
    call prt_polarize (subevt%prt(i), h)
  end subroutine subevt_polarize
```

Accessing contents

Return true if the subevent has entries.

⟨*Subevents: public*⟩+≡

```
  public :: subevt_is_nonempty
```

⟨*Subevents: procedures*⟩+≡

```
  function subevt_is_nonempty (subevt) result (flag)
    logical :: flag
    type(subevt_t), intent(in) :: subevt
    flag = subevt%n_active /= 0
  end function subevt_is_nonempty
```

Return the number of entries

⟨*Subevents: public*⟩+≡

```
  public :: subevt_get_length
```

⟨*Subevents: procedures*⟩+≡

```
  function subevt_get_length (subevt) result (length)
    integer :: length
    type(subevt_t), intent(in) :: subevt
    length = subevt%n_active
  end function subevt_get_length
```

Return a specific particle. The index is not checked for validity.

⟨*Subevents: public*⟩+≡

```
  public :: subevt_get_prt
```

⟨*Subevents: procedures*⟩+≡

```
function subevt_get_prt (subevt, i) result (prt)
  type(prt_t) :: prt
  type(subevt_t), intent(in) :: subevt
  integer, intent(in) :: i
  prt = subevt%prt(i)
end function subevt_get_prt
```

Return the partonic energy squared. We take the particles with flag PRT_INCOMING and compute their total invariant mass.

⟨*Subevents: public*⟩+≡

```
public :: subevt_get_sqrts_hat
```

⟨*Subevents: procedures*⟩+≡

```
function subevt_get_sqrts_hat (subevt) result (sqrts_hat)
  type(subevt_t), intent(in) :: subevt
  real(default) :: sqrts_hat
  type(vector4_t) :: p
  integer :: i
  do i = 1, subevt%n_active
     if (subevt%prt(i)%type == PRT_INCOMING) then
        p = p + prt_get_momentum (subevt%prt(i))
     end if
  end do
  sqrts_hat = p ** 1
end function subevt_get_sqrts_hat
```

Return the number of incoming (outgoing) particles, respectively. Beam particles or composites are not counted.

⟨*Subevents: public*⟩+≡

```
public :: subevt_get_n_in
public :: subevt_get_n_out
```

⟨*Subevents: procedures*⟩+≡

```
function subevt_get_n_in (subevt) result (n_in)
  type(subevt_t), intent(in) :: subevt
  integer :: n_in
  n_in = count (subevt%prt(:subevt%n_active)%type == PRT_INCOMING)
end function subevt_get_n_in

function subevt_get_n_out (subevt) result (n_out)
  type(subevt_t), intent(in) :: subevt
  integer :: n_out
  n_out = count (subevt%prt(:subevt%n_active)%type == PRT_OUTGOING)
end function subevt_get_n_out
```

⟨*Subevents: interfaces*⟩+≡

```
interface c_prt
   module procedure c_prt_from_subevt
   module procedure c_prt_array_from_subevt
end interface
```

⟨*Subevents: procedures*⟩+≡

```
function c_prt_from_subevt (subevt, i) result (c_prt)
  type(c_prt_t) :: c_prt
  type(subevt_t), intent(in) :: subevt
  integer, intent(in) :: i
  c_prt = c_prt_from_prt (subevt%prt(i))
end function c_prt_from_subevt

function c_prt_array_from_subevt (subevt) result (c_prt_array)
  type(subevt_t), intent(in) :: subevt
  type(c_prt_t), dimension(subevt%n_active) :: c_prt_array
  c_prt_array = c_prt_from_prt (subevt%prt(1:subevt%n_active))
end function c_prt_array_from_subevt
```

Operations with subevents

The join operation joins two subevents. When appending the elements of the second list, we check for each particle whether it is already in the first list. If yes, it is discarded. The result list should be initialized already.

If a mask is present, it refers to the second subevent. Particles where the mask is not set are discarded.

⟨*Subevents: public*⟩+≡

```
public :: subevt_join
```

⟨*Subevents: procedures*⟩+≡

```
subroutine subevt_join (subevt, pl1, pl2, mask2)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: pl1, pl2
  logical, dimension(:), intent(in), optional :: mask2
  integer :: n1, n2, i, n
  n1 = pl1%n_active
  n2 = pl2%n_active
  call subevt_reset (subevt, n1 + n2)
  subevt%prt(:n1)   = pl1%prt(:n1)
  n = n1
  if (present (mask2)) then
     do i = 1, pl2%n_active
        if (mask2(i)) then
           if (disjoint (i)) then
              n = n + 1
              subevt%prt(n) = pl2%prt(i)
           end if
        end if
     end do
  else
     do i = 1, pl2%n_active
        if (disjoint (i)) then
           n = n + 1
           subevt%prt(n) = pl2%prt(i)
        end if
     end do
  end if
  subevt%n_active = n
```

```
contains
  function disjoint (i) result (flag)
    integer, intent(in) :: i
    logical :: flag
    integer :: j
    do j = 1, pl1%n_active
       if (.not. are_disjoint (pl1%prt(j), pl2%prt(i))) then
          flag = .false.
          return
       end if
    end do
    flag = .true.
  end function disjoint
end subroutine subevt_join
```

The combine operation makes a subevent whose entries are the result of adding (the momenta of) each pair of particles in the input lists. We trace the particles from which a particles is built by storing a `src` array. Each particle entry in the `src` list contains a list of indices which indicates its building blocks. The indices refer to an original list of particles. Index lists are sorted, and they contain no element more than once.

We thus require that in a given pseudoparticle, each original particle occurs at most once.

⟨*Subevents: public*⟩+≡

```
public :: subevt_combine
```

⟨*Subevents: procedures*⟩+≡

```
subroutine subevt_combine (subevt, pl1, pl2, mask12)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: pl1, pl2
  logical, dimension(:,:), intent(in), optional :: mask12
  integer :: n1, n2, i1, i2, n, j
  logical :: ok
  n1 = pl1%n_active
  n2 = pl2%n_active
  call subevt_reset (subevt, n1 * n2)
  n = 1
  do i1 = 1, n1
     do i2 = 1, n2
        if (present (mask12)) then
           ok = mask12(i1,i2)
        else
           ok = .true.
        end if
        if (ok)  call prt_combine &
             (subevt%prt(n), pl1%prt(i1), pl2%prt(i2), ok)
        if (ok) then
           CHECK_DOUBLES: do j = 1, n - 1
              if (subevt%prt(n) .match. subevt%prt(j)) then
                 ok = .false.;  exit CHECK_DOUBLES
              end if
           end do CHECK_DOUBLES
           if (ok)  n = n + 1
```

```
         end if
      end do
   end do
   subevt%n_active = n - 1
end subroutine subevt_combine
```

The collect operation makes a single-entry subevent which results from combining (the momenta of) all particles in the input list. As above, the result does not contain an original particle more than once; this is checked for each particle when it is collected. Furthermore, each entry has a mask; where the mask is false, the entry is dropped.

(Thus, if the input particles are already composite, there is some chance that the result depends on the order of the input list and is not as expected. This situation should be avoided.)

⟨*Subevents: public*⟩+≡

```
public :: subevt_collect
```

⟨*Subevents: procedures*⟩+≡

```
subroutine subevt_collect (subevt, pl1, mask1)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: pl1
  logical, dimension(:), intent(in) :: mask1
  type(prt_t) :: prt
  integer :: i
  logical :: ok
  call subevt_reset (subevt, 1)
  subevt%n_active = 0
  do i = 1, pl1%n_active
     if (mask1(i)) then
        if (subevt%n_active == 0) then
           subevt%n_active = 1
           subevt%prt(1) = pl1%prt(i)
        else
           call prt_combine (prt, subevt%prt(1), pl1%prt(i), ok)
           if (ok)  subevt%prt(1) = prt
        end if
     end if
  end do
end subroutine subevt_collect
```

The cluster operation is similar to `collect`, but applies a jet algorithm. The result is a subevent consisting of jets and, possibly, unclustered extra particles. As above, the result does not contain an original particle more than once; this is checked for each particle when it is collected. Furthermore, each entry has a mask; where the mask is false, the entry is dropped.

The algorithm: first determine the (pseudo)particles that participate in the clustering. They should not overlap, and the mask entry must be set. We then cluster the particles, using the given jet definition. The result particles are retrieved from the cluster sequence. We still have to determine the source indices for each jet: for each input particle, we get the jet index. Accumulating the source entries for all particles that are part of a given jet, we derive the jet

source entries. Finally, we delete the C structures that have been constructed by FastJet and its interface.

⟨*Subevents: public*⟩+≡

```
public :: subevt_cluster
```

⟨*Subevents: procedures*⟩+≡

```
subroutine subevt_cluster (subevt, pl1, mask1, jet_def, keep_jets)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: pl1
  logical, dimension(:), intent(in) :: mask1
  type(jet_definition_t), intent(in) :: jet_def
  logical, intent(in) :: keep_jets
  integer, dimension(:), allocatable :: src, src_tmp
  integer, dimension(:), allocatable :: map, jet_idx
  type(pseudojet_t), dimension(:), allocatable :: jet_in, jet_out
  type(pseudojet_vector_t) :: jv_in, jv_out
  type(cluster_sequence_t) :: cs
  integer :: i, prt_idx, k, n_src, n_active, this_jet, combined_pdg, pdg, n_quarks
  n_active = 0
  allocate (map (pl1%n_active), source = 0)
  allocate (src (0))
  do i = 1, pl1%n_active
     if (mask1(i)) then
        call combine_index_lists (src_tmp, src, pl1%prt(i)%src)
        if (allocated (src_tmp)) then
           call move_alloc (from=src_tmp, to=src)
           n_active = n_active + 1
           map(n_active) = i
        end if
     end if
  end do
  allocate (jet_in (count (map /= 0)))
  do i = 1, size (jet_in)
     call jet_in(i)%init (prt_get_momentum (pl1%prt(map(i))))
  end do
  call jv_in%init (jet_in)
  call cs%init (jv_in, jet_def)
  jv_out = cs%inclusive_jets ()
  allocate (jet_idx (size (jet_in)))
  call cs%assign_jet_indices (jv_out, jet_idx)
  allocate (jet_out (jv_out%size ()))
  jet_out = jv_out
  call subevt_reset (subevt, size (jet_out))
  do this_jet = 1, size (jet_out)
     src = 0
     n_src = 0
     combined_pdg = 0
     n_quarks = 0
     do prt_idx = 1, size (jet_idx)
        if (jet_idx(prt_idx) == this_jet) then
           associate (prt => pl1%prt(map(prt_idx)))
             do k = 1, size (prt%src)
                src(n_src + k) = prt%src(k)
             end do
```

```
            n_src = n_src + size (prt%src)
            if (is_quark (prt%pdg)) then
               n_quarks = n_quarks + 1
               if (combined_pdg == 0) then
                  combined_pdg = prt%pdg
               end if
            end if
         end associate
      end if
   end do
   if (keep_jets .and. n_quarks == 1) then
      pdg = combined_pdg
   else
      pdg = 0
   end if
   call prt_init_pseudojet (subevt%prt(this_jet), jet_out(this_jet), src(:n_src), pdg)
end do
do i = 1, size (jet_out)
   call jet_out(i)%final ()
end do
call jv_out%final ()
call cs%final ()
call jv_in%final ()
do i = 1, size (jet_in)
   call jet_in(i)%final ()
end do
end subroutine subevt_cluster
```

Return a list of all particles for which the mask is true.

⟨*Subevents: public*⟩+≡

```
public :: subevt_select
```

⟨*Subevents: procedures*⟩+≡

```
subroutine subevt_select (subevt, pl, mask1)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: pl
  logical, dimension(:), intent(in) :: mask1
  integer :: i, n
  call subevt_reset (subevt, pl%n_active)
  n = 0
  do i = 1, pl%n_active
     if (mask1(i)) then
        n = n + 1
        subevt%prt(n) = pl%prt(i)
     end if
  end do
  subevt%n_active = n
end subroutine subevt_select
```

Return a subevent which consists of the single particle with specified `index`. If `index` is negative, count from the end. If it is out of bounds, return an empty list.

⟨*Subevents: public*⟩+≡

```
public :: subevt_extract
```

⟨*Subevents: procedures*⟩+≡

```
subroutine subevt_extract (subevt, pl, index)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: pl
  integer, intent(in) :: index
  if (index > 0) then
     if (index <= pl%n_active) then
        call subevt_reset (subevt, 1)
        subevt%prt(1) = pl%prt(index)
     else
        call subevt_reset (subevt, 0)
     end if
  else if (index < 0) then
     if (abs (index) <= pl%n_active) then
        call subevt_reset (subevt, 1)
        subevt%prt(1) = pl%prt(pl%n_active + 1 + index)
     else
        call subevt_reset (subevt, 0)
     end if
  else
     call subevt_reset (subevt, 0)
  end if
end subroutine subevt_extract
```

Return the list of particles sorted according to increasing values of the provided integer or real array. If no array is given, sort by PDG value.

⟨*Subevents: public*⟩+≡

```
public :: subevt_sort
```

⟨*Subevents: interfaces*⟩+≡

```
interface subevt_sort
   module procedure subevt_sort_pdg
   module procedure subevt_sort_int
   module procedure subevt_sort_real
end interface
```

⟨*Subevents: procedures*⟩+≡

```
subroutine subevt_sort_pdg (subevt, pl)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: pl
  integer :: n
  n = subevt%n_active
  call subevt_sort_int (subevt, pl, abs (3 * subevt%prt(:n)%pdg - 1))
end subroutine subevt_sort_pdg

subroutine subevt_sort_int (subevt, pl, ival)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: pl
  integer, dimension(:), intent(in) :: ival
  call subevt_reset (subevt, pl%n_active)
  subevt%n_active = pl%n_active
  subevt%prt = pl%prt( order (ival) )
```

```
end subroutine subevt_sort_int

subroutine subevt_sort_real (subevt, pl, rval)
  type(subevt_t), intent(inout) :: subevt
  type(subevt_t), intent(in) :: pl
  real(default), dimension(:), intent(in) :: rval
  integer, dimension(size(rval)) :: idx
  integer :: i
  call subevt_reset (subevt, pl%n_active)
  subevt%n_active = pl%n_active
  if (allocated (subevt%prt))  deallocate (subevt%prt)
  allocate (subevt%prt (size(pl%prt)))
  !!! !!! !!! Workaround for the ifort 16.0/1
  !subevt%prt = pl%prt(order (rval))
  idx = order (rval)
  do i = 1, size(idx)
     subevt%prt(i) = pl%prt(idx(i))
  end do
end subroutine subevt_sort_real
```

Return the list of particles which have any of the specified PDG codes (and optionally particle type: beam, incoming, outgoing).

The `pack` command was buggy in some gfortran versions, therefore it is unrolled. The unrolled version may be more efficient, actually.

⟨*Subevents: public*⟩+≡

```
public :: subevt_select_pdg_code
```

⟨*Subevents: procedures*⟩+≡

```
subroutine subevt_select_pdg_code (subevt, aval, subevt_in, prt_type)
  type(subevt_t), intent(inout) :: subevt
  type(pdg_array_t), intent(in) :: aval
  type(subevt_t), intent(in) :: subevt_in
  integer, intent(in), optional :: prt_type
  integer :: n_active, n_match
  logical, dimension(:), allocatable :: mask
  integer :: i, j
  n_active = subevt_in%n_active
  allocate (mask (n_active))
  forall (i = 1:n_active) &
       mask(i) = aval .match. subevt_in%prt(i)%pdg
  if (present (prt_type)) &
       mask = mask .and. subevt_in%prt(:n_active)%type == prt_type
  n_match = count (mask)
  call subevt_reset (subevt, n_match)
  !!! !!! !!! Workaround for gfortran compiler bug
  ! subevt%prt(:n_match) = pack (subevt_in%prt(:n_active), mask)
  j = 0
  do i = 1, n_active
     if (mask(i)) then
        j = j + 1
        subevt%prt(j) = subevt_in%prt(i)
     end if
  end do
end subroutine subevt_select_pdg_code
```

12.4.3 Eliminate numerical noise

This is useful for testing purposes: set entries to zero that are smaller in absolute values than a given tolerance parameter.

Note: instead of setting the tolerance in terms of EPSILON (kind-dependent), we fix it to 10^{-16}, which is the typical value for double precision. The reason is that there are situations where intermediate representations (external libraries, files) are limited to double precision, even if the main program uses higher precision.

⟨*Subevents: public*⟩+≡
```
public :: pacify
```

⟨*Subevents: interfaces*⟩+≡
```
interface pacify
   module procedure pacify_prt
   module procedure pacify_subevt
end interface pacify
```

⟨*Subevents: procedures*⟩+≡
```
subroutine pacify_prt (prt)
  class(prt_t), intent(inout) :: prt
  real(default) :: e
  e = max (1E-10_default * energy (prt%p), 1E-13_default)
  call pacify (prt%p, e)
  call pacify (prt%p2, 1E3_default * e)
end subroutine pacify_prt

subroutine pacify_subevt (subevt)
  class(subevt_t), intent(inout) :: subevt
  integer :: i
  do i = 1, subevt%n_active
     call pacify (subevt%prt(i))
  end do
end subroutine pacify_subevt
```

12.5 Analysis tools

This module defines structures useful for data analysis. These include observables, histograms, and plots.

Observables are quantities that are calculated and summed up event by event. At the end, one can compute the average and error.

Histograms have their bins in addition to the observable properties. Histograms are usually written out in tables and displayed graphically.

In plots, each record creates its own entry in a table. This can be used for scatter plots if called event by event, or for plotting dependencies on parameters if called once per integration run.

Graphs are container for histograms and plots, which carry their own graphics options.

The type layout is still somewhat obfuscated. This would become much simpler if type extension could be used.

⟨analysis.f90⟩≡

```
⟨File header⟩

module analysis

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_utils, only: quote_underscore, tex_format
  use system_defs, only: TAB
  use diagnostics
  use os_interface
  use ifiles

⟨Standard module head⟩

⟨Analysis: public⟩

⟨Analysis: parameters⟩

⟨Analysis: types⟩

⟨Analysis: interfaces⟩

⟨Analysis: variables⟩

contains

⟨Analysis: procedures⟩

end module analysis
```

12.5.1 Output formats

These formats share a common field width (alignment).

⟨*Analysis: parameters*⟩≡

```
  character(*), parameter, public :: HISTOGRAM_HEAD_FORMAT = "1x,A15,3x"
```

```
character(*), parameter, public :: HISTOGRAM_INTG_FORMAT = "3x,I9,3x"
character(*), parameter, public :: HISTOGRAM_DATA_FORMAT = "ES19.12"
```

12.5.2 Graph options

These parameters are used for displaying data. They apply to a whole graph, which may contain more than one plot element.

The GAMELAN code chunks are part of both `graph_options` and `drawing_options`. The `drawing_options` copy is used in histograms and plots, also as graph elements. The `graph_options` copy is used for `graph` objects as a whole. Both copies are usually identical.

⟨*Analysis: public*⟩≡

```
public :: graph_options_t
```

⟨*Analysis: types*⟩≡

```
type :: graph_options_t
   private
   type(string_t) :: id
   type(string_t) :: title
   type(string_t) :: description
   type(string_t) :: x_label
   type(string_t) :: y_label
   integer :: width_mm = 130
   integer :: height_mm = 90
   logical :: x_log = .false.
   logical :: y_log = .false.
   real(default) :: x_min = 0
   real(default) :: x_max = 1
   real(default) :: y_min = 0
   real(default) :: y_max = 1
   logical :: x_min_set = .false.
   logical :: x_max_set = .false.
   logical :: y_min_set = .false.
   logical :: y_max_set = .false.
   type(string_t) :: gmlcode_bg
   type(string_t) :: gmlcode_fg
end type graph_options_t
```

Initialize the record, all strings are empty. The limits are undefined.

⟨*Analysis: public*⟩+≡

```
public :: graph_options_init
```

⟨*Analysis: procedures*⟩≡

```
subroutine graph_options_init (graph_options)
  type(graph_options_t), intent(out) :: graph_options
  graph_options%id = ""
  graph_options%title = ""
  graph_options%description = ""
  graph_options%x_label = ""
  graph_options%y_label = ""
  graph_options%gmlcode_bg = ""
  graph_options%gmlcode_fg = ""
```

```
end subroutine graph_options_init
```

Set individual options.

⟨*Analysis: public*⟩+≡

```
public :: graph_options_set
```

⟨*Analysis: procedures*⟩+≡

```
subroutine graph_options_set (graph_options, id, &
    title, description, x_label, y_label, width_mm, height_mm, &
    x_log, y_log, x_min, x_max, y_min, y_max, &
    gmlcode_bg, gmlcode_fg)
  type(graph_options_t), intent(inout) :: graph_options
  type(string_t), intent(in), optional :: id
  type(string_t), intent(in), optional :: title
  type(string_t), intent(in), optional :: description
  type(string_t), intent(in), optional :: x_label, y_label
  integer, intent(in), optional :: width_mm, height_mm
  logical, intent(in), optional :: x_log, y_log
  real(default), intent(in), optional :: x_min, x_max, y_min, y_max
  type(string_t), intent(in), optional :: gmlcode_bg, gmlcode_fg
  if (present (id))  graph_options%id = id
  if (present (title))  graph_options%title = title
  if (present (description))  graph_options%description = description
  if (present (x_label))  graph_options%x_label = x_label
  if (present (y_label))  graph_options%y_label = y_label
  if (present (width_mm))   graph_options%width_mm  = width_mm
  if (present (height_mm))  graph_options%height_mm = height_mm
  if (present (x_log))  graph_options%x_log = x_log
  if (present (y_log))  graph_options%y_log = y_log
  if (present (x_min))  graph_options%x_min = x_min
  if (present (x_max))  graph_options%x_max = x_max
  if (present (y_min))  graph_options%y_min = y_min
  if (present (y_max))  graph_options%y_max = y_max
  if (present (x_min))  graph_options%x_min_set = .true.
  if (present (x_max))  graph_options%x_max_set = .true.
  if (present (y_min))  graph_options%y_min_set = .true.
  if (present (y_max))  graph_options%y_max_set = .true.
  if (present (gmlcode_bg))  graph_options%gmlcode_bg = gmlcode_bg
  if (present (gmlcode_fg))  graph_options%gmlcode_fg = gmlcode_fg
end subroutine graph_options_set
```

Write a simple account of all options.

⟨*Analysis: public*⟩+≡

```
public :: graph_options_write
```

⟨*Analysis: procedures*⟩+≡

```
subroutine graph_options_write (gro, unit)
  type(graph_options_t), intent(in) :: gro
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
1 format (A,1x,'"',A,'"')
2 format (A,1x,L1)
3 format (A,1x,ES19.12)
```

```
4   format (A,1x,I0)
5   format (A,1x,'[undefined]')
    write (u, 1)  "title       =", char (gro%title)
    write (u, 1)  "description =", char (gro%description)
    write (u, 1)  "x_label     =", char (gro%x_label)
    write (u, 1)  "y_label     =", char (gro%y_label)
    write (u, 2)  "x_log       =", gro%x_log
    write (u, 2)  "y_log       =", gro%y_log
    if (gro%x_min_set) then
       write (u, 3)  "x_min       =", gro%x_min
    else
       write (u, 5)  "x_min       ="
    end if
    if (gro%x_max_set) then
       write (u, 3)  "x_max       =", gro%x_max
    else
       write (u, 5)  "x_max       ="
    end if
    if (gro%y_min_set) then
       write (u, 3)  "y_min       =", gro%y_min
    else
       write (u, 5)  "y_min       ="
    end if
    if (gro%y_max_set) then
       write (u, 3)  "y_max       =", gro%y_max
    else
       write (u, 5)  "y_max       ="
    end if
    write (u, 4)  "width_mm    =", gro%width_mm
    write (u, 4)  "height_mm   =", gro%height_mm
    write (u, 1)  "gmlcode_bg  =", char (gro%gmlcode_bg)
    write (u, 1)  "gmlcode_fg  =", char (gro%gmlcode_fg)
  end subroutine graph_options_write
```

Write a LaTeX header/footer for the analysis file.

⟨*Analysis: procedures*⟩+≡

```
  subroutine graph_options_write_tex_header (gro, unit)
    type(graph_options_t), intent(in) :: gro
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    if (gro%title /= "") then
       write (u, "(A)")
       write (u, "(A)")  "\section{" // char (gro%title) // "}"
    else
       write (u, "(A)")  "\section{" // char (quote_underscore (gro%id)) // "}"
    end if
    if (gro%description /= "") then
       write (u, "(A)")  char (gro%description)
       write (u, *)
       write (u, "(A)")  "\vspace*{\baselineskip}"
    end if
    write (u, "(A)")  "\vspace*{\baselineskip}"
```

```
    write (u, "(A)")  "\unitlength 1mm"
    write (u, "(A,I0,',',I0,A)")  &
         "\begin{gmlgraph*}(", &
         gro%width_mm, gro%height_mm, &
         ")[dat]"
  end subroutine graph_options_write_tex_header

  subroutine graph_options_write_tex_footer (gro, unit)
    type(graph_options_t), intent(in) :: gro
    integer, intent(in), optional :: unit
    integer :: u, width, height
    width = gro%width_mm - 10
    height = gro%height_mm - 10
    u = given_output_unit (unit)
    write (u, "(A)")  "  begingmleps ""Whizard-Logo.eps"";"
    write (u, "(A,I0,A,I0,A)")  &
         "    base := (", width, "*unitlength,", height, "*unitlength);"
    write (u, "(A)")  "    height := 9.6*unitlength;"
    write (u, "(A)")  "    width := 11.2*unitlength;"
    write (u, "(A)")  "  endgmleps;"
    write (u, "(A)")  "\end{gmlgraph*}"
  end subroutine graph_options_write_tex_footer
```

Return the analysis object ID.

⟨*Analysis: procedures*⟩+≡

```
  function graph_options_get_id (gro) result (id)
    type(string_t) :: id
    type(graph_options_t), intent(in) :: gro
    id = gro%id
  end function graph_options_get_id
```

Create an appropriate `setup` command (linear/log).

⟨*Analysis: procedures*⟩+≡

```
  function graph_options_get_gml_setup (gro) result (cmd)
    type(string_t) :: cmd
    type(graph_options_t), intent(in) :: gro
    type(string_t) :: x_str, y_str
    if (gro%x_log) then
       x_str = "log"
    else
       x_str = "linear"
    end if
    if (gro%y_log) then
       y_str = "log"
    else
       y_str = "linear"
    end if
    cmd = "setup (" // x_str // ", " // y_str // ");"
  end function graph_options_get_gml_setup
```

Return the labels in GAMELAN form.

⟨*Analysis: procedures*⟩+≡

```
function graph_options_get_gml_x_label (gro) result (cmd)
  type(string_t) :: cmd
  type(graph_options_t), intent(in) :: gro
  cmd = 'label.bot (<' // '<' // gro%x_label // '>' // '>, out);'
end function graph_options_get_gml_x_label

function graph_options_get_gml_y_label (gro) result (cmd)
  type(string_t) :: cmd
  type(graph_options_t), intent(in) :: gro
  cmd = 'label.ulft (<' // '<' // gro%y_label // '>' // '>, out);'
end function graph_options_get_gml_y_label
```

Create an appropriate `graphrange` statement for the given graph options. Where the graph options are not set, use the supplied arguments, if any, otherwise set the undefined value.

⟨*Analysis: procedures*⟩+≡

```
function graph_options_get_gml_graphrange &
     (gro, x_min, x_max, y_min, y_max) result (cmd)
  type(string_t) :: cmd
  type(graph_options_t), intent(in) :: gro
  real(default), intent(in), optional :: x_min, x_max, y_min, y_max
  type(string_t) :: x_min_str, x_max_str, y_min_str, y_max_str
  character(*), parameter :: fmt = "(ES15.8)"
  if (gro%x_min_set) then
     x_min_str = "#" // trim (adjustl (real2string (gro%x_min, fmt)))
  else if (present (x_min)) then
     x_min_str = "#" // trim (adjustl (real2string (x_min, fmt)))
  else
     x_min_str = "??"
  end if
  if (gro%x_max_set) then
     x_max_str = "#" // trim (adjustl (real2string (gro%x_max, fmt)))
  else if (present (x_max)) then
     x_max_str = "#" // trim (adjustl (real2string (x_max, fmt)))
  else
     x_max_str = "??"
  end if
  if (gro%y_min_set) then
     y_min_str = "#" // trim (adjustl (real2string (gro%y_min, fmt)))
  else if (present (y_min)) then
     y_min_str = "#" // trim (adjustl (real2string (y_min, fmt)))
  else
     y_min_str = "??"
  end if
  if (gro%y_max_set) then
     y_max_str = "#" // trim (adjustl (real2string (gro%y_max, fmt)))
  else if (present (y_max)) then
     y_max_str = "#" // trim (adjustl (real2string (y_max, fmt)))
  else
     y_max_str = "??"
  end if
  cmd = "graphrange (" // x_min_str // ", " // y_min_str // "), " &
       // "(" // x_max_str // ", " // y_max_str // ");"
```

```
end function graph_options_get_gml_graphrange
```

Get extra GAMELAN code to be executed before and after the usual drawing commands.

⟨*Analysis: procedures*⟩+≡

```
function graph_options_get_gml_bg_command (gro) result (cmd)
  type(string_t) :: cmd
  type(graph_options_t), intent(in) :: gro
  cmd = gro%gmlcode_bg
end function graph_options_get_gml_bg_command

function graph_options_get_gml_fg_command (gro) result (cmd)
  type(string_t) :: cmd
  type(graph_options_t), intent(in) :: gro
  cmd = gro%gmlcode_fg
end function graph_options_get_gml_fg_command
```

Append the header for generic data output in ifile format. We print only labels, not graphics parameters.

⟨*Analysis: procedures*⟩+≡

```
subroutine graph_options_get_header (pl, header, comment)
  type(graph_options_t), intent(in) :: pl
  type(ifile_t), intent(inout) :: header
  type(string_t), intent(in), optional :: comment
  type(string_t) :: c
  if (present (comment)) then
     c = comment
  else
     c = ""
  end if
  call ifile_append (header, &
       c // "ID: " // pl%id)
  call ifile_append (header, &
       c // "title: " // pl%title)
  call ifile_append (header, &
       c // "description: " // pl%description)
  call ifile_append (header, &
       c // "x axis label: " // pl%x_label)
  call ifile_append (header, &
       c // "y axis label: " // pl%y_label)
end subroutine graph_options_get_header
```

12.5.3 Drawing options

These options apply to an individual graph element (histogram or plot).

⟨*Analysis: public*⟩+≡

```
public :: drawing_options_t
```

⟨*Analysis: types*⟩+≡

```
type :: drawing_options_t
   type(string_t) :: dataset
   logical :: with_hbars = .false.
```

```
    logical :: with_base = .false.
    logical :: piecewise = .false.
    logical :: fill = .false.
    logical :: draw = .false.
    logical :: err = .false.
    logical :: symbols = .false.
    type(string_t) :: fill_options
    type(string_t) :: draw_options
    type(string_t) :: err_options
    type(string_t) :: symbol
    type(string_t) :: gmlcode_bg
    type(string_t) :: gmlcode_fg
end type drawing_options_t
```

Write a simple account of all options.

⟨*Analysis: public*⟩+≡

```
public :: drawing_options_write
```

⟨*Analysis: procedures*⟩+≡

```
subroutine drawing_options_write (dro, unit)
    type(drawing_options_t), intent(in) :: dro
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
1 format (A,1x,'"',A,'"')
2 format (A,1x,L1)
  write (u, 2)  "with_hbars  =", dro%with_hbars
  write (u, 2)  "with_base   =", dro%with_base
  write (u, 2)  "piecewise   =", dro%piecewise
  write (u, 2)  "fill        =", dro%fill
  write (u, 2)  "draw        =", dro%draw
  write (u, 2)  "err         =", dro%err
  write (u, 2)  "symbols     =", dro%symbols
  write (u, 1)  "fill_options=", char (dro%fill_options)
  write (u, 1)  "draw_options=", char (dro%draw_options)
  write (u, 1)  "err_options =", char (dro%err_options)
  write (u, 1)  "symbol      =", char (dro%symbol)
  write (u, 1)  "gmlcode_bg  =", char (dro%gmlcode_bg)
  write (u, 1)  "gmlcode_fg  =", char (dro%gmlcode_fg)
end subroutine drawing_options_write
```

Init with empty strings and default options, appropriate for either histogram or plot.

⟨*Analysis: public*⟩+≡

```
public :: drawing_options_init_histogram
public :: drawing_options_init_plot
```

⟨*Analysis: procedures*⟩+≡

```
subroutine drawing_options_init_histogram (dro)
  type(drawing_options_t), intent(out) :: dro
  dro%dataset = "dat"
  dro%with_hbars = .true.
  dro%with_base = .true.
  dro%piecewise = .true.
```

```
    dro%fill = .true.
    dro%draw = .true.
    dro%fill_options = "withcolor col.default"
    dro%draw_options = ""
    dro%err_options = ""
    dro%symbol = "fshape(circle scaled 1mm)()"
    dro%gmlcode_bg = ""
    dro%gmlcode_fg = ""
  end subroutine drawing_options_init_histogram

  subroutine drawing_options_init_plot (dro)
    type(drawing_options_t), intent(out) :: dro
    dro%dataset = "dat"
    dro%draw = .true.
    dro%fill_options = "withcolor col.default"
    dro%draw_options = ""
    dro%err_options = ""
    dro%symbol = "fshape(circle scaled 1mm)()"
    dro%gmlcode_bg = ""
    dro%gmlcode_fg = ""
  end subroutine drawing_options_init_plot
```

Set individual options.

⟨*Analysis: public*⟩+≡

```
  public :: drawing_options_set
```

⟨*Analysis: procedures*⟩+≡

```
  subroutine drawing_options_set (dro, dataset, &
      with_hbars, with_base, piecewise, fill, draw, err, symbols, &
      fill_options, draw_options, err_options, symbol, &
      gmlcode_bg, gmlcode_fg)
    type(drawing_options_t), intent(inout) :: dro
    type(string_t), intent(in), optional :: dataset
    logical, intent(in), optional :: with_hbars, with_base, piecewise
    logical, intent(in), optional :: fill, draw, err, symbols
    type(string_t), intent(in), optional :: fill_options, draw_options
    type(string_t), intent(in), optional :: err_options, symbol
    type(string_t), intent(in), optional :: gmlcode_bg, gmlcode_fg
    if (present (dataset))  dro%dataset = dataset
    if (present (with_hbars))  dro%with_hbars = with_hbars
    if (present (with_base))  dro%with_base = with_base
    if (present (piecewise))  dro%piecewise = piecewise
    if (present (fill))  dro%fill = fill
    if (present (draw))  dro%draw = draw
    if (present (err))  dro%err = err
    if (present (symbols))  dro%symbols = symbols
    if (present (fill_options))  dro%fill_options = fill_options
    if (present (draw_options))  dro%draw_options = draw_options
    if (present (err_options))  dro%err_options = err_options
    if (present (symbol))  dro%symbol = symbol
    if (present (gmlcode_bg))  dro%gmlcode_bg = gmlcode_bg
    if (present (gmlcode_fg))  dro%gmlcode_fg = gmlcode_fg
  end subroutine drawing_options_set
```

There are sepate commands for drawing the curve and for drawing errors. The symbols are applied to the latter. First of all, we may have to compute a baseline:

⟨*Analysis: procedures*⟩+≡

```
function drawing_options_get_calc_command (dro) result (cmd)
  type(string_t) :: cmd
  type(drawing_options_t), intent(in) :: dro
  if (dro%with_base) then
     cmd = "calculate " // dro%dataset // ".base (" // dro%dataset // ") " &
          // "(x, #0);"
  else
     cmd = ""
  end if
end function drawing_options_get_calc_command
```

Return the drawing command.

⟨*Analysis: procedures*⟩+≡

```
function drawing_options_get_draw_command (dro) result (cmd)
  type(string_t) :: cmd
  type(drawing_options_t), intent(in) :: dro
  if (dro%fill) then
     cmd = "fill"
  else if (dro%draw) then
     cmd = "draw"
  else
     cmd = ""
  end if
  if (dro%fill .or. dro%draw) then
     if (dro%piecewise)  cmd = cmd // " piecewise"
     if (dro%draw .and. dro%with_base)  cmd = cmd // " cyclic"
     cmd = cmd // " from (" // dro%dataset
     if (dro%with_base) then
        if (dro%piecewise) then
           cmd = cmd // ", " // dro%dataset // ".base/\"  ! "
        else
           cmd = cmd // " ~ " // dro%dataset // ".base\"  ! "
        end if
     end if
     cmd = cmd // ")"
     if (dro%fill) then
        cmd = cmd // " " // dro%fill_options
        if (dro%draw)  cmd = cmd // " outlined"
     end if
     if (dro%draw)  cmd = cmd // " " // dro%draw_options
     cmd = cmd // ";"
  end if
end function drawing_options_get_draw_command
```

The error command draws error bars, if any.

⟨*Analysis: procedures*⟩+≡

```
function drawing_options_get_err_command (dro) result (cmd)
  type(string_t) :: cmd
  type(drawing_options_t), intent(in) :: dro
```

```
    if (dro%err) then
       cmd = "draw piecewise " &
            // "from (" // dro%dataset // ".err)" &
            // " " // dro%err_options // ";"
    else
       cmd = ""
    end if
  end function drawing_options_get_err_command
```

The symbol command draws symbols, if any.

⟨*Analysis: procedures*⟩+≡

```
  function drawing_options_get_symb_command (dro) result (cmd)
    type(string_t) :: cmd
    type(drawing_options_t), intent(in) :: dro
    if (dro%symbols) then
       cmd = "phantom" &
            // " from (" // dro%dataset // ")" &
            // " withsymbol (" // dro%symbol // ");"
    else
       cmd = ""
    end if
  end function drawing_options_get_symb_command
```

Get extra GAMELAN code to be executed before and after the usual drawing commands.

⟨*Analysis: procedures*⟩+≡

```
  function drawing_options_get_gml_bg_command (dro) result (cmd)
    type(string_t) :: cmd
    type(drawing_options_t), intent(in) :: dro
    cmd = dro%gmlcode_bg
  end function drawing_options_get_gml_bg_command

  function drawing_options_get_gml_fg_command (dro) result (cmd)
    type(string_t) :: cmd
    type(drawing_options_t), intent(in) :: dro
    cmd = dro%gmlcode_fg
  end function drawing_options_get_gml_fg_command
```

12.5.4 Observables

The observable type holds the accumulated observable values and weight sums which are necessary for proper averaging.

⟨*Analysis: types*⟩+≡

```
  type :: observable_t
     private
     real(default) :: sum_values = 0
     real(default) :: sum_squared_values = 0
     real(default) :: sum_weights = 0
     real(default) :: sum_squared_weights = 0
     integer :: count = 0
     type(string_t) :: obs_label
```

```
      type(string_t) :: obs_unit
      type(graph_options_t) :: graph_options
   end type observable_t
```

Initialize with defined properties

⟨*Analysis: procedures*⟩+≡

```
   subroutine observable_init (obs, obs_label, obs_unit, graph_options)
     type(observable_t), intent(out) :: obs
     type(string_t), intent(in), optional :: obs_label, obs_unit
     type(graph_options_t), intent(in), optional :: graph_options
     if (present (obs_label)) then
        obs%obs_label = obs_label
     else
        obs%obs_label = ""
     end if
     if (present (obs_unit)) then
        obs%obs_unit = obs_unit
     else
        obs%obs_unit = ""
     end if
     if (present (graph_options)) then
        obs%graph_options = graph_options
     else
        call graph_options_init (obs%graph_options)
     end if
   end subroutine observable_init
```

Reset all numeric entries.

⟨*Analysis: procedures*⟩+≡

```
   subroutine observable_clear (obs)
     type(observable_t), intent(inout) :: obs
     obs%sum_values = 0
     obs%sum_squared_values = 0
     obs%sum_weights = 0
     obs%sum_squared_weights = 0
     obs%count = 0
   end subroutine observable_clear
```

Record a a value. Always successful for observables.

⟨*Analysis: interfaces*⟩≡

```
   interface observable_record_value
      module procedure observable_record_value_unweighted
      module procedure observable_record_value_weighted
   end interface
```

⟨*Analysis: procedures*⟩+≡

```
   subroutine observable_record_value_unweighted (obs, value, success)
     type(observable_t), intent(inout) :: obs
     real(default), intent(in) :: value
     logical, intent(out), optional :: success
     obs%sum_values = obs%sum_values + value
     obs%sum_squared_values = obs%sum_squared_values + value**2
```

```
  obs%sum_weights = obs%sum_weights + 1
  obs%sum_squared_weights = obs%sum_squared_weights + 1
  obs%count = obs%count + 1
  if (present (success))  success = .true.
end subroutine observable_record_value_unweighted

subroutine observable_record_value_weighted (obs, value, weight, success)
  type(observable_t), intent(inout) :: obs
  real(default), intent(in) :: value, weight
  logical, intent(out), optional :: success
  obs%sum_values = obs%sum_values + value * weight
  obs%sum_squared_values = obs%sum_squared_values + value**2 * weight
  obs%sum_weights = obs%sum_weights + abs (weight)
  obs%sum_squared_weights = obs%sum_squared_weights + weight**2
  obs%count = obs%count + 1
  if (present (success))  success = .true.
end subroutine observable_record_value_weighted
```

Here are the statistics formulas:

1. Unweighted case: Given a sample of n values x_i, the average is

$$\langle x \rangle = \frac{\sum x_i}{n} \tag{12.4}$$

and the error estimate

$$\Delta x = \sqrt{\frac{1}{n-1}\langle \sum (x_i - \langle x \rangle)^2} \tag{12.5}$$

$$= \sqrt{\frac{1}{n-1}\left(\frac{\sum x_i^2}{n} - \frac{(\sum x_i)^2}{n^2}\right)} \tag{12.6}$$

2. Weighted case: Instead of weight 1, each event comes with weight w_i.

$$\langle x \rangle = \frac{\sum x_i w_i}{\sum w_i} \tag{12.7}$$

and

$$\Delta x = \sqrt{\frac{1}{n-1}\left(\frac{\sum x_i^2 w_i}{\sum w_i} - \frac{(\sum x_i w_i)^2}{(\sum w_i)^2}\right)} \tag{12.8}$$

For $w_i = 1$, this specializes to the previous formula.

⟨*Analysis: procedures*⟩+≡

```
function observable_get_n_entries (obs) result (n)
  integer :: n
  type(observable_t), intent(in) :: obs
  n = obs%count
end function observable_get_n_entries

function observable_get_average (obs) result (avg)
  real(default) :: avg
  type(observable_t), intent(in) :: obs
```

```
  if (obs%sum_weights /= 0) then
     avg = obs%sum_values / obs%sum_weights
  else
     avg = 0
  end if
end function observable_get_average

function observable_get_error (obs) result (err)
  real(default) :: err
  type(observable_t), intent(in) :: obs
  real(default) :: var, n
  if (obs%sum_weights /= 0) then
     select case (obs%count)
     case (0:1)
        err = 0
     case default
        n = obs%count
        var = obs%sum_squared_values / obs%sum_weights &
              - (obs%sum_values / obs%sum_weights) ** 2
        err = sqrt (max (var, 0._default) / (n - 1))
     end select
  else
     err = 0
  end if
end function observable_get_error
```

Write label and/or physical unit to a string.

⟨*Analysis: procedures*⟩+≡

```
function observable_get_label (obs, wl, wu) result (string)
  type(string_t) :: string
  type(observable_t), intent(in) :: obs
  logical, intent(in) :: wl, wu
  type(string_t) :: obs_label, obs_unit
  if (wl) then
     if (obs%obs_label /= "") then
        obs_label = obs%obs_label
     else
        obs_label = "\textrm{Observable}"
     end if
  else
     obs_label = ""
  end if
  if (wu) then
     if (obs%obs_unit /= "") then
        if (wl) then
           obs_unit = "\;[" // obs%obs_unit // "]"
        else
           obs_unit = obs%obs_unit
        end if
     else
        obs_unit = ""
     end if
  else
```

```
      obs_unit = ""
    end if
    string = obs_label // obs_unit
  end function observable_get_label
```

12.5.5 Output

⟨*Analysis: procedures*⟩+≡

```
  subroutine observable_write (obs, unit)
    type(observable_t), intent(in) :: obs
    integer, intent(in), optional :: unit
    real(default) :: avg, err, relerr
    integer :: n
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    avg = observable_get_average (obs)
    err = observable_get_error (obs)
    if (avg /= 0) then
       relerr = err / abs (avg)
    else
       relerr = 0
    end if
    n = observable_get_n_entries (obs)
    if (obs%graph_options%title /= "") then
       write (u, "(A,1x,3A)") &
            "title       =", '"', char (obs%graph_options%title), '"'
    end if
    if (obs%graph_options%title /= "") then
       write (u, "(A,1x,3A)") &
            "description =", '"', char (obs%graph_options%description), '"'
    end if
    write (u, "(A,1x," // HISTOGRAM_DATA_FORMAT // ")", advance = "no") &
         "average     =", avg
    call write_unit ()
    write (u, "(A,1x," // HISTOGRAM_DATA_FORMAT // ")", advance = "no") &
         "error[abs]  =", err
    call write_unit ()
    write (u, "(A,1x," // HISTOGRAM_DATA_FORMAT // ")") &
         "error[rel]  =", relerr
    write (u, "(A,1x,I0)") &
         "n_entries   =", n
  contains
    subroutine write_unit ()
      if (obs%obs_unit /= "") then
         write (u, "(1x,A)")  char (obs%obs_unit)
      else
         write (u, *)
      end if
    end subroutine write_unit
  end subroutine observable_write
```

LaTeX output.

⟨*Analysis: procedures*⟩+≡

```
subroutine observable_write_driver (obs, unit, write_heading)
  type(observable_t), intent(in) :: obs
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: write_heading
  real(default) :: avg, err
  integer :: n_digits
  logical :: heading
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  heading = .true.;  if (present (write_heading))  heading = write_heading
  avg = observable_get_average (obs)
  err = observable_get_error (obs)
  if (avg /= 0 .and. err /= 0) then
     n_digits = max (2, 2 - int (log10 (abs (err / real (avg, default)))))
  else if (avg /= 0) then
     n_digits = 100
  else
     n_digits = 1
  end if
  if (heading) then
     write (u, "(A)")
     if (obs%graph_options%title /= "") then
        write (u, "(A)")  "\section{" // char (obs%graph_options%title) &
             // "}"
     else
        write (u, "(A)")  "\section{Observable}"
     end if
     if (obs%graph_options%description /= "") then
        write (u, "(A)")  char (obs%graph_options%description)
        write (u, *)
     end if
     write (u, "(A)")  "\begin{flushleft}"
  end if
  write (u, "(A)", advance="no")  "  $\langle{" ! $ sign
  write (u, "(A)", advance="no")  char (observable_get_label (obs, wl=.true., wu=.false.))
  write (u, "(A)", advance="no")  "}\rangle = "
  write (u, "(A)", advance="no")  char (tex_format (avg, n_digits))
  write (u, "(A)", advance="no")  "\pm"
  write (u, "(A)", advance="no")  char (tex_format (err, 2))
  write (u, "(A)", advance="no")  "\;{"
  write (u, "(A)", advance="no")  char (observable_get_label (obs, wl=.false., wu=.true.))
  write (u, "(A)")  "}"
  write (u, "(A)", advance="no")  "     \quad[n_{\text{entries}} = "
  write (u, "(I0)",advance="no")  observable_get_n_entries (obs)
  write (u, "(A)")  "]$"          ! $ fool Emacs' noweb mode
  if (heading) then
     write (u, "(A)") "\end{flushleft}"
  end if
end subroutine observable_write_driver
```

12.5.6 Histograms

Bins

⟨*Analysis: types*⟩+≡

```
type :: bin_t
   private
   real(default) :: midpoint = 0
   real(default) :: width = 0
   real(default) :: sum_weights = 0
   real(default) :: sum_squared_weights = 0
   real(default) :: sum_excess_weights = 0
   integer :: count = 0
end type bin_t
```

⟨*Analysis: procedures*⟩+≡

```
subroutine bin_init (bin, midpoint, width)
  type(bin_t), intent(out) :: bin
  real(default), intent(in) :: midpoint, width
  bin%midpoint = midpoint
  bin%width = width
end subroutine bin_init
```

⟨*Analysis: procedures*⟩+≡

```
elemental subroutine bin_clear (bin)
  type(bin_t), intent(inout) :: bin
  bin%sum_weights = 0
  bin%sum_squared_weights = 0
  bin%sum_excess_weights = 0
  bin%count = 0
end subroutine bin_clear
```

⟨*Analysis: procedures*⟩+≡

```
subroutine bin_record_value (bin, normalize, weight, excess)
  type(bin_t), intent(inout) :: bin
  logical, intent(in) :: normalize
  real(default), intent(in) :: weight
  real(default), intent(in), optional :: excess
  real(default) :: w, e
  if (normalize) then
     if (bin%width /= 0) then
        w = weight / bin%width
        if (present (excess))  e = excess / bin%width
     else
        w = 0
        if (present (excess))  e = 0
     end if
  else
     w = weight
     if (present (excess))  e = excess
  end if
  bin%sum_weights = bin%sum_weights + abs (w)
  bin%sum_squared_weights = bin%sum_squared_weights + w ** 2
  if (present (excess)) &
```

```
        bin%sum_excess_weights = bin%sum_excess_weights + abs (e)
    bin%count = bin%count + 1
  end subroutine bin_record_value
```

⟨*Analysis: procedures*⟩+≡

```
  function bin_get_midpoint (bin) result (x)
    real(default) :: x
    type(bin_t), intent(in) :: bin
    x = bin%midpoint
  end function bin_get_midpoint

  function bin_get_width (bin) result (w)
    real(default) :: w
    type(bin_t), intent(in) :: bin
    w = bin%width
  end function bin_get_width

  function bin_get_n_entries (bin) result (n)
    integer :: n
    type(bin_t), intent(in) :: bin
    n = bin%count
  end function bin_get_n_entries

  function bin_get_sum (bin) result (s)
    real(default) :: s
    type(bin_t), intent(in) :: bin
    s = bin%sum_weights
  end function bin_get_sum

  function bin_get_error (bin) result (err)
    real(default) :: err
    type(bin_t), intent(in) :: bin
    err = sqrt (bin%sum_squared_weights)
  end function bin_get_error

  function bin_get_excess (bin) result (excess)
    real(default) :: excess
    type(bin_t), intent(in) :: bin
    excess = bin%sum_excess_weights
  end function bin_get_excess
```

⟨*Analysis: procedures*⟩+≡

```
  subroutine bin_write_header (unit)
    integer, intent(in), optional :: unit
    character(120) :: buffer
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    write (buffer, "(A,4(1x," //HISTOGRAM_HEAD_FORMAT // "),2x,A)") &
         "#", "bin midpoint", "value    ", "error    ", &
         "excess     ", "n"
    write (u, "(A)")  trim (buffer)
  end subroutine bin_write_header
```

```
subroutine bin_write (bin, unit)
  type(bin_t), intent(in) :: bin
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,4(1x," // HISTOGRAM_DATA_FORMAT // "),2x,I0)") &
       bin_get_midpoint (bin), &
       bin_get_sum (bin), &
       bin_get_error (bin), &
       bin_get_excess (bin), &
       bin_get_n_entries (bin)
end subroutine bin_write
```

Histograms

⟨*Analysis: types*⟩+≡

```
type :: histogram_t
   private
   real(default) :: lower_bound = 0
   real(default) :: upper_bound = 0
   real(default) :: width = 0
   integer :: n_bins = 0
   logical :: normalize_bins = .false.
   type(observable_t) :: obs
   type(observable_t) :: obs_within_bounds
   type(bin_t) :: underflow
   type(bin_t), dimension(:), allocatable :: bin
   type(bin_t) :: overflow
   type(graph_options_t) :: graph_options
   type(drawing_options_t) :: drawing_options
end type histogram_t
```

Initializer/finalizer

Initialize a histogram. We may provide either the bin width or the number of bins. A finalizer is not needed, since the histogram contains no pointer (sub)components.

⟨*Analysis: interfaces*⟩+≡

```
interface histogram_init
   module procedure histogram_init_n_bins
   module procedure histogram_init_bin_width
end interface
```

⟨*Analysis: procedures*⟩+≡

```
subroutine histogram_init_n_bins (h, id, &
     lower_bound, upper_bound, n_bins, normalize_bins, &
     obs_label, obs_unit, graph_options, drawing_options)
  type(histogram_t), intent(out) :: h
  type(string_t), intent(in) :: id
  real(default), intent(in) :: lower_bound, upper_bound
```

```
  integer, intent(in) :: n_bins
  logical, intent(in) :: normalize_bins
  type(string_t), intent(in), optional :: obs_label, obs_unit
  type(graph_options_t), intent(in), optional :: graph_options
  type(drawing_options_t), intent(in), optional :: drawing_options
  real(default) :: bin_width
  integer :: i
  call observable_init (h%obs_within_bounds, obs_label, obs_unit)
  call observable_init (h%obs, obs_label, obs_unit)
  h%lower_bound = lower_bound
  h%upper_bound = upper_bound
  h%n_bins = max (n_bins, 1)
  h%width = h%upper_bound - h%lower_bound
  h%normalize_bins = normalize_bins
  bin_width = h%width / h%n_bins
  allocate (h%bin (h%n_bins))
  call bin_init (h%underflow, h%lower_bound, 0._default)
  do i = 1, h%n_bins
     call bin_init (h%bin(i), &
          h%lower_bound - bin_width/2 + i * bin_width, bin_width)
  end do
  call bin_init (h%overflow, h%upper_bound, 0._default)
  if (present (graph_options)) then
     h%graph_options = graph_options
  else
     call graph_options_init (h%graph_options)
  end if
  call graph_options_set (h%graph_options, id = id)
  if (present (drawing_options)) then
     h%drawing_options = drawing_options
  else
     call drawing_options_init_histogram (h%drawing_options)
  end if
end subroutine histogram_init_n_bins

subroutine histogram_init_bin_width (h, id, &
     lower_bound, upper_bound, bin_width, normalize_bins, &
     obs_label, obs_unit, graph_options, drawing_options)
  type(histogram_t), intent(out) :: h
  type(string_t), intent(in) :: id
  real(default), intent(in) :: lower_bound, upper_bound, bin_width
  logical, intent(in) :: normalize_bins
  type(string_t), intent(in), optional :: obs_label, obs_unit
  type(graph_options_t), intent(in), optional :: graph_options
  type(drawing_options_t), intent(in), optional :: drawing_options
  integer :: n_bins
  if (bin_width /= 0) then
     n_bins = nint ((upper_bound - lower_bound) / bin_width)
  else
     n_bins = 1
  end if
  call histogram_init_n_bins (h, id, &
       lower_bound, upper_bound, n_bins, normalize_bins, &
       obs_label, obs_unit, graph_options, drawing_options)
```

```
end subroutine histogram_init_bin_width
```

Initialize a histogram by copying another one.

Since h has no pointer (sub)components, intrinsic assignment is sufficient. Optionally, we replace the drawing options.

⟨*Analysis: procedures*⟩+≡

```
subroutine histogram_init_histogram (h, h_in, drawing_options)
  type(histogram_t), intent(out) :: h
  type(histogram_t), intent(in) :: h_in
  type(drawing_options_t), intent(in), optional :: drawing_options
  h = h_in
  if (present (drawing_options)) then
     h%drawing_options = drawing_options
  end if
end subroutine histogram_init_histogram
```

Fill histograms

Clear the histogram contents, but do not modify the structure.

⟨*Analysis: procedures*⟩+≡

```
subroutine histogram_clear (h)
  type(histogram_t), intent(inout) :: h
  call observable_clear (h%obs)
  call observable_clear (h%obs_within_bounds)
  call bin_clear (h%underflow)
  if (allocated (h%bin))  call bin_clear (h%bin)
  call bin_clear (h%overflow)
end subroutine histogram_clear
```

Record a value. Successful if the value is within bounds, otherwise it is recorded as under-/overflow. Optionally, we may provide an excess weight that could be returned by the unweighting procedure.

⟨*Analysis: procedures*⟩+≡

```
subroutine histogram_record_value_unweighted (h, value, excess, success)
  type(histogram_t), intent(inout) :: h
  real(default), intent(in) :: value
  real(default), intent(in), optional :: excess
  logical, intent(out), optional :: success
  integer :: i_bin
  call observable_record_value (h%obs, value)
  if (h%width /= 0) then
     i_bin = floor (((value - h%lower_bound) / h%width) * h%n_bins) + 1
  else
     i_bin = 0
  end if
  if (i_bin <= 0) then
     call bin_record_value (h%underflow, .false., 1._default, excess)
     if (present (success))  success = .false.
  else if (i_bin <= h%n_bins) then
     call observable_record_value (h%obs_within_bounds, value)
     call bin_record_value &
```

```
            (h%bin(i_bin), h%normalize_bins, 1._default, excess)
         if (present (success))  success = .true.
      else
         call bin_record_value (h%overflow, .false., 1._default, excess)
         if (present (success))  success = .false.
      end if
   end subroutine histogram_record_value_unweighted
```

Weighted events: analogous, but no excess weight.

⟨*Analysis: procedures*⟩+≡

```
   subroutine histogram_record_value_weighted (h, value, weight, success)
      type(histogram_t), intent(inout) :: h
      real(default), intent(in) :: value, weight
      logical, intent(out), optional :: success
      integer :: i_bin
      call observable_record_value (h%obs, value, weight)
      if (h%width /= 0) then
         i_bin = floor (((value - h%lower_bound) / h%width) * h%n_bins) + 1
      else
         i_bin = 0
      end if
      if (i_bin <= 0) then
         call bin_record_value (h%underflow, .false., weight)
         if (present (success))  success = .false.
      else if (i_bin <= h%n_bins) then
         call observable_record_value (h%obs_within_bounds, value, weight)
         call bin_record_value (h%bin(i_bin), h%normalize_bins, weight)
         if (present (success))  success = .true.
      else
         call bin_record_value (h%overflow, .false., weight)
         if (present (success))  success = .false.
      end if
   end subroutine histogram_record_value_weighted
```

Access contents

Inherited from the observable component (all-over average etc.)

⟨*Analysis: procedures*⟩+≡

```
   function histogram_get_n_entries (h) result (n)
      integer :: n
      type(histogram_t), intent(in) :: h
      n = observable_get_n_entries (h%obs)
   end function histogram_get_n_entries

   function histogram_get_average (h) result (avg)
      real(default) :: avg
      type(histogram_t), intent(in) :: h
      avg = observable_get_average (h%obs)
   end function histogram_get_average

   function histogram_get_error (h) result (err)
      real(default) :: err
```

```
    type(histogram_t), intent(in) :: h
    err = observable_get_error (h%obs)
  end function histogram_get_error
```

Analogous, but applied only to events within bounds.

⟨*Analysis: procedures*⟩+≡

```
  function histogram_get_n_entries_within_bounds (h) result (n)
    integer :: n
    type(histogram_t), intent(in) :: h
    n = observable_get_n_entries (h%obs_within_bounds)
  end function histogram_get_n_entries_within_bounds

  function histogram_get_average_within_bounds (h) result (avg)
    real(default) :: avg
    type(histogram_t), intent(in) :: h
    avg = observable_get_average (h%obs_within_bounds)
  end function histogram_get_average_within_bounds

  function histogram_get_error_within_bounds (h) result (err)
    real(default) :: err
    type(histogram_t), intent(in) :: h
    err = observable_get_error (h%obs_within_bounds)
  end function histogram_get_error_within_bounds
```

Get the number of bins

⟨*Analysis: procedures*⟩+≡

```
  function histogram_get_n_bins (h) result (n)
    type(histogram_t), intent(in) :: h
    integer :: n
    n = h%n_bins
  end function histogram_get_n_bins
```

Check bins. If the index is zero or above the limit, return the results for underflow or overflow, respectively.

⟨*Analysis: procedures*⟩+≡

```
  function histogram_get_n_entries_for_bin (h, i) result (n)
    integer :: n
    type(histogram_t), intent(in) :: h
    integer, intent(in) :: i
    if (i <= 0) then
       n = bin_get_n_entries (h%underflow)
    else if (i <= h%n_bins) then
       n = bin_get_n_entries (h%bin(i))
    else
       n = bin_get_n_entries (h%overflow)
    end if
  end function histogram_get_n_entries_for_bin

  function histogram_get_sum_for_bin (h, i) result (avg)
    real(default) :: avg
    type(histogram_t), intent(in) :: h
    integer, intent(in) :: i
```

```
    if (i <= 0) then
       avg = bin_get_sum (h%underflow)
    else if (i <= h%n_bins) then
       avg = bin_get_sum (h%bin(i))
    else
       avg = bin_get_sum (h%overflow)
    end if
  end function histogram_get_sum_for_bin

  function histogram_get_error_for_bin (h, i) result (err)
    real(default) :: err
    type(histogram_t), intent(in) :: h
    integer, intent(in) :: i
    if (i <= 0) then
       err = bin_get_error (h%underflow)
    else if (i <= h%n_bins) then
       err = bin_get_error (h%bin(i))
    else
       err = bin_get_error (h%overflow)
    end if
  end function histogram_get_error_for_bin

  function histogram_get_excess_for_bin (h, i) result (err)
    real(default) :: err
    type(histogram_t), intent(in) :: h
    integer, intent(in) :: i
    if (i <= 0) then
       err = bin_get_excess (h%underflow)
    else if (i <= h%n_bins) then
       err = bin_get_excess (h%bin(i))
    else
       err = bin_get_excess (h%overflow)
    end if
  end function histogram_get_excess_for_bin
```

Return a pointer to the graph options.

⟨*Analysis: procedures*⟩+≡

```
  function histogram_get_graph_options_ptr (h) result (ptr)
    type(graph_options_t), pointer :: ptr
    type(histogram_t), intent(in), target :: h
    ptr => h%graph_options
  end function histogram_get_graph_options_ptr
```

Return a pointer to the drawing options.

⟨*Analysis: procedures*⟩+≡

```
  function histogram_get_drawing_options_ptr (h) result (ptr)
    type(drawing_options_t), pointer :: ptr
    type(histogram_t), intent(in), target :: h
    ptr => h%drawing_options
  end function histogram_get_drawing_options_ptr
```

Output

⟨*Analysis: procedures*⟩+≡

```
subroutine histogram_write (h, unit)
  type(histogram_t), intent(in) :: h
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  call bin_write_header (u)
  if (allocated (h%bin)) then
     do i = 1, h%n_bins
        call bin_write (h%bin(i), u)
     end do
  end if
  write (u, "(A)")
  write (u, "(A,1x,A)")  "#", "Underflow:"
  call bin_write (h%underflow, u)
  write (u, "(A)")
  write (u, "(A,1x,A)")  "#", "Overflow:"
  call bin_write (h%overflow, u)
  write (u, "(A)")
  write (u, "(A,1x,A)")  "#", "Summary: data within bounds"
  call observable_write (h%obs_within_bounds, u)
  write (u, "(A)")
  write (u, "(A,1x,A)")  "#", "Summary: all data"
  call observable_write (h%obs, u)
  write (u, "(A)")
end subroutine histogram_write
```

Write the GAMELAN reader for histogram contents.

⟨*Analysis: procedures*⟩+≡

```
subroutine histogram_write_gml_reader (h, filename, unit)
  type(histogram_t), intent(in) :: h
  type(string_t), intent(in) :: filename
  integer, intent(in), optional :: unit
  character(*), parameter :: fmt = "(ES15.8)"
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(2x,A)")  'fromfile "' // char (filename) // '":'
  write (u, "(4x,A)")  'key "# Histogram:";'
  write (u, "(4x,A)")  'dx := #' &
       // real2char (h%width / h%n_bins / 2, fmt) // ';'
  write (u, "(4x,A)")  'for i withinblock:'
  write (u, "(6x,A)")  'get x, y, y.d, y.n, y.e;'
  if (h%drawing_options%with_hbars) then
     write (u, "(6x,A)")  'plot (' // char (h%drawing_options%dataset) &
          // ') (x,y) hbar dx;'
  else
     write (u, "(6x,A)")  'plot (' // char (h%drawing_options%dataset) &
          // ') (x,y);'
  end if
  if (h%drawing_options%err) then
     write (u, "(6x,A)")  'plot (' // char (h%drawing_options%dataset) &
       // '.err) ' &
```

```
        // '(x,y) vbar y.d;'
  end if
  !!! Future excess options for plots
  ! write (u, "(6x,A)")  'if show_excess: ' // &
  !            & 'plot(dat.e)(x, y plus y.e) hbar dx; fi'
  write (u, "(4x,A)")  'endfor'
  write (u, "(2x,A)")  'endfrom'
end subroutine histogram_write_gml_reader
```

LaTeX and GAMELAN output.

⟨*Analysis: procedures*⟩+≡

```
subroutine histogram_write_gml_driver (h, filename, unit)
  type(histogram_t), intent(in) :: h
  type(string_t), intent(in) :: filename
  integer, intent(in), optional :: unit
  type(string_t) :: calc_cmd, bg_cmd, draw_cmd, err_cmd, symb_cmd, fg_cmd
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call graph_options_write_tex_header (h%graph_options, unit)
  write (u, "(2x,A)")  char (graph_options_get_gml_setup (h%graph_options))
  write (u, "(2x,A)")  char (graph_options_get_gml_graphrange &
       (h%graph_options, x_min=h%lower_bound, x_max=h%upper_bound))
  call histogram_write_gml_reader (h, filename, unit)
  calc_cmd = drawing_options_get_calc_command (h%drawing_options)
  if (calc_cmd /= "")  write (u, "(2x,A)")  char (calc_cmd)
  bg_cmd = drawing_options_get_gml_bg_command (h%drawing_options)
  if (bg_cmd /= "")  write (u, "(2x,A)")  char (bg_cmd)
  draw_cmd = drawing_options_get_draw_command (h%drawing_options)
  if (draw_cmd /= "")  write (u, "(2x,A)")  char (draw_cmd)
  err_cmd = drawing_options_get_err_command (h%drawing_options)
  if (err_cmd /= "")  write (u, "(2x,A)")  char (err_cmd)
  symb_cmd = drawing_options_get_symb_command (h%drawing_options)
  if (symb_cmd /= "")  write (u, "(2x,A)")  char (symb_cmd)
  fg_cmd = drawing_options_get_gml_fg_command (h%drawing_options)
  if (fg_cmd /= "")  write (u, "(2x,A)")  char (fg_cmd)
  write (u, "(2x,A)")  char (graph_options_get_gml_x_label (h%graph_options))
  write (u, "(2x,A)")  char (graph_options_get_gml_y_label (h%graph_options))
  call graph_options_write_tex_footer (h%graph_options, unit)
  write (u, "(A)") "\vspace*{2\baselineskip}"
  write (u, "(A)") "\begin{flushleft}"
  write (u, "(A)") "\textbf{Data within bounds:} \\"
  call observable_write_driver (h%obs_within_bounds, unit, &
                                write_heading=.false.)
  write (u, "(A)") "\\[0.5\baselineskip]"
  write (u, "(A)") "\textbf{All data:} \\"
  call observable_write_driver (h%obs, unit, write_heading=.false.)
  write (u, "(A)") "\end{flushleft}"
end subroutine histogram_write_gml_driver
```

Return the header for generic data output as an ifile.

⟨*Analysis: procedures*⟩+≡

```
subroutine histogram_get_header (h, header, comment)
  type(histogram_t), intent(in) :: h
```

```
    type(ifile_t), intent(inout) :: header
    type(string_t), intent(in), optional :: comment
    type(string_t) :: c
    if (present (comment)) then
       c = comment
    else
       c = ""
    end if
    call ifile_append (header, c // "WHIZARD histogram data")
    call graph_options_get_header (h%graph_options, header, comment)
    call ifile_append (header, &
         c // "range: " // real2string (h%lower_bound) &
         // " - " // real2string (h%upper_bound))
    call ifile_append (header, &
         c // "counts total: " &
         // int2char (histogram_get_n_entries_within_bounds (h)))
    call ifile_append (header, &
         c // "total average: " &
         // real2string (histogram_get_average_within_bounds (h)) // " +- " &
         // real2string (histogram_get_error_within_bounds (h)))
  end subroutine histogram_get_header
```

12.5.7 Plots

Points

⟨*Analysis: types*⟩+≡

```
  type :: point_t
     private
     real(default) :: x = 0
     real(default) :: y = 0
     real(default) :: yerr = 0
     real(default) :: xerr = 0
     type(point_t), pointer :: next => null ()
  end type point_t
```

⟨*Analysis: interfaces*⟩+≡

```
  interface point_init
    module procedure point_init_contents
    module procedure point_init_point
  end interface
```

⟨*Analysis: procedures*⟩+≡

```
  subroutine point_init_contents (point, x, y, yerr, xerr)
    type(point_t), intent(out) :: point
    real(default), intent(in) :: x, y
    real(default), intent(in), optional :: yerr, xerr
    point%x = x
    point%y = y
    if (present (yerr))  point%yerr = yerr
    if (present (xerr))  point%xerr = xerr
  end subroutine point_init_contents
```

```
subroutine point_init_point (point, point_in)
  type(point_t), intent(out) :: point
  type(point_t), intent(in) :: point_in
  point%x = point_in%x
  point%y = point_in%y
  point%yerr = point_in%yerr
  point%xerr = point_in%xerr
end subroutine point_init_point
```

⟨*Analysis: procedures*⟩+≡

```
function point_get_x (point) result (x)
  real(default) :: x
  type(point_t), intent(in) :: point
  x = point%x
end function point_get_x

function point_get_y (point) result (y)
  real(default) :: y
  type(point_t), intent(in) :: point
  y = point%y
end function point_get_y

function point_get_xerr (point) result (xerr)
  real(default) :: xerr
  type(point_t), intent(in) :: point
  xerr = point%xerr
end function point_get_xerr

function point_get_yerr (point) result (yerr)
  real(default) :: yerr
  type(point_t), intent(in) :: point
  yerr = point%yerr
end function point_get_yerr
```

⟨*Analysis: procedures*⟩+≡

```
subroutine point_write_header (unit)
  integer, intent(in) :: unit
  character(120) :: buffer
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (buffer, "(A,4(1x," // HISTOGRAM_HEAD_FORMAT // "))") &
       "#", "x       ", "y       ", "yerr     ", "xerr     "
  write (u, "(A)")  trim (buffer)
end subroutine point_write_header

subroutine point_write (point, unit)
  type(point_t), intent(in) :: point
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,4(1x," // HISTOGRAM_DATA_FORMAT // "))") &
       point_get_x (point), &
       point_get_y (point), &
```

```
        point_get_yerr (point), &
        point_get_xerr (point)
end subroutine point_write
```

Plots

⟨*Analysis: types*⟩+≡

```
type :: plot_t
   private
   type(point_t), pointer :: first => null ()
   type(point_t), pointer :: last => null ()
   integer :: count = 0
   type(graph_options_t) :: graph_options
   type(drawing_options_t) :: drawing_options
end type plot_t
```

Initializer/finalizer

Initialize a plot. We provide the lower and upper bound in the x direction.

⟨*Analysis: interfaces*⟩+≡

```
interface plot_init
   module procedure plot_init_empty
   module procedure plot_init_plot
end interface
```

⟨*Analysis: procedures*⟩+≡

```
subroutine plot_init_empty (p, id, graph_options, drawing_options)
  type(plot_t), intent(out) :: p
  type(string_t), intent(in) :: id
  type(graph_options_t), intent(in), optional :: graph_options
  type(drawing_options_t), intent(in), optional :: drawing_options
  if (present (graph_options)) then
     p%graph_options = graph_options
  else
     call graph_options_init (p%graph_options)
  end if
  call graph_options_set (p%graph_options, id = id)
  if (present (drawing_options)) then
     p%drawing_options = drawing_options
  else
     call drawing_options_init_plot (p%drawing_options)
  end if
end subroutine plot_init_empty
```

Initialize a plot by copying another one, optionally merging in a new set of drawing options.

Since `p` has pointer (sub)components, we have to explicitly deep-copy the original.

⟨*Analysis: procedures*⟩+≡

```
subroutine plot_init_plot (p, p_in, drawing_options)
  type(plot_t), intent(out) :: p
```

```
    type(plot_t), intent(in) :: p_in
    type(drawing_options_t), intent(in), optional :: drawing_options
    type(point_t), pointer :: current, new
    current => p_in%first
    do while (associated (current))
       allocate (new)
       call point_init (new, current)
       if (associated (p%last)) then
          p%last%next => new
       else
          p%first => new
       end if
       p%last => new
       current => current%next
    end do
    p%count = p_in%count
    p%graph_options = p_in%graph_options
    if (present (drawing_options)) then
       p%drawing_options = drawing_options
    else
       p%drawing_options = p_in%drawing_options
    end if
  end subroutine plot_init_plot
```

Finalize the plot by deallocating the list of points.

⟨*Analysis: procedures*⟩+≡

```
  subroutine plot_final (plot)
    type(plot_t), intent(inout) :: plot
    type(point_t), pointer :: current
    do while (associated (plot%first))
       current => plot%first
       plot%first => current%next
       deallocate (current)
    end do
    plot%last => null ()
  end subroutine plot_final
```

Fill plots

Clear the plot contents, but do not modify the structure.

⟨*Analysis: procedures*⟩+≡

```
  subroutine plot_clear (plot)
    type(plot_t), intent(inout) :: plot
    plot%count = 0
    call plot_final (plot)
  end subroutine plot_clear
```

Record a value. Successful if the value is within bounds, otherwise it is recorded as under-/overflow.

⟨*Analysis: procedures*⟩+≡

```
  subroutine plot_record_value (plot, x, y, yerr, xerr, success)
```

```
    type(plot_t), intent(inout) :: plot
    real(default), intent(in) :: x, y
    real(default), intent(in), optional :: yerr, xerr
    logical, intent(out), optional :: success
    type(point_t), pointer :: point
    plot%count = plot%count + 1
    allocate (point)
    call point_init (point, x, y, yerr, xerr)
    if (associated (plot%first)) then
       plot%last%next => point
    else
       plot%first => point
    end if
    plot%last => point
    if (present (success))  success = .true.
  end subroutine plot_record_value
```

Access contents

The number of points.

⟨*Analysis: procedures*⟩+≡

```
  function plot_get_n_entries (plot) result (n)
    integer :: n
    type(plot_t), intent(in) :: plot
    n = plot%count
  end function plot_get_n_entries
```

Return a pointer to the graph options.

⟨*Analysis: procedures*⟩+≡

```
  function plot_get_graph_options_ptr (p) result (ptr)
    type(graph_options_t), pointer :: ptr
    type(plot_t), intent(in), target :: p
    ptr => p%graph_options
  end function plot_get_graph_options_ptr
```

Return a pointer to the drawing options.

⟨*Analysis: procedures*⟩+≡

```
  function plot_get_drawing_options_ptr (p) result (ptr)
    type(drawing_options_t), pointer :: ptr
    type(plot_t), intent(in), target :: p
    ptr => p%drawing_options
  end function plot_get_drawing_options_ptr
```

Output

This output format is used by the GAMELAN driver below.

⟨*Analysis: procedures*⟩+≡

```
  subroutine plot_write (plot, unit)
    type(plot_t), intent(in) :: plot
    integer, intent(in), optional :: unit
```

```
    type(point_t), pointer :: point
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    call point_write_header (u)
    point => plot%first
    do while (associated (point))
       call point_write (point, unit)
       point => point%next
    end do
    write (u, *)
    write (u, "(A,1x,A)")  "#", "Summary:"
    write (u, "(A,1x,I0)") &
         "n_entries =", plot_get_n_entries (plot)
    write (u, *)
  end subroutine plot_write
```

Write the GAMELAN reader for plot contents.

⟨*Analysis: procedures*⟩+≡

```
  subroutine plot_write_gml_reader (p, filename, unit)
    type(plot_t), intent(in) :: p
    type(string_t), intent(in) :: filename
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(2x,A)")  'fromfile "' // char (filename) // '":'
    write (u, "(4x,A)")  'key "# Plot:";'
    write (u, "(4x,A)")  'for i withinblock:'
    write (u, "(6x,A)")  'get x, y, y.err, x.err;'
    write (u, "(6x,A)")  'plot (' // char (p%drawing_options%dataset) &
         // ') (x,y);'
    if (p%drawing_options%err) then
       write (u, "(6x,A)")  'plot (' // char (p%drawing_options%dataset) &
            // '.err) (x,y) vbar y.err hbar x.err;'
    end if
    write (u, "(4x,A)")  'endfor'
    write (u, "(2x,A)")  'endfrom'
  end subroutine plot_write_gml_reader
```

LaTeX and GAMELAN output. Analogous to histogram output.

⟨*Analysis: procedures*⟩+≡

```
  subroutine plot_write_gml_driver (p, filename, unit)
    type(plot_t), intent(in) :: p
    type(string_t), intent(in) :: filename
    integer, intent(in), optional :: unit
    type(string_t) :: calc_cmd, bg_cmd, draw_cmd, err_cmd, symb_cmd, fg_cmd
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    call graph_options_write_tex_header (p%graph_options, unit)
    write (u, "(2x,A)") &
         char (graph_options_get_gml_setup (p%graph_options))
    write (u, "(2x,A)") &
         char (graph_options_get_gml_graphrange (p%graph_options))
    call plot_write_gml_reader (p, filename, unit)
```

```
    calc_cmd = drawing_options_get_calc_command (p%drawing_options)
    if (calc_cmd /= "")  write (u, "(2x,A)")  char (calc_cmd)
    bg_cmd = drawing_options_get_gml_bg_command (p%drawing_options)
    if (bg_cmd /= "")  write (u, "(2x,A)")  char (bg_cmd)
    draw_cmd = drawing_options_get_draw_command (p%drawing_options)
    if (draw_cmd /= "")  write (u, "(2x,A)")  char (draw_cmd)
    err_cmd = drawing_options_get_err_command (p%drawing_options)
    if (err_cmd /= "")  write (u, "(2x,A)")  char (err_cmd)
    symb_cmd = drawing_options_get_symb_command (p%drawing_options)
    if (symb_cmd /= "")  write (u, "(2x,A)")  char (symb_cmd)
    fg_cmd = drawing_options_get_gml_fg_command (p%drawing_options)
    if (fg_cmd /= "")  write (u, "(2x,A)")  char (fg_cmd)
    write (u, "(2x,A)")  char (graph_options_get_gml_x_label (p%graph_options))
    write (u, "(2x,A)")  char (graph_options_get_gml_y_label (p%graph_options))
    call graph_options_write_tex_footer (p%graph_options, unit)
  end subroutine plot_write_gml_driver
```

Append header for generic data output in ifile format.

⟨*Analysis: procedures*⟩+≡

```
  subroutine plot_get_header (plot, header, comment)
    type(plot_t), intent(in) :: plot
    type(ifile_t), intent(inout) :: header
    type(string_t), intent(in), optional :: comment
    type(string_t) :: c
    if (present (comment)) then
       c = comment
    else
       c = ""
    end if
    call ifile_append (header, c // "WHIZARD plot data")
    call graph_options_get_header (plot%graph_options, header, comment)
    call ifile_append (header, &
         c // "number of points: " &
         // int2char (plot_get_n_entries (plot)))
  end subroutine plot_get_header
```

12.5.8 Graphs

A graph is a container for several graph elements. Each graph element is either a plot or a histogram. There is an appropriate base type below (the `analysis_object_t`), but to avoid recursion, we define a separate base type here. Note that there is no actual recursion: a graph is an analysis object, but a graph cannot contain graphs.

(If we could use type extension, the implementation would be much more transparent.)

Graph elements

Graph elements cannot be filled by the `record` command directly. The contents are always copied from elementary histograms or plots.

⟨*Analysis: types*⟩+≡

```
type :: graph_element_t
   private
   integer :: type = AN_UNDEFINED
   type(histogram_t), pointer :: h => null ()
   type(plot_t), pointer :: p => null ()
end type graph_element_t
```

⟨*Analysis: procedures*⟩+≡

```
subroutine graph_element_final (el)
  type(graph_element_t), intent(inout) :: el
  select case (el%type)
  case (AN_HISTOGRAM)
     deallocate (el%h)
  case (AN_PLOT)
     call plot_final (el%p)
     deallocate (el%p)
  end select
  el%type = AN_UNDEFINED
end subroutine graph_element_final
```

Return the number of entries in the graph element:

⟨*Analysis: procedures*⟩+≡

```
function graph_element_get_n_entries (el) result (n)
  integer :: n
  type(graph_element_t), intent(in) :: el
  select case (el%type)
  case (AN_HISTOGRAM);  n = histogram_get_n_entries (el%h)
  case (AN_PLOT);       n = plot_get_n_entries (el%p)
  case default;         n = 0
  end select
end function graph_element_get_n_entries
```

Return a pointer to the graph / drawing options.

⟨*Analysis: procedures*⟩+≡

```
function graph_element_get_graph_options_ptr (el) result (ptr)
  type(graph_options_t), pointer :: ptr
  type(graph_element_t), intent(in) :: el
  select case (el%type)
  case (AN_HISTOGRAM);  ptr => histogram_get_graph_options_ptr (el%h)
  case (AN_PLOT);       ptr => plot_get_graph_options_ptr (el%p)
  case default;         ptr => null ()
  end select
end function graph_element_get_graph_options_ptr

function graph_element_get_drawing_options_ptr (el) result (ptr)
  type(drawing_options_t), pointer :: ptr
  type(graph_element_t), intent(in) :: el
  select case (el%type)
  case (AN_HISTOGRAM);  ptr => histogram_get_drawing_options_ptr (el%h)
  case (AN_PLOT);       ptr => plot_get_drawing_options_ptr (el%p)
  case default;         ptr => null ()
  end select
```

```
end function graph_element_get_drawing_options_ptr
```

Output, simple wrapper for the plot/histogram writer.

⟨*Analysis: procedures*⟩+≡

```
subroutine graph_element_write (el, unit)
  type(graph_element_t), intent(in) :: el
  integer, intent(in), optional :: unit
  type(graph_options_t), pointer :: gro
  type(string_t) :: id
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  gro => graph_element_get_graph_options_ptr (el)
  id = graph_options_get_id (gro)
  write (u, "(A,A)")  '#', repeat ("-", 78)
  select case (el%type)
  case (AN_HISTOGRAM)
     write (u, "(A)", advance="no")  "# Histogram: "
     write (u, "(1x,A)")  char (id)
     call histogram_write (el%h, unit)
  case (AN_PLOT)
     write (u, "(A)", advance="no")  "# Plot: "
     write (u, "(1x,A)")  char (id)
     call plot_write (el%p, unit)
  end select
end subroutine graph_element_write
```

⟨*Analysis: procedures*⟩+≡

```
subroutine graph_element_write_gml_reader (el, filename, unit)
  type(graph_element_t), intent(in) :: el
  type(string_t), intent(in) :: filename
  integer, intent(in), optional :: unit
  select case (el%type)
  case (AN_HISTOGRAM);  call histogram_write_gml_reader (el%h, filename, unit)
  case (AN_PLOT);       call plot_write_gml_reader (el%p, filename, unit)
  end select
end subroutine graph_element_write_gml_reader
```

The graph type

The actual graph type contains its own `graph_options`, which override the individual settings. The `drawing_options` are set in the graph elements. This distinction motivates the separation of the two types.

⟨*Analysis: types*⟩+≡

```
type :: graph_t
   private
   type(graph_element_t), dimension(:), allocatable :: el
   type(graph_options_t) :: graph_options
end type graph_t
```

Initializer/finalizer

The graph is created with a definite number of elements. The elements are filled one by one, optionally with modified drawing options.

⟨*Analysis: procedures*⟩+≡

```
subroutine graph_init (g, id, n_elements, graph_options)
  type(graph_t), intent(out) :: g
  type(string_t), intent(in) :: id
  integer, intent(in) :: n_elements
  type(graph_options_t), intent(in), optional :: graph_options
  allocate (g%el (n_elements))
  if (present (graph_options)) then
     g%graph_options = graph_options
  else
     call graph_options_init (g%graph_options)
  end if
  call graph_options_set (g%graph_options, id = id)
end subroutine graph_init
```

⟨*Analysis: procedures*⟩+≡

```
subroutine graph_insert_histogram (g, i, h, drawing_options)
  type(graph_t), intent(inout), target :: g
  integer, intent(in) :: i
  type(histogram_t), intent(in) :: h
  type(drawing_options_t), intent(in), optional :: drawing_options
  type(graph_options_t), pointer :: gro
  type(drawing_options_t), pointer :: dro
  type(string_t) :: id
  g%el(i)%type = AN_HISTOGRAM
  allocate (g%el(i)%h)
  call histogram_init_histogram (g%el(i)%h, h, drawing_options)
  gro => histogram_get_graph_options_ptr (g%el(i)%h)
  dro => histogram_get_drawing_options_ptr (g%el(i)%h)
  id = graph_options_get_id (gro)
  call drawing_options_set (dro, dataset = "dat." // id)
end subroutine graph_insert_histogram
```

⟨*Analysis: procedures*⟩+≡

```
subroutine graph_insert_plot (g, i, p, drawing_options)
  type(graph_t), intent(inout) :: g
  integer, intent(in) :: i
  type(plot_t), intent(in) :: p
  type(drawing_options_t), intent(in), optional :: drawing_options
  type(graph_options_t), pointer :: gro
  type(drawing_options_t), pointer :: dro
  type(string_t) :: id
  g%el(i)%type = AN_PLOT
  allocate (g%el(i)%p)
  call plot_init_plot (g%el(i)%p, p, drawing_options)
  gro => plot_get_graph_options_ptr (g%el(i)%p)
  dro => plot_get_drawing_options_ptr (g%el(i)%p)
  id = graph_options_get_id (gro)
  call drawing_options_set (dro, dataset = "dat." // id)
```

```
end subroutine graph_insert_plot
```

Finalizer.

⟨*Analysis: procedures*⟩+≡

```
subroutine graph_final (g)
  type(graph_t), intent(inout) :: g
  integer :: i
  do i = 1, size (g%el)
     call graph_element_final (g%el(i))
  end do
  deallocate (g%el)
end subroutine graph_final
```

Access contents

The number of elements.

⟨*Analysis: procedures*⟩+≡

```
function graph_get_n_elements (graph) result (n)
  integer :: n
  type(graph_t), intent(in) :: graph
  n = size (graph%el)
end function graph_get_n_elements
```

Retrieve a pointer to the drawing options of an element, so they can be modified. (The `target` attribute is not actually needed because the components are pointers.)

⟨*Analysis: procedures*⟩+≡

```
function graph_get_drawing_options_ptr (g, i) result (ptr)
  type(drawing_options_t), pointer :: ptr
  type(graph_t), intent(in), target :: g
  integer, intent(in) :: i
  ptr => graph_element_get_drawing_options_ptr (g%el(i))
end function graph_get_drawing_options_ptr
```

Output

The default output format just writes histogram and plot data.

⟨*Analysis: procedures*⟩+≡

```
subroutine graph_write (graph, unit)
  type(graph_t), intent(in) :: graph
  integer, intent(in), optional :: unit
  integer :: i
  do i = 1, size (graph%el)
     call graph_element_write (graph%el(i), unit)
  end do
end subroutine graph_write
```

The GAMELAN driver is not a simple wrapper, but it writes the plot/histogram contents embedded the complete graph. First, data are read in, global background commands next, then individual elements, then global foreground commands.

⟨*Analysis: procedures*⟩+≡

```
subroutine graph_write_gml_driver (g, filename, unit)
  type(graph_t), intent(in) :: g
  type(string_t), intent(in) :: filename
  type(string_t) :: calc_cmd, bg_cmd, draw_cmd, err_cmd, symb_cmd, fg_cmd
  integer, intent(in), optional :: unit
  type(drawing_options_t), pointer :: dro
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  call graph_options_write_tex_header (g%graph_options, unit)
  write (u, "(2x,A)") &
       char (graph_options_get_gml_setup (g%graph_options))
  write (u, "(2x,A)") &
       char (graph_options_get_gml_graphrange (g%graph_options))
  do i = 1, size (g%el)
     call graph_element_write_gml_reader (g%el(i), filename, unit)
     calc_cmd = drawing_options_get_calc_command &
                    (graph_element_get_drawing_options_ptr (g%el(i)))
     if (calc_cmd /= "")  write (u, "(2x,A)")  char (calc_cmd)
  end do
  bg_cmd = graph_options_get_gml_bg_command (g%graph_options)
  if (bg_cmd /= "")  write (u, "(2x,A)")  char (bg_cmd)
  do i = 1, size (g%el)
     dro => graph_element_get_drawing_options_ptr (g%el(i))
     bg_cmd = drawing_options_get_gml_bg_command (dro)
     if (bg_cmd /= "")  write (u, "(2x,A)")  char (bg_cmd)
     draw_cmd = drawing_options_get_draw_command (dro)
     if (draw_cmd /= "")  write (u, "(2x,A)")  char (draw_cmd)
     err_cmd = drawing_options_get_err_command (dro)
     if (err_cmd /= "")  write (u, "(2x,A)")  char (err_cmd)
     symb_cmd = drawing_options_get_symb_command (dro)
     if (symb_cmd /= "")  write (u, "(2x,A)")  char (symb_cmd)
     fg_cmd = drawing_options_get_gml_fg_command (dro)
     if (fg_cmd /= "")  write (u, "(2x,A)")  char (fg_cmd)
  end do
  fg_cmd = graph_options_get_gml_fg_command (g%graph_options)
  if (fg_cmd /= "")  write (u, "(2x,A)")  char (fg_cmd)
  write (u, "(2x,A)")  char (graph_options_get_gml_x_label (g%graph_options))
  write (u, "(2x,A)")  char (graph_options_get_gml_y_label (g%graph_options))
  call graph_options_write_tex_footer (g%graph_options, unit)
end subroutine graph_write_gml_driver
```

Append header for generic data output in ifile format.

⟨*Analysis: procedures*⟩+≡

```
subroutine graph_get_header (graph, header, comment)
  type(graph_t), intent(in) :: graph
  type(ifile_t), intent(inout) :: header
  type(string_t), intent(in), optional :: comment
  type(string_t) :: c
```

```
    if (present (comment)) then
       c = comment
    else
       c = ""
    end if
    call ifile_append (header, c // "WHIZARD graph data")
    call graph_options_get_header (graph%graph_options, header, comment)
    call ifile_append (header, &
         c // "number of graph elements: " &
         // int2char (graph_get_n_elements (graph)))
  end subroutine graph_get_header
```

12.5.9 Analysis objects

This data structure holds all observables, histograms and such that are currently active. We have one global store; individual items are identified by their ID strings.

(This should rather be coded by type extension.)

⟨*Analysis: parameters*⟩+≡

```
  integer, parameter :: AN_UNDEFINED = 0
  integer, parameter :: AN_OBSERVABLE = 1
  integer, parameter :: AN_HISTOGRAM = 2
  integer, parameter :: AN_PLOT = 3
  integer, parameter :: AN_GRAPH = 4
```

⟨*Analysis: public*⟩+≡

```
  public :: AN_UNDEFINED, AN_HISTOGRAM, AN_OBSERVABLE, AN_PLOT, AN_GRAPH
```

⟨*Analysis: types*⟩+≡

```
  type :: analysis_object_t
     private
     type(string_t) :: id
     integer :: type = AN_UNDEFINED
     type(observable_t), pointer :: obs => null ()
     type(histogram_t), pointer :: h => null ()
     type(plot_t), pointer :: p => null ()
     type(graph_t), pointer :: g => null ()
     type(analysis_object_t), pointer :: next => null ()
  end type analysis_object_t
```

Initializer/finalizer

Allocate with the correct type but do not fill initial values.

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_object_init (obj, id, type)
    type(analysis_object_t), intent(out) :: obj
    type(string_t), intent(in) :: id
    integer, intent(in) :: type
    obj%id = id
    obj%type = type
    select case (obj%type)
```

```
    case (AN_OBSERVABLE);  allocate (obj%obs)
    case (AN_HISTOGRAM);   allocate (obj%h)
    case (AN_PLOT);        allocate (obj%p)
    case (AN_GRAPH);       allocate (obj%g)
    end select
  end subroutine analysis_object_init
```

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_object_final (obj)
    type(analysis_object_t), intent(inout) :: obj
    select case (obj%type)
    case (AN_OBSERVABLE)
       deallocate (obj%obs)
    case (AN_HISTOGRAM)
       deallocate (obj%h)
    case (AN_PLOT)
       call plot_final (obj%p)
       deallocate (obj%p)
    case (AN_GRAPH)
       call graph_final (obj%g)
       deallocate (obj%g)
    end select
    obj%type = AN_UNDEFINED
  end subroutine analysis_object_final
```

Clear the analysis object, i.e., reset it to its initial state. Not applicable to graphs, which are always combinations of other existing objects.

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_object_clear (obj)
    type(analysis_object_t), intent(inout) :: obj
    select case (obj%type)
    case (AN_OBSERVABLE)
       call observable_clear (obj%obs)
    case (AN_HISTOGRAM)
       call histogram_clear (obj%h)
    case (AN_PLOT)
       call plot_clear (obj%p)
    end select
  end subroutine analysis_object_clear
```

Fill with data

Record data. The effect depends on the type of analysis object.

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_object_record_data (obj, &
       x, y, yerr, xerr, weight, excess, success)
    type(analysis_object_t), intent(inout) :: obj
    real(default), intent(in) :: x
    real(default), intent(in), optional :: y, yerr, xerr, weight, excess
    logical, intent(out), optional :: success
    select case (obj%type)
```

```
    case (AN_OBSERVABLE)
       if (present (weight)) then
          call observable_record_value_weighted (obj%obs, x, weight, success)
       else
          call observable_record_value_unweighted (obj%obs, x, success)
       end if
    case (AN_HISTOGRAM)
       if (present (weight)) then
          call histogram_record_value_weighted (obj%h, x, weight, success)
       else
          call histogram_record_value_unweighted (obj%h, x, excess, success)
       end if
    case (AN_PLOT)
       if (present (y)) then
          call plot_record_value (obj%p, x, y, yerr, xerr, success)
       else
          if (present (success))  success = .false.
       end if
    case default
       if (present (success))  success = .false.
    end select
  end subroutine analysis_object_record_data
```

Explicitly set the pointer to the next object in the list.

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_object_set_next_ptr (obj, next)
    type(analysis_object_t), intent(inout) :: obj
    type(analysis_object_t), pointer :: next
    obj%next => next
  end subroutine analysis_object_set_next_ptr
```

Access contents

Return a pointer to the next object in the list.

⟨*Analysis: procedures*⟩+≡

```
  function analysis_object_get_next_ptr (obj) result (next)
    type(analysis_object_t), pointer :: next
    type(analysis_object_t), intent(in) :: obj
    next => obj%next
  end function analysis_object_get_next_ptr
```

Return data as appropriate for the object type.

⟨*Analysis: procedures*⟩+≡

```
  function analysis_object_get_n_elements (obj) result (n)
    integer :: n
    type(analysis_object_t), intent(in) :: obj
    select case (obj%type)
    case (AN_HISTOGRAM)
       n = 1
    case (AN_PLOT)
       n = 1
```

```
  case (AN_GRAPH)
     n = graph_get_n_elements (obj%g)
  case default
     n = 0
  end select
end function analysis_object_get_n_elements

function analysis_object_get_n_entries (obj, within_bounds) result (n)
  integer :: n
  type(analysis_object_t), intent(in) :: obj
  logical, intent(in), optional :: within_bounds
  logical :: wb
  select case (obj%type)
  case (AN_OBSERVABLE)
     n = observable_get_n_entries (obj%obs)
  case (AN_HISTOGRAM)
     wb = .false.;  if (present (within_bounds)) wb = within_bounds
     if (wb) then
        n = histogram_get_n_entries_within_bounds (obj%h)
     else
        n = histogram_get_n_entries (obj%h)
     end if
  case (AN_PLOT)
     n = plot_get_n_entries (obj%p)
  case default
     n = 0
  end select
end function analysis_object_get_n_entries

function analysis_object_get_average (obj, within_bounds) result (avg)
  real(default) :: avg
  type(analysis_object_t), intent(in) :: obj
  logical, intent(in), optional :: within_bounds
  logical :: wb
  select case (obj%type)
  case (AN_OBSERVABLE)
     avg = observable_get_average (obj%obs)
  case (AN_HISTOGRAM)
     wb = .false.;  if (present (within_bounds)) wb = within_bounds
     if (wb) then
        avg = histogram_get_average_within_bounds (obj%h)
     else
        avg = histogram_get_average (obj%h)
     end if
  case default
     avg = 0
  end select
end function analysis_object_get_average

function analysis_object_get_error (obj, within_bounds) result (err)
  real(default) :: err
  type(analysis_object_t), intent(in) :: obj
  logical, intent(in), optional :: within_bounds
  logical :: wb
```

```
  select case (obj%type)
  case (AN_OBSERVABLE)
     err = observable_get_error (obj%obs)
  case (AN_HISTOGRAM)
     wb = .false.;  if (present (within_bounds)) wb = within_bounds
     if (wb) then
        err = histogram_get_error_within_bounds (obj%h)
     else
        err = histogram_get_error (obj%h)
     end if
  case default
     err = 0
  end select
end function analysis_object_get_error
```

Return pointers to the actual contents:

⟨*Analysis: procedures*⟩+≡

```
function analysis_object_get_observable_ptr (obj) result (obs)
  type(observable_t), pointer :: obs
  type(analysis_object_t), intent(in) :: obj
  select case (obj%type)
  case (AN_OBSERVABLE);  obs => obj%obs
  case default;          obs => null ()
  end select
end function analysis_object_get_observable_ptr

function analysis_object_get_histogram_ptr (obj) result (h)
  type(histogram_t), pointer :: h
  type(analysis_object_t), intent(in) :: obj
  select case (obj%type)
  case (AN_HISTOGRAM);  h => obj%h
  case default;         h => null ()
  end select
end function analysis_object_get_histogram_ptr

function analysis_object_get_plot_ptr (obj) result (plot)
  type(plot_t), pointer :: plot
  type(analysis_object_t), intent(in) :: obj
  select case (obj%type)
  case (AN_PLOT);  plot => obj%p
  case default;    plot => null ()
  end select
end function analysis_object_get_plot_ptr

function analysis_object_get_graph_ptr (obj) result (g)
  type(graph_t), pointer :: g
  type(analysis_object_t), intent(in) :: obj
  select case (obj%type)
  case (AN_GRAPH);  g => obj%g
  case default;     g => null ()
  end select
end function analysis_object_get_graph_ptr
```

Return true if the object has a graphical representation:

⟨*Analysis: procedures*⟩+≡

```
function analysis_object_has_plot (obj) result (flag)
  logical :: flag
  type(analysis_object_t), intent(in) :: obj
  select case (obj%type)
  case (AN_HISTOGRAM);  flag = .true.
  case (AN_PLOT);       flag = .true.
  case (AN_GRAPH);      flag = .true.
  case default;         flag = .false.
  end select
end function analysis_object_has_plot
```

Output

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_object_write (obj, unit, verbose)
  type(analysis_object_t), intent(in) :: obj
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  logical :: verb
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  verb = .false.;  if (present (verbose))  verb = verbose
  write (u, "(A)")  repeat ("#", 79)
  select case (obj%type)
  case (AN_OBSERVABLE)
     write (u, "(A)", advance="no")  "# Observable:"
  case (AN_HISTOGRAM)
     write (u, "(A)", advance="no")  "# Histogram: "
  case (AN_PLOT)
     write (u, "(A)", advance="no")  "# Plot: "
  case (AN_GRAPH)
     write (u, "(A)", advance="no")  "# Graph: "
  case default
     write (u, "(A)") "# [undefined analysis object]"
     return
  end select
  write (u, "(1x,A)")  char (obj%id)
  select case (obj%type)
  case (AN_OBSERVABLE)
     call observable_write (obj%obs, unit)
  case (AN_HISTOGRAM)
     if (verb) then
        call graph_options_write (obj%h%graph_options, unit)
        write (u, *)
        call drawing_options_write (obj%h%drawing_options, unit)
        write (u, *)
     end if
     call histogram_write (obj%h, unit)
  case (AN_PLOT)
     if (verb) then
        call graph_options_write (obj%p%graph_options, unit)
```

```
          write (u, *)
          call drawing_options_write (obj%p%drawing_options, unit)
          write (u, *)
       end if
       call plot_write (obj%p, unit)
    case (AN_GRAPH)
       call graph_write (obj%g, unit)
    end select
  end subroutine analysis_object_write
```

Write the object part of the LaTeX driver file.

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_object_write_driver (obj, filename, unit)
    type(analysis_object_t), intent(in) :: obj
    type(string_t), intent(in) :: filename
    integer, intent(in), optional :: unit
    select case (obj%type)
    case (AN_OBSERVABLE)
       call observable_write_driver (obj%obs, unit)
    case (AN_HISTOGRAM)
       call histogram_write_gml_driver (obj%h, filename, unit)
    case (AN_PLOT)
       call plot_write_gml_driver (obj%p, filename, unit)
    case (AN_GRAPH)
       call graph_write_gml_driver (obj%g, filename, unit)
    end select
  end subroutine analysis_object_write_driver
```

Return a data header for external formats, in ifile form.

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_object_get_header (obj, header, comment)
    type(analysis_object_t), intent(in) :: obj
    type(ifile_t), intent(inout) :: header
    type(string_t), intent(in), optional :: comment
    select case (obj%type)
    case (AN_HISTOGRAM)
       call histogram_get_header (obj%h, header, comment)
    case (AN_PLOT)
       call plot_get_header (obj%p, header, comment)
    end select
  end subroutine analysis_object_get_header
```

12.5.10 Analysis object iterator

Analysis objects are containers which have iterable data structures: histograms/bins and plots/points. If they are to be treated on a common basis, it is useful to have an iterator which hides the implementation details.

The iterator is used only for elementary analysis objects that contain plot data: histograms or plots. It is invalid for meta-objects (graphs) and non-graphical objects (observables).

⟨*Analysis: public*⟩+≡

```
public :: analysis_iterator_t
```

⟨*Analysis: types*⟩+≡

```
type :: analysis_iterator_t
  private
  integer :: type = AN_UNDEFINED
  type(analysis_object_t), pointer :: object => null ()
  integer :: index = 1
  type(point_t), pointer :: point => null ()
end type
```

The initializer places the iterator at the beginning of the analysis object.

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_iterator_init (iterator, object)
  type(analysis_iterator_t), intent(out) :: iterator
  type(analysis_object_t), intent(in), target :: object
  iterator%object => object
  if (associated (iterator%object)) then
     iterator%type = iterator%object%type
     select case (iterator%type)
     case (AN_PLOT)
        iterator%point => iterator%object%p%first
     end select
  end if
end subroutine analysis_iterator_init
```

The iterator is valid as long as it points to an existing entry. An iterator for a data object without array data (observable) is always invalid.

⟨*Analysis: public*⟩+≡

```
public :: analysis_iterator_is_valid
```

⟨*Analysis: procedures*⟩+≡

```
function analysis_iterator_is_valid (iterator) result (valid)
  logical :: valid
  type(analysis_iterator_t), intent(in) :: iterator
  if (associated (iterator%object)) then
     select case (iterator%type)
     case (AN_HISTOGRAM)
        valid = iterator%index <= histogram_get_n_bins (iterator%object%h)
     case (AN_PLOT)
        valid = associated (iterator%point)
     case default
        valid = .false.
     end select
  else
     valid = .false.
  end if
end function analysis_iterator_is_valid
```

Advance the iterator.

⟨*Analysis: public*⟩+≡

```
public :: analysis_iterator_advance
```

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_iterator_advance (iterator)
  type(analysis_iterator_t), intent(inout) :: iterator
  if (associated (iterator%object)) then
     select case (iterator%type)
     case (AN_PLOT)
        iterator%point => iterator%point%next
     end select
     iterator%index = iterator%index + 1
  end if
end subroutine analysis_iterator_advance
```

Retrieve the object type:

⟨*Analysis: public*⟩+≡

```
public :: analysis_iterator_get_type
```

⟨*Analysis: procedures*⟩+≡

```
function analysis_iterator_get_type (iterator) result (type)
  integer :: type
  type(analysis_iterator_t), intent(in) :: iterator
  type = iterator%type
end function analysis_iterator_get_type
```

Use the iterator to retrieve data. We implement a common routine which takes the data descriptors as optional arguments. Data which do not occur in the selected type trigger to an error condition.

The iterator must point to a valid entry.

⟨*Analysis: public*⟩+≡

```
public :: analysis_iterator_get_data
```

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_iterator_get_data (iterator, &
     x, y, yerr, xerr, width, excess, index, n_total)
  type(analysis_iterator_t), intent(in) :: iterator
  real(default), intent(out), optional :: x, y, yerr, xerr, width, excess
  integer, intent(out), optional :: index, n_total
  select case (iterator%type)
  case (AN_HISTOGRAM)
     if (present (x)) &
          x = bin_get_midpoint (iterator%object%h%bin(iterator%index))
     if (present (y)) &
          y = bin_get_sum (iterator%object%h%bin(iterator%index))
     if (present (yerr)) &
          yerr = bin_get_error (iterator%object%h%bin(iterator%index))
     if (present (xerr)) &
          call invalid ("histogram", "xerr")
     if (present (width)) &
          width = bin_get_width (iterator%object%h%bin(iterator%index))
     if (present (excess)) &
          excess = bin_get_excess (iterator%object%h%bin(iterator%index))
     if (present (index)) &
          index = iterator%index
     if (present (n_total)) &
          n_total = histogram_get_n_bins (iterator%object%h)
```

```
    case (AN_PLOT)
       if (present (x)) &
            x = point_get_x (iterator%point)
       if (present (y)) &
            y = point_get_y (iterator%point)
       if (present (yerr)) &
            yerr = point_get_yerr (iterator%point)
       if (present (xerr)) &
            xerr = point_get_xerr (iterator%point)
       if (present (width)) &
            call invalid ("plot", "width")
       if (present (excess)) &
            call invalid ("plot", "excess")
       if (present (index)) &
            index = iterator%index
       if (present (n_total)) &
            n_total = plot_get_n_entries (iterator%object%p)
    case default
       call msg_bug ("analysis_iterator_get_data: called " &
            // "for unsupported analysis object type")
    end select
  contains
    subroutine invalid (typestr, objstr)
      character(*), intent(in) :: typestr, objstr
      call msg_bug ("analysis_iterator_get_data: attempt to get '" &
           // objstr // "' for type '" // typestr // "'")
    end subroutine invalid
  end subroutine analysis_iterator_get_data
```

12.5.11 Analysis store

This data structure holds all observables, histograms and such that are currently active. We have one global store; individual items are identified by their ID strings and types.

⟨*Analysis: variables*⟩≡

```
  type(analysis_store_t), save :: analysis_store
```

⟨*Analysis: types*⟩+≡

```
  type :: analysis_store_t
     private
     type(analysis_object_t), pointer :: first => null ()
     type(analysis_object_t), pointer :: last => null ()
  end type analysis_store_t
```

Delete the analysis store

⟨*Analysis: public*⟩+≡

```
  public :: analysis_final
```

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_final ()
    type(analysis_object_t), pointer :: current
```

```
    do while (associated (analysis_store%first))
       current => analysis_store%first
       analysis_store%first => current%next
       call analysis_object_final (current)
    end do
    analysis_store%last => null ()
  end subroutine analysis_final
```

Append a new analysis object

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_store_append_object (id, type)
    type(string_t), intent(in) :: id
    integer, intent(in) :: type
    type(analysis_object_t), pointer :: obj
    allocate (obj)
    call analysis_object_init (obj, id, type)
    if (associated (analysis_store%last)) then
       analysis_store%last%next => obj
    else
       analysis_store%first => obj
    end if
    analysis_store%last => obj
  end subroutine analysis_store_append_object
```

Return a pointer to the analysis object with given ID.

⟨*Analysis: procedures*⟩+≡

```
  function analysis_store_get_object_ptr (id) result (obj)
    type(string_t), intent(in) :: id
    type(analysis_object_t), pointer :: obj
    obj => analysis_store%first
    do while (associated (obj))
       if (obj%id == id)  return
       obj => obj%next
    end do
  end function analysis_store_get_object_ptr
```

Initialize an analysis object: either reset it if present, or append a new entry.

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_store_init_object (id, type, obj)
    type(string_t), intent(in) :: id
    integer, intent(in) :: type
    type(analysis_object_t), pointer :: obj, next
    obj => analysis_store_get_object_ptr (id)
    if (associated (obj)) then
       next => analysis_object_get_next_ptr (obj)
       call analysis_object_final (obj)
       call analysis_object_init (obj, id, type)
       call analysis_object_set_next_ptr (obj, next)
    else
       call analysis_store_append_object (id, type)
       obj => analysis_store%last
    end if
```

```
end subroutine analysis_store_init_object
```

Get the type of a analysis object

⟨*Analysis: public*⟩+≡

```
public :: analysis_store_get_object_type
```

⟨*Analysis: procedures*⟩+≡

```
function analysis_store_get_object_type (id) result (type)
  type(string_t), intent(in) :: id
  integer :: type
  type(analysis_object_t), pointer :: object
  object => analysis_store_get_object_ptr (id)
  if (associated (object)) then
     type = object%type
  else
     type = AN_UNDEFINED
  end if
end function analysis_store_get_object_type
```

Return the number of objects in the store.

⟨*Analysis: procedures*⟩+≡

```
function analysis_store_get_n_objects () result (n)
  integer :: n
  type(analysis_object_t), pointer :: current
  n = 0
  current => analysis_store%first
  do while (associated (current))
     n = n + 1
     current => current%next
  end do
end function analysis_store_get_n_objects
```

Allocate an array and fill it with all existing IDs.

⟨*Analysis: public*⟩+≡

```
public :: analysis_store_get_ids
```

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_store_get_ids (id)
  type(string_t), dimension(:), allocatable, intent(out) :: id
  type(analysis_object_t), pointer :: current
  integer :: i
  allocate (id (analysis_store_get_n_objects()))
  i = 0
  current => analysis_store%first
  do while (associated (current))
     i = i + 1
     id(i) = current%id
     current => current%next
  end do
end subroutine analysis_store_get_ids
```

12.5.12 LaTeX driver file

Write a driver file for all objects in the store.

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_store_write_driver_all (filename_data, unit)
  type(string_t), intent(in) :: filename_data
  integer, intent(in), optional :: unit
  type(analysis_object_t), pointer :: obj
  call analysis_store_write_driver_header (unit)
  obj => analysis_store%first
  do while (associated (obj))
     call analysis_object_write_driver (obj, filename_data, unit)
     obj => obj%next
  end do
  call analysis_store_write_driver_footer (unit)
end subroutine analysis_store_write_driver_all
```

Write a driver file for an array of objects.

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_store_write_driver_obj (filename_data, id, unit)
  type(string_t), intent(in) :: filename_data
  type(string_t), dimension(:), intent(in) :: id
  integer, intent(in), optional :: unit
  type(analysis_object_t), pointer :: obj
  integer :: i
  call analysis_store_write_driver_header (unit)
  do i = 1, size (id)
     obj => analysis_store_get_object_ptr (id(i))
     if (associated (obj))  &
          call analysis_object_write_driver (obj, filename_data, unit)
  end do
  call analysis_store_write_driver_footer (unit)
end subroutine analysis_store_write_driver_obj
```

The beginning of the driver file.

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_store_write_driver_header (unit)
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, '(A)') "\documentclass[12pt]{article}"
  write (u, *)
  write (u, '(A)') "\usepackage{gamelan}"
  write (u, '(A)') "\usepackage{amsmath}"
  write (u, '(A)') "\usepackage{ifpdf}"
  write (u, '(A)') "\ifpdf"
  write (u, '(A)') "   \DeclareGraphicsRule{*}{mps}{*}{}"
  write (u, '(A)') "\else"
  write (u, '(A)') "   \DeclareGraphicsRule{*}{eps}{*}{}"
  write (u, '(A)') "\fi"
  write (u, *)
  write (u, '(A)') "\begin{document}"
  write (u, '(A)') "\begin{gmlfile}"
```

```
    write (u, *)
    write (u, '(A)') "\begin{gmlcode}"
    write (u, '(A)') "  color col.default, col.excess;"
    write (u, '(A)') "  col.default = 0.9white;"
    write (u, '(A)') "  col.excess  = red;"
    write (u, '(A)') "  boolean show_excess;"
    !!! Future excess options for plots
    ! if (mcs(1)%plot_excess .and. mcs(1)%unweighted) then
    !    write (u, '(A)') "  show_excess = true;"
    ! else
    write (u, '(A)') "  show_excess = false;"
    ! end if
    write (u, '(A)') "\end{gmlcode}"
    write (u, *)
  end subroutine analysis_store_write_driver_header
```

The end of the driver file.

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_store_write_driver_footer (unit)
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    write(u, *)
    write(u, '(A)') "\end{gmlfile}"
    write(u, '(A)') "\end{document}"
  end subroutine analysis_store_write_driver_footer
```

12.5.13 API

Creating new objects

The specific versions below:

⟨*Analysis: public*⟩+≡

```
  public :: analysis_init_observable
```

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_init_observable (id, obs_label, obs_unit, graph_options)
    type(string_t), intent(in) :: id
    type(string_t), intent(in), optional :: obs_label, obs_unit
    type(graph_options_t), intent(in), optional :: graph_options
    type(analysis_object_t), pointer :: obj
    type(observable_t), pointer :: obs
    call analysis_store_init_object (id, AN_OBSERVABLE, obj)
    obs => analysis_object_get_observable_ptr (obj)
    call observable_init (obs, obs_label, obs_unit, graph_options)
  end subroutine analysis_init_observable
```

⟨*Analysis: public*⟩+≡

```
  public :: analysis_init_histogram
```

⟨*Analysis: interfaces*⟩+≡

```
  interface analysis_init_histogram
     module procedure analysis_init_histogram_n_bins
```

```
   module procedure analysis_init_histogram_bin_width
end interface
```

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_init_histogram_n_bins &
     (id, lower_bound, upper_bound, n_bins, normalize_bins, &
      obs_label, obs_unit, graph_options, drawing_options)
  type(string_t), intent(in) :: id
  real(default), intent(in) :: lower_bound, upper_bound
  integer, intent(in) :: n_bins
  logical, intent(in) :: normalize_bins
  type(string_t), intent(in), optional :: obs_label, obs_unit
  type(graph_options_t), intent(in), optional :: graph_options
  type(drawing_options_t), intent(in), optional :: drawing_options
  type(analysis_object_t), pointer :: obj
  type(histogram_t), pointer :: h
  call analysis_store_init_object (id, AN_HISTOGRAM, obj)
  h => analysis_object_get_histogram_ptr (obj)
  call histogram_init (h, id, &
       lower_bound, upper_bound, n_bins, normalize_bins, &
       obs_label, obs_unit, graph_options, drawing_options)
end subroutine analysis_init_histogram_n_bins

subroutine analysis_init_histogram_bin_width &
     (id, lower_bound, upper_bound, bin_width, normalize_bins, &
      obs_label, obs_unit, graph_options, drawing_options)
  type(string_t), intent(in) :: id
  real(default), intent(in) :: lower_bound, upper_bound, bin_width
  logical, intent(in) :: normalize_bins
  type(string_t), intent(in), optional :: obs_label, obs_unit
  type(graph_options_t), intent(in), optional :: graph_options
  type(drawing_options_t), intent(in), optional :: drawing_options
  type(analysis_object_t), pointer :: obj
  type(histogram_t), pointer :: h
  call analysis_store_init_object (id, AN_HISTOGRAM, obj)
  h => analysis_object_get_histogram_ptr (obj)
  call histogram_init (h, id, &
       lower_bound, upper_bound, bin_width, normalize_bins, &
       obs_label, obs_unit, graph_options, drawing_options)
end subroutine analysis_init_histogram_bin_width
```

⟨*Analysis: public*⟩+≡

```
public :: analysis_init_plot
```

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_init_plot (id, graph_options, drawing_options)
  type(string_t), intent(in) :: id
  type(graph_options_t), intent(in), optional :: graph_options
  type(drawing_options_t), intent(in), optional :: drawing_options
  type(analysis_object_t), pointer :: obj
  type(plot_t), pointer :: plot
  call analysis_store_init_object (id, AN_PLOT, obj)
  plot => analysis_object_get_plot_ptr (obj)
  call plot_init (plot, id, graph_options, drawing_options)
```

```
end subroutine analysis_init_plot
```

⟨*Analysis: public*⟩+≡

```
public :: analysis_init_graph
```

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_init_graph (id, n_elements, graph_options)
  type(string_t), intent(in) :: id
  integer, intent(in) :: n_elements
  type(graph_options_t), intent(in), optional :: graph_options
  type(analysis_object_t), pointer :: obj
  type(graph_t), pointer :: graph
  call analysis_store_init_object (id, AN_GRAPH, obj)
  graph => analysis_object_get_graph_ptr (obj)
  call graph_init (graph, id, n_elements, graph_options)
end subroutine analysis_init_graph
```

Recording data

This procedure resets an object or the whole store to its initial state.

⟨*Analysis: public*⟩+≡

```
public :: analysis_clear
```

⟨*Analysis: interfaces*⟩+≡

```
interface analysis_clear
   module procedure analysis_store_clear_obj
   module procedure analysis_store_clear_all
end interface
```

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_store_clear_obj (id)
  type(string_t), intent(in) :: id
  type(analysis_object_t), pointer :: obj
  obj => analysis_store_get_object_ptr (id)
  if (associated (obj)) then
     call analysis_object_clear (obj)
  end if
end subroutine analysis_store_clear_obj

subroutine analysis_store_clear_all ()
  type(analysis_object_t), pointer :: obj
  obj => analysis_store%first
  do while (associated (obj))
     call analysis_object_clear (obj)
     obj => obj%next
  end do
end subroutine analysis_store_clear_all
```

There is one generic recording function whose behavior depends on the type of analysis object.

⟨*Analysis: public*⟩+≡

```
public :: analysis_record_data
```

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_record_data (id, x, y, yerr, xerr, &
     weight, excess, success, exist)
  type(string_t), intent(in) :: id
  real(default), intent(in) :: x
  real(default), intent(in), optional :: y, yerr, xerr, weight, excess
  logical, intent(out), optional :: success, exist
  type(analysis_object_t), pointer :: obj
  obj => analysis_store_get_object_ptr (id)
  if (associated (obj)) then
     call analysis_object_record_data (obj, x, y, yerr, xerr, &
          weight, excess, success)
     if (present (exist))  exist = .true.
  else
     if (present (success))  success = .false.
     if (present (exist))  exist = .false.
  end if
end subroutine analysis_record_data
```

Build a graph

This routine sets up the array of graph elements by copying the graph elements given as input. The object must exist and already be initialized as a graph.

⟨*Analysis: public*⟩+≡

```
public :: analysis_fill_graph
```

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_fill_graph (id, i, id_in, drawing_options)
  type(string_t), intent(in) :: id
  integer, intent(in) :: i
  type(string_t), intent(in) :: id_in
  type(drawing_options_t), intent(in), optional :: drawing_options
  type(analysis_object_t), pointer :: obj
  type(graph_t), pointer :: g
  type(histogram_t), pointer :: h
  type(plot_t), pointer :: p
  obj => analysis_store_get_object_ptr (id)
  g => analysis_object_get_graph_ptr (obj)
  obj => analysis_store_get_object_ptr (id_in)
  if (associated (obj)) then
     select case (obj%type)
     case (AN_HISTOGRAM)
        h => analysis_object_get_histogram_ptr (obj)
        call graph_insert_histogram (g, i, h, drawing_options)
     case (AN_PLOT)
        p => analysis_object_get_plot_ptr (obj)
        call graph_insert_plot (g, i, p, drawing_options)
     case default
        call msg_error ("Graph '" // char (id) // "': Element '" &
             // char (id_in) // "' is neither histogram nor plot.")
     end select
  else
     call msg_error ("Graph '" // char (id) // "': Element '" &
```

```
            // char (id_in) // "' is undefined.")
  end if
end subroutine analysis_fill_graph
```

Retrieve generic results

Check if a named object exists.

⟨*Analysis: public*⟩+≡

```
public :: analysis_exists
```

⟨*Analysis: procedures*⟩+≡

```
function analysis_exists (id) result (flag)
  type(string_t), intent(in) :: id
  logical :: flag
  type(analysis_object_t), pointer :: obj
  flag = .true.
  obj => analysis_store%first
  do while (associated (obj))
     if (obj%id == id)  return
     obj => obj%next
  end do
  flag = .false.
end function analysis_exists
```

The following functions should work for all kinds of analysis object:

⟨*Analysis: public*⟩+≡

```
public :: analysis_get_n_elements
public :: analysis_get_n_entries
public :: analysis_get_average
public :: analysis_get_error
```

⟨*Analysis: procedures*⟩+≡

```
function analysis_get_n_elements (id) result (n)
  integer :: n
  type(string_t), intent(in) :: id
  type(analysis_object_t), pointer :: obj
  obj => analysis_store_get_object_ptr (id)
  if (associated (obj)) then
     n = analysis_object_get_n_elements (obj)
  else
     n = 0
  end if
end function analysis_get_n_elements

function analysis_get_n_entries (id, within_bounds) result (n)
  integer :: n
  type(string_t), intent(in) :: id
  logical, intent(in), optional :: within_bounds
  type(analysis_object_t), pointer :: obj
  obj => analysis_store_get_object_ptr (id)
  if (associated (obj)) then
     n = analysis_object_get_n_entries (obj, within_bounds)
  else
```

```
      n = 0
    end if
  end function analysis_get_n_entries

  function analysis_get_average (id, within_bounds) result (avg)
    real(default) :: avg
    type(string_t), intent(in) :: id
    type(analysis_object_t), pointer :: obj
    logical, intent(in), optional :: within_bounds
    obj => analysis_store_get_object_ptr (id)
    if (associated (obj)) then
       avg = analysis_object_get_average (obj, within_bounds)
    else
       avg = 0
    end if
  end function analysis_get_average

  function analysis_get_error (id, within_bounds) result (err)
    real(default) :: err
    type(string_t), intent(in) :: id
    type(analysis_object_t), pointer :: obj
    logical, intent(in), optional :: within_bounds
    obj => analysis_store_get_object_ptr (id)
    if (associated (obj)) then
       err = analysis_object_get_error (obj, within_bounds)
    else
       err = 0
    end if
  end function analysis_get_error
```

Return true if any analysis object is graphical

⟨*Analysis: public*⟩+≡

```
  public :: analysis_has_plots
```

⟨*Analysis: interfaces*⟩+≡

```
  interface analysis_has_plots
     module procedure analysis_has_plots_any
     module procedure analysis_has_plots_obj
  end interface
```

⟨*Analysis: procedures*⟩+≡

```
  function analysis_has_plots_any () result (flag)
    logical :: flag
    type(analysis_object_t), pointer :: obj
    flag = .false.
    obj => analysis_store%first
    do while (associated (obj))
       flag = analysis_object_has_plot (obj)
       if (flag)  return
    end do
  end function analysis_has_plots_any

  function analysis_has_plots_obj (id) result (flag)
    logical :: flag
```

```
    type(string_t), dimension(:), intent(in) :: id
    type(analysis_object_t), pointer :: obj
    integer :: i
    flag = .false.
    do i = 1, size (id)
       obj => analysis_store_get_object_ptr (id(i))
       if (associated (obj)) then
          flag = analysis_object_has_plot (obj)
          if (flag)  return
       end if
    end do
  end function analysis_has_plots_obj
```

Iterators

Initialize an iterator for the given object. If the object does not exist or has wrong type, the iterator will be invalid.

⟨*Analysis: public*⟩+≡

```
  public :: analysis_init_iterator
```

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_init_iterator (id, iterator)
    type(string_t), intent(in) :: id
    type(analysis_iterator_t), intent(out) :: iterator
    type(analysis_object_t), pointer :: obj
    obj => analysis_store_get_object_ptr (id)
    if (associated (obj))  call analysis_iterator_init (iterator, obj)
  end subroutine analysis_init_iterator
```

Output

⟨*Analysis: public*⟩+≡

```
  public :: analysis_write
```

⟨*Analysis: interfaces*⟩+≡

```
  interface analysis_write
     module procedure analysis_write_object
     module procedure analysis_write_all
  end interface
```

⟨*Analysis: procedures*⟩+≡

```
  subroutine analysis_write_object (id, unit, verbose)
    type(string_t), intent(in) :: id
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    type(analysis_object_t), pointer :: obj
    obj => analysis_store_get_object_ptr (id)
    if (associated (obj)) then
       call analysis_object_write (obj, unit, verbose)
    else
       call msg_error ("Analysis object '" // char (id) // "' not found")
    end if
```

```
end subroutine analysis_write_object

subroutine analysis_write_all (unit, verbose)
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  type(analysis_object_t), pointer :: obj
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  obj => analysis_store%first
  do while (associated (obj))
     call analysis_object_write (obj, unit, verbose)
     obj => obj%next
  end do
end subroutine analysis_write_all
```

⟨*Analysis: public*⟩+≡

```
public :: analysis_write_driver
```

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_write_driver (filename_data, id, unit)
  type(string_t), intent(in) :: filename_data
  type(string_t), dimension(:), intent(in), optional :: id
  integer, intent(in), optional :: unit
  if (present (id)) then
     call analysis_store_write_driver_obj (filename_data, id, unit)
  else
     call analysis_store_write_driver_all (filename_data, unit)
  end if
end subroutine analysis_write_driver
```

⟨*Analysis: public*⟩+≡

```
public :: analysis_compile_tex
```

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_compile_tex (file, has_gmlcode, os_data)
  type(string_t), intent(in) :: file
  logical, intent(in) :: has_gmlcode
  type(os_data_t), intent(in) :: os_data
  integer :: status
  if (os_data%event_analysis_ps) then
     call os_system_call ("make compile " // os_data%makeflags // " -f " // &
          char (file) // "_ana.makefile", status)
     if (status /= 0) then
        call msg_error ("Unable to compile analysis output file")
     end if
  else
     call msg_warning ("Skipping results display because " &
          // "latex/mpost/dvips is not available")
  end if
end subroutine analysis_compile_tex
```

Write header for generic data output to an ifile.

⟨*Analysis: public*⟩+≡

```
public :: analysis_get_header
```

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_get_header (id, header, comment)
  type(string_t), intent(in) :: id
  type(ifile_t), intent(inout) :: header
  type(string_t), intent(in), optional :: comment
  type(analysis_object_t), pointer :: object
  object => analysis_store_get_object_ptr (id)
  if (associated (object)) then
     call analysis_object_get_header (object, header, comment)
  end if
end subroutine analysis_get_header
```

Write a makefile in order to do the compile steps.

⟨*Analysis: public*⟩+≡

```
public :: analysis_write_makefile
```

⟨*Analysis: procedures*⟩+≡

```
subroutine analysis_write_makefile (filename, unit, has_gmlcode, os_data)
  type(string_t), intent(in) :: filename
  integer, intent(in) :: unit
  logical, intent(in) :: has_gmlcode
  type(os_data_t), intent(in) :: os_data
  write (unit, "(3A)")  "# WHIZARD: Makefile for analysis '", &
       char (filename), "'"
  write (unit, "(A)")  "# Automatically generated file, do not edit"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "# LaTeX setup"
  write (unit, "(A)")  "LATEX = " // char (os_data%latex)
  write (unit, "(A)")  "MPOST = " // char (os_data%mpost)
  write (unit, "(A)")  "GML = " // char (os_data%gml)
  write (unit, "(A)")  "DVIPS = " // char (os_data%dvips)
  write (unit, "(A)")  "PS2PDF = " // char (os_data%ps2pdf)
  write (unit, "(A)")  'TEX_FLAGS = "$$TEXINPUTS:' // &
       char(os_data%whizard_texpath) // '"'
  write (unit, "(A)")  'MP_FLAGS  = "$$MPINPUTS:' // &
       char(os_data%whizard_texpath) // '"'
  write (unit, "(A)")  ""
  write (unit, "(5A)")  "TEX_SOURCES = ", char (filename), ".tex"
  if (os_data%event_analysis_pdf) then
     write (unit, "(5A)")  "TEX_OBJECTS = ", char (filename), ".pdf"
  else
     write (unit, "(5A)")  "TEX_OBJECTS = ", char (filename), ".ps"
  end if
  if (os_data%event_analysis_ps) then
     if (os_data%event_analysis_pdf) then
        write (unit, "(5A)")  char (filename), ".pdf: ", &
             char (filename), ".tex"
     else
        write (unit, "(5A)")  char (filename), ".ps: ", &
             char (filename), ".tex"
     end if
     write (unit, "(5A)")  TAB, "-TEXINPUTS=$(TEX_FLAGS) $(LATEX) " // &
          char (filename) // ".tex"
     if (has_gmlcode) then
```

```
         write (unit, "(5A)")  TAB, "$(GML) " // char (filename)
         write (unit, "(5A)")  TAB, "TEXINPUTS=$(TEX_FLAGS) $(LATEX) " // &
           char (filename) // ".tex"
      end if
      write (unit, "(5A)")  TAB, "$(DVIPS) -o " // char (filename) // ".ps " // &
           char (filename) // ".dvi"
      if (os_data%event_analysis_pdf) then
         write (unit, "(5A)")  TAB, "$(PS2PDF) " // char (filename) // ".ps"
      end if
   end if
   write (unit, "(A)")
   write (unit, "(A)")  "compile: $(TEX_OBJECTS)"
   write (unit, "(A)")  ".PHONY: compile"
   write (unit, "(A)")
   write (unit, "(5A)")  "CLEAN_OBJECTS = ",  char (filename), ".aux"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".log"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".dvi"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".out"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".[1-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".[1-9][0-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".[1-9][0-9][0-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".t[1-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".t[1-9][0-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".t[1-9][0-9][0-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".ltp"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".mp"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".mpx"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".dvi"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".ps"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (filename), ".pdf"
   write (unit, "(A)")
   write (unit, "(A)")  "# Generic cleanup targets"
   write (unit, "(A)")  "clean-objects:"
   write (unit, "(A)")  TAB // "rm -f $(CLEAN_OBJECTS)"
   write (unit, "(A)")  ""
   write (unit, "(A)")  "clean: clean-objects"
   write (unit, "(A)")  ".PHONY: clean"
end subroutine analysis_write_makefile
```

12.5.14 Unit tests

Test module, followed by the corresponding implementation module.

⟨analysis_ut.f90⟩≡

```
⟨File header⟩

module analysis_ut
  use unit_tests
  use analysis_uti

⟨Standard module head⟩

⟨Analysis: public test⟩
```

```
  contains

  ⟨Analysis: test driver⟩

  end module analysis_ut
```

⟨analysis_uti.f90⟩≡

```
  ⟨File header⟩

  module analysis_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use format_defs, only: FMT_19

    use analysis

  ⟨Standard module head⟩

  ⟨Analysis: test declarations⟩

  contains

  ⟨Analysis: tests⟩

  end module analysis_uti
```

API: driver for the unit tests below.

⟨*Analysis: public test*⟩≡

```
    public :: analysis_test
```

⟨*Analysis: test driver*⟩≡

```
    subroutine analysis_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Analysis: execute tests⟩
    end subroutine analysis_test
```

⟨*Analysis: execute tests*⟩≡

```
    call test (analysis_1, "analysis_1", &
         "check elementary analysis building blocks", &
         u, results)
```

⟨*Analysis: test declarations*⟩≡

```
    public :: analysis_1
```

⟨*Analysis: tests*⟩≡

```
    subroutine analysis_1 (u)
      integer, intent(in) :: u
      type(string_t) :: id1, id2, id3, id4
      integer :: i
      id1 = "foo"
      id2 = "bar"
      id3 = "hist"
      id4 = "plot"
```

```
write (u, "(A)")  "* Test output: Analysis"
write (u, "(A)")  "*   Purpose: test the analysis routines"
write (u, "(A)")

call analysis_init_observable (id1)
call analysis_init_observable (id2)
call analysis_init_histogram &
     (id3, 0.5_default, 5.5_default, 1._default, normalize_bins=.false.)
call analysis_init_plot (id4)
do i = 1, 3
   write (u, "(A,1x," // FMT_19 // ")")  "data = ", real(i,default)
   call analysis_record_data (id1, real(i,default))
   call analysis_record_data (id2, real(i,default), &
                                   weight=real(i,default))
   call analysis_record_data (id3, real(i,default))
   call analysis_record_data (id4, real(i,default), real(i,default)**2)
end do
write (u, "(A,10(1x,I5))") "n_entries = ", &
     analysis_get_n_entries (id1), &
     analysis_get_n_entries (id2), &
     analysis_get_n_entries (id3), &
     analysis_get_n_entries (id3, within_bounds = .true.), &
     analysis_get_n_entries (id4), &
     analysis_get_n_entries (id4, within_bounds = .true.)
write (u, "(A,10(1x," // FMT_19 // "))")  "average   = ", &
     analysis_get_average (id1), &
     analysis_get_average (id2), &
     analysis_get_average (id3), &
     analysis_get_average (id3, within_bounds = .true.)
write (u, "(A,10(1x," // FMT_19 // "))")  "error     = ", &
     analysis_get_error (id1), &
     analysis_get_error (id2), &
     analysis_get_error (id3), &
     analysis_get_error (id3, within_bounds = .true.)

write (u, "(A)")
write (u, "(A)") "* Clear analysis #2"
write (u, "(A)")

call analysis_clear (id2)
do i = 4, 6
   print *, "data = ", real(i,default)
   call analysis_record_data (id1, real(i,default))
   call analysis_record_data (id2, real(i,default), &
                                   weight=real(i,default))
   call analysis_record_data (id3, real(i,default))
   call analysis_record_data (id4, real(i,default), real(i,default)**2)
end do
write (u, "(A,10(1x,I5))")  "n_entries = ", &
     analysis_get_n_entries (id1), &
     analysis_get_n_entries (id2), &
     analysis_get_n_entries (id3), &
     analysis_get_n_entries (id3, within_bounds = .true.), &
```

```
      analysis_get_n_entries (id4), &
      analysis_get_n_entries (id4, within_bounds = .true.)
  write (u, "(A,10(1x," // FMT_19 // "))")  "average   = ", &
      analysis_get_average (id1), &
      analysis_get_average (id2), &
      analysis_get_average (id3), &
      analysis_get_average (id3, within_bounds = .true.)
  write (u, "(A,10(1x," // FMT_19 // "))")  "error     = ", &
      analysis_get_error (id1), &
      analysis_get_error (id2), &
      analysis_get_error (id3), &
      analysis_get_error (id3, within_bounds = .true.)
  write (u, "(A)")
  call analysis_write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call analysis_clear ()
  call analysis_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: analysis_1"
end subroutine analysis_1
```

Chapter 13

Matrix Element Handling

In this chapter, we support internal and external matrix elements: initialization, automatic generation where necessary, and numerical evaluation. We provide the interface for code generation and linking. Matrix-element code is organized in processes and process libraries.

process_constants A record of static process properties, for easy transfer between various `WHIZARD` modules.

prclib_interfaces This module deals with matrix-element code which is accessible via external libraries (Fortran libraries or generic C-compatible libraries) and must either be generated by the program or provided by the user explicitly.

The module defines and uses an abstract type `prc_writer_t` and two abstract extensions, one for a Fortran module and one for a C-compatible library. The implementation provides the specific methods for writing the appropriate parts in external matrix element code.

prc_core_def This module defines the abstract types `prc_core_def_t` and `prc_driver_t`. The implementation of the former provides the configuration for processes of a certain class, while the latter accesses the corresponding matrix element, in particular those generated by the appropriate `prc_writer_t` object.

process_libraries This module combines the functionality of the previous module with the means for holding processes definitions (the internal counterpart of appropriate declarations in the user interface), for handling matrix elements which do not need external code, and for accessing the matrix elements by the procedures for matrix-element evaluation, integration and event generation.

prclib_stacks Collect process libraries.

test_me This module provides a test implementation for the abstract types in the `prc_core_def` module. The implementation is intended for self-tests of several later modules. The implementation is internal, i.e., no external code has is generated.

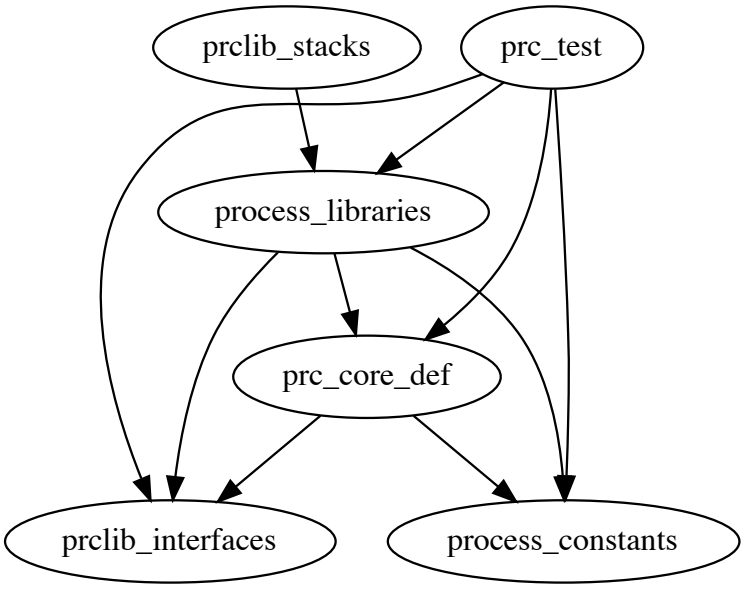

Figure 13.1: Module dependencies in `src/matrix_elements`.

All data structures which are specific for a particular way of generating code or evaluating matrix element are kept abstract and thus generic. Later modules such as `prc_omega` provide implementations, in the form of type extensions for the various abstract types.

13.1 Process data block

We define a simple transparent type that contains universal constant process data. We will reference objects of this type for the phase-space setup, for interfacing with process libraries, for implementing matrix-element generation, and in the master process-handling module.

⟨process_constants.f90⟩≡

```
⟨File header⟩

module process_constants

⟨Use kinds⟩
⟨Use strings⟩

  use io_units, only: given_output_unit, free_unit
  use format_utils, only: write_integer_array
  use md5, only: md5sum

  use pdg_arrays

⟨Standard module head⟩

⟨Process Constants: public⟩

⟨Process Constants: types⟩

contains

⟨Process Constants: procedures⟩

end module process_constants
```

The data type is just a block of public objects, only elementary types, no type-bound procedures.

⟨*Process Constants: public*⟩≡

```
public :: process_constants_t
```

⟨*Process Constants: types*⟩≡

```
type :: process_constants_t
   type(string_t) :: id
   type(string_t) :: model_name
   character(32) :: md5sum = ""
   logical :: openmp_supported = .false.
   integer :: n_in  = 0
   integer :: n_out = 0
   integer :: n_flv = 0
   integer :: n_hel = 0
   integer :: n_col = 0
   integer :: n_cin = 0
   integer :: n_cf  = 0
   integer, dimension(:,:), allocatable :: flv_state
   integer, dimension(:,:), allocatable :: hel_state
   integer, dimension(:,:,:), allocatable :: col_state
   logical, dimension(:,:), allocatable :: ghost_flag
   complex(default), dimension(:), allocatable :: color_factors
```

```
    integer, dimension(:,:), allocatable :: cf_index
  contains
  ⟨Process Constants: process constants: TBP⟩
  end type process_constants_t
```

⟨*Process Constants: process constants: TBP*⟩≡

```
  procedure :: get_n_tot => process_constants_get_n_tot
```

⟨*Process Constants: procedures*⟩≡

```
  elemental function process_constants_get_n_tot (prc_const) result (n_tot)
    integer :: n_tot
    class(process_constants_t), intent(in) :: prc_const
    n_tot = prc_const%n_in + prc_const%n_out
  end function process_constants_get_n_tot
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
  procedure :: get_flv_state => process_constants_get_flv_state
```

⟨*Process Constants: procedures*⟩+≡

```
  subroutine process_constants_get_flv_state (prc_const, flv_state)
    class(process_constants_t), intent(in) :: prc_const
    integer, dimension(:,:), allocatable, intent(out) :: flv_state
    allocate (flv_state (size (prc_const%flv_state, 1), &
         size (prc_const%flv_state, 2)))
    flv_state = prc_const%flv_state
  end subroutine process_constants_get_flv_state
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
  procedure :: get_n_flv => process_constants_get_n_flv
```

⟨*Process Constants: procedures*⟩+≡

```
  function process_constants_get_n_flv (data) result (n_flv)
    integer :: n_flv
    class(process_constants_t), intent(in) :: data
    n_flv = data%n_flv
  end function process_constants_get_n_flv
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
  procedure :: get_hel_state => process_constants_get_hel_state
```

⟨*Process Constants: procedures*⟩+≡

```
  subroutine process_constants_get_hel_state (prc_const, hel_state)
    class(process_constants_t), intent(in) :: prc_const
    integer, dimension(:,:), allocatable, intent(out) :: hel_state
    allocate (hel_state (size (prc_const%hel_state, 1), &
         size (prc_const%hel_state, 2)))
    hel_state = prc_const%hel_state
  end subroutine process_constants_get_hel_state
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
  procedure :: get_col_state => process_constants_get_col_state
```

⟨*Process Constants: procedures*⟩+≡

```
subroutine process_constants_get_col_state (prc_const, col_state)
  class(process_constants_t), intent(in) :: prc_const
  integer, dimension(:,:,:), allocatable, intent(out) :: col_state
  allocate (col_state (size (prc_const%col_state, 1), &
       size (prc_const%col_state, 2), size (prc_const%col_state, 3)))
  col_state = prc_const%col_state
end subroutine process_constants_get_col_state
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
procedure :: get_ghost_flag => process_constants_get_ghost_flag
```

⟨*Process Constants: procedures*⟩+≡

```
subroutine process_constants_get_ghost_flag (prc_const, ghost_flag)
  class(process_constants_t), intent(in) :: prc_const
  logical, dimension(:,:), allocatable, intent(out) :: ghost_flag
  allocate (ghost_flag (size (prc_const%ghost_flag, 1), &
       size (prc_const%ghost_flag, 2)))
  ghost_flag = prc_const%ghost_flag
end subroutine process_constants_get_ghost_flag
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
procedure :: get_color_factors => process_constants_get_color_factors
```

⟨*Process Constants: procedures*⟩+≡

```
subroutine process_constants_get_color_factors (prc_const, col_facts)
  class(process_constants_t), intent(in) :: prc_const
  complex(default), dimension(:), allocatable, intent(out) :: col_facts
  allocate (col_facts (size (prc_const%color_factors)))
  col_facts = prc_const%color_factors
end subroutine process_constants_get_color_factors
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
procedure :: get_cf_index => process_constants_get_cf_index
```

⟨*Process Constants: procedures*⟩+≡

```
subroutine process_constants_get_cf_index (prc_const, cf_index)
  class(process_constants_t), intent(in) :: prc_const
  integer, intent(out), dimension(:,:), allocatable :: cf_index
  allocate (cf_index (size (prc_const%cf_index, 1), &
       size (prc_const%cf_index, 2)))
  cf_index = prc_const%cf_index
end subroutine process_constants_get_cf_index
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
procedure :: set_flv_state => process_constants_set_flv_state
```

⟨*Process Constants: procedures*⟩+≡

```
subroutine process_constants_set_flv_state (prc_const, flv_state)
  class(process_constants_t), intent(inout) :: prc_const
  integer, intent(in), dimension(:,:), allocatable :: flv_state
  if (allocated (prc_const%flv_state)) deallocate (prc_const%flv_state)
  allocate (prc_const%flv_state (size (flv_state, 1), &
       size (flv_state, 2)))
```

```
    prc_const%flv_state = flv_state
    prc_const%n_flv = size (flv_state, 2)
  end subroutine process_constants_set_flv_state
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
  procedure :: set_col_state => process_constants_set_col_state
```

⟨*Process Constants: procedures*⟩+≡

```
  subroutine process_constants_set_col_state (prc_const, col_state)
    class(process_constants_t), intent(inout) :: prc_const
    integer, intent(in), dimension(:,:,:), allocatable :: col_state
    allocate (prc_const%col_state (size (col_state, 1), &
         size (col_state, 2), size (col_state, 3)))
    prc_const%col_state = col_state
  end subroutine process_constants_set_col_state
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
  procedure :: set_cf_index => process_constants_set_cf_index
```

⟨*Process Constants: procedures*⟩+≡

```
  subroutine process_constants_set_cf_index (prc_const, cf_index)
    class(process_constants_t), intent(inout) :: prc_const
    integer, dimension(:,:), intent(in), allocatable :: cf_index
    allocate (prc_const%cf_index (size (cf_index, 1), &
         size (cf_index, 2)))
    prc_const%cf_index = cf_index
  end subroutine process_constants_set_cf_index
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
  procedure :: set_color_factors => process_constants_set_color_factors
```

⟨*Process Constants: procedures*⟩+≡

```
  subroutine process_constants_set_color_factors (prc_const, color_factors)
    class(process_constants_t), intent(inout) :: prc_const
    complex(default), dimension(:), intent(in), allocatable :: color_factors
    allocate (prc_const%color_factors (size (color_factors)))
    prc_const%color_factors = color_factors
  end subroutine process_constants_set_color_factors
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
  procedure :: set_ghost_flag => process_constants_set_ghost_flag
```

⟨*Process Constants: procedures*⟩+≡

```
  subroutine process_constants_set_ghost_flag (prc_const, ghost_flag)
    class(process_constants_t), intent(inout) :: prc_const
    logical, dimension(:,:), allocatable, intent(in) :: ghost_flag
    allocate (prc_const%ghost_flag (size (ghost_flag, 1), &
         size (ghost_flag, 2)))
    prc_const%ghost_flag = ghost_flag
  end subroutine process_constants_set_ghost_flag
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
  procedure :: get_pdg_in => process_constants_get_pdg_in
```

⟨*Process Constants: procedures*⟩+≡

```
function process_constants_get_pdg_in (prc_const) result (pdg_in)
  type(pdg_array_t), dimension(:), allocatable :: pdg_in
  class(process_constants_t), intent(in) :: prc_const
  type(pdg_array_t) :: pdg_tmp
  integer :: i
  allocate (pdg_in (prc_const%n_in))
  do i = 1, prc_const%n_in
     pdg_tmp = prc_const%flv_state(i,:)
     pdg_in(i) = sort_abs (pdg_tmp, unique = .true.)
  end do
end function process_constants_get_pdg_in
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
procedure :: compute_md5sum => process_constants_compute_md5sum
```

⟨*Process Constants: procedures*⟩+≡

```
subroutine process_constants_compute_md5sum (prc_const, include_id)
  class(process_constants_t), intent(inout) :: prc_const
  logical, intent(in) :: include_id
  integer :: unit
  unit = prc_const%fill_unit_for_md5sum (include_id)
  rewind (unit)
  prc_const%md5sum = md5sum (unit)
  close (unit)
end subroutine process_constants_compute_md5sum
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
procedure :: fill_unit_for_md5sum => process_constants_fill_unit_for_md5sum
```

⟨*Process Constants: procedures*⟩+≡

```
function process_constants_fill_unit_for_md5sum (prc_const, include_id) result (unit)
  integer :: unit
  class(process_constants_t), intent(in) :: prc_const
  logical, intent(in) :: include_id
  integer :: i, j, k
  unit = free_unit ()
  open (unit, status="scratch", action="readwrite")
  if (include_id) write (unit, '(A)') char (prc_const%id)
  write (unit, '(A)') char (prc_const%model_name)
  write (unit, '(L1)') prc_const%openmp_supported
  write (unit, '(I0)') prc_const%n_in
  write (unit, '(I0)') prc_const%n_out
  write (unit, '(I0)') prc_const%n_flv
  write (unit, '(I0)') prc_const%n_hel
  write (unit, '(I0)') prc_const%n_col
  write (unit, '(I0)') prc_const%n_cin
  write (unit, '(I0)') prc_const%n_cf
  do i = 1, size (prc_const%flv_state, dim=1)
     do j = 1, size (prc_const%flv_state, dim=2)
        write (unit, '(I0)') prc_const%flv_state (i, j)
     end do
  end do
  do i = 1, size (prc_const%hel_state, dim=1)
```

```
      do j = 1, size (prc_const%hel_state, dim=2)
         write (unit, '(I0)') prc_const%hel_state (i, j)
      end do
   end do
   do i = 1, size (prc_const%col_state, dim=1)
      do j = 1, size (prc_const%col_state, dim=2)
         do k = 1, size (prc_const%col_state, dim=3)
            write (unit, '(I0)') prc_const%col_state (i, j, k)
         end do
     end do
   end do
   do i = 1, size (prc_const%ghost_flag, dim=1)
      do j = 1, size (prc_const%ghost_flag, dim=2)
         write (unit, '(L1)') prc_const%ghost_flag (i, j)
      end do
   end do
   do i = 1, size (prc_const%color_factors)
      write (unit, '(F0.0,F0.0)') real (prc_const%color_factors(i)), &
         aimag (prc_const%color_factors(i))
   end do
   do i = 1, size (prc_const%cf_index, dim=1)
      do j = 1, size (prc_const%cf_index, dim=2)
         write (unit, '(I0)') prc_const%cf_index(i, j)
      end do
   end do
end function process_constants_fill_unit_for_md5sum
```

⟨*Process Constants: process constants: TBP*⟩+≡

```
procedure :: write => process_constants_write
```

⟨*Process Constants: procedures*⟩+≡

```
subroutine process_constants_write (prc_const, unit)
  class(process_constants_t), intent(in) :: prc_const
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1x,A,A)") "Process data of id: ", char (prc_const%id)
  write (u, "(1x,A,A)") "Associated model: ", char (prc_const%model_name)
  write (u, "(1x,A,I0)") "n_in: ", prc_const%n_in
  write (u, "(1x,A,I0)") "n_out: ", prc_const%n_out
  write (u, "(1x,A,I0)") "n_flv: ", prc_const%n_flv
  write (u, "(1x,A,I0)") "n_hel: ", prc_const%n_hel
  write (u, "(1x,A,I0)") "n_col: ", prc_const%n_col
  write (u, "(1x,A,I0)") "n_cin: ", prc_const%n_cin
  write (u, "(1x,A,I0)") "n_cf: ", prc_const%n_cf
  write (u, "(1x,A)") "Flavors: "
  do i = 1, prc_const%n_flv
     write (u, "(1x,A,I0)") "i_flv: ", i
     call write_integer_array (prc_const%flv_state (:,i))
  end do
end subroutine process_constants_write
```

13.2 Process library interface

The module prclib_interfaces handles external matrix-element code.

13.2.1 Overview

The top-level data structure is the prclib_driver_t data type. The associated type-bound procedures deal with the generation of external code, compilation and linking, and accessing the active external library.

An object of type prclib_driver_t consists of the following parts:

1. Metadata that identify name and status of the library driver, etc.
2. An array of process records (prclib_driver_record_t), one for each external matrix element.
3. A record of type dlaccess_t which handles the operating-system part of linking a dynamically loadable library.
4. A collection of procedure pointers which have a counterpart in the external library interface. Given the unique identifier of a matrix element, the procedures retrieve generic matrix-element information such as the particle content and helicity combination tables. There is also a procedure which returns pointers to the more specific procedures that a matrix element provides, called *features*.

The process records of type prclib_driver_record_t handle the individual matrix elements. Each record identifies a process by name (id), names the physics model to be loaded for this process, lists the features that the associated matrix-element code provides, and holds a writer object which handles all operations that depend on the process type. The numbering of process records is identical to the numbering of matrix-element codes in the external library.

The writer object is of abstract type prc_writer_t. The module defines two basic, also abstract, extensions: prc_writer_f_module_t and prc_writer_c_lib_t. The first version is for matrix-element code that is available in form of Fortran modules. The writer contains type-bound procedures which create appropriate use directives and C-compatible wrapper functions for the given set of Fortran modules and their features. The second version is for matrix-element code that is available in form of a C-compatible library (this includes Fortran libraries with proper C bindings). The writer needs not write wrapper function, but explicit interface blocks for the matrix-element features.

Each matrix-element variant is encoded in an appropriate extension of prc_writer_t. For instance, O'MEGA matrix elements provide an implementation omega_writer_t which extends prc_writer_f_module_t.

13.2.2 Workflow

We expect that the functionality provided by this module is called in the following order:

1. The caller initializes the prclib_driver_t object and fills the array of prclib_record_t entries with the appropriate process data and process-specific writer objects.

2. It calls the `generate_makefile` method to set up an appropriate makefile in the current directory. The makefile will handle source generation, compilation and linking both for the individual matrix elements (unless this has to be done manually) and for the common external driver code which interfaces those matrix element.

3. The `generate_driver_code` writes the common driver as source code to file.

4. The methods `make_source`, `make_compile`, and `make_link` individually perform the corresponding steps in building the library. Wherever possible, they simply use the generated makefile. By calling `make`, we make sure that we can avoid unnecessary recompilation. For the compilation and linking steps, the makefile will employ `libtool`.

5. The `load` method loads the library procedures into the corresponding procedure pointers, using the `dlopen` mechanism via the `dlaccess` subobject.

13.2.3 The module

⟨`prclib_interfaces.f90`⟩≡

```
⟨File header⟩

module prclib_interfaces

  use, intrinsic :: iso_c_binding !NODEP!

  use kinds
⟨Use strings⟩
  use io_units
  use system_defs, only: TAB
  use string_utils, only: lower_case
  use diagnostics
  use os_interface

⟨Standard module head⟩

⟨Prclib interfaces: public⟩

⟨Prclib interfaces: types⟩

⟨Prclib interfaces: interfaces⟩

contains

⟨Prclib interfaces: procedures⟩

end module prclib_interfaces
```

13.2.4 Writers

External matrix element code provides externally visible procedures, which we denote as *features*. The features consist of informational subroutines and func-

tions which are mandatory (universal features) and matrix-element specific subroutines and functions (specific features). The driver interfaces the generic features directly, while it returns the specific features in form of bind(C) procedure pointers to the caller. For instance, function `n_in` is generic, while the matrix matrix-element value itself is specific.

To implement these tasks, the driver needs `use` directives for Fortran module procedures, interface blocks for other external stuff, wrapper code, and Makefile snippets.

Generic writer

In the `prc_writer_t` data type, we collect the procedures which implement the writing tasks. The type is abstract. The concrete implementations are defined by an extension which is specific for the process type.

The MD5 sum stored here should be the MD5 checksum of the current process component, which can be calculated once the process is configured completely. It can be used by implementations which work with external files, such as `O'MEGA`.

⟨*Prclib interfaces: public*⟩≡
```
public :: prc_writer_t
```

⟨*Prclib interfaces: types*⟩≡
```
type, abstract :: prc_writer_t
   character(32) :: md5sum = ""
 contains
 ⟨Prclib interfaces: prc writer: TBP⟩
end type prc_writer_t
```

In any case, it is useful to have a string representation of the writer type. This must be implemented by all extensions.

⟨*Prclib interfaces: prc writer: TBP*⟩≡
```
procedure(get_const_string), nopass, deferred :: type_name
```

⟨*Prclib interfaces: interfaces*⟩≡
```
abstract interface
   function get_const_string () result (string)
     import
     type(string_t) :: string
   end function get_const_string
end interface
```

Return the name of a procedure that implements a given feature, as it is provided by the external matrix-element code. For a reasonable default, we take the feature name unchanged.

⟨*Prclib interfaces: prc writer: TBP*⟩+≡
```
procedure, nopass :: get_procname => prc_writer_get_procname
```

⟨*Prclib interfaces: procedures*⟩≡
```
function prc_writer_get_procname (feature) result (name)
  type(string_t) :: name
  type(string_t), intent(in) :: feature
  name = feature
end function prc_writer_get_procname
```

Return the name of a procedure that implements a given feature with the bind(C) property, so it can be accessed via a C procedure pointer and handled by dlopen. We need this for all special features of a matrix element, since the interface has to return a C function pointer for it. For a default implementation, we prefix the external procedure name by the process ID.

⟨*Prclib interfaces: prc writer: TBP*⟩+≡

```
procedure :: get_c_procname => prc_writer_get_c_procname
```

⟨*Prclib interfaces: procedures*⟩+≡

```
function prc_writer_get_c_procname (writer, id, feature) result (name)
  class(prc_writer_t), intent(in) :: writer
  type(string_t), intent(in) :: id, feature
  type(string_t) :: name
  name = id // "_" // feature
end function prc_writer_get_c_procname
```

Common signature of code-writing procedures. The procedure may use the process ID, and the feature name. (Not necessarily all of them.)

⟨*Prclib interfaces: interfaces*⟩+≡

```
abstract interface
   subroutine write_code_file (writer, id)
     import
     class(prc_writer_t), intent(in) :: writer
     type(string_t), intent(in) :: id
   end subroutine write_code_file
end interface

abstract interface
   subroutine write_code (writer, unit, id)
     import
     class(prc_writer_t), intent(in) :: writer
     integer, intent(in) :: unit
     type(string_t), intent(in) :: id
   end subroutine write_code
end interface

abstract interface
   subroutine write_code_os (writer, unit, id, os_data, testflag)
     import
     class(prc_writer_t), intent(in) :: writer
     integer, intent(in) :: unit
     type(string_t), intent(in) :: id
     type(os_data_t), intent(in) :: os_data
     logical, intent(in), optional :: testflag
   end subroutine write_code_os
end interface

abstract interface
   subroutine write_feature_code (writer, unit, id, feature)
     import
     class(prc_writer_t), intent(in) :: writer
     integer, intent(in) :: unit
```

```
      type(string_t), intent(in) :: id, feature
    end subroutine write_feature_code
end interface
```

There must be a procedure which writes an interface block for a given feature. If the external matrix element is implemented as a Fortran module, this is required only for the specific features which are returned as procedure pointers.

⟨*Prclib interfaces: prc writer: TBP*⟩+≡

```
procedure(write_feature_code), deferred :: write_interface
```

There must also be a procedure which writes Makefile code which is specific for the current process, but not the feature.

⟨*Prclib interfaces: prc writer: TBP*⟩+≡

```
procedure(write_code_os), deferred :: write_makefile_code
```

This procedure writes code process-specific source-code file (which need not be Fortran). It may be a no-op, if the source code is generated by Make instead.

⟨*Prclib interfaces: prc writer: TBP*⟩+≡

```
procedure(write_code_file), deferred :: write_source_code
```

Writer for Fortran-module matrix elements

If the matrix element is available as a Fortran module, we have specific requirements: (i) the features are imported via `use` directives, (ii) the specific features require bind(C) wrappers.

The type is still abstract, all methods must be implemented explicitly for a specific matrix-element variant.

⟨*Prclib interfaces: public*⟩+≡

```
public :: prc_writer_f_module_t
```

⟨*Prclib interfaces: types*⟩+≡

```
type, extends (prc_writer_t), abstract :: prc_writer_f_module_t
 contains
 ⟨Prclib interfaces: prc writer f module: TBP⟩
end type prc_writer_f_module_t
```

Return the name of the Fortran module. As a default implementation, we take the process ID unchanged.

⟨*Prclib interfaces: prc writer f module: TBP*⟩≡

```
procedure, nopass :: get_module_name => prc_writer_get_module_name
```

⟨*Prclib interfaces: procedures*⟩+≡

```
function prc_writer_get_module_name (id) result (name)
  type(string_t) :: name
  type(string_t), intent(in) :: id
  name = id
end function prc_writer_get_module_name
```

Write a `use` directive that associates the driver reference with the procedure in the matrix element code. By default, we use the C name for this.

⟨*Prclib interfaces: prc writer f module: TBP*⟩+≡

```
procedure :: write_use_line => prc_writer_write_use_line
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine prc_writer_write_use_line (writer, unit, id, feature)
  class(prc_writer_f_module_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t) :: id, feature
  write (unit, "(2x,9A)")  "use ", char (writer%get_module_name (id)), &
       ", only: ", char (writer%get_c_procname (id, feature)), &
       " => ", char (writer%get_procname (feature))
end subroutine prc_writer_write_use_line
```

Write a wrapper routine for a feature. This also associates a C name the module procedure. The details depend on the writer variant.

⟨*Prclib interfaces: prc writer f module: TBP*⟩+≡
```
procedure(prc_write_wrapper), deferred :: write_wrapper
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
abstract interface
   subroutine prc_write_wrapper (writer, unit, id, feature)
     import
     class(prc_writer_f_module_t), intent(in) :: writer
     integer, intent(in) :: unit
     type(string_t), intent(in) :: id, feature
   end subroutine prc_write_wrapper
end interface
```

This is used for testing only: initialize the writer with a specific MD5 sum string.

⟨*Prclib interfaces: prc writer: TBP*⟩+≡
```
procedure :: init_test => prc_writer_init_test
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine prc_writer_init_test (writer)
  class(prc_writer_t), intent(out) :: writer
  writer%md5sum = "1234567890abcdef1234567890abcdef"
end subroutine prc_writer_init_test
```

Writer for C-library matrix elements

This applies if the matrix element is available as a C library or a Fortran library with bind(C) compatible interface. We can use the basic version.

The type is still abstract, all methods must be implemented explicitly for a specific matrix-element variant.

⟨*Prclib interfaces: public*⟩+≡
```
public :: prc_writer_c_lib_t
```

⟨*Prclib interfaces: types*⟩+≡
```
type, extends (prc_writer_t), abstract :: prc_writer_c_lib_t
 contains
 ⟨Prclib interfaces: prc writer c lib: TBP⟩
end type prc_writer_c_lib_t
```

13.2.5 Process records in the library driver

A process record holds the process (component) `ID`, the physics `model_name`, and the array of `feature`s that are implemented by the corresponding matrix element code.

The `writer` component holds procedures. The procedures write source code for the current record, either for the driver or for the Makefile.

⟨*Prclib interfaces: types*⟩+≡

```
type :: prclib_driver_record_t
   type(string_t) :: id
   type(string_t) :: model_name
   type(string_t), dimension(:), allocatable :: feature
   class(prc_writer_t), pointer :: writer => null ()
 contains
 ⟨Prclib interfaces: prclib driver record: TBP⟩
end type prclib_driver_record_t
```

Output routine. We indent the output, so it smoothly integrates into the output routine for the whole driver.

Note: the pointer `writer` is introduced as a workaround for a NAG compiler bug.

⟨*Prclib interfaces: prclib driver record: TBP*⟩≡

```
procedure :: write => prclib_driver_record_write
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_record_write (object, unit)
  class(prclib_driver_record_t), intent(in) :: object
  integer, intent(in) :: unit
  integer :: j
  class(prc_writer_t), pointer :: writer
  write (unit, "(3x,A,2x,'[',A,']')")  &
       char (object%id), char (object%model_name)
  if (allocated (object%feature)) then
     writer => object%writer
     write (unit, "(5x,A,A)", advance="no") &
          char (writer%type_name ()), ":"
     do j = 1, size (object%feature)
        write (unit, "(1x,A)", advance="no") &
             char (object%feature(j))
     end do
     write (unit, *)
  end if
end subroutine prclib_driver_record_write
```

Get the C procedure name for a feature.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡

```
procedure :: get_c_procname => prclib_driver_record_get_c_procname
```

⟨*Prclib interfaces: procedures*⟩+≡

```
function prclib_driver_record_get_c_procname (record, feature) result (name)
  type(string_t) :: name
  class(prclib_driver_record_t), intent(in) :: record
  type(string_t), intent(in) :: feature
```

```
    name = record%writer%get_c_procname (record%id, feature)
  end function prclib_driver_record_get_c_procname
```

Write a USE directive for a given feature. Applies only if the record corresponds to a Fortran module.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡

```
  procedure :: write_use_line => prclib_driver_record_write_use_line
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prclib_driver_record_write_use_line (record, unit, feature)
    class(prclib_driver_record_t), intent(in) :: record
    integer, intent(in) :: unit
    type(string_t), intent(in) :: feature
    select type (writer => record%writer)
    class is (prc_writer_f_module_t)
       call writer%write_use_line (unit, record%id, feature)
    end select
  end subroutine prclib_driver_record_write_use_line
```

The alternative: write an interface block for a given feature, unless the record corresponds to a Fortran module.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡

```
  procedure :: write_interface => prclib_driver_record_write_interface
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prclib_driver_record_write_interface (record, unit, feature)
    class(prclib_driver_record_t), intent(in) :: record
    integer, intent(in) :: unit
    type(string_t), intent(in) :: feature
    select type (writer => record%writer)
    class is (prc_writer_f_module_t)
    class default
       call writer%write_interface (unit, record%id, feature)
    end select
  end subroutine prclib_driver_record_write_interface
```

Write all special feature interfaces for the current record. Do this for all process variants.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡

```
  procedure :: write_interfaces => prclib_driver_record_write_interfaces
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prclib_driver_record_write_interfaces (record, unit)
    class(prclib_driver_record_t), intent(in) :: record
    integer, intent(in) :: unit
    integer :: i
    do i = 1, size (record%feature)
       call record%writer%write_interface (unit, record%id, record%feature(i))
    end do
  end subroutine prclib_driver_record_write_interfaces
```

Write the wrapper routines for this record, if it corresponds to a Fortran module.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡

```
  procedure :: write_wrappers => prclib_driver_record_write_wrappers
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_record_write_wrappers (record, unit)
  class(prclib_driver_record_t), intent(in) :: record
  integer, intent(in) :: unit
  integer :: i
  select type (writer => record%writer)
  class is (prc_writer_f_module_t)
     do i = 1, size (record%feature)
        call writer%write_wrapper (unit, record%id, record%feature(i))
     end do
  end select
end subroutine prclib_driver_record_write_wrappers
```

Write the Makefile code for this record.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡

```
procedure :: write_makefile_code => prclib_driver_record_write_makefile_code
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_record_write_makefile_code &
     (record, unit, os_data, testflag)
  class(prclib_driver_record_t), intent(in) :: record
  integer, intent(in) :: unit
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: testflag
  call record%writer%write_makefile_code (unit, record%id, os_data, testflag)
end subroutine prclib_driver_record_write_makefile_code
```

Write source-code files for this record. (Does nothing if the source code is handled by Makefile rules.)

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡

```
procedure :: write_source_code => prclib_driver_record_write_source_code
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_record_write_source_code (record)
  class(prclib_driver_record_t), intent(in) :: record
  call record%writer%write_source_code (record%id)
end subroutine prclib_driver_record_write_source_code
```

13.2.6 The process library driver object

A `prclib_driver_t` object provides the interface to external matrix element code. The code is provided by an external library which is either statically or dynamically linked.

The dynamic and static versions of the library are two different implementations of the abstract base type.

The `basename` identifies the library, both by file names and by Fortran variable names.

The `loaded` flag becomes true once all procedure pointers to the matrix element have been assigned.

For a dynamical external library, the communication proceeds via a `dlaccess` object.

n_processes is the number of external process code components that are referenced by this library. The code is addressed by index (i_lib in the process library entry above). This number should be equal to the number returned by get_n_prc.

For each external process, there is a separate record which holds the data that are needed for the driver parts which are specific for a given process component. The actual pointers for the loaded library will be assigned elsewhere.

The remainder is a collection of procedure pointers, which can be assigned once all external code has been compiled and linked. The procedure pointers all take a process component code index as an argument. Most return information about the process component that should match the process definition. The get_fptr procedures return a function pointer, which is the actual means to compute matrix elements or retrieve associated data.

Finally, the unload_hook and reload_hook pointers allow for the insertion of additional code when a library is loaded.

⟨*Prclib interfaces: public*⟩+≡

```
public :: prclib_driver_t
```

⟨*Prclib interfaces: types*⟩+≡

```
type, abstract :: prclib_driver_t
   type(string_t) :: basename
   character(32) :: md5sum = ""
   logical :: loaded = .false.
   type(string_t) :: libname
   type(string_t) :: modellibs_ldflags
   integer :: n_processes = 0
   type(prclib_driver_record_t), dimension(:), allocatable :: record
   procedure(prc_get_n_processes), nopass, pointer :: &
        get_n_processes => null ()
   procedure(prc_get_stringptr), nopass, pointer :: &
        get_process_id_ptr => null ()
   procedure(prc_get_stringptr), nopass, pointer :: &
        get_model_name_ptr => null ()
   procedure(prc_get_stringptr), nopass, pointer :: &
        get_md5sum_ptr => null ()
   procedure(prc_get_log), nopass, pointer :: &
        get_openmp_status => null ()
   procedure(prc_get_int), nopass, pointer :: get_n_in  => null ()
   procedure(prc_get_int), nopass, pointer :: get_n_out => null ()
   procedure(prc_get_int), nopass, pointer :: get_n_flv => null ()
   procedure(prc_get_int), nopass, pointer :: get_n_hel => null ()
   procedure(prc_get_int), nopass, pointer :: get_n_col => null ()
   procedure(prc_get_int), nopass, pointer :: get_n_cin => null ()
   procedure(prc_get_int), nopass, pointer :: get_n_cf  => null ()
   procedure(prc_set_int_tab1), nopass, pointer :: &
        set_flv_state_ptr => null ()
   procedure(prc_set_int_tab1), nopass, pointer :: &
        set_hel_state_ptr => null ()
   procedure(prc_set_col_state), nopass, pointer :: &
        set_col_state_ptr => null ()
   procedure(prc_set_color_factors), nopass, pointer :: &
        set_color_factors_ptr => null ()
   procedure(prc_get_fptr), nopass, pointer :: get_fptr => null ()
```

```
 contains
 ⟨Prclib interfaces: prclib driver: TBP⟩
end type prclib_driver_t
```

This is the dynamic version. It contains a `dlaccess` object for communicating with the OS.

⟨*Prclib interfaces: public*⟩+≡

```
public :: prclib_driver_dynamic_t
```

⟨*Prclib interfaces: types*⟩+≡

```
type, extends (prclib_driver_t) :: prclib_driver_dynamic_t
   type(dlaccess_t) :: dlaccess
 contains
 ⟨Prclib interfaces: prclib driver dynamic: TBP⟩
end type prclib_driver_dynamic_t
```

Print just the metadata. Procedure pointers cannot be printed.

⟨*Prclib interfaces: prclib driver: TBP*⟩≡

```
procedure :: write => prclib_driver_write
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_write (object, unit, libpath)
  class(prclib_driver_t), intent(in) :: object
  integer, intent(in) :: unit
  logical, intent(in), optional :: libpath
  logical :: write_lib
  integer :: i
  write_lib = .true.
  if (present (libpath))  write_lib = libpath
  write (unit, "(1x,A,A)")  &
       "External matrix-element code library: ", char (object%basename)
  select type (object)
  type is (prclib_driver_dynamic_t)
     write (unit, "(3x,A,L1)")  "static    = F"
  class default
     write (unit, "(3x,A,L1)")  "static    = T"
  end select
  write (unit, "(3x,A,L1)")  "loaded    = ", object%loaded
  write (unit, "(3x,A,A,A)") "MD5 sum   = '", object%md5sum, "'"
  if (write_lib) then
     write (unit, "(3x,A,A,A)") "Mdl flags = '", &
          char (object%modellibs_ldflags), "'"
  end if
  select type (object)
  type is (prclib_driver_dynamic_t)
     write (unit, *)
     call object%dlaccess%write (unit)
  end select
  write (unit, *)
  if (allocated (object%record)) then
     write (unit, "(1x,A)")  "Matrix-element code entries:"
     do i = 1, object%n_processes
        call object%record(i)%write (unit)
     end do
```

```
    else
       write (unit, "(1x,A)")  "Matrix-element code entries: [undefined]"
    end if
  end subroutine prclib_driver_write
```

Allocate a library as either static or dynamic. For static libraries, the procedure defers control to an external procedure which knows about the available static libraries. By default, this procedure is empty, but when we build a stand-alone executable, we replace the dummy by an actual dispatcher for the available static libraries. If the static dispatcher was not successful, we allocate a dynamic library.

The default version of `dispatch_prclib_static` resides in the `prebuilt` section of the `WHIZARD` tree, in a separate library. It does nothing, but can be replaced by a different procedure that allocates a static library driver if requested by name.

Note: `intent(out)` for the `driver` argument segfaults with gfortran 4.7.

⟨*Prclib interfaces: public*⟩+≡

```
  public :: dispatch_prclib_driver
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine dispatch_prclib_driver &
       (driver, basename, modellibs_ldflags)
    class(prclib_driver_t), intent(inout), allocatable :: driver
    type(string_t), intent(in) :: basename
    type(string_t), intent(in), optional :: modellibs_ldflags
    procedure(dispatch_prclib_driver) :: dispatch_prclib_static
    if (allocated (driver))  deallocate (driver)
    call dispatch_prclib_static (driver, basename)
    if (.not. allocated (driver)) then
       allocate (prclib_driver_dynamic_t :: driver)
    end if
    driver%basename = basename
    driver%modellibs_ldflags = modellibs_ldflags
  end subroutine dispatch_prclib_driver
```

Initialize the ID array and set `n_processes` accordingly.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
  procedure :: init => prclib_driver_init
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prclib_driver_init (driver, n_processes)
    class(prclib_driver_t), intent(inout) :: driver
    integer, intent(in) :: n_processes
    driver%n_processes = n_processes
    allocate (driver%record (n_processes))
  end subroutine prclib_driver_init
```

Set the MD5 sum. This is separate because the MD5 sum may be known only after initialization.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
  procedure :: set_md5sum => prclib_driver_set_md5sum
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_set_md5sum (driver, md5sum)
  class(prclib_driver_t), intent(inout) :: driver
  character(32), intent(in) :: md5sum
  driver%md5sum = md5sum
end subroutine prclib_driver_set_md5sum
```

Set the process record for a specific library entry. If the index is zero, we do nothing.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: set_record => prclib_driver_set_record
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_set_record (driver, i, &
     id, model_name, features, writer)
  class(prclib_driver_t), intent(inout) :: driver
  integer, intent(in) :: i
  type(string_t), intent(in) :: id
  type(string_t), intent(in) :: model_name
  type(string_t), dimension(:), intent(in) :: features
  class(prc_writer_t), intent(in), pointer :: writer
  if (i > 0) then
     associate (record => driver%record(i))
       record%id = id
       record%model_name = model_name
       allocate (record%feature (size (features)))
       record%feature = features
       record%writer => writer
     end associate
  end if
end subroutine prclib_driver_set_record
```

Write all USE directives for a given feature, scanning the array of processes. Only Fortran-module processes count. Then, write interface blocks for the remaining processes.

The `implicit none` statement must go in-between.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: write_interfaces => prclib_driver_write_interfaces
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_write_interfaces (driver, unit, feature)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: feature
  integer :: i
  do i = 1, driver%n_processes
     call driver%record(i)%write_use_line (unit, feature)
  end do
  write (unit, "(2x,9A)")  "implicit none"
  do i = 1, driver%n_processes
     call driver%record(i)%write_interface (unit, feature)
  end do
end subroutine prclib_driver_write_interfaces
```

13.2.7 Write makefile

The makefile contains constant parts, parts that depend on the library name, and parts that depend on the specific processes and their types.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: generate_makefile => prclib_driver_generate_makefile
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_generate_makefile (driver, unit, os_data, testflag)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: testflag
  integer :: i
  write (unit, "(A)")  "# WHIZARD: Makefile for process library '" &
       // char (driver%basename) // "'"
  write (unit, "(A)")  "# Automatically generated file, do not edit"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "# Integrity check (don't modify the following line!)"
  write (unit, "(A)")  "MD5SUM = '" // driver%md5sum // "'"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "# Library name"
  write (unit, "(A)")  "BASE = " // char (driver%basename)
  write (unit, "(A)")  ""
  write (unit, "(A)")  "# Compiler"
  write (unit, "(A)")  "FC = " // char (os_data%fc)
  write (unit, "(A)")  "CC = " // char (os_data%cc)
  write (unit, "(A)")  ""
  write (unit, "(A)")  "# Included libraries"
  write (unit, "(A)")  "FCINCL = " // char (os_data%whizard_includes)
  write (unit, "(A)")  ""
  write (unit, "(A)")  "# Compiler flags"
  write (unit, "(A)")  "FCFLAGS = " // char (os_data%fcflags)
  write (unit, "(A)")  "FCFLAGS_PIC = " // char (os_data%fcflags_pic)
  write (unit, "(A)")  "CFLAGS = " // char (os_data%cflags)
  write (unit, "(A)")  "CFLAGS_PIC = " // char (os_data%cflags_pic)
  write (unit, "(A)")  "LDFLAGS = " // char (os_data%whizard_ldflags) &
       // " " // char (os_data%ldflags) // " " // &
       char (driver%modellibs_ldflags)
  write (unit, "(A)")  ""
  write (unit, "(A)")  "# LaTeX setup"
  write (unit, "(A)")  "LATEX = " // char (os_data%latex)
  write (unit, "(A)")  "MPOST = " // char (os_data%mpost)
  write (unit, "(A)")  "DVIPS = " // char (os_data%dvips)
  write (unit, "(A)")  "PS2PDF = " // char (os_data%ps2pdf)
  write (unit, "(A)")  'TEX_FLAGS = "$$TEXINPUTS:' // &
       char(os_data%whizard_texpath) // '"'
  write (unit, "(A)")  'MP_FLAGS  = "$$MPINPUTS:' // &
       char(os_data%whizard_texpath) // '"'
  write (unit, "(A)")  ""
  write (unit, "(A)")  "# Libtool"
  write (unit, "(A)")  "LIBTOOL = " // char (os_data%whizard_libtool)
  write (unit, "(A)")  "FCOMPILE = $(LIBTOOL) --tag=FC --mode=compile"
  write (unit, "(A)")  "CCOMPILE = $(LIBTOOL) --tag=CC --mode=compile"
  write (unit, "(A)")  "LINK = $(LIBTOOL) --tag=FC --mode=link"
```

```
write (unit, "(A)")  ""
write (unit, "(A)")  "# Compile commands (default)"
write (unit, "(A)")  "LTFCOMPILE = $(FCOMPILE) $(FC) -c &
     &$(FCINCL) $(FCFLAGS) $(FCFLAGS_PIC)"
write (unit, "(A)")  "LTCCOMPILE = $(CCOMPILE) $(CC) -c &
     &$(CFLAGS) $(CFLAGS_PIC)"
write (unit, "(A)")  ""
write (unit, "(A)")  "# Default target"
write (unit, "(A)")  "all: link diags"
write (unit, "(A)")  ""
write (unit, "(A)")  "# Matrix-element code files"
do i = 1, size (driver%record)
   call driver%record(i)%write_makefile_code (unit, os_data, testflag)
end do
write (unit, "(A)")  ""
write (unit, "(A)")  "# Library driver"
write (unit, "(A)")  "$(BASE).lo: $(BASE).f90 $(OBJECTS)"
write (unit, "(A)")  TAB // "$(LTFCOMPILE) $<"
write (unit, "(A)")  ""
write (unit, "(A)")  "# Library"
write (unit, "(A)")  "$(BASE).la: $(BASE).lo $(OBJECTS)"
write (unit, "(A)")  TAB // "$(LINK) $(FC) -module -rpath /dev/null &
     &$(FCFLAGS) $(LDFLAGS) -o $(BASE).la $^"
write (unit, "(A)")  ""
write (unit, "(A)")  "# Main targets"
write (unit, "(A)")  "link: compile $(BASE).la"
write (unit, "(A)")  "compile: source $(OBJECTS) $(TEX_OBJECTS) $(BASE).lo"
write (unit, "(A)")  "compile_tex: $(TEX_OBJECTS)"
write (unit, "(A)")  "source: $(SOURCES) $(BASE).f90 $(TEX_SOURCES)"
write (unit, "(A)")  ".PHONY: link diags compile compile_tex source"
write (unit, "(A)")  ""
write (unit, "(A)")  "# Specific cleanup targets"
do i = 1, size (driver%record)
   write (unit, "(A)")  "clean-" // char (driver%record(i)%id) // ":"
   write (unit, "(A)")  ".PHONY: clean-" // char (driver%record(i)%id)
end do
write (unit, "(A)")  ""
write (unit, "(A)")  "# Generic cleanup targets"
write (unit, "(A)")  "clean-library:"
write (unit, "(A)")  TAB // "rm -f $(BASE).la"
write (unit, "(A)")  "clean-objects:"
write (unit, "(A)")  TAB // "rm -f $(BASE).lo $(BASE)_driver.mod &
     &$(CLEAN_OBJECTS)"
write (unit, "(A)")  "clean-source:"
write (unit, "(A)")  TAB // "rm -f $(CLEAN_SOURCES)"
write (unit, "(A)")  "clean-driver:"
write (unit, "(A)")  TAB // "rm -f $(BASE).f90"
write (unit, "(A)")  "clean-makefile:"
write (unit, "(A)")  TAB // "rm -f $(BASE).makefile"
write (unit, "(A)")  ".PHONY: clean-library clean-objects &
     &clean-source clean-driver clean-makefile"
write (unit, "(A)")  ""
write (unit, "(A)")  "clean: clean-library clean-objects clean-source"
write (unit, "(A)")  "distclean: clean clean-driver clean-makefile"
```

```
    write (unit, "(A)")  ".PHONY: clean distclean"
  end subroutine prclib_driver_generate_makefile
```

13.2.8 Write driver file

This procedure writes the process library driver source code to the specified output unit. The individual routines for writing source-code procedures are given below.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
  procedure :: generate_driver_code => prclib_driver_generate_code
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prclib_driver_generate_code (driver, unit)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: unit
    type(string_t) :: prefix
    integer :: i

    prefix = driver%basename // "_"

    write (unit, "(A)")  "! WHIZARD matrix-element code interface"
    write (unit, "(A)")  "!"
    write (unit, "(A)")  "! Automatically generated file, do not edit"
    call driver%write_module (unit, prefix)
    call driver%write_lib_md5sum_fun (unit, prefix)
    call driver%write_get_n_processes_fun (unit, prefix)
    call driver%write_get_process_id_fun (unit, prefix)
    call driver%write_get_model_name_fun (unit, prefix)
    call driver%write_get_md5sum_fun (unit, prefix)
    call driver%write_string_to_array_fun (unit, prefix)
    call driver%write_get_openmp_status_fun (unit, prefix)
    call driver%write_get_int_fun (unit, prefix, var_str ("n_in"))
    call driver%write_get_int_fun (unit, prefix, var_str ("n_out"))
    call driver%write_get_int_fun (unit, prefix, var_str ("n_flv"))
    call driver%write_get_int_fun (unit, prefix, var_str ("n_hel"))
    call driver%write_get_int_fun (unit, prefix, var_str ("n_col"))
    call driver%write_get_int_fun (unit, prefix, var_str ("n_cin"))
    call driver%write_get_int_fun (unit, prefix, var_str ("n_cf"))
    call driver%write_set_int_sub (unit, prefix, var_str ("flv_state"))
    call driver%write_set_int_sub (unit, prefix, var_str ("hel_state"))
    call driver%write_set_col_state_sub (unit, prefix)
    call driver%write_set_color_factors_sub (unit, prefix)
    call driver%write_get_fptr_sub (unit, prefix)
    do i = 1, driver%n_processes
       call driver%record(i)%write_wrappers (unit)
    end do
  end subroutine prclib_driver_generate_code
```

The driver module is used and required *only* if we intend to link the library statically. Then, it provides the (static) driver type as a concrete implementation of the abstract library driver. This type contains the internal dispatcher for assigning the library procedures to their appropriate procedure pointers. In

the dynamical case, the assignment is done via the base-type dispatcher which invokes the DL mechanism.

However, compiling this together with the rest in any case should not do any harm.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure, nopass :: write_module => prclib_driver_write_module
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_write_module (unit, prefix)
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  write (unit, "(A)")  ""
  write (unit, "(A)")  "! Module: define library driver as an extension &
       &of the abstract driver type."
  write (unit, "(A)")  "! This is used _only_ by the library dispatcher &
       &of a static executable."
  write (unit, "(A)")  "! For a dynamical library, the stand-alone proce&
       &dures are linked via libdl."
  write (unit, "(A)")  ""
  write (unit, "(A)")  "module " &
       // char (prefix) // "driver"
  write (unit, "(A)")  "  use iso_c_binding"
  write (unit, "(A)")  "  use iso_varying_string, string_t => varying_string"
  write (unit, "(A)")  "  use diagnostics"
  write (unit, "(A)")  "  use prclib_interfaces"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "  implicit none"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "  type, extends (prclib_driver_t) :: " &
       // char (prefix) // "driver_t"
  write (unit, "(A)")  "   contains"
  write (unit, "(A)")  "     procedure :: get_c_funptr => " &
       // char (prefix) // "driver_get_c_funptr"
  write (unit, "(A)")  "  end type " &
       // char (prefix) // "driver_t"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "contains"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "  function " &
       // char (prefix) // "driver_get_c_funptr (driver, feature) result &
       &(c_fptr)"
  write (unit, "(A)")  "    class(" &
       // char (prefix) // "driver_t), intent(inout) :: driver"
  write (unit, "(A)")  "    type(string_t), intent(in) :: feature"
  write (unit, "(A)")  "    type(c_funptr) :: c_fptr"
  call write_decl ("get_n_processes", "get_n_processes")
  call write_decl ("get_stringptr", "get_process_id_ptr")
  call write_decl ("get_stringptr", "get_model_name_ptr")
  call write_decl ("get_stringptr", "get_md5sum_ptr")
  call write_decl ("get_log", "get_openmp_status")
  call write_decl ("get_int", "get_n_in")
  call write_decl ("get_int", "get_n_out")
  call write_decl ("get_int", "get_n_flv")
  call write_decl ("get_int", "get_n_hel")
```

```
  call write_decl ("get_int", "get_n_col")
  call write_decl ("get_int", "get_n_cin")
  call write_decl ("get_int", "get_n_cf")
  call write_decl ("set_int_tab1", "set_flv_state_ptr")
  call write_decl ("set_int_tab1", "set_hel_state_ptr")
  call write_decl ("set_col_state", "set_col_state_ptr")
  call write_decl ("set_color_factors", "set_color_factors_ptr")
  call write_decl ("get_fptr", "get_fptr")
  write (unit, "(A)")  "    select case (char (feature))"
  call write_case ("get_n_processes")
  call write_case ("get_process_id_ptr")
  call write_case ("get_model_name_ptr")
  call write_case ("get_md5sum_ptr")
  call write_case ("get_openmp_status")
  call write_case ("get_n_in")
  call write_case ("get_n_out")
  call write_case ("get_n_flv")
  call write_case ("get_n_hel")
  call write_case ("get_n_col")
  call write_case ("get_n_cin")
  call write_case ("get_n_cf")
  call write_case ("set_flv_state_ptr")
  call write_case ("set_hel_state_ptr")
  call write_case ("set_col_state_ptr")
  call write_case ("set_color_factors_ptr")
  call write_case ("get_fptr")
  write (unit, "(A)")  "    case default"
  write (unit, "(A)")  "       call msg_bug ('prclib2 driver setup: unknown &
       &function name')"
  write (unit, "(A)")  "    end select"
  write (unit, "(A)")  "  end function " &
       // char (prefix) // "driver_get_c_funptr"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "end module " &
       // char (prefix) // "driver"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "! Stand-alone external procedures: used for both &
       &static and dynamic linkage"
contains
  subroutine write_decl (template, feature)
    character(*), intent(in) :: template, feature
    write (unit, "(A)")  "    procedure(prc_" // template // ") &"
    write (unit, "(A)")  "         :: " &
         // char (prefix) // feature
  end subroutine write_decl
  subroutine write_case (feature)
    character(*), intent(in) :: feature
    write (unit, "(A)")  "    case ('" // feature // "')"
    write (unit, "(A)")  "       c_fptr = c_funloc (" &
         // char (prefix) // feature // ")"
  end subroutine write_case
end subroutine prclib_driver_write_module
```

This function provides the overall library MD5sum. The function is for internal

use (therefore not bind(C)), the external interface is via the `get_md5sum_ptr` procedure with index 0.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
procedure :: write_lib_md5sum_fun => prclib_driver_write_lib_md5sum_fun
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine prclib_driver_write_lib_md5sum_fun (driver, unit, prefix)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  write (unit, "(A)")  ""
  write (unit, "(A)")  "! The MD5 sum of the library"
  write (unit, "(A)")  "function " // char (prefix) &
       // "md5sum () result (md5sum)"
  write (unit, "(A)")  "  implicit none"
  write (unit, "(A)")  "  character(32) :: md5sum"
  write (unit, "(A)")  "  md5sum = '" // driver%md5sum // "'"
  write (unit, "(A)")  "end function " // char (prefix) // "md5sum"
end subroutine prclib_driver_write_lib_md5sum_fun
```

13.2.9 Interface bodies for informational functions

These interfaces implement the communication between WHIZARD (the main program) and the process-library driver. The procedures are all BIND(C), so they can safely be exposed by the library and handled by the `dlopen` mechanism, which apparently understands only C calling conventions.

In the sections below, for each procedure, we provide both the interface itself and a procedure that writes the correponding procedure as source code to the process library driver.

Process count

Return the number of processes contained in the library.

⟨*Prclib interfaces: public*⟩+≡
```
public :: prc_get_n_processes
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
abstract interface
   function prc_get_n_processes () result (n) bind(C)
     import
     integer(c_int) :: n
   end function prc_get_n_processes
end interface
```

Here is the code.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
procedure :: write_get_n_processes_fun
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine write_get_n_processes_fun (driver, unit, prefix)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
```

```
    write (unit, "(A)")  ""
    write (unit, "(A)")  "! Return the number of processes in this library"
    write (unit, "(A)")  "function " // char (prefix) &
         // "get_n_processes () result (n) bind(C)"
    write (unit, "(A)")  "  use iso_c_binding"
    write (unit, "(A)")  "  implicit none"
    write (unit, "(A)")  "  integer(c_int) :: n"
    write (unit, "(A,IO)")  "  n = ", driver%n_processes
    write (unit, "(A)")  "end function " // char (prefix) &
         // "get_n_processes"
end subroutine write_get_n_processes_fun
```

Informational string functions

These functions return constant information about the matrix-element code.

The following procedures have to return strings. With the BIND(C) constraint, we choose to return the C pointer to a string, and its length, so the procedures implement this interface. They are actually subroutines.

⟨*Prclib interfaces: public*⟩+≡

```
public :: prc_get_stringptr
```

⟨*Prclib interfaces: interfaces*⟩+≡

```
abstract interface
   subroutine prc_get_stringptr (i, cptr, len) bind(C)
     import
     integer(c_int), intent(in) :: i
     type(c_ptr), intent(out) :: cptr
     integer(c_int), intent(out) :: len
   end subroutine prc_get_stringptr
end interface
```

To hide this complication, we introduce a subroutine that converts the returned C pointer to a `string_t` object. As a side effect, we deallocate the original after conversion – otherwise, we might have a memory leak.

For the conversion, we first pointer-convert the C pointer to a Fortran character array pointer, length 1 and size `len`. Using argument association and an internal subroutine, we convert this to a character array with length `len` and size 1. Using ordinary assignment, we finally convert this to `string_t`.

The function takes the pointer-returning function as an argument. The index `i` identifies the process in the library.

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine get_string_via_cptr (string, i, get_stringptr)
  type(string_t), intent(out) :: string
  integer, intent(in) :: i
  procedure(prc_get_stringptr) :: get_stringptr
  type(c_ptr) :: cptr
  integer(c_int) :: pid, len
  character(kind=c_char), dimension(:), pointer :: c_array
  pid = i
  call get_stringptr (pid, cptr, len)
  if (c_associated (cptr)) then
     call c_f_pointer (cptr, c_array, shape = [len])
```

```
      call set_string (c_array)
      call get_stringptr (0_c_int, cptr, len)
    else
      string = ""
    end if
  contains
    subroutine set_string (buffer)
      character(len, kind=c_char), dimension(1), intent(in) :: buffer
      string = buffer(1)
    end subroutine set_string
  end subroutine get_string_via_cptr
```

Since the module procedures return Fortran strings, we have to convert them. This is the necessary auxiliary routine. The routine is not BIND(C), it is not accessed from outside.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
  procedure, nopass :: write_string_to_array_fun
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine write_string_to_array_fun (unit, prefix)
    integer, intent(in) :: unit
    type(string_t), intent(in) :: prefix
    write (unit, "(A)")  ""
    write (unit, "(A)")  "! Auxiliary: convert character string &
         &to array pointer"
    write (unit, "(A)")  "subroutine " // char (prefix) &
         // "string_to_array (string, a)"
    write (unit, "(A)")  "  use iso_c_binding"
    write (unit, "(A)")  "  implicit none"
    write (unit, "(A)")  "  character(*), intent(in) :: string"
    write (unit, "(A)")  "  character(kind=c_char), dimension(:), &
         &allocatable, intent(out) :: a"
    write (unit, "(A)")  "  integer :: i"
    write (unit, "(A)")  "  allocate (a (len (string)))"
    write (unit, "(A)")  "  do i = 1, size (a)"
    write (unit, "(A)")  "     a(i) = string(i:i)"
    write (unit, "(A)")  "  end do"
    write (unit, "(A)")  "end subroutine " // char (prefix) &
         // "string_to_array"
  end subroutine write_string_to_array_fun
```

The above routine is called by other functions. It is not in a module, so they need its interface explicitly.

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine write_string_to_array_interface (unit, prefix)
    integer, intent(in) :: unit
    type(string_t), intent(in) :: prefix
    write (unit, "(2x,A)")  "interface"
    write (unit, "(2x,A)")  "   subroutine " // char (prefix) &
         // "string_to_array (string, a)"
    write (unit, "(2x,A)")  "     use iso_c_binding"
    write (unit, "(2x,A)")  "     implicit none"
    write (unit, "(2x,A)")  "     character(*), intent(in) :: string"
```

```
    write (unit, "(2x,A)")  "     character(kind=c_char), dimension(:), &
         &allocatable, intent(out) :: a"
    write (unit, "(2x,A)")  "   end subroutine " // char (prefix) &
         // "string_to_array"
    write (unit, "(2x,A)")  "end interface"
  end subroutine write_string_to_array_interface
```

Here are the info functions which return strings, implementing the interface `prc_get_stringptr`.

Return the process ID for each process.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
  procedure :: write_get_process_id_fun
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine write_get_process_id_fun (driver, unit, prefix)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: unit
    type(string_t), intent(in) :: prefix
    integer :: i
    write (unit, "(A)")  ""
    write (unit, "(A)")  "! Return the process ID of process #i &
         &(as a C pointer to a character array)"
    write (unit, "(A)")  "subroutine " // char (prefix) &
         // "get_process_id_ptr (i, cptr, len) bind(C)"
    write (unit, "(A)")  "  use iso_c_binding"
    write (unit, "(A)")  "  implicit none"
    write (unit, "(A)")  "  integer(c_int), intent(in) :: i"
    write (unit, "(A)")  "  type(c_ptr), intent(out) :: cptr"
    write (unit, "(A)")  "  integer(c_int), intent(out) :: len"
    write (unit, "(A)")  "  character(kind=c_char), dimension(:), &
         &allocatable, target, save :: a"
    call write_string_to_array_interface (unit, prefix)
    write (unit, "(A)")  "  select case (i)"
    write (unit, "(A)")  "  case (0);  if (allocated (a))  deallocate (a)"
    do i = 1, driver%n_processes
       write (unit, "(A,I0,9A)")  "  case (", i, ");  ", &
            "call ", char (prefix), "string_to_array ('", &
            char (driver%record(i)%id), "', a)"
    end do
    write (unit, "(A)")  "  end select"
    write (unit, "(A)")  "  if (allocated (a)) then"
    write (unit, "(A)")  "     cptr = c_loc (a)"
    write (unit, "(A)")  "     len = size (a)"
    write (unit, "(A)")  "  else"
    write (unit, "(A)")  "     cptr = c_null_ptr"
    write (unit, "(A)")  "     len = 0"
    write (unit, "(A)")  "  end if"
    write (unit, "(A)")  "end subroutine " // char (prefix) &
         // "get_process_id_ptr"
  end subroutine write_get_process_id_fun
```

Return the model name, given explicitly.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
  procedure :: write_get_model_name_fun
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine write_get_model_name_fun (driver, unit, prefix)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  integer :: i
  write (unit, "(A)")  ""
  write (unit, "(A)")  "! Return the model name for process #i &
       &(as a C pointer to a character array)"
  write (unit, "(A)")  "subroutine " // char (prefix) &
       // "get_model_name_ptr (i, cptr, len) bind(C)"
  write (unit, "(A)")  "  use iso_c_binding"
  write (unit, "(A)")  "  implicit none"
  write (unit, "(A)")  "  integer(c_int), intent(in) :: i"
  write (unit, "(A)")  "  type(c_ptr), intent(out) :: cptr"
  write (unit, "(A)")  "  integer(c_int), intent(out) :: len"
  write (unit, "(A)")  "  character(kind=c_char), dimension(:), &
       &allocatable, target, save :: a"
  call write_string_to_array_interface (unit, prefix)
  write (unit, "(A)")  "  select case (i)"
  write (unit, "(A)")  "  case (0);  if (allocated (a))  deallocate (a)"
  do i = 1, driver%n_processes
     write (unit, "(A,I0,9A)")  "  case (", i, ");  ", &
          "call ", char (prefix), "string_to_array ('" , &
          char (driver%record(i)%model_name), &
          "', a)"
  end do
  write (unit, "(A)")  "  end select"
  write (unit, "(A)")  "  if (allocated (a)) then"
  write (unit, "(A)")  "     cptr = c_loc (a)"
  write (unit, "(A)")  "     len = size (a)"
  write (unit, "(A)")  "  else"
  write (unit, "(A)")  "     cptr = c_null_ptr"
  write (unit, "(A)")  "     len = 0"
  write (unit, "(A)")  "  end if"
  write (unit, "(A)")  "end subroutine " // char (prefix) &
       // "get_model_name_ptr"
end subroutine write_get_model_name_fun
```

Call the MD5 sum function for the process. The function calls the corresponding function of the matrix-element code, and it returns the C address of a character array with length 32.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: write_get_md5sum_fun
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine write_get_md5sum_fun (driver, unit, prefix)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  integer :: i
  write (unit, "(A)")  ""
  write (unit, "(A)")  "! Return the MD5 sum for the process configuration &
       &(as a C pointer to a character array)"
```

```
  write (unit, "(A)")  "subroutine " // char (prefix) &
       // "get_md5sum_ptr (i, cptr, len) bind(C)"
  write (unit, "(A)")  "  use iso_c_binding"
  call driver%write_interfaces (unit, var_str ("md5sum"))
  write (unit, "(A)")  "  interface"
  write (unit, "(A)")  "     function " // char (prefix) &
       // "md5sum () result (md5sum)"
  write (unit, "(A)")  "       character(32) :: md5sum"
  write (unit, "(A)")  "     end function " // char (prefix) // "md5sum"
  write (unit, "(A)")  "  end interface"
  write (unit, "(A)")  "  integer(c_int), intent(in) :: i"
  write (unit, "(A)")  "  type(c_ptr), intent(out) :: cptr"
  write (unit, "(A)")  "  integer(c_int), intent(out) :: len"
  write (unit, "(A)")  "  character(kind=c_char), dimension(32), &
       &target, save :: md5sum"
  write (unit, "(A)")  "  select case (i)"
  write (unit, "(A)")  "  case (0)"
  write (unit, "(A)")  "     call copy (" // char (prefix) // "md5sum ())"
  write (unit, "(A)")  "     cptr = c_loc (md5sum)"
  do i = 1, driver%n_processes
     write (unit, "(A,I0,A)")  "  case (", i, ")"
     call driver%record(i)%write_md5sum_call (unit)
  end do
  write (unit, "(A)")  "  case default"
  write (unit, "(A)")  "     cptr = c_null_ptr"
  write (unit, "(A)")  "  end select"
  write (unit, "(A)")  "  len = 32"
  write (unit, "(A)")  "contains"
  write (unit, "(A)")  "  subroutine copy (md5sum_tmp)"
  write (unit, "(A)")  "    character, dimension(32), intent(in) :: &
       &md5sum_tmp"
  write (unit, "(A)")  "    md5sum = md5sum_tmp"
  write (unit, "(A)")  "  end subroutine copy"
  write (unit, "(A)")  "end subroutine " // char (prefix) &
         // "get_md5sum_ptr"
end subroutine write_get_md5sum_fun
```

The actual call depends on the type of matrix element.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡

```
procedure :: write_md5sum_call => prclib_driver_record_write_md5sum_call
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_record_write_md5sum_call (record, unit)
  class(prclib_driver_record_t), intent(in) :: record
  integer, intent(in) :: unit
  call record%writer%write_md5sum_call (unit, record%id)
end subroutine prclib_driver_record_write_md5sum_call
```

The interface goes into the writer base type:

⟨*Prclib interfaces: prc writer: TBP*⟩+≡

```
procedure(write_code), deferred :: write_md5sum_call
```

In the Fortran module case, we take a detour. The string returned by the Fortran function is copied into a fixed-size array. The copy routine is an internal

subroutine of get_md5sum_ptr. We return the C address of the target array.

⟨*Prclib interfaces: prc writer f module: TBP*⟩+≡
```
procedure :: write_md5sum_call => prc_writer_f_module_write_md5sum_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine prc_writer_f_module_write_md5sum_call (writer, unit, id)
  class(prc_writer_f_module_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  write (unit, "(5x,9A)")  "call copy (", &
       char (writer%get_c_procname (id, var_str ("md5sum"))), " ())"
  write (unit, "(5x,9A)")  "cptr = c_loc (md5sum)"
end subroutine prc_writer_f_module_write_md5sum_call
```

In the C library case, the library function returns a C pointer, which we can just copy.

⟨*Prclib interfaces: prc writer c lib: TBP*⟩≡
```
procedure :: write_md5sum_call => prc_writer_c_lib_write_md5sum_call
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine prc_writer_c_lib_write_md5sum_call (writer, unit, id)
  class(prc_writer_c_lib_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  write (unit, "(5x,9A)") &
       "cptr =  ", &
       char (writer%get_c_procname (id, var_str ("get_md5sum"))), " ()"
end subroutine prc_writer_c_lib_write_md5sum_call
```

Actual references to the info functions

The string-valued info functions return C character arrays. For the API of the library driver, we provide convenience functions which (re)convert those arrays into string_t objects.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
procedure :: get_process_id => prclib_driver_get_process_id
procedure :: get_model_name => prclib_driver_get_model_name
procedure :: get_md5sum => prclib_driver_get_md5sum
```

⟨*Prclib interfaces: procedures*⟩+≡
```
function prclib_driver_get_process_id (driver, i) result (string)
  type(string_t) :: string
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: i
  call get_string_via_cptr (string, i, driver%get_process_id_ptr)
end function prclib_driver_get_process_id

function prclib_driver_get_model_name (driver, i) result (string)
  type(string_t) :: string
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: i
  call get_string_via_cptr (string, i, driver%get_model_name_ptr)
end function prclib_driver_get_model_name
```

```
function prclib_driver_get_md5sum (driver, i) result (string)
  type(string_t) :: string
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: i
  call get_string_via_cptr (string, i, driver%get_md5sum_ptr)
end function prclib_driver_get_md5sum
```

Informational logical functions

When returning a logical value, we use the C boolean type, which may differ from Fortran.

⟨*Prclib interfaces: public*⟩+≡

```
public :: prc_get_log
```

⟨*Prclib interfaces: interfaces*⟩+≡

```
abstract interface
   function prc_get_log (pid) result (l) bind(C)
     import
     integer(c_int), intent(in) :: pid
     logical(c_bool) :: l
   end function prc_get_log
end interface
```

Return a logical flag which tells whether OpenMP is supported for a specific process code.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: write_get_openmp_status_fun
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine write_get_openmp_status_fun (driver, unit, prefix)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  integer :: i
  write (unit, "(A)")  ""
  write (unit, "(A)")  "! Return the OpenMP support status"
  write (unit, "(A)")  "function " // char (prefix) &
       // "get_openmp_status (i) result (openmp_status) bind(C)"
  write (unit, "(A)")  "  use iso_c_binding"
  call driver%write_interfaces (unit, var_str ("openmp_supported"))
  write (unit, "(A)")  "  integer(c_int), intent(in) :: i"
  write (unit, "(A)")  "  logical(c_bool) :: openmp_status"
  write (unit, "(A)")  "  select case (i)"
  do i = 1, driver%n_processes
     write (unit, "(A,I0,9A)")  "  case (", i, ");  ", &
          "openmp_status = ", &
          char (driver%record(i)%get_c_procname &
          (var_str ("openmp_supported"))), " ()"
  end do
  write (unit, "(A)")  "  end select"
  write (unit, "(A)")  "end function " // char (prefix) &
       // "get_openmp_status"
end subroutine write_get_openmp_status_fun
```

Informational integer functions

Various process metadata are integer values. We can use a single interface for all of them.

⟨*Prclib interfaces: public*⟩+≡
```
public :: prc_get_int
```

⟨*Prclib interfaces: interfaces*⟩+≡
```
abstract interface
   function prc_get_int (pid) result (n) bind(C)
     import
     integer(c_int), intent(in) :: pid
     integer(c_int) :: n
   end function prc_get_int
end interface
```

This function returns any data of type integer, for each process.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡
```
procedure :: write_get_int_fun
```

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine write_get_int_fun (driver, unit, prefix, feature)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  type(string_t), intent(in) :: feature
  integer :: i
  write (unit, "(A)")  ""
  write (unit, "(9A)")  "! Return the value of ", char (feature)
  write (unit, "(9A)")  "function ", char (prefix), &
       "get_", char (feature), " (pid)", &
       " result (", char (feature), ") bind(C)"
  write (unit, "(9A)")  "  use iso_c_binding"
  call driver%write_interfaces (unit, feature)
  write (unit, "(9A)")  "  integer(c_int), intent(in) :: pid"
  write (unit, "(9A)")  "  integer(c_int) :: ", char (feature)
  write (unit, "(9A)")  "  select case (pid)"
  do i = 1, driver%n_processes
     write (unit, "(2x,A,I0,9A)")  "case (", i, ");  ", &
          char (feature), " = ", &
          char (driver%record(i)%get_c_procname (feature)), &
          " ()"
  end do
  write (unit, "(9A)")  "  end select"
  write (unit, "(9A)")  "end function ", char (prefix), &
       "get_", char (feature)
end subroutine write_get_int_fun
```

Write a `case` line that assigns the value of the external function to the current return value.

⟨*Prclib interfaces: procedures*⟩+≡
```
subroutine write_case_int_fun (record, unit, i, feature)
```

```
    class(prclib_driver_record_t), intent(in) :: record
    integer, intent(in) :: unit
    integer, intent(in) :: i
    type(string_t), intent(in) :: feature
    write (unit, "(5x,A,I0,9A)")  "case (", i, ");  ", &
         char (feature), " = ", char (record%get_c_procname (feature))
  end subroutine write_case_int_fun
```

Flavor and helicity tables

Transferring tables is more complicated. First, a two-dimensional array.

⟨*Prclib interfaces: public*⟩+≡

```
  public :: prc_set_int_tab1
```

⟨*Prclib interfaces: interfaces*⟩+≡

```
  abstract interface
     subroutine prc_set_int_tab1 (pid, tab, shape) bind(C)
       import
       integer(c_int), intent(in) :: pid
       integer(c_int), dimension(*), intent(out) :: tab
       integer(c_int), dimension(2), intent(in) :: shape
     end subroutine prc_set_int_tab1
  end interface
```

This subroutine returns a table of integers.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
  procedure :: write_set_int_sub
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine write_set_int_sub (driver, unit, prefix, feature)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: unit
    type(string_t), intent(in) :: prefix
    type(string_t), intent(in) :: feature
    integer :: i
    write (unit, "(A)")  ""
    write (unit, "(9A)")  "! Set table: ", char (feature)
    write (unit, "(9A)")  "subroutine ", char (prefix), &
         "set_", char (feature), "_ptr (pid, ", char (feature), &
         ", shape) bind(C)"
    write (unit, "(9A)")  "  use iso_c_binding"
    call driver%write_interfaces (unit, feature)
    write (unit, "(9A)")  "  integer(c_int), intent(in) :: pid"
    write (unit, "(9A)")  "  integer(c_int), dimension(*), intent(out) :: ", &
         char (feature)
    write (unit, "(9A)")  "  integer(c_int), dimension(2), intent(in) :: shape"
    write (unit, "(9A)")  "  integer, dimension(:,:), allocatable :: ", &
         char (feature), "_tmp"
    write (unit, "(9A)")  "  integer :: i, j"
    write (unit, "(9A)")  "  select case (pid)"
    do i = 1, driver%n_processes
       write (unit, "(2x,A,I0,A)")  "case (", i, ")"
       call driver%record(i)%write_int_sub_call (unit, feature)
    end do
```

```
    write (unit, "(9A)")  "  end select"
    write (unit, "(9A)")  "end subroutine ", char (prefix), &
         "set_", char (feature), "_ptr"
  end subroutine write_set_int_sub
```

The actual call depends on the type of matrix element.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡

```
  procedure :: write_int_sub_call => prclib_driver_record_write_int_sub_call
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prclib_driver_record_write_int_sub_call (record, unit, feature)
    class(prclib_driver_record_t), intent(in) :: record
    integer, intent(in) :: unit
    type(string_t), intent(in) :: feature
    call record%writer%write_int_sub_call (unit, record%id, feature)
  end subroutine prclib_driver_record_write_int_sub_call
```

The interface goes into the writer base type:

⟨*Prclib interfaces: prc writer: TBP*⟩+≡

```
  procedure(write_feature_code), deferred :: write_int_sub_call
```

In the Fortran module case, we need an extra copy in the (academical) situation where default integer and `c_int` differ. Otherwise, we just associate a Fortran array with the C pointer and let the matrix-element subroutine fill the array.

⟨*Prclib interfaces: prc writer f module: TBP*⟩+≡

```
  procedure :: write_int_sub_call => prc_writer_f_module_write_int_sub_call
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prc_writer_f_module_write_int_sub_call (writer, unit, id, feature)
    class(prc_writer_f_module_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id, feature
    write (unit, "(5x,9A)")  "allocate (",  char (feature), "_tmp ", &
         "(shape(1), shape(2)))"
    write (unit, "(5x,9A)")  "call ", &
         char (writer%get_c_procname (id, feature)), &
         " (", char (feature), "_tmp)"
    write (unit, "(5x,9A)")  "forall (i=1:shape(1), j=1:shape(2)) "
    write (unit, "(8x,9A)")  char (feature), "(i + shape(1)*(j-1)) = ", &
         char (feature), "_tmp", "(i,j)"
    write (unit, "(5x,9A)")  "end forall"
  end subroutine prc_writer_f_module_write_int_sub_call
```

In the C library case, we just transfer the C pointer to the library function.

⟨*Prclib interfaces: prc writer c lib: TBP*⟩+≡

```
  procedure :: write_int_sub_call => prc_writer_c_lib_write_int_sub_call
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prc_writer_c_lib_write_int_sub_call (writer, unit, id, feature)
    class(prc_writer_c_lib_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id, feature
    write (unit, "(5x,9A)")  "call ", &
         char (writer%get_c_procname (id, feature)), " (", char (feature), ")"
```

```
end subroutine prc_writer_c_lib_write_int_sub_call
```

Color state table

The color-state specification needs a table of integers (one array per color flow) and a corresponding array of color-ghost flags.

⟨*Prclib interfaces: public*⟩+≡

```
public :: prc_set_col_state
```

⟨*Prclib interfaces: interfaces*⟩+≡

```
abstract interface
   subroutine prc_set_col_state (pid, col_state, ghost_flag, shape) bind(C)
     import
     integer(c_int), intent(in) :: pid
     integer(c_int), dimension(*), intent(out) :: col_state
     logical(c_bool), dimension(*), intent(out) :: ghost_flag
     integer(c_int), dimension(3), intent(in) :: shape
   end subroutine prc_set_col_state
end interface
```

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: write_set_col_state_sub
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine write_set_col_state_sub (driver, unit, prefix)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  integer :: i
  type(string_t) :: feature
  feature = "col_state"
  write (unit, "(A)")  ""
  write (unit, "(9A)")  "! Set tables: col_state, ghost_flag"
  write (unit, "(9A)")  "subroutine ", char (prefix), &
       "set_col_state_ptr (pid, col_state, ghost_flag, shape) bind(C)"
  write (unit, "(9A)")  "  use iso_c_binding"
  call driver%write_interfaces (unit, feature)
  write (unit, "(9A)")  "  integer(c_int), intent(in) :: pid"
  write (unit, "(9A)") &
       "  integer(c_int), dimension(*), intent(out) :: col_state"
  write (unit, "(9A)") &
       "  logical(c_bool), dimension(*), intent(out) :: ghost_flag"
  write (unit, "(9A)") &
       "  integer(c_int), dimension(3), intent(in) :: shape"
  write (unit, "(9A)") &
       "  integer, dimension(:,:,:), allocatable :: col_state_tmp"
  write (unit, "(9A)") &
       "  logical, dimension(:,:), allocatable :: ghost_flag_tmp"
  write (unit, "(9A)") "  integer :: i, j, k"
  write (unit, "(A)")  "  select case (pid)"
  do i = 1, driver%n_processes
     write (unit, "(A,I0,A)")  "  case (", i, ")"
     call driver%record(i)%write_col_state_call (unit)
  end do
```

```
    write (unit, "(A)")  "  end select"
    write (unit, "(9A)")  "end subroutine ", char (prefix), &
         "set_col_state_ptr"
  end subroutine write_set_col_state_sub
```

The actual call depends on the type of matrix element.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡

```
  procedure :: write_col_state_call => prclib_driver_record_write_col_state_call
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prclib_driver_record_write_col_state_call (record, unit)
    class(prclib_driver_record_t), intent(in) :: record
    integer, intent(in) :: unit
    call record%writer%write_col_state_call (unit, record%id)
  end subroutine prclib_driver_record_write_col_state_call
```

The interface goes into the writer base type:

⟨*Prclib interfaces: prc writer: TBP*⟩+≡

```
  procedure(write_code), deferred :: write_col_state_call
```

In the Fortran module case, we need an extra copy in the (academical) situation where default integer and `c_int` differ. Otherwise, we just associate a Fortran array with the C pointer and let the matrix-element subroutine fill the array.

⟨*Prclib interfaces: prc writer f module: TBP*⟩+≡

```
  procedure :: write_col_state_call => prc_writer_f_module_write_col_state_call
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prc_writer_f_module_write_col_state_call (writer, unit, id)
    class(prc_writer_f_module_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id
    write (unit, "(9A)")  "  allocate (col_state_tmp ", &
            "(shape(1), shape(2), shape(3)))"
    write (unit, "(5x,9A)")  "allocate (ghost_flag_tmp ", &
               "(shape(2), shape(3)))"
    write (unit, "(5x,9A)")  "call ", &
         char (writer%get_c_procname (id, var_str ("col_state"))), &
         " (col_state_tmp, ghost_flag_tmp)"
    write (unit, "(5x,9A)")  "forall (i = 1:shape(2), j = 1:shape(3))"
    write (unit, "(8x,9A)")  "forall (k = 1:shape(1))"
    write (unit, "(11x,9A)")  &
         "col_state(k + shape(1) * (i + shape(2)*(j-1) - 1)) ", &
         "= col_state_tmp(k,i,j)"
    write (unit, "(8x,9A)")  "end forall"
    write (unit, "(8x,9A)")  &
         "ghost_flag(i + shape(2)*(j-1)) = ghost_flag_tmp(i,j)"
    write (unit, "(5x,9A)")  "end forall"
  end subroutine prc_writer_f_module_write_col_state_call
```

In the C library case, we just transfer the C pointer to the library function.

⟨*Prclib interfaces: prc writer c lib: TBP*⟩+≡

```
  procedure :: write_col_state_call => prc_writer_c_lib_write_col_state_call
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prc_writer_c_lib_write_col_state_call (writer, unit, id)
  class(prc_writer_c_lib_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  write (unit, "(5x,9A)")  "call ", &
       char (writer%get_c_procname (id, var_str ("col_state"))), &
       " (col_state, ghost_flag)"
end subroutine prc_writer_c_lib_write_col_state_call
```

Color factors

For the color-factor information, we return two integer arrays and a complex array.

⟨*Prclib interfaces: public*⟩+≡

```
public :: prc_set_color_factors
```

⟨*Prclib interfaces: interfaces*⟩+≡

```
abstract interface
   subroutine prc_set_color_factors &
        (pid, cf_index1, cf_index2, color_factors, shape) bind(C)
     import
     integer(c_int), intent(in) :: pid
     integer(c_int), dimension(*), intent(out) :: cf_index1, cf_index2
     complex(c_default_complex), dimension(*), intent(out) :: color_factors
     integer(c_int), dimension(1), intent(in) :: shape
   end subroutine prc_set_color_factors
end interface
```

This subroutine returns the color-flavor factor table.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: write_set_color_factors_sub
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine write_set_color_factors_sub (driver, unit, prefix)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  integer :: i
  type(string_t) :: feature
  feature = "color_factors"
  write (unit, "(A)")  ""
  write (unit, "(A)")  "! Set tables: color factors"
  write (unit, "(9A)")  "subroutine ", char (prefix), &
       "set_color_factors_ptr (pid, cf_index1, cf_index2, color_factors, ", &
       "shape) bind(C)"
  write (unit, "(A)")  "  use iso_c_binding"
  write (unit, "(A)")  "  use kinds"
  write (unit, "(A)")  "  use omega_color"
  call driver%write_interfaces (unit, feature)
  write (unit, "(A)")  "  integer(c_int), intent(in) :: pid"
  write (unit, "(A)")  "  integer(c_int), dimension(1), intent(in) :: shape"
  write (unit, "(A)")  "  integer(c_int), dimension(*), intent(out) :: &
```

```
        &cf_index1, cf_index2"
    write (unit, "(A)")  "  complex(c_default_complex), dimension(*), &
        &intent(out) :: color_factors"
    write (unit, "(A)")  "  type(omega_color_factor), dimension(:), &
        &allocatable :: cf"
    write (unit, "(A)")  "  select case (pid)"
    do i = 1, driver%n_processes
       write (unit, "(2x,A,I0,A)")  "case (", i, ")"
       call driver%record(i)%write_color_factors_call (unit)
    end do
    write (unit, "(A)")  "  end select"
    write (unit, "(A)")  "end subroutine " // char (prefix) &
        // "set_color_factors_ptr"
  end subroutine write_set_color_factors_sub
```

The actual call depends on the type of matrix element.

⟨*Prclib interfaces: prclib driver record: TBP*⟩+≡

```
  procedure :: write_color_factors_call => prclib_driver_record_write_color_factors_call
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prclib_driver_record_write_color_factors_call (record, unit)
    class(prclib_driver_record_t), intent(in) :: record
    integer, intent(in) :: unit
    call record%writer%write_color_factors_call (unit, record%id)
  end subroutine prclib_driver_record_write_color_factors_call
```

The interface goes into the writer base type:

⟨*Prclib interfaces: prc writer: TBP*⟩+≡

```
  procedure(write_code), deferred :: write_color_factors_call
```

In the Fortran module case, the matrix-element procedure fills an array of `omega_color_factor` elements. We distribute this array among two integer arrays and one complex-valued array, for which we have the C pointers.

⟨*Prclib interfaces: prc writer f module: TBP*⟩+≡

```
  procedure :: write_color_factors_call => prc_writer_f_module_write_color_factors_call
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prc_writer_f_module_write_color_factors_call (writer, unit, id)
    class(prc_writer_f_module_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id
    write (unit, "(5x,A)")  "allocate (cf (shape(1)))"
    write (unit, "(5x,9A)")  "call ", &
        char (writer%get_c_procname (id, var_str ("color_factors"))), " (cf)"
    write (unit, "(5x,9A)")  "cf_index1(1:shape(1)) = cf%i1"
    write (unit, "(5x,9A)")  "cf_index2(1:shape(1)) = cf%i2"
    write (unit, "(5x,9A)")  "color_factors(1:shape(1)) = cf%factor"
  end subroutine prc_writer_f_module_write_color_factors_call
```

In the C library case, we just transfer the C pointers to the library function. There are three arrays.

⟨*Prclib interfaces: prc writer c lib: TBP*⟩+≡

```
  procedure :: write_color_factors_call => &
       prc_writer_c_lib_write_color_factors_call
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prc_writer_c_lib_write_color_factors_call (writer, unit, id)
  class(prc_writer_c_lib_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  write (unit, "(5x,9A)")  "call ", &
       char (writer%get_c_procname (id, var_str ("color_factors"))), &
       " (cf_index1, cf_index2, color_factors)"
end subroutine prc_writer_c_lib_write_color_factors_call
```

13.2.10 Interfaces for C-library matrix element

If the matrix element code is not provided as a Fortran module but as a C or bind(C) Fortran library, we need explicit interfaces for the library functions. They are not identical to the Fortran module versions. They transfer pointers directly.

The implementation is part of the `prc_writer_c_lib` type, which serves as base type for all C-library writers. It writes specific interfaces depending on the feature.

We bind this as the method `write_standard_interface` instead of `write_interface`, because we have to override the latter. Otherwise we could not call the method because the writer type is abstract.

⟨*Prclib interfaces: prc writer c lib: TBP*⟩+≡

```
procedure :: write_standard_interface => prc_writer_c_lib_write_interface
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prc_writer_c_lib_write_interface (writer, unit, id, feature)
  class(prc_writer_c_lib_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  select case (char (feature))
  case ("md5sum")
     write (unit, "(2x,9A)")  "interface"
     write (unit, "(5x,9A)")  "function ", &
          char (writer%get_c_procname (id, var_str ("get_md5sum"))), &
          " () result (cptr) bind(C)"
     write (unit, "(7x,9A)")  "import"
     write (unit, "(7x,9A)")  "implicit none"
     write (unit, "(7x,9A)")  "type(c_ptr) :: cptr"
     write (unit, "(5x,9A)")  "end function ", &
          char (writer%get_c_procname (id, var_str ("get_md5sum")))
     write (unit, "(2x,9A)")  "end interface"
  case ("openmp_supported")
     write (unit, "(2x,9A)")  "interface"
     write (unit, "(5x,9A)")  "function ", &
          char (writer%get_c_procname (id, feature)), &
          " () result (status) bind(C)"
     write (unit, "(7x,9A)")  "import"
     write (unit, "(7x,9A)")  "implicit none"
     write (unit, "(7x,9A)")  "logical(c_bool) :: status"
     write (unit, "(5x,9A)")  "end function ", &
          char (writer%get_c_procname (id, feature))
```

```
   write (unit, "(2x,9A)")  "end interface"
case ("n_in", "n_out", "n_flv", "n_hel", "n_col", "n_cin", "n_cf")
   write (unit, "(2x,9A)")  "interface"
   write (unit, "(5x,9A)")  "function ", &
        char (writer%get_c_procname (id, feature)), &
        " () result (n) bind(C)"
   write (unit, "(7x,9A)")  "import"
   write (unit, "(7x,9A)")  "implicit none"
   write (unit, "(7x,9A)")  "integer(c_int) :: n"
   write (unit, "(5x,9A)")  "end function ", &
        char (writer%get_c_procname (id, feature))
   write (unit, "(2x,9A)")  "end interface"
case ("flv_state", "hel_state")
   write (unit, "(2x,9A)")  "interface"
   write (unit, "(5x,9A)")  "subroutine ", &
        char (writer%get_c_procname (id, feature)), &
        " (", char (feature), ") bind(C)"
   write (unit, "(7x,9A)")  "import"
   write (unit, "(7x,9A)")  "implicit none"
   write (unit, "(7x,9A)")  "integer(c_int), dimension(*), intent(out) ", &
        ":: ", char (feature)
   write (unit, "(5x,9A)")  "end subroutine ", &
        char (writer%get_c_procname (id, feature))
   write (unit, "(2x,9A)")  "end interface"
case ("col_state")
   write (unit, "(2x,9A)")  "interface"
   write (unit, "(5x,9A)")  "subroutine ", &
        char (writer%get_c_procname (id, feature)), &
        " (col_state, ghost_flag) bind(C)"
   write (unit, "(7x,9A)")  "import"
   write (unit, "(7x,9A)")  "implicit none"
   write (unit, "(7x,9A)")  "integer(c_int), dimension(*), intent(out) ", &
        ":: col_state"
   write (unit, "(7x,9A)")  "logical(c_bool), dimension(*), intent(out) ", &
        ":: ghost_flag"
   write (unit, "(5x,9A)")  "end subroutine ", &
        char (writer%get_c_procname (id, feature))
   write (unit, "(2x,9A)")  "end interface"
case ("color_factors")
   write (unit, "(2x,9A)")  "interface"
   write (unit, "(5x,9A)")  "subroutine ", &
        char (writer%get_c_procname (id, feature)), &
        " (cf_index1, cf_index2, color_factors) bind(C)"
   write (unit, "(7x,9A)")  "import"
   write (unit, "(7x,9A)")  "implicit none"
   write (unit, "(7x,9A)")  "integer(c_int), dimension(*), &
        &intent(out) :: cf_index1"
   write (unit, "(7x,9A)")  "integer(c_int), dimension(*), &
        &intent(out) :: cf_index2"
   write (unit, "(7x,9A)")  "complex(c_default_complex), dimension(*), &
        &intent(out) :: color_factors"
   write (unit, "(5x,9A)")  "end subroutine ", &
        char (writer%get_c_procname (id, feature))
   write (unit, "(2x,9A)")  "end interface"
```

```
    end select
  end subroutine prc_writer_c_lib_write_interface
```

13.2.11 Retrieving the tables

In the previous section we had the writer routines for procedures that return tables, actually C pointers to tables. Here, we write convenience routines that unpack them and move the contents to suitable Fortran arrays.

The flavor and helicity tables are two-dimensional integer arrays. We use intermediate storage for correctly transforming C to Fortran data types.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
  procedure :: set_flv_state => prclib_driver_set_flv_state
  procedure :: set_hel_state => prclib_driver_set_hel_state
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prclib_driver_set_flv_state (driver, i, flv_state)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: i
    integer, dimension(:,:), allocatable, intent(out) :: flv_state
    integer :: n_tot, n_flv
    integer(c_int) :: pid
    integer(c_int), dimension(:,:), allocatable :: c_flv_state
    pid = i
    n_tot = driver%get_n_in (pid) + driver%get_n_out (pid)
    n_flv = driver%get_n_flv (pid)
    allocate (flv_state (n_tot, n_flv))
    allocate (c_flv_state (n_tot, n_flv))
    call driver%set_flv_state_ptr &
         (pid, c_flv_state, int ([n_tot, n_flv], kind=c_int))
    flv_state = c_flv_state
  end subroutine prclib_driver_set_flv_state

  subroutine prclib_driver_set_hel_state (driver, i, hel_state)
    class(prclib_driver_t), intent(in) :: driver
    integer, intent(in) :: i
    integer, dimension(:,:), allocatable, intent(out) :: hel_state
    integer :: n_tot, n_hel
    integer(c_int) :: pid
    integer(c_int), dimension(:,:), allocatable, target :: c_hel_state
    pid = i
    n_tot = driver%get_n_in (pid) + driver%get_n_out (pid)
    n_hel = driver%get_n_hel (pid)
    allocate (hel_state (n_tot, n_hel))
    allocate (c_hel_state (n_tot, n_hel))
    call driver%set_hel_state_ptr &
         (pid, c_hel_state, int ([n_tot, n_hel], kind=c_int))
    hel_state = c_hel_state
  end subroutine prclib_driver_set_hel_state
```

The color-flow table is three-dimensional, otherwise similar. We simultaneously set the ghost-flag table, which consists of logical entries.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: set_col_state => prclib_driver_set_col_state
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_set_col_state (driver, i, col_state, ghost_flag)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: i
  integer, dimension(:,:,:), allocatable, intent(out) :: col_state
  logical, dimension(:,:), allocatable, intent(out) :: ghost_flag
  integer :: n_cin, n_tot, n_col
  integer(c_int) :: pid
  integer(c_int), dimension(:,:,:), allocatable :: c_col_state
  logical(c_bool), dimension(:,:), allocatable :: c_ghost_flag
  pid = i
  n_cin = driver%get_n_cin (pid)
  n_tot = driver%get_n_in (pid) + driver%get_n_out (pid)
  n_col = driver%get_n_col (pid)
  allocate (col_state (n_cin, n_tot, n_col))
  allocate (c_col_state (n_cin, n_tot, n_col))
  allocate (ghost_flag (n_tot, n_col))
  allocate (c_ghost_flag (n_tot, n_col))
  call driver%set_col_state_ptr (pid, &
       c_col_state, c_ghost_flag, int ([n_cin, n_tot, n_col], kind=c_int))
  col_state = c_col_state
  ghost_flag = c_ghost_flag
end subroutine prclib_driver_set_col_state
```

The color-factor table is a sparse matrix: a two-column array of indices and one array which contains the corresponding factors.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: set_color_factors => prclib_driver_set_color_factors
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_set_color_factors (driver, i, color_factors, cf_index)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: i
  complex(default), dimension(:), allocatable, intent(out) :: color_factors
  integer, dimension(:,:), allocatable, intent(out) :: cf_index
  integer :: n_cf
  integer(c_int) :: pid
  complex(c_default_complex), dimension(:), allocatable, target :: c_color_factors
  integer(c_int), dimension(:), allocatable, target :: c_cf_index1
  integer(c_int), dimension(:), allocatable, target :: c_cf_index2
  pid = i
  n_cf = driver%get_n_cf (pid)
  allocate (color_factors (n_cf))
  allocate (c_color_factors (n_cf))
  allocate (c_cf_index1 (n_cf))
  allocate (c_cf_index2 (n_cf))
  call driver%set_color_factors_ptr (pid, &
       c_cf_index1, c_cf_index2, &
       c_color_factors, int ([n_cf], kind=c_int))
  color_factors = c_color_factors
  allocate (cf_index (2, n_cf))
  cf_index(1,:) = c_cf_index1
  cf_index(2,:) = c_cf_index2
```

```
end subroutine prclib_driver_set_color_factors
```

13.2.12 Returning a procedure pointer

The functions that directly access the matrix element, event by event, are assigned to a process-specific driver object as procedure pointers. For the `dlopen` interface, we use C function pointers. This subroutine returns such a pointer:

⟨*Prclib interfaces: public*⟩+≡

```
public :: prc_get_fptr
```

⟨*Prclib interfaces: interfaces*⟩+≡

```
abstract interface
   subroutine prc_get_fptr (pid, fid, fptr) bind(C)
     import
     integer(c_int), intent(in) :: pid
     integer(c_int), intent(in) :: fid
     type(c_funptr), intent(out) :: fptr
   end subroutine prc_get_fptr
end interface
```

This procedure writes the source code for the procedure pointer returning subroutine.

All C functions that are provided by the matrix element code of a specific process are handled here. The selection consists of a double layered `select case` construct.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: write_get_fptr_sub
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine write_get_fptr_sub (driver, unit, prefix)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: unit
  type(string_t), intent(in) :: prefix
  integer :: i, j
  write (unit, "(A)")  ""
  write (unit, "(A)")  "! Return C pointer to a procedure:"
  write (unit, "(A)")  "! pid = process index;  fid = function index"
  write (unit, "(4A)")  "subroutine ", char (prefix), "get_fptr ", &
       "(pid, fid, fptr) bind(C)"
  write (unit, "(A)")  "  use iso_c_binding"
  write (unit, "(A)")  "  use kinds"
  write (unit, "(A)")  "  implicit none"
  write (unit, "(A)")  "  integer(c_int), intent(in) :: pid"
  write (unit, "(A)")  "  integer(c_int), intent(in) :: fid"
  write (unit, "(A)")  "  type(c_funptr), intent(out) :: fptr"
  do i = 1, driver%n_processes
     call driver%record(i)%write_interfaces (unit)
  end do
  write (unit, "(A)")  "  select case (pid)"
  do i = 1, driver%n_processes
     write (unit, "(2x,A,I0,A)")  "case (", i, ")"
     write (unit, "(5x,A)")  "select case (fid)"
```

```
      associate (record => driver%record(i))
        do j = 1, size (record%feature)
           write (unit, "(5x,A,I0,9A)")  "case (", j, ");  ", &
                "fptr = c_funloc (", &
                char (record%get_c_procname (record%feature(j))), &
                ")"
        end do
      end associate
      write (unit, "(5x,A)")  "end select"
    end do
    write (unit, "(A)")  "  end select"
    write (unit, "(3A)")  "end subroutine ", char (prefix), "get_fptr"
  end subroutine write_get_fptr_sub
```

The procedures for which we want to return a pointer (the 'features' of the matrix element code) are actually Fortran module procedures. If we want to have a C signature, we must write wrapper functions for all of them. The procedures, their signatures, and the appropriate writer routines are specific for the process type.

To keep this generic, we do not provide the writer routines here, but just the interface for a writer routine. The actual routines are stored in the process record.

The `prefix` indicates the library, the `id` indicates the process, and `procname` is the bare name of the procedure to be written.

⟨*Prclib interfaces: public*⟩+≡

```
public :: write_driver_code
```

⟨*Prclib interfaces: interfaces*⟩+≡

```
abstract interface
   subroutine write_driver_code (unit, prefix, id, procname)
     import
     integer, intent(in) :: unit
     type(string_t), intent(in) :: prefix
     type(string_t), intent(in) :: id
     type(string_t), intent(in) :: procname
   end subroutine write_driver_code
end interface
```

13.2.13 Hooks

Interface for additional library unload / reload hooks (currently unused!)

⟨*Prclib interfaces: public*⟩+≡

```
public :: prclib_unload_hook
public :: prclib_reload_hook
```

⟨*Prclib interfaces: interfaces*⟩+≡

```
abstract interface
   subroutine prclib_unload_hook (libname)
     import
     type(string_t), intent(in) :: libname
   end subroutine prclib_unload_hook
```

```
    subroutine prclib_reload_hook (libname)
      import
      type(string_t), intent(in) :: libname
    end subroutine prclib_reload_hook
end interface
```

13.2.14 Make source, compile, link

Since we should have written a Makefile, these tasks amount to simple `make` calls. Note that the Makefile targets depend on each other, so calling `link` executes also the `source` and `compile` steps, when necessary.

The first routine writes source-code files for the individual processes. First it calls the writer routines directly for each process, then it calls `make source`. The make command may either post-process the files, or it may do the complete work, e.g., calling an external program the generates the files.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: make_source => prclib_driver_make_source
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_make_source (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  integer :: i
  do i = 1, driver%n_processes
     call driver%record(i)%write_source_code ()
  end do
  call os_system_call ("make source " // os_data%makeflags &
       // " -f " // driver%basename // ".makefile")
end subroutine prclib_driver_make_source
```

Compile matrix element source code and the driver source code.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: make_compile => prclib_driver_make_compile
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_make_compile (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  call os_system_call ("make compile " // os_data%makeflags &
       // " -f " // driver%basename // ".makefile")
end subroutine prclib_driver_make_compile
```

Combine all matrix-element code together with the driver in a process library on disk.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: make_link => prclib_driver_make_link
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_make_link (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  call os_system_call ("make link " // os_data%makeflags &
       // " -f " // driver%basename // ".makefile")
```

```
end subroutine prclib_driver_make_link
```

13.2.15 Clean up generated files

The task of cleaning any generated files should also be deferred to Makefile targets. Apart from removing everything, removing specific files may be useful for partial rebuilds. (Note that removing the makefile itself can only be done once, for obvious reasons.)

If there is no makefile, do nothing.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: clean_library => prclib_driver_clean_library
procedure :: clean_objects => prclib_driver_clean_objects
procedure :: clean_source => prclib_driver_clean_source
procedure :: clean_driver => prclib_driver_clean_driver
procedure :: clean_makefile => prclib_driver_clean_makefile
procedure :: clean => prclib_driver_clean
procedure :: distclean => prclib_driver_distclean
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_clean_library (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  if (driver%makefile_exists ()) then
     call os_system_call ("make clean-library " // os_data%makeflags &
          // " -f " // driver%basename // ".makefile")
  end if
end subroutine prclib_driver_clean_library

subroutine prclib_driver_clean_objects (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  if (driver%makefile_exists ()) then
     call os_system_call ("make clean-objects " // os_data%makeflags &
          // " -f " // driver%basename // ".makefile")
  end if
end subroutine prclib_driver_clean_objects

subroutine prclib_driver_clean_source (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  if (driver%makefile_exists ()) then
     call os_system_call ("make clean-source " // os_data%makeflags &
          // " -f " // driver%basename // ".makefile")
  end if
end subroutine prclib_driver_clean_source

subroutine prclib_driver_clean_driver (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  if (driver%makefile_exists ()) then
     call os_system_call ("make clean-driver " // os_data%makeflags &
          // " -f " // driver%basename // ".makefile")
```

```
  end if
end subroutine prclib_driver_clean_driver

subroutine prclib_driver_clean_makefile (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  if (driver%makefile_exists ()) then
     call os_system_call ("make clean-makefile " // os_data%makeflags &
          // " -f " // driver%basename // ".makefile")
  end if
end subroutine prclib_driver_clean_makefile

subroutine prclib_driver_clean (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  if (driver%makefile_exists ()) then
     call os_system_call ("make clean " // os_data%makeflags &
          // " -f " // driver%basename // ".makefile")
  end if
end subroutine prclib_driver_clean

subroutine prclib_driver_distclean (driver, os_data)
  class(prclib_driver_t), intent(in) :: driver
  type(os_data_t), intent(in) :: os_data
  if (driver%makefile_exists ()) then
     call os_system_call ("make distclean " // os_data%makeflags &
          // " -f " // driver%basename // ".makefile")
  end if
end subroutine prclib_driver_distclean
```

This Make target should remove all files that apply to a specific process. We execute this when we want to force remaking source code. Note that source targets need not have prerequisites, so just calling `make_source` would not do anything if the files exist.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: clean_proc => prclib_driver_clean_proc
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_clean_proc (driver, i, os_data)
  class(prclib_driver_t), intent(in) :: driver
  integer, intent(in) :: i
  type(os_data_t), intent(in) :: os_data
  type(string_t) :: id
  if (driver%makefile_exists ()) then
     id = driver%record(i)%id
    call os_system_call ("make clean-" // driver%record(i)%id // " " &
         // os_data%makeflags &
         // " -f " // driver%basename // ".makefile")
  end if
end subroutine prclib_driver_clean_proc
```

13.2.16 Further Tools

Check for the appropriate makefile.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: makefile_exists => prclib_driver_makefile_exists
```

⟨*Prclib interfaces: procedures*⟩+≡

```
function prclib_driver_makefile_exists (driver) result (flag)
  class(prclib_driver_t), intent(in) :: driver
  logical :: flag
  inquire (file = char (driver%basename) // ".makefile", exist = flag)
end function prclib_driver_makefile_exists
```

13.2.17 Load the library

Once the library has been linked, we can dlopen it and assign all procedure pointers to their proper places in the library driver object. The `loaded` flag is set only if all required pointers have become assigned.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: load => prclib_driver_load
```

⟨*Prclib interfaces: procedures*⟩+≡

```
subroutine prclib_driver_load (driver, os_data, noerror)
  class(prclib_driver_t), intent(inout) :: driver
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: noerror
  type(c_funptr) :: c_fptr
  logical :: ignore

  ignore = .false.;  if (present (noerror))  ignore = noerror

  driver%libname = os_get_dlname (driver%basename, os_data, noerror, noerror)
  if (driver%libname == "")  return
  select type (driver)
  type is (prclib_driver_dynamic_t)
     if (.not. dlaccess_is_open (driver%dlaccess)) then
        call dlaccess_init &
             (driver%dlaccess, var_str (."), driver%libname, os_data)
        if (.not. ignore)  call driver%check_dlerror ()
     end if
     driver%loaded = dlaccess_is_open (driver%dlaccess)
  class default
     driver%loaded = .true.
  end select
  if (.not. driver%loaded)  return

  c_fptr = driver%get_c_funptr (var_str ("get_n_processes"))
  call c_f_procpointer (c_fptr, driver%get_n_processes)
  driver%loaded = driver%loaded .and. associated (driver%get_n_processes)

  c_fptr = driver%get_c_funptr (var_str ("get_process_id_ptr"))
  call c_f_procpointer (c_fptr, driver%get_process_id_ptr)
  driver%loaded = driver%loaded .and. associated (driver%get_process_id_ptr)
```

```
c_fptr = driver%get_c_funptr (var_str ("get_model_name_ptr"))
call c_f_procpointer (c_fptr, driver%get_model_name_ptr)
driver%loaded = driver%loaded .and. associated (driver%get_model_name_ptr)

c_fptr = driver%get_c_funptr (var_str ("get_md5sum_ptr"))
call c_f_procpointer (c_fptr, driver%get_md5sum_ptr)
driver%loaded = driver%loaded .and. associated (driver%get_md5sum_ptr)

c_fptr = driver%get_c_funptr (var_str ("get_openmp_status"))
call c_f_procpointer (c_fptr, driver%get_openmp_status)
driver%loaded = driver%loaded .and. associated (driver%get_openmp_status)

c_fptr = driver%get_c_funptr (var_str ("get_n_in"))
call c_f_procpointer (c_fptr, driver%get_n_in)
driver%loaded = driver%loaded .and. associated (driver%get_n_in)

c_fptr = driver%get_c_funptr (var_str ("get_n_out"))
call c_f_procpointer (c_fptr, driver%get_n_out)
driver%loaded = driver%loaded .and. associated (driver%get_n_out)

c_fptr = driver%get_c_funptr (var_str ("get_n_flv"))
call c_f_procpointer (c_fptr, driver%get_n_flv)
driver%loaded = driver%loaded .and. associated (driver%get_n_flv)

c_fptr = driver%get_c_funptr (var_str ("get_n_hel"))
call c_f_procpointer (c_fptr, driver%get_n_hel)
driver%loaded = driver%loaded .and. associated (driver%get_n_hel)

c_fptr = driver%get_c_funptr (var_str ("get_n_col"))
call c_f_procpointer (c_fptr, driver%get_n_col)
driver%loaded = driver%loaded .and. associated (driver%get_n_col)

c_fptr = driver%get_c_funptr (var_str ("get_n_cin"))
call c_f_procpointer (c_fptr, driver%get_n_cin)
driver%loaded = driver%loaded .and. associated (driver%get_n_cin)

c_fptr = driver%get_c_funptr (var_str ("get_n_cf"))
call c_f_procpointer (c_fptr, driver%get_n_cf)
driver%loaded = driver%loaded .and. associated (driver%get_n_cf)

c_fptr = driver%get_c_funptr (var_str ("set_flv_state_ptr"))
call c_f_procpointer (c_fptr, driver%set_flv_state_ptr)
driver%loaded = driver%loaded .and. associated (driver%set_flv_state_ptr)

c_fptr = driver%get_c_funptr (var_str ("set_hel_state_ptr"))
call c_f_procpointer (c_fptr, driver%set_hel_state_ptr)
driver%loaded = driver%loaded .and. associated (driver%set_hel_state_ptr)

c_fptr = driver%get_c_funptr (var_str ("set_col_state_ptr"))
call c_f_procpointer (c_fptr, driver%set_col_state_ptr)
driver%loaded = driver%loaded .and. associated (driver%set_col_state_ptr)

c_fptr = driver%get_c_funptr (var_str ("set_color_factors_ptr"))
```

```
    call c_f_procpointer (c_fptr, driver%set_color_factors_ptr)
    driver%loaded = driver%loaded .and. associated (driver%set_color_factors_ptr)

    c_fptr = driver%get_c_funptr (var_str ("get_fptr"))
    call c_f_procpointer (c_fptr, driver%get_fptr)
    driver%loaded = driver%loaded .and. associated (driver%get_fptr)

  end subroutine prclib_driver_load
```

Unload. To be sure, nullify the procedure pointers.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
  procedure :: unload => prclib_driver_unload
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prclib_driver_unload (driver)
    class(prclib_driver_t), intent(inout) :: driver
    select type (driver)
    type is (prclib_driver_dynamic_t)
       if (dlaccess_is_open (driver%dlaccess)) then
          call dlaccess_final (driver%dlaccess)
          call driver%check_dlerror ()
       end if
    end select
    driver%loaded = .false.
    nullify (driver%get_n_processes)
    nullify (driver%get_process_id_ptr)
    nullify (driver%get_model_name_ptr)
    nullify (driver%get_md5sum_ptr)
    nullify (driver%get_openmp_status)
    nullify (driver%get_n_in)
    nullify (driver%get_n_out)
    nullify (driver%get_n_flv)
    nullify (driver%get_n_hel)
    nullify (driver%get_n_col)
    nullify (driver%get_n_cin)
    nullify (driver%get_n_cf)
    nullify (driver%set_flv_state_ptr)
    nullify (driver%set_hel_state_ptr)
    nullify (driver%set_col_state_ptr)
    nullify (driver%set_color_factors_ptr)
    nullify (driver%get_fptr)
  end subroutine prclib_driver_unload
```

This subroutine checks the `dlerror` content and issues a fatal error if it finds an error there.

⟨*Prclib interfaces: prclib driver dynamic: TBP*⟩≡

```
  procedure :: check_dlerror => prclib_driver_check_dlerror
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  subroutine prclib_driver_check_dlerror (driver)
    class(prclib_driver_dynamic_t), intent(in) :: driver
    if (dlaccess_has_error (driver%dlaccess)) then
       call msg_fatal (char (dlaccess_get_error (driver%dlaccess)))
    end if
```

```
end subroutine prclib_driver_check_dlerror
```

Get the handle (C function pointer) for a given "feature" of the matrix element code, so it can be assigned to the appropriate procedure pointer slot. In the static case, this is a trivial pointer assignment, hard-coded into the driver type implementation.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure (prclib_driver_get_c_funptr), deferred :: get_c_funptr
```

⟨*Prclib interfaces: interfaces*⟩+≡

```
abstract interface
   function prclib_driver_get_c_funptr (driver, feature) result (c_fptr)
     import
     class(prclib_driver_t), intent(inout) :: driver
     type(string_t), intent(in) :: feature
     type(c_funptr) :: c_fptr
   end function prclib_driver_get_c_funptr
end interface
```

In the dynamic-library case, we call the DL interface to retrieve the C pointer to a named procedure.

⟨*Prclib interfaces: prclib driver dynamic: TBP*⟩+≡

```
procedure :: get_c_funptr => prclib_driver_dynamic_get_c_funptr
```

⟨*Prclib interfaces: procedures*⟩+≡

```
function prclib_driver_dynamic_get_c_funptr (driver, feature) result (c_fptr)
  class(prclib_driver_dynamic_t), intent(inout) :: driver
  type(string_t), intent(in) :: feature
  type(c_funptr) :: c_fptr
  type(string_t) :: prefix, full_name
  prefix = lower_case (driver%basename) // "_"
  full_name = prefix // feature
  c_fptr = dlaccess_get_c_funptr (driver%dlaccess, full_name)
  call driver%check_dlerror ()
end function prclib_driver_dynamic_get_c_funptr
```

13.2.18 MD5 sums

Recall the MD5 sum written in the Makefile

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
procedure :: get_md5sum_makefile => prclib_driver_get_md5sum_makefile
```

⟨*Prclib interfaces: procedures*⟩+≡

```
function prclib_driver_get_md5sum_makefile (driver) result (md5sum)
  class(prclib_driver_t), intent(in) :: driver
  character(32) :: md5sum
  type(string_t) :: filename
  character(80) :: buffer
  logical :: exist
  integer :: u, iostat
  md5sum = ""
  filename = driver%basename // ".makefile"
```

```
    inquire (file = char (filename), exist = exist)
    if (exist) then
       u = free_unit ()
       open (u, file = char (filename), action = "read", status = "old")
       iostat = 0
       do
          read (u, "(A)", iostat = iostat)  buffer
          if (iostat /= 0)  exit
          buffer = adjustl (buffer)
          select case (buffer(1:9))
          case ("MD5SUM = ")
             read (buffer(11:), "(A32)")  md5sum
             exit
          end select
       end do
       close (u)
    end if
  end function prclib_driver_get_md5sum_makefile
```

Recall the MD5 sum written in the driver source code.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
  procedure :: get_md5sum_driver => prclib_driver_get_md5sum_driver
```

⟨*Prclib interfaces: procedures*⟩+≡

```
  function prclib_driver_get_md5sum_driver (driver) result (md5sum)
    class(prclib_driver_t), intent(in) :: driver
    character(32) :: md5sum
    type(string_t) :: filename
    character(80) :: buffer
    logical :: exist
    integer :: u, iostat
    md5sum = ""
    filename = driver%basename // ".f90"
    inquire (file = char (filename), exist = exist)
    if (exist) then
       u = free_unit ()
       open (u, file = char (filename), action = "read", status = "old")
       iostat = 0
       do
          read (u, "(A)", iostat = iostat)  buffer
          if (iostat /= 0)  exit
          buffer = adjustl (buffer)
          select case (buffer(1:9))
          case ("md5sum = ")
             read (buffer(11:), "(A32)")  md5sum
             exit
          end select
       end do
       close (u)
    end if
  end function prclib_driver_get_md5sum_driver
```

Recall the MD5 sum written in the matrix element source code.

⟨*Prclib interfaces: prclib driver: TBP*⟩+≡

```
    procedure :: get_md5sum_source => prclib_driver_get_md5sum_source
```

⟨*Prclib interfaces: procedures*⟩+≡

```
    function prclib_driver_get_md5sum_source (driver, i) result (md5sum)
      class(prclib_driver_t), intent(in) :: driver
      integer, intent(in) :: i
      character(32) :: md5sum
      type(string_t) :: filename
      character(80) :: buffer
      logical :: exist
      integer :: u, iostat
      md5sum = ""

      filename = driver%record(i)%id // ".f90"
      inquire (file = char (filename), exist = exist)
      if (exist) then
         u = free_unit ()
         open (u, file = char (filename), action = "read", status = "old")
         iostat = 0
         do
            read (u, "(A)", iostat = iostat)  buffer
            if (iostat /= 0)  exit
            buffer = adjustl (buffer)
            select case (buffer(1:9))
            case ("md5sum = ")
               read (buffer(11:), "(A32)")  md5sum
               exit
            end select
         end do
         close (u)
      end if
    end function prclib_driver_get_md5sum_source
```

13.2.19 Unit Test

Test module, followed by the corresponding implementation module.

⟨prclib_interfaces_ut.f90⟩≡

```
  ⟨File header⟩

  module prclib_interfaces_ut
    use kinds
    use system_dependencies, only: CC_IS_GNU, CC_HAS_QUADMATH
    use unit_tests
    use prclib_interfaces_uti

  ⟨Standard module head⟩

  ⟨Prclib interfaces: public test⟩

  ⟨Prclib interfaces: public test auxiliary⟩

  contains
```

```
  ⟨Prclib interfaces: test driver⟩

  end module prclib_interfaces_ut
```

⟨prclib_interfaces_uti.f90⟩≡

```
  ⟨File header⟩

  module prclib_interfaces_uti

    use, intrinsic :: iso_c_binding !NODEP!

    use kinds
    use system_dependencies, only: CC_HAS_QUADMATH, DEFAULT_FC_PRECISION
  ⟨Use strings⟩
    use io_units
    use system_defs, only: TAB
    use os_interface

    use prclib_interfaces

  ⟨Standard module head⟩

  ⟨Prclib interfaces: public test auxiliary⟩

  ⟨Prclib interfaces: test declarations⟩

  ⟨Prclib interfaces: test types⟩

  contains

  ⟨Prclib interfaces: tests⟩

  ⟨Prclib interfaces: test auxiliary⟩

  end module prclib_interfaces_uti
```

API: driver for the unit tests below.

⟨*Prclib interfaces: public test*⟩≡

```
    public :: prclib_interfaces_test
```

⟨*Prclib interfaces: test driver*⟩≡

```
    subroutine prclib_interfaces_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Prclib interfaces: execute tests⟩
    end subroutine prclib_interfaces_test
```

Empty process list

Test 1: Create a driver object and display its contents. One of the feature lists references a writer procedure; this is just a dummy that does nothing useful.

⟨*Prclib interfaces: execute tests*⟩≡

```
    call test (prclib_interfaces_1, "prclib_interfaces_1", &
         "create driver object", &
```

```
        u, results)
```

⟨*Prclib interfaces: test declarations*⟩≡

```
    public :: prclib_interfaces_1
```

⟨*Prclib interfaces: tests*⟩≡

```
    subroutine prclib_interfaces_1 (u)
      integer, intent(in) :: u
      class(prclib_driver_t), allocatable :: driver
      character(32), parameter :: md5sum = "prclib_interfaces_1_md5sum      "
      class(prc_writer_t), pointer :: test_writer_1


      write (u, "(A)")  "* Test output: prclib_interfaces_1"
      write (u, "(A)")  "*   Purpose: display the driver object contents"
      write (u, *)
      write (u, "(A)")  "* Create a prclib driver object"
      write (u, "(A)")

      call dispatch_prclib_driver (driver, var_str ("prclib"), var_str (""))
      call driver%init (3)
      call driver%set_md5sum (md5sum)

      allocate (test_writer_1_t :: test_writer_1)

      call driver%set_record (1, var_str ("test1"), var_str ("test_model"), &
           [var_str ("init")], test_writer_1)

      call driver%set_record (2, var_str ("test2"), var_str ("foo_model"), &
           [var_str ("another_proc")], test_writer_1)

      call driver%set_record (3, var_str ("test3"), var_str ("test_model"), &
           [var_str ("init"), var_str ("some_proc")], test_writer_1)

      call driver%write (u)

      deallocate (test_writer_1)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: prclib_interfaces_1"
    end subroutine prclib_interfaces_1
```

The writer: the procedures write just comment lines. We can fix an instance of this as a parameter (since it has no mutable content) and just reference the fixed parameter.

NOTE: temporarily made public.

⟨*Prclib interfaces: test types*⟩≡

```
    type, extends (prc_writer_t) :: test_writer_1_t
     contains
       procedure, nopass :: type_name => test_writer_1_type_name
       procedure :: write_makefile_code => test_writer_1_mk
       procedure :: write_source_code => test_writer_1_src
       procedure :: write_interface => test_writer_1_if
       procedure :: write_md5sum_call => test_writer_1_md5sum
```

```
   procedure :: write_int_sub_call => test_writer_1_int_sub
   procedure :: write_col_state_call => test_writer_1_col_state
   procedure :: write_color_factors_call => test_writer_1_col_factors
end type test_writer_1_t
```

⟨*Prclib interfaces: test auxiliary*⟩≡

```
function test_writer_1_type_name () result (string)
  type(string_t) :: string
  string = "test_1"
end function test_writer_1_type_name

subroutine test_writer_1_mk (writer, unit, id, os_data, testflag)
  class(test_writer_1_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: testflag
  write (unit, "(5A)")  "# Makefile code for process ", char (id), &
       " goes here."
end subroutine test_writer_1_mk

subroutine test_writer_1_src (writer, id)
  class(test_writer_1_t), intent(in) :: writer
  type(string_t), intent(in) :: id
end subroutine test_writer_1_src

subroutine test_writer_1_if (writer, unit, id, feature)
  class(test_writer_1_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  write (unit, "(2x,9A)")  "! Interface code for ", &
     char (id), "_", char (writer%get_procname (feature)), &
     " goes here."
end subroutine test_writer_1_if

subroutine test_writer_1_md5sum (writer, unit, id)
  class(test_writer_1_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  write (unit, "(5x,9A)")  "! MD5sum call for ", char (id), " goes here."
end subroutine test_writer_1_md5sum

subroutine test_writer_1_int_sub (writer, unit, id, feature)
  class(test_writer_1_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  write (unit, "(5x,9A)")  "! ", char (feature), " call for ", &
       char (id), " goes here."
end subroutine test_writer_1_int_sub

subroutine test_writer_1_col_state (writer, unit, id)
  class(test_writer_1_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
```

```
  write (unit, "(5x,9A)")  "! col_state call for ", &
       char (id), " goes here."
end subroutine test_writer_1_col_state

subroutine test_writer_1_col_factors (writer, unit, id)
  class(test_writer_1_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  write (unit, "(5x,9A)")  "! color_factors call for ", &
       char (id), " goes here."
end subroutine test_writer_1_col_factors
```

Process library driver file

Test 2: Write the driver file for a test case with two processes. The first process needs no wrapper (C library), the second one needs wrappers (Fortran module library).

⟨*Prclib interfaces: execute tests*⟩+≡

```
call test (prclib_interfaces_2, "prclib_interfaces_2", &
     "write driver file", &
     u, results)
```

⟨*Prclib interfaces: test declarations*⟩+≡

```
public :: prclib_interfaces_2
```

⟨*Prclib interfaces: tests*⟩+≡

```
subroutine prclib_interfaces_2 (u)
  integer, intent(in) :: u
  class(prclib_driver_t), allocatable :: driver
  character(32), parameter :: md5sum = "prclib_interfaces_2_md5sum      "
  class(prc_writer_t), pointer :: test_writer_1, test_writer_2

  write (u, "(A)")  "* Test output: prclib_interfaces_2"
  write (u, "(A)")  "*   Purpose: check the generated driver source code"
  write (u, "(A)")
  write (u, "(A)")  "* Create a prclib driver object (2 processes)"
  write (u, "(A)")

  call dispatch_prclib_driver (driver, var_str ("prclib2"), var_str (""))
  call driver%init (2)
  call driver%set_md5sum (md5sum)

  allocate (test_writer_1_t :: test_writer_1)
  allocate (test_writer_2_t :: test_writer_2)

  call driver%set_record (1, var_str ("test1"), var_str ("Test_model"), &
       [var_str ("proc1")], test_writer_1)

  call driver%set_record (2, var_str ("test2"), var_str ("Test_model"), &
       [var_str ("proc1"), var_str ("proc2")], test_writer_2)

  call driver%write (u)

  write (u, "(A)")
```

```
  write (u, "(A)")  "* Write the driver file"
  write (u, "(A)")  "* File contents:"
  write (u, "(A)")

  call driver%generate_driver_code (u)

  deallocate (test_writer_1)
  deallocate (test_writer_2)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: prclib_interfaces_2"
end subroutine prclib_interfaces_2
```

A writer with wrapper code: the procedures again write just comment lines. Since all procedures are NOPASS, we can reuse two of the TBP.

⟨*Prclib interfaces: test types*⟩+≡

```
type, extends (prc_writer_f_module_t) :: test_writer_2_t
 contains
   procedure, nopass :: type_name => test_writer_2_type_name
   procedure :: write_makefile_code => test_writer_2_mk
   procedure :: write_source_code => test_writer_2_src
   procedure :: write_interface => test_writer_2_if
   procedure :: write_wrapper => test_writer_2_wr
end type test_writer_2_t
```

⟨*Prclib interfaces: test auxiliary*⟩+≡

```
function test_writer_2_type_name () result (string)
  type(string_t) :: string
  string = "test_2"
end function test_writer_2_type_name

subroutine test_writer_2_mk (writer, unit, id, os_data, testflag)
  class(test_writer_2_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: testflag
  write (unit, "(5A)")  "# Makefile code for process ", char (id), &
       " goes here."
end subroutine test_writer_2_mk

subroutine test_writer_2_src (writer, id)
  class(test_writer_2_t), intent(in) :: writer
  type(string_t), intent(in) :: id
end subroutine test_writer_2_src

subroutine test_writer_2_if (writer, unit, id, feature)
  class(test_writer_2_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  write (unit, "(2x,9A)")  "! Interface code for ", &
     char (writer%get_module_name (id)), "_", &
     char (writer%get_procname (feature)), " goes here."
```

```
end subroutine test_writer_2_if

subroutine test_writer_2_wr (writer, unit, id, feature)
  class(test_writer_2_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  write (unit, *)
  write (unit, "(9A)")  "! Wrapper code for ", &
     char (writer%get_c_procname (id, feature)), " goes here."
end subroutine test_writer_2_wr
```

Process library makefile

Test 3: Write the makefile for compiling and linking the process library (processes and driver code). There are two processes, one with one method, one with two methods.

To have predictable output, we reset the system-dependent initial components of os_data to known values.

⟨*Prclib interfaces: execute tests*⟩+≡

```
call test (prclib_interfaces_3, "prclib_interfaces_3", &
     "write makefile", &
     u, results)
```

⟨*Prclib interfaces: test declarations*⟩+≡

```
public :: prclib_interfaces_3
```

⟨*Prclib interfaces: tests*⟩+≡

```
subroutine prclib_interfaces_3 (u)
  integer, intent(in) :: u
  class(prclib_driver_t), allocatable :: driver
  type(os_data_t) :: os_data
  character(32), parameter :: md5sum = "prclib_interfaces_3_md5sum      "
  class(prc_writer_t), pointer :: test_writer_1, test_writer_2

  call os_data_init (os_data)
  os_data%fc = "fortran-compiler"
  os_data%whizard_includes = "-I module-dir"
  os_data%fcflags = "-C=all"
  os_data%fcflags_pic = "-PIC"
  os_data%cc = "c-compiler"
  os_data%cflags = "-I include-dir"
  os_data%cflags_pic = "-PIC"
  os_data%whizard_ldflags = ""
  os_data%ldflags = ""
  os_data%whizard_libtool = "my-libtool"
  os_data%latex = "latex -halt-on-error"
  os_data%mpost = "mpost --math=scaled -halt-on-error"
  os_data%dvips = "dvips"
  os_data%ps2pdf = "ps2pdf14"
  os_data%whizard_texpath = ""

  write (u, "(A)")  "* Test output: prclib_interfaces_3"
  write (u, "(A)")  "*   Purpose: check the generated Makefile"
```

```
    write (u, *)
    write (u, "(A)")  "* Create a prclib driver object (2 processes)"
    write (u, "(A)")

    call dispatch_prclib_driver (driver, var_str ("prclib3"), var_str (""))
    call driver%init (2)
    call driver%set_md5sum (md5sum)

    allocate (test_writer_1_t :: test_writer_1)
    allocate (test_writer_2_t :: test_writer_2)

    call driver%set_record (1, var_str ("test1"), var_str ("Test_model"), &
         [var_str ("proc1")], test_writer_1)

    call driver%set_record (2, var_str ("test2"), var_str ("Test_model"), &
         [var_str ("proc1"), var_str ("proc2")], test_writer_2)

    call driver%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Write Makefile"
    write (u, "(A)")  "* File contents:"
    write (u, "(A)")

    call driver%generate_makefile (u, os_data)

    deallocate (test_writer_1)
    deallocate (test_writer_2)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prclib_interfaces_3"
  end subroutine prclib_interfaces_3
```

Compile test with Fortran module

Test 4: Write driver and makefile and try to compile and link the library driver.

There is a single test process with a single feature. The process code is provided as a Fortran module, therefore we need a wrapper for the featured procedure.

⟨*Prclib interfaces: execute tests*⟩+≡

```
  call test (prclib_interfaces_4, "prclib_interfaces_4", &
       "compile and link (Fortran module)", &
       u, results)
```

⟨*Prclib interfaces: test declarations*⟩+≡

```
  public :: prclib_interfaces_4
```

⟨*Prclib interfaces: tests*⟩+≡

```
  subroutine prclib_interfaces_4 (u)
    integer, intent(in) :: u
    class(prclib_driver_t), allocatable :: driver
    class(prc_writer_t), pointer :: test_writer_4
    type(os_data_t) :: os_data
```

```
integer :: u_file

integer, dimension(:,:), allocatable :: flv_state
integer, dimension(:,:), allocatable :: hel_state
integer, dimension(:,:,:), allocatable :: col_state
logical, dimension(:,:), allocatable :: ghost_flag
integer, dimension(:,:), allocatable :: cf_index
complex(default), dimension(:), allocatable :: color_factors
character(32), parameter :: md5sum = "prclib_interfaces_4_md5sum      "
character(32) :: md5sum_file

type(c_funptr) :: proc1_ptr
interface
   subroutine proc1_t (n) bind(C)
     import
     integer(c_int), intent(out) :: n
   end subroutine proc1_t
end interface
procedure(proc1_t), pointer :: proc1
integer(c_int) :: n

write (u, "(A)")  "* Test output: prclib_interfaces_4"
write (u, "(A)")  "*   Purpose: compile, link, and load process library"
write (u, "(A)")  "*            with (fake) matrix-element code &
     &as a Fortran module"
write (u, *)
write (u, "(A)")  "* Create a prclib driver object (1 process)"
write (u, "(A)")

call os_data_init (os_data)

allocate (test_writer_4_t :: test_writer_4)
call test_writer_4%init_test ()

call dispatch_prclib_driver (driver, var_str ("prclib4"), var_str (""))
call driver%init (1)
call driver%set_md5sum (md5sum)

call driver%set_record (1, var_str ("test4"), var_str ("Test_model"), &
     [var_str ("proc1")], test_writer_4)

call driver%write (u)

write (u, *)
write (u, "(A)")  "* Write Makefile"
u_file = free_unit ()
open (u_file, file="prclib4.makefile", status="replace", action="write")
call driver%generate_makefile (u_file, os_data)
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Recall MD5 sum from Makefile"
write (u, "(A)")
```

```
md5sum_file = driver%get_md5sum_makefile ()
write (u, "(1x,A,A,A)")  "MD5 sum = '", md5sum_file, "'"

write (u, "(A)")
write (u, "(A)")  "* Write driver source code"

u_file = free_unit ()
open (u_file, file="prclib4.f90", status="replace", action="write")
call driver%generate_driver_code (u_file)
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Recall MD5 sum from driver source"
write (u, "(A)")

md5sum_file = driver%get_md5sum_driver ()
write (u, "(1x,A,A,A)")  "MD5 sum = '", md5sum_file, "'"

write (u, "(A)")
write (u, "(A)")  "* Write matrix-element source code"
call driver%make_source (os_data)

write (u, "(A)")
write (u, "(A)")  "* Recall MD5 sum from matrix-element source"
write (u, "(A)")

md5sum_file = driver%get_md5sum_source (1)
write (u, "(1x,A,A,A)")  "MD5 sum = '", md5sum_file, "'"

write (u, "(A)")
write (u, "(A)")  "* Compile source code"
call driver%make_compile (os_data)

write (u, "(A)")  "* Link library"
call driver%make_link (os_data)

write (u, "(A)")  "* Load library"
call driver%load (os_data)

write (u, *)
call driver%write (u)
write (u, *)

if (driver%loaded) then
   write (u, "(A)")  "* Call library functions:"
   write (u, *)
   write (u, "(1x,A,I0)")  "n_processes   = ", driver%get_n_processes ()
   write (u, "(1x,A,A,A)")  "process_id    = '", &
        char (driver%get_process_id (1)), "'"
   write (u, "(1x,A,A,A)")  "model_name    = '", &
        char (driver%get_model_name (1)), "'"
   write (u, "(1x,A,A,A)")  "md5sum (lib)  = '", &
        char (driver%get_md5sum (0)), "'"
   write (u, "(1x,A,A,A)")  "md5sum (proc) = '", &
```

```
         char (driver%get_md5sum (1)), "'"
      write (u, "(1x,A,L1)")  "openmp_status = ", driver%get_openmp_status (1)
      write (u, "(1x,A,I0)")  "n_in  = ", driver%get_n_in (1)
      write (u, "(1x,A,I0)")  "n_out = ", driver%get_n_out (1)
      write (u, "(1x,A,I0)")  "n_flv = ", driver%get_n_flv (1)
      write (u, "(1x,A,I0)")  "n_hel = ", driver%get_n_hel (1)
      write (u, "(1x,A,I0)")  "n_col = ", driver%get_n_col (1)
      write (u, "(1x,A,I0)")  "n_cin = ", driver%get_n_cin (1)
      write (u, "(1x,A,I0)")  "n_cf  = ", driver%get_n_cf (1)

      call driver%set_flv_state (1, flv_state)
      write (u, "(1x,A,10(1x,I0))")  "flv_state =", flv_state

      call driver%set_hel_state (1, hel_state)
      write (u, "(1x,A,10(1x,I0))")  "hel_state =", hel_state

      call driver%set_col_state (1, col_state, ghost_flag)
      write (u, "(1x,A,10(1x,I0))")  "col_state =", col_state
      write (u, "(1x,A,10(1x,L1))")  "ghost_flag =", ghost_flag

      call driver%set_color_factors (1, color_factors, cf_index)
      write (u, "(1x,A,10(1x,F5.3))")  "color_factors =", color_factors
      write (u, "(1x,A,10(1x,I0))")  "cf_index =", cf_index

      call driver%get_fptr (1, 1, proc1_ptr)
      call c_f_procpointer (proc1_ptr, proc1)
      if (associated (proc1)) then
         write (u, *)
         call proc1 (n)
         write (u, "(1x,A,I0)")  "proc1(1) = ", n
      end if

   end if

   deallocate (test_writer_4)

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: prclib_interfaces_4"
end subroutine prclib_interfaces_4
```

This version of test-code writer actually writes an interface and wrapper code. The wrapped function is a no-parameter function with integer result.

The stored MD5 sum may be modified.

We will reuse this later, therefore public.

⟨*Prclib interfaces: public test auxiliary*⟩≡

```
public :: test_writer_4_t
```

⟨*Prclib interfaces: test types*⟩+≡

```
type, extends (prc_writer_f_module_t) :: test_writer_4_t
 contains
   procedure, nopass :: type_name => test_writer_4_type_name
   procedure, nopass :: get_module_name => &
        test_writer_4_get_module_name
   procedure :: write_makefile_code => test_writer_4_mk
```

```
   procedure :: write_source_code => test_writer_4_src
   procedure :: write_interface => test_writer_4_if
   procedure :: write_wrapper => test_writer_4_wr
end type test_writer_4_t
```

⟨*Prclib interfaces: test auxiliary*⟩+≡

```
function test_writer_4_type_name () result (string)
  type(string_t) :: string
  string = "test_4"
end function test_writer_4_type_name

function test_writer_4_get_module_name (id) result (name)
  type(string_t), intent(in) :: id
  type(string_t) :: name
  name = "tpr_" // id
end function test_writer_4_get_module_name

subroutine test_writer_4_mk (writer, unit, id, os_data, testflag)
  class(test_writer_4_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: testflag
  write (unit, "(5A)")  "SOURCES += ", char (id), ".f90"
  write (unit, "(5A)")  "OBJECTS += ", char (id), ".lo"
  write (unit, "(5A)")  "CLEAN_SOURCES += ", char (id), ".f90"
  write (unit, "(5A)")  "CLEAN_OBJECTS += tpr_", char (id), ".mod"
  write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), ".lo"
  write (unit, "(5A)")  char (id), ".lo: ", char (id), ".f90"
  write (unit, "(5A)")  TAB, "$(LTFCOMPILE) $<"
end subroutine test_writer_4_mk

subroutine test_writer_4_src (writer, id)
  class(test_writer_4_t), intent(in) :: writer
  type(string_t), intent(in) :: id
  call write_test_module_file (id, var_str ("proc1"), writer%md5sum)
end subroutine test_writer_4_src

subroutine test_writer_4_if (writer, unit, id, feature)
  class(test_writer_4_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  write (unit, "(2x,9A)")  "interface"
  write (unit, "(5x,9A)")  "subroutine ", &
     char (writer%get_c_procname (id, feature)), &
     " (n) bind(C)"
  write (unit, "(7x,9A)")  "import"
  write (unit, "(7x,9A)")  "implicit none"
  write (unit, "(7x,9A)")  "integer(c_int), intent(out) :: n"
  write (unit, "(5x,9A)")  "end subroutine ", &
     char (writer%get_c_procname (id, feature))
  write (unit, "(2x,9A)")  "end interface"
end subroutine test_writer_4_if
```

```
subroutine test_writer_4_wr (writer, unit, id, feature)
  class(test_writer_4_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  write (unit, *)
  write (unit, "(9A)")  "subroutine ", &
       char (writer%get_c_procname (id, feature)), &
     " (n) bind(C)"
  write (unit, "(2x,9A)")  "use iso_c_binding"
  write (unit, "(2x,9A)")  "use tpr_", char (id), ", only: ", &
       char (writer%get_procname (feature))
  write (unit, "(2x,9A)")  "implicit none"
  write (unit, "(2x,9A)")  "integer(c_int), intent(out) :: n"
  write (unit, "(2x,9A)")  "call ", char (feature), " (n)"
  write (unit, "(9A)")  "end subroutine ", &
     char (writer%get_c_procname (id, feature))
end subroutine test_writer_4_wr
```

We need a test module file (actually, one for each process in the test above) that allows us to check compilation and linking. The test module implements a colorless $1 \to 2$ process, and it implements one additional function (feature), the name given as an argument.

⟨*Prclib interfaces: test auxiliary*⟩+≡

```
subroutine write_test_module_file (basename, feature, md5sum)
  type(string_t), intent(in) :: basename
  type(string_t), intent(in) :: feature
  character(32), intent(in) :: md5sum
  integer :: u
  u = free_unit ()
  open (u, file = char (basename) // ".f90", &
       status = "replace", action = "write")
  write (u, "(A)")  "! (Pseudo) matrix element code file &
       &for WHIZARD self-test"
  write (u, *)
  write (u, "(A)")  "module tpr_" // char (basename)
  write (u, *)
  write (u, "(2x,A)")  "use kinds"
  write (u, "(2x,A)")  "use omega_color, OCF => omega_color_factor"
  write (u, *)
  write (u, "(2x,A)")  "implicit none"
  write (u, "(2x,A)")  "private"
  write (u, *)
  call write_test_me_code_1 (u)
  write (u, *)
  write (u, "(2x,A)")  "public :: " // char (feature)
  write (u, *)
  write (u, "(A)")  "contains"
  write (u, *)
  call write_test_me_code_2 (u, md5sum)
  write (u, *)
  write (u, "(2x,A)")  "subroutine " // char (feature) // " (n)"
  write (u, "(2x,A)")  "  integer, intent(out) :: n"
  write (u, "(2x,A)")  "  n = 42"
```

```
    write (u, "(2x,A)")  "end subroutine " // char (feature)
    write (u, *)
    write (u, "(A)")  "end module tpr_" // char (basename)
    close (u)
  end subroutine write_test_module_file
```

The following two subroutines provide building blocks for a matrix-element source code file, useful only for testing the workflow. The first routine writes the header part, the other routine the implementation of the procedures listed in the header.

⟨*Prclib interfaces: test auxiliary*⟩+≡

```
  subroutine write_test_me_code_1 (u)
    integer, intent(in) :: u
    write (u, "(2x,A)")  "public :: md5sum"
    write (u, "(2x,A)")  "public :: openmp_supported"
    write (u, *)
    write (u, "(2x,A)")  "public :: n_in"
    write (u, "(2x,A)")  "public :: n_out"
    write (u, "(2x,A)")  "public :: n_flv"
    write (u, "(2x,A)")  "public :: n_hel"
    write (u, "(2x,A)")  "public :: n_cin"
    write (u, "(2x,A)")  "public :: n_col"
    write (u, "(2x,A)")  "public :: n_cf"
    write (u, *)
    write (u, "(2x,A)")  "public :: flv_state"
    write (u, "(2x,A)")  "public :: hel_state"
    write (u, "(2x,A)")  "public :: col_state"
    write (u, "(2x,A)")  "public :: color_factors"
  end subroutine write_test_me_code_1

  subroutine write_test_me_code_2 (u, md5sum)
    integer, intent(in) :: u
    character(32), intent(in) :: md5sum
    write (u, "(2x,A)")  "pure function md5sum ()"
    write (u, "(2x,A)")  "  character(len=32) :: md5sum"
    write (u, "(2x,A)")  "  md5sum = '" // md5sum // "'"
    write (u, "(2x,A)")  "end function md5sum"
    write (u, *)
    write (u, "(2x,A)")  "pure function openmp_supported () result (status)"
    write (u, "(2x,A)")  "  logical :: status"
    write (u, "(2x,A)")  "  status = .false."
    write (u, "(2x,A)")  "end function openmp_supported"
    write (u, *)
    write (u, "(2x,A)")  "pure function n_in () result (n)"
    write (u, "(2x,A)")  "  integer :: n"
    write (u, "(2x,A)")  "  n = 1"
    write (u, "(2x,A)")  "end function n_in"
    write (u, *)
    write (u, "(2x,A)")  "pure function n_out () result (n)"
    write (u, "(2x,A)")  "  integer :: n"
    write (u, "(2x,A)")  "  n = 2"
    write (u, "(2x,A)")  "end function n_out"
    write (u, *)
```

```
  write (u, "(2x,A)")  "pure function n_flv () result (n)"
  write (u, "(2x,A)")  "  integer :: n"
  write (u, "(2x,A)")  "  n = 1"
  write (u, "(2x,A)")  "end function n_flv"
  write (u, *)
  write (u, "(2x,A)")  "pure function n_hel () result (n)"
  write (u, "(2x,A)")  "  integer :: n"
  write (u, "(2x,A)")  "  n = 1"
  write (u, "(2x,A)")  "end function n_hel"
  write (u, *)
  write (u, "(2x,A)")  "pure function n_cin () result (n)"
  write (u, "(2x,A)")  "  integer :: n"
  write (u, "(2x,A)")  "  n = 2"
  write (u, "(2x,A)")  "end function n_cin"
  write (u, *)
  write (u, "(2x,A)")  "pure function n_col () result (n)"
  write (u, "(2x,A)")  "  integer :: n"
  write (u, "(2x,A)")  "  n = 1"
  write (u, "(2x,A)")  "end function n_col"
  write (u, *)
  write (u, "(2x,A)")  "pure function n_cf () result (n)"
  write (u, "(2x,A)")  "  integer :: n"
  write (u, "(2x,A)")  "  n = 1"
  write (u, "(2x,A)")  "end function n_cf"
  write (u, *)
  write (u, "(2x,A)")  "pure subroutine flv_state (a)"
  write (u, "(2x,A)")  "  integer, dimension(:,:), intent(out) :: a"
  write (u, "(2x,A)")  "  a = reshape ([1,2,3], [3,1])"
  write (u, "(2x,A)")  "end subroutine flv_state"
  write (u, *)
  write (u, "(2x,A)")  "pure subroutine hel_state (a)"
  write (u, "(2x,A)")  "  integer, dimension(:,:), intent(out) :: a"
  write (u, "(2x,A)")  "  a = reshape ([0,0,0], [3,1])"
  write (u, "(2x,A)")  "end subroutine hel_state"
  write (u, *)
  write (u, "(2x,A)")  "pure subroutine col_state (a, g)"
  write (u, "(2x,A)")  "  integer, dimension(:,:,:), intent(out) :: a"
  write (u, "(2x,A)")  "  logical, dimension(:,:), intent(out) :: g"
  write (u, "(2x,A)")  "  a = reshape ([0,0, 0,0, 0,0], [2,3,1])"
  write (u, "(2x,A)")  "  g = reshape ([.false., .false., .false.], [3,1])"
  write (u, "(2x,A)")  "end subroutine col_state"
  write (u, *)
  write (u, "(2x,A)")  "pure subroutine color_factors (cf)"
  write (u, "(2x,A)")  "  type(OCF), dimension(:), intent(out) :: cf"
  write (u, "(2x,A)")  "  cf = [ OCF(1,1,+1._default) ]"
  write (u, "(2x,A)")  "end subroutine color_factors"
end subroutine write_test_me_code_2
```

Compile test with Fortran bind(C) library

Test 5: Write driver and makefile and try to compile and link the library driver.

There is a single test process with a single feature. The process code is provided as a Fortran library of independent procedures. These procedures are

bind(C).

⟨*Prclib interfaces: execute tests*⟩+≡

```
call test (prclib_interfaces_5, "prclib_interfaces_5", &
     "compile and link (Fortran library)", &
     u, results)
```

⟨*Prclib interfaces: test declarations*⟩+≡

```
public :: prclib_interfaces_5
```

⟨*Prclib interfaces: tests*⟩+≡

```
subroutine prclib_interfaces_5 (u)
  integer, intent(in) :: u
  class(prclib_driver_t), allocatable :: driver
  class(prc_writer_t), pointer :: test_writer_5
  type(os_data_t) :: os_data
  integer :: u_file

  integer, dimension(:,:), allocatable :: flv_state
  integer, dimension(:,:), allocatable :: hel_state
  integer, dimension(:,:,:), allocatable :: col_state
  logical, dimension(:,:), allocatable :: ghost_flag
  integer, dimension(:,:), allocatable :: cf_index
  complex(default), dimension(:), allocatable :: color_factors
  character(32), parameter :: md5sum = "prclib_interfaces_5_md5sum      "

  type(c_funptr) :: proc1_ptr
  interface
     subroutine proc1_t (n) bind(C)
       import
       integer(c_int), intent(out) :: n
     end subroutine proc1_t
  end interface
  procedure(proc1_t), pointer :: proc1
  integer(c_int) :: n

  write (u, "(A)")  "* Test output: prclib_interfaces_5"
  write (u, "(A)")  "*   Purpose: compile, link, and load process library"
  write (u, "(A)")  "*            with (fake) matrix-element code &
       &as a Fortran bind(C) library"
  write (u, *)
  write (u, "(A)")  "* Create a prclib driver object (1 process)"
  write (u, "(A)")

  call os_data_init (os_data)
  allocate (test_writer_5_t :: test_writer_5)

  call dispatch_prclib_driver (driver, var_str ("prclib5"), var_str (""))
  call driver%init (1)
  call driver%set_md5sum (md5sum)

  call driver%set_record (1, var_str ("test5"), var_str ("Test_model"), &
       [var_str ("proc1")], test_writer_5)

  call driver%write (u)
```

```
write (u, *)
write (u, "(A)")  "* Write makefile"
u_file = free_unit ()
open (u_file, file="prclib5.makefile", status="replace", action="write")
call driver%generate_makefile (u_file, os_data)
close (u_file)

write (u, "(A)")  "* Write driver source code"
u_file = free_unit ()
open (u_file, file="prclib5.f90", status="replace", action="write")
call driver%generate_driver_code (u_file)
close (u_file)

write (u, "(A)")  "* Write matrix-element source code"
call driver%make_source (os_data)

write (u, "(A)")  "* Compile source code"
call driver%make_compile (os_data)

write (u, "(A)")  "* Link library"
call driver%make_link (os_data)

write (u, "(A)")  "* Load library"
call driver%load (os_data)

write (u, *)
call driver%write (u)
write (u, *)

if (driver%loaded) then
   write (u, "(A)")  "* Call library functions:"
   write (u, *)
   write (u, "(1x,A,I0)")  "n_processes   = ", driver%get_n_processes ()
   write (u, "(1x,A,A)")  "process_id    = ", &
        char (driver%get_process_id (1))
   write (u, "(1x,A,A)")  "model_name    = ", &
        char (driver%get_model_name (1))
   write (u, "(1x,A,A)")  "md5sum        = ", &
        char (driver%get_md5sum (1))
   write (u, "(1x,A,L1)")  "openmp_status = ", driver%get_openmp_status (1)
   write (u, "(1x,A,I0)")  "n_in  = ", driver%get_n_in (1)
   write (u, "(1x,A,I0)")  "n_out = ", driver%get_n_out (1)
   write (u, "(1x,A,I0)")  "n_flv = ", driver%get_n_flv (1)
   write (u, "(1x,A,I0)")  "n_hel = ", driver%get_n_hel (1)
   write (u, "(1x,A,I0)")  "n_col = ", driver%get_n_col (1)
   write (u, "(1x,A,I0)")  "n_cin = ", driver%get_n_cin (1)
   write (u, "(1x,A,I0)")  "n_cf  = ", driver%get_n_cf (1)

   call driver%set_flv_state (1, flv_state)
   write (u, "(1x,A,10(1x,I0))")  "flv_state =", flv_state

   call driver%set_hel_state (1, hel_state)
   write (u, "(1x,A,10(1x,I0))")  "hel_state =", hel_state
```

```
      call driver%set_col_state (1, col_state, ghost_flag)
      write (u, "(1x,A,10(1x,I0))")  "col_state =", col_state
      write (u, "(1x,A,10(1x,L1))")  "ghost_flag =", ghost_flag

      call driver%set_color_factors (1, color_factors, cf_index)
      write (u, "(1x,A,10(1x,F5.3))")  "color_factors =", color_factors
      write (u, "(1x,A,10(1x,I0))")  "cf_index =", cf_index

      call driver%get_fptr (1, 1, proc1_ptr)
      call c_f_procpointer (proc1_ptr, proc1)
      if (associated (proc1)) then
         write (u, *)
         call proc1 (n)
         write (u, "(1x,A,I0)")  "proc1(1) = ", n
      end if

   end if

   deallocate (test_writer_5)

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: prclib_interfaces_5"
end subroutine prclib_interfaces_5
```

This version of test-code writer writes interfaces for all standard features plus one specific feature. The interfaces are all bind(C), so no wrapper is needed.

⟨*Prclib interfaces: test types*⟩+≡

```
type, extends (prc_writer_c_lib_t) :: test_writer_5_t
 contains
   procedure, nopass :: type_name => test_writer_5_type_name
   procedure :: write_makefile_code => test_writer_5_mk
   procedure :: write_source_code => test_writer_5_src
   procedure :: write_interface => test_writer_5_if
end type test_writer_5_t
```

The

⟨*Prclib interfaces: test auxiliary*⟩+≡

```
function test_writer_5_type_name () result (string)
  type(string_t) :: string
  string = "test_5"
end function test_writer_5_type_name

subroutine test_writer_5_mk (writer, unit, id, os_data, testflag)
  class(test_writer_5_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: testflag
  write (unit, "(5A)")  "SOURCES += ", char (id), ".f90"
  write (unit, "(5A)")  "OBJECTS += ", char (id), ".lo"
  write (unit, "(5A)")  char (id), ".lo: ", char (id), ".f90"
  write (unit, "(5A)")  TAB, "$(LTFCOMPILE) $<"
end subroutine test_writer_5_mk
```

```
subroutine test_writer_5_src (writer, id)
  class(test_writer_5_t), intent(in) :: writer
  type(string_t), intent(in) :: id
  call write_test_f_lib_file (id, var_str ("proc1"))
end subroutine test_writer_5_src

subroutine test_writer_5_if (writer, unit, id, feature)
  class(test_writer_5_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  select case (char (feature))
  case ("proc1")
     write (unit, "(2x,9A)")  "interface"
     write (unit, "(5x,9A)")  "subroutine ", &
          char (writer%get_c_procname (id, feature)), &
          " (n) bind(C)"
     write (unit, "(7x,9A)")  "import"
     write (unit, "(7x,9A)")  "implicit none"
     write (unit, "(7x,9A)")  "integer(c_int), intent(out) :: n"
     write (unit, "(5x,9A)")  "end subroutine ", &
          char (writer%get_c_procname (id, feature))
     write (unit, "(2x,9A)")  "end interface"
  case default
     call writer%write_standard_interface (unit, id, feature)
  end select
end subroutine test_writer_5_if
```

We need a test module file (actually, one for each process in the test above) that allows us to check compilation and linking. The test module implements a colorless $1 \to 2$ process, and it implements one additional function (feature), the name given as an argument.

⟨*Prclib interfaces: test auxiliary*⟩+≡

```
subroutine write_test_f_lib_file (basename, feature)
  type(string_t), intent(in) :: basename
  type(string_t), intent(in) :: feature
  integer :: u
  u = free_unit ()
  open (u, file = char (basename) // ".f90", &
       status = "replace", action = "write")
  write (u, "(A)")  "! (Pseudo) matrix element code file &
       &for WHIZARD self-test"
  call write_test_me_code_3 (u, char (basename))
  write (u, *)
  write (u, "(A)")  "subroutine " // char (basename) // "_" &
       // char (feature) // " (n) bind(C)"
  write (u, "(A)")  "  use iso_c_binding"
  write (u, "(A)")  "  implicit none"
  write (u, "(A)")  "  integer(c_int), intent(out) :: n"
  write (u, "(A)")  "  n = 42"
  write (u, "(A)")  "end subroutine " // char (basename) // "_" &
       // char (feature)
  close (u)
```

```
end subroutine write_test_f_lib_file
```

The following matrix-element source code is identical to the previous one, but modified such as to provide independent procedures without a module envelope.

⟨*Prclib interfaces: test auxiliary*⟩+≡

```
subroutine write_test_me_code_3 (u, id)
  integer, intent(in) :: u
  character(*), intent(in) :: id
  write (u, "(A)")  "function " // id // "_get_md5sum () &
       &result (cptr) bind(C)"
  write (u, "(A)")  "  use iso_c_binding"
  write (u, "(A)")  "  implicit none"
  write (u, "(A)")  "  type(c_ptr) :: cptr"
  write (u, "(A)")  "  character(c_char), dimension(32), &
       &target, save :: md5sum"
  write (u, "(A)")  "  md5sum = copy (c_char_&
       &'1234567890abcdef1234567890abcdef')"
  write (u, "(A)")  "  cptr = c_loc (md5sum)"
  write (u, "(A)")  "contains"
  write (u, "(A)")  "  function copy (md5sum)"
  write (u, "(A)")  "    character(c_char), dimension(32) :: copy"
  write (u, "(A)")  "    character(c_char), dimension(32), intent(in) :: &
       &md5sum"
  write (u, "(A)")  "    copy = md5sum"
  write (u, "(A)")  "  end function copy"
  write (u, "(A)")  "end function " // id // "_get_md5sum"
  write (u, *)
  write (u, "(A)")  "function " // id // "_openmp_supported () &
       &result (status) bind(C)"
  write (u, "(A)")  "  use iso_c_binding"
  write (u, "(A)")  "  implicit none"
  write (u, "(A)")  "  logical(c_bool) :: status"
  write (u, "(A)")  "  status = .false."
  write (u, "(A)")  "end function " // id // "_openmp_supported"
  write (u, *)
  write (u, "(A)")  "function " // id // "_n_in () result (n) bind(C)"
  write (u, "(A)")  "  use iso_c_binding"
  write (u, "(A)")  "  implicit none"
  write (u, "(A)")  "  integer(c_int) :: n"
  write (u, "(A)")  "  n = 1"
  write (u, "(A)")  "end function " // id // "_n_in"
  write (u, *)
  write (u, "(A)")  "function " // id // "_n_out () result (n) bind(C)"
  write (u, "(A)")  "  use iso_c_binding"
  write (u, "(A)")  "  implicit none"
  write (u, "(A)")  "  integer(c_int) :: n"
  write (u, "(A)")  "  n = 2"
  write (u, "(A)")  "end function " // id // "_n_out"
  write (u, *)
  write (u, "(A)")  "function " // id // "_n_flv () result (n) bind(C)"
  write (u, "(A)")  "  use iso_c_binding"
  write (u, "(A)")  "  implicit none"
  write (u, "(A)")  "  integer(c_int) :: n"
```

```
write (u, "(A)")  "  n = 1"
write (u, "(A)")  "end function " // id // "_n_flv"
write (u, *)
write (u, "(A)")  "function " // id // "_n_hel () result (n) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int) :: n"
write (u, "(A)")  "  n = 1"
write (u, "(A)")  "end function " // id // "_n_hel"
write (u, *)
write (u, "(A)")  "function " // id // "_n_cin () result (n) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int) :: n"
write (u, "(A)")  "  n = 2"
write (u, "(A)")  "end function " // id // "_n_cin"
write (u, *)
write (u, "(A)")  "function " // id // "_n_col () result (n) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int) :: n"
write (u, "(A)")  "  n = 1"
write (u, "(A)")  "end function " // id // "_n_col"
write (u, *)
write (u, "(A)")  "function " // id // "_n_cf () result (n) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int) :: n"
write (u, "(A)")  "  n = 1"
write (u, "(A)")  "end function " // id // "_n_cf"
write (u, *)
write (u, "(A)")  "subroutine " // id // "_flv_state (flv_state) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int), dimension(*), intent(out) :: flv_state"
write (u, "(A)")  "  flv_state(1:3) = [1,2,3]"
write (u, "(A)")  "end subroutine " // id // "_flv_state"
write (u, *)
write (u, "(A)")  "subroutine " // id // "_hel_state (hel_state) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int), dimension(*), intent(out) :: hel_state"
write (u, "(A)")  "  hel_state(1:3) = [0,0,0]"
write (u, "(A)")  "end subroutine " // id // "_hel_state"
write (u, *)
write (u, "(A)")  "subroutine " // id // "_col_state &
     &(col_state, ghost_flag) bind(C)"
write (u, "(A)")  "  use iso_c_binding"
write (u, "(A)")  "  implicit none"
write (u, "(A)")  "  integer(c_int), dimension(*), intent(out) &
     &:: col_state"
write (u, "(A)")  "  logical(c_bool), dimension(*), intent(out) &
     &:: ghost_flag"
write (u, "(A)")  "  col_state(1:6) = [0,0, 0,0, 0,0]"
```

```
    write (u, "(A)")  "  ghost_flag(1:3) = [.false., .false., .false.]"
    write (u, "(A)")  "end subroutine " // id // "_col_state"
    write (u, *)
    write (u, "(A)")  "subroutine " // id // "_color_factors &
         &(cf_index1, cf_index2, color_factors) bind(C)"
    write (u, "(A)")  "  use iso_c_binding"
    write (u, "(A)")  "  use kinds"
    write (u, "(A)")  "  implicit none"
    write (u, "(A)")  "  integer(c_int), dimension(*), intent(out) :: cf_index1"
    write (u, "(A)")  "  integer(c_int), dimension(*), intent(out) :: cf_index2"
    write (u, "(A)")  "  complex(c_default_complex), dimension(*), &
         &intent(out) :: color_factors"
    write (u, "(A)")  "  cf_index1(1:1) = [1]"
    write (u, "(A)")  "  cf_index2(1:1) = [1]"
    write (u, "(A)")  "  color_factors(1:1) = [1]"
    write (u, "(A)")  "end subroutine " // id // "_color_factors"
  end subroutine write_test_me_code_3
```

Compile test with genuine C library

Test 6: Write driver and makefile and try to compile and link the library driver.

There is a single test process with a single feature. The process code is provided as a C library of independent procedures. These procedures should match the Fortran bind(C) interface.

⟨*Prclib interfaces: execute tests*⟩+≡

```
  if (default == double .or. (CC_IS_GNU .and. CC_HAS_QUADMATH)) then
     call test (prclib_interfaces_6, "prclib_interfaces_6", &
          "compile and link (C library)", &
          u, results)
  end if
```

⟨*Prclib interfaces: test declarations*⟩+≡

```
  public :: prclib_interfaces_6
```

⟨*Prclib interfaces: tests*⟩+≡

```
  subroutine prclib_interfaces_6 (u)
    integer, intent(in) :: u
    class(prclib_driver_t), allocatable :: driver
    class(prc_writer_t), pointer :: test_writer_6
    type(os_data_t) :: os_data
    integer :: u_file

    integer, dimension(:,:), allocatable :: flv_state
    integer, dimension(:,:), allocatable :: hel_state
    integer, dimension(:,:,:), allocatable :: col_state
    logical, dimension(:,:), allocatable :: ghost_flag
    integer, dimension(:,:), allocatable :: cf_index
    complex(default), dimension(:), allocatable :: color_factors
    character(32), parameter :: md5sum = "prclib_interfaces_6_md5sum      "

    type(c_funptr) :: proc1_ptr
    interface
       subroutine proc1_t (n) bind(C)
```

```
     import
     integer(c_int), intent(out) :: n
   end subroutine proc1_t
end interface
procedure(proc1_t), pointer :: proc1
integer(c_int) :: n

write (u, "(A)")  "* Test output: prclib_interfaces_6"
write (u, "(A)")  "*   Purpose: compile, link, and load process library"
write (u, "(A)")  "*            with (fake) matrix-element code &
     &as a C library"
write (u, *)
write (u, "(A)")  "* Create a prclib driver object (1 process)"
write (u, "(A)")

call os_data_init (os_data)
allocate (test_writer_6_t :: test_writer_6)

call dispatch_prclib_driver (driver, var_str ("prclib6"), var_str (""))
call driver%init (1)
call driver%set_md5sum (md5sum)

call driver%set_record (1, var_str ("test6"), var_str ("Test_model"), &
     [var_str ("proc1")], test_writer_6)

call driver%write (u)

write (u, *)
write (u, "(A)")  "* Write makefile"
u_file = free_unit ()
open (u_file, file="prclib6.makefile", status="replace", action="write")
call driver%generate_makefile (u_file, os_data)
close (u_file)

write (u, "(A)")  "* Write driver source code"
u_file = free_unit ()
open (u_file, file="prclib6.f90", status="replace", action="write")
call driver%generate_driver_code (u_file)
close (u_file)

write (u, "(A)")  "* Write matrix-element source code"
call driver%make_source (os_data)

write (u, "(A)")  "* Compile source code"
call driver%make_compile (os_data)

write (u, "(A)")  "* Link library"
call driver%make_link (os_data)

write (u, "(A)")  "* Load library"
call driver%load (os_data)

write (u, *)
call driver%write (u)
```

```
  write (u, *)

  if (driver%loaded) then
     write (u, "(A)")  "* Call library functions:"
     write (u, *)
     write (u, "(1x,A,I0)")  "n_processes   = ", driver%get_n_processes ()
     write (u, "(1x,A,A)")  "process_id    = ", &
          char (driver%get_process_id (1))
     write (u, "(1x,A,A)")  "model_name    = ", &
          char (driver%get_model_name (1))
     write (u, "(1x,A,A)")  "md5sum        = ", &
          char (driver%get_md5sum (1))
     write (u, "(1x,A,L1)")  "openmp_status = ", driver%get_openmp_status (1)
     write (u, "(1x,A,I0)")  "n_in  = ", driver%get_n_in (1)
     write (u, "(1x,A,I0)")  "n_out = ", driver%get_n_out (1)
     write (u, "(1x,A,I0)")  "n_flv = ", driver%get_n_flv (1)
     write (u, "(1x,A,I0)")  "n_hel = ", driver%get_n_hel (1)
     write (u, "(1x,A,I0)")  "n_col = ", driver%get_n_col (1)
     write (u, "(1x,A,I0)")  "n_cin = ", driver%get_n_cin (1)
     write (u, "(1x,A,I0)")  "n_cf  = ", driver%get_n_cf (1)

     call driver%set_flv_state (1, flv_state)
     write (u, "(1x,A,10(1x,I0))")  "flv_state =", flv_state

     call driver%set_hel_state (1, hel_state)
     write (u, "(1x,A,10(1x,I0))")  "hel_state =", hel_state

     call driver%set_col_state (1, col_state, ghost_flag)
     write (u, "(1x,A,10(1x,I0))")  "col_state =", col_state
     write (u, "(1x,A,10(1x,L1))")  "ghost_flag =", ghost_flag

     call driver%set_color_factors (1, color_factors, cf_index)
     write (u, "(1x,A,10(1x,F5.3))")  "color_factors =", color_factors
     write (u, "(1x,A,10(1x,I0))")  "cf_index =", cf_index

     call driver%get_fptr (1, 1, proc1_ptr)
     call c_f_procpointer (proc1_ptr, proc1)
     if (associated (proc1)) then
        write (u, *)
        call proc1 (n)
        write (u, "(1x,A,I0)")  "proc1(1) = ", n
     end if

  end if

  deallocate (test_writer_6)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: prclib_interfaces_6"
end subroutine prclib_interfaces_6
```

This version of test-code writer writes interfaces for all standard features plus one specific feature. The interfaces are all bind(C), so no wrapper is needed.

The driver part is identical to the Fortran case, so we simply extend the previous test_writer_5 type. We only have to override the Makefile writer.

⟨*Prclib interfaces: test types*⟩+≡

```
type, extends (test_writer_5_t) :: test_writer_6_t
 contains
   procedure, nopass :: type_name => test_writer_6_type_name
   procedure :: write_makefile_code => test_writer_6_mk
   procedure :: write_source_code => test_writer_6_src
end type test_writer_6_t
```

⟨*Prclib interfaces: test auxiliary*⟩+≡

```
function test_writer_6_type_name () result (string)
  type(string_t) :: string
  string = "test_6"
end function test_writer_6_type_name

subroutine test_writer_6_mk (writer, unit, id, os_data, testflag)
  class(test_writer_6_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: testflag
  write (unit, "(5A)")  "SOURCES += ", char (id), ".c"
  write (unit, "(5A)")  "OBJECTS += ", char (id), ".lo"
  write (unit, "(5A)")  char (id), ".lo: ", char (id), ".c"
  write (unit, "(5A)")  TAB, "$(LTCCOMPILE) $<"
end subroutine test_writer_6_mk

subroutine test_writer_6_src (writer, id)
  class(test_writer_6_t), intent(in) :: writer
  type(string_t), intent(in) :: id
  call write_test_c_lib_file (id, var_str ("proc1"))
end subroutine test_writer_6_src
```

We need a test module file (actually, one for each process in the test above) that allows us to check compilation and linking. The test module implements a colorless $1 \to 2$ process, and it implements one additional function (feature), the name given as an argument.

⟨*Prclib interfaces: test auxiliary*⟩+≡

```
subroutine write_test_c_lib_file (basename, feature)
  type(string_t), intent(in) :: basename
  type(string_t), intent(in) :: feature
  integer :: u
  u = free_unit ()
  open (u, file = char (basename) // ".c", &
       status = "replace", action = "write")
  write (u, "(A)")  "/* (Pseudo) matrix element code file &
       &for WHIZARD self-test */"
  write (u, "(A)")  "#include <stdbool.h>"
  if (CC_HAS_QUADMATH) then
     write (u, "(A)")  "#include <quadmath.h>"
  end if
```

```
  write (u, *)
  call write_test_me_code_4 (u, char (basename))
  write (u, *)
  write (u, "(A)")  "void " // char (basename) // "_" &
       // char (feature) // "(int* n) {"
  write (u, "(A)")  "  *n = 42;"
  write (u, "(A)")  "}"
  close (u)
end subroutine write_test_c_lib_file
```

The following matrix-element source code is equivalent to the code in the previous example, but coded in C.

⟨*Prclib interfaces: test auxiliary*⟩+≡

```
subroutine write_test_me_code_4 (u, id)
  integer, intent(in) :: u
  character(*), intent(in) :: id
  write (u, "(A)")  "char* " // id // "_get_md5sum() {"
  write (u, "(A)")  "  return ""1234567890abcdef1234567890abcdef"";"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "bool " // id // "_openmp_supported() {"
  write (u, "(A)")  "  return false;"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "int " // id // "_n_in() {"
  write (u, "(A)")  "  return 1;"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "int " // id // "_n_out() {"
  write (u, "(A)")  "  return 2;"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "int " // id // "_n_flv() {"
  write (u, "(A)")  "  return 1;"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "int " // id // "_n_hel() {"
  write (u, "(A)")  "  return 1;"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "int " // id // "_n_cin() {"
  write (u, "(A)")  "  return 2;"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "int " // id // "_n_col() {"
  write (u, "(A)")  "  return 1;"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "int " // id // "_n_cf() {"
  write (u, "(A)")  "  return 1;"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "void " // id // "_flv_state( int (*a)[] ) {"
```

```
  write (u, "(A)")  "  static int flv_state[1][3] =  { { 1, 2, 3 } };"
  write (u, "(A)")  "  int j;"
  write (u, "(A)")  "  for (j = 0; j < 3; j++) { (*a)[j] &
       &= flv_state[0][j]; }"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "void " // id // "_hel_state( int (*a)[] ) {"
  write (u, "(A)")  "  static int hel_state[1][3] =  { { 0, 0, 0 } };"
  write (u, "(A)")  "  int j;"
  write (u, "(A)")  "  for (j = 0; j < 3; j++) { (*a)[j] &
       &= hel_state[0][j]; }"
  write (u, "(A)")  "}"
  write (u, *)
  write (u, "(A)")  "void " // id // "_col_state&
       &( int (*a)[], bool (*g)[] ) {"
  write (u, "(A)")  "  static int col_state[1][3][2] = &
       &{ { {0, 0}, {0, 0}, {0, 0} } };"
  write (u, "(A)")  "  static bool ghost_flag[1][3] =  &
       &{ { false, false, false } };"
  write (u, "(A)")  "  int j,k;"
  write (u, "(A)")  "  for (j = 0; j < 3; j++) {"
  write (u, "(A)")  "    for (k = 0; k < 2; k++) {"
  write (u, "(A)")  "       (*a)[j*2+k] = col_state[0][j][k];"
  write (u, "(A)")  "    }"
  write (u, "(A)")  "    (*g)[j] = ghost_flag[0][j];"
  write (u, "(A)")  "  }"
  write (u, "(A)")  "}"
  write (u, *)
  select case (DEFAULT_FC_PRECISION)
  case ("quadruple")
     write (u, "(A)")  "void " // id // "_color_factors&
          &( int (*cf_index1)[], int (*cf_index2)[], &
          &__complex128 (*color_factors)[] ) {"
  case ("extended")
     write (u, "(A)")  "void " // id // "_color_factors&
          &( int (*cf_index1)[], int (*cf_index2)[], &
          &long double _Complex (*color_factors)[] ) {"
  case default
     write (u, "(A)")  "void " // id // "_color_factors&
          &( int (*cf_index1)[], int (*cf_index2)[], &
          &double _Complex (*color_factors)[] ) {"
  end select
  write (u, "(A)")  "  (*color_factors)[0] = 1;"
  write (u, "(A)")  "  (*cf_index1)[0] = 1;"
  write (u, "(A)")  "  (*cf_index2)[0] = 1;"
  write (u, "(A)")  "}"
end subroutine write_test_me_code_4
```

Test cleanup targets

Test 7: Repeat test 4 (create, compile, link Fortran module and driver) and properly clean up all generated files.

⟨*Prclib interfaces: execute tests*⟩+≡

```
    call test (prclib_interfaces_7, "prclib_interfaces_7", &
         "cleanup", &
         u, results)
```

⟨*Prclib interfaces: test declarations*⟩+≡

```
    public :: prclib_interfaces_7
```

⟨*Prclib interfaces: tests*⟩+≡

```
    subroutine prclib_interfaces_7 (u)
      integer, intent(in) :: u
      class(prclib_driver_t), allocatable :: driver
      class(prc_writer_t), pointer :: test_writer_4
      type(os_data_t) :: os_data
      integer :: u_file
      character(32), parameter :: md5sum = "1234567890abcdef1234567890abcdef"

      write (u, "(A)")  "* Test output: prclib_interfaces_7"
      write (u, "(A)")  "*   Purpose: compile and link process library"
      write (u, "(A)")  "*            with (fake) matrix-element code &
           &as a Fortran module"
      write (u, "(A)")  "*            then clean up generated files"
      write (u, *)
      write (u, "(A)")  "* Create a prclib driver object (1 process)"

      allocate (test_writer_4_t :: test_writer_4)

      call os_data_init (os_data)
      call dispatch_prclib_driver (driver, var_str ("prclib7"), var_str (""))
      call driver%init (1)
      call driver%set_md5sum (md5sum)
      call driver%set_record (1, var_str ("test7"), var_str ("Test_model"), &
           [var_str ("proc1")], test_writer_4)

      write (u, "(A)")  "* Write makefile"
      u_file = free_unit ()
      open (u_file, file="prclib7.makefile", status="replace", action="write")
      call driver%generate_makefile (u_file, os_data)
      close (u_file)

      write (u, "(A)")  "* Write driver source code"
      u_file = free_unit ()
      open (u_file, file="prclib7.f90", status="replace", action="write")
      call driver%generate_driver_code (u_file)
      close (u_file)

      write (u, "(A)")  "* Write matrix-element source code"
      call driver%make_source (os_data)

      write (u, "(A)")  "* Compile source code"
      call driver%make_compile (os_data)

      write (u, "(A)")  "* Link library"
      call driver%make_link (os_data)
```

```
    write (u, "(A)")  "* File check"
    write (u, *)
    call check_file (u, "test7.f90")
    call check_file (u, "tpr_test7.mod")
    call check_file (u, "test7.lo")
    call check_file (u, "prclib7.makefile")
    call check_file (u, "prclib7.f90")
    call check_file (u, "prclib7.lo")
    call check_file (u, "prclib7.la")

    write (u, *)
    write (u, "(A)")  "* Delete library"
    write (u, *)
    call driver%clean_library (os_data)
    call check_file (u, "prclib7.la")

    write (u, *)
    write (u, "(A)")  "* Delete object code"
    write (u, *)
    call driver%clean_objects (os_data)
    call check_file (u, "test7.lo")
    call check_file (u, "tpr_test7.mod")
    call check_file (u, "prclib7.lo")

    write (u, *)
    write (u, "(A)")  "* Delete source code"
    write (u, *)
    call driver%clean_source (os_data)
    call check_file (u, "test7.f90")

    write (u, *)
    write (u, "(A)")  "* Delete driver source code"
    write (u, *)
    call driver%clean_driver (os_data)
    call check_file (u, "prclib7.f90")

    write (u, *)
    write (u, "(A)")  "* Delete makefile"
    write (u, *)
    call driver%clean_makefile (os_data)
    call check_file (u, "prclib7.makefile")

    deallocate (test_writer_4)

    write (u, *)
    write (u, "(A)")  "* Test output end: prclib_interfaces_7"
  end subroutine prclib_interfaces_7
```

Auxiliary routine: check and report existence of a file

⟨*Prclib interfaces: test auxiliary*⟩+≡

```
  subroutine check_file (u, file)
    integer, intent(in) :: u
    character(*), intent(in) :: file
```

```
    logical :: exist
    inquire (file=file, exist=exist)
    write (u, "(2x,A,A,L1)")  file, " = ", exist
  end subroutine check_file
```

13.3 Abstract process core configuration

In this module, we define abstract data types that handle the method-specific part of defining a process (including all of its options) and accessing an external matrix element.

There are no unit tests, these are deferred to the `process_libraries` module below.

⟨`prc_core_def.f90`⟩≡

```
⟨File header⟩

module prc_core_def

⟨Use strings⟩
  use io_units
  use diagnostics

  use process_constants
  use prclib_interfaces

⟨Standard module head⟩

⟨Prc core def: public⟩

⟨Prc core def: types⟩

⟨Prc core def: interfaces⟩

contains

⟨Prc core def: procedures⟩

end module prc_core_def
```

13.3.1 Process core definition type

For storing configuration data that depend on the specific process variant, we introduce a polymorphic type. At this point, we just declare an abstract base type. This allows us to defer the implementation to later modules.

There should be no components that need explicit finalization, otherwise we would have to call a finalizer from the `process_component_def_t` wrapper.

Translate a `prc_core_def_t` to above named integers

⟨*Prc core def: public*⟩≡

```
  public :: prc_core_def_t
```

⟨*Prc core def: types*⟩≡
```
type, abstract :: prc_core_def_t
   class(prc_writer_t), allocatable :: writer
 contains
 ⟨Prc core def: process core def: TBP⟩
end type prc_core_def_t
```

Interfaces for the deferred methods.

This returns a string. No passed argument; the string is constant and depends just on the type.

⟨*Prc core def: process core def: TBP*⟩≡
```
procedure (prc_core_def_get_string), nopass, deferred :: type_string
```

⟨*Prc core def: interfaces*⟩≡
```
abstract interface
   function prc_core_def_get_string () result (string)
     import
     type(string_t) :: string
   end function prc_core_def_get_string
end interface
```

The `write` method should display the content completely.

⟨*Prc core def: process core def: TBP*⟩+≡
```
procedure (prc_core_def_write), deferred :: write
```

⟨*Prc core def: interfaces*⟩+≡
```
abstract interface
   subroutine prc_core_def_write (object, unit)
     import
     class(prc_core_def_t), intent(in) :: object
     integer, intent(in) :: unit
   end subroutine prc_core_def_write
end interface
```

The `read` method should fill the content completely.

⟨*Prc core def: process core def: TBP*⟩+≡
```
procedure (prc_core_def_read), deferred :: read
```

⟨*Prc core def: interfaces*⟩+≡
```
abstract interface
   subroutine prc_core_def_read (object, unit)
     import
     class(prc_core_def_t), intent(out) :: object
     integer, intent(in) :: unit
   end subroutine prc_core_def_read
end interface
```

This communicates a MD5 checksum to the writer inside the `core_def` object, if there is any. Usually, this checksum is not yet known at the time when the writer is initialized.

⟨*Prc core def: process core def: TBP*⟩+≡
```
procedure :: set_md5sum => prc_core_def_set_md5sum
```

⟨*Prc core def: procedures*⟩≡

```
subroutine prc_core_def_set_md5sum (core_def, md5sum)
  class(prc_core_def_t), intent(inout) :: core_def
  character(32) :: md5sum
  if (allocated (core_def%writer))  core_def%writer%md5sum = md5sum
end subroutine prc_core_def_set_md5sum
```

Allocate an appropriate driver object which corresponds to the chosen process core definition.

For internal matrix element (i.e., those which do not need external code), the driver should have access to all matrix element information from the beginning. In short, it is the matrix-element code.

For external matrix elements, the driver will get access to the external matrix element code.

⟨*Prc core def: process core def: TBP*⟩+≡

```
procedure(prc_core_def_allocate_driver), deferred :: allocate_driver
```

⟨*Prc core def: interfaces*⟩+≡

```
abstract interface
   subroutine prc_core_def_allocate_driver (object, driver, basename)
     import
     class(prc_core_def_t), intent(in) :: object
     class(prc_core_driver_t), intent(out), allocatable :: driver
     type(string_t), intent(in) :: basename
   end subroutine prc_core_def_allocate_driver
end interface
```

This flag tells whether the particular variant needs external code. We implement a default function which returns false. The flag depends only on the type, therefore we implement it as `nopass`.

⟨*Prc core def: process core def: TBP*⟩+≡

```
procedure, nopass :: needs_code => prc_core_def_needs_code
```

⟨*Prc core def: procedures*⟩+≡

```
function prc_core_def_needs_code () result (flag)
  logical :: flag
  flag = .false.
end function prc_core_def_needs_code
```

This subroutine allocates an array which holds the name of all features that this process core implements. This feature applies to matrix element code that is not coded as a Fortran module but communicates via independent library functions, which follow the C calling conventions. The addresses of those functions are returned as C function pointers, which can be converted into Fortran procedure pointers. The conversion is done in code specific for the process variant; here we just retrieve the C function pointer.

The array returned here serves the purpose of writing specific driver code. The driver interfaces only those C functions which are supported for the given process core.

If the process core does not require external code, this array is meaningless.

⟨*Prc core def: process core def: TBP*⟩+≡

```
procedure(prc_core_def_get_features), nopass, deferred &
     :: get_features
```

⟨*Prc core def: interfaces*⟩+≡

```
abstract interface
   subroutine prc_core_def_get_features (features)
     import
     type(string_t), dimension(:), allocatable, intent(out) :: features
   end subroutine prc_core_def_get_features
end interface
```

Assign pointers to the process-specific procedures to the driver, if the process is external.

⟨*Prc core def: process core def: TBP*⟩+≡

```
procedure(prc_core_def_connect), deferred :: connect
```

⟨*Prc core def: interfaces*⟩+≡

```
abstract interface
   subroutine prc_core_def_connect (def, lib_driver, i, proc_driver)
     import
     class(prc_core_def_t), intent(in) :: def
     class(prclib_driver_t), intent(in) :: lib_driver
     integer, intent(in) :: i
     class(prc_core_driver_t), intent(inout) :: proc_driver
   end subroutine prc_core_def_connect
end interface
```

13.3.2 Process core template

We must be able to automatically allocate a process core definition object with the appropriate type, given only the type name.

To this end, we introduce a `prc_template_t` type which is simply a wrapper for an empty `prc_core_def_t` object. Choosing one of the templates from an array, we can allocate the target object.

⟨*Prc core def: public*⟩+≡

```
public :: prc_template_t
```

⟨*Prc core def: types*⟩+≡

```
type :: prc_template_t
   class(prc_core_def_t), allocatable :: core_def
end type prc_template_t
```

The allocation routine. We use the `source` option of the `allocate` statement. The `mold` option would probably more appropriate, but is a F2008 feature.

⟨*Prc core def: public*⟩+≡

```
public :: allocate_core_def
```

⟨*Prc core def: procedures*⟩+≡

```
subroutine allocate_core_def (template, name, core_def)
  type(prc_template_t), dimension(:), intent(in) :: template
  type(string_t), intent(in) :: name
  class(prc_core_def_t), allocatable :: core_def
```

```
  integer :: i
  do i = 1, size (template)
     if (template(i)%core_def%type_string () == name) then
        allocate (core_def, source = template(i)%core_def)
        return
     end if
  end do
end subroutine allocate_core_def
```

13.3.3 Process driver

For each process component, we implement a driver object which holds the calls to the matrix element and various auxiliary routines as procedure pointers. Any actual calculation will use this object to communicate with the process.

Depending on the type of process (as described by a corresponding `prc_core_def` object), the procedure pointers may refer to external or internal code, and there may be additional procedures for certain types. The base type defined here is abstract.

⟨*Prc core def: public*⟩+≡

```
public :: prc_core_driver_t
```

⟨*Prc core def: types*⟩+≡

```
type, abstract :: prc_core_driver_t
 contains
 ⟨Prc core def: process driver: TBP⟩
end type prc_core_driver_t
```

This returns the process type. No reference to contents.

⟨*Prc core def: process driver: TBP*⟩≡

```
procedure(prc_core_driver_type_name), nopass, deferred :: type_name
```

⟨*Prc core def: interfaces*⟩+≡

```
abstract interface
   function prc_core_driver_type_name () result (type)
     import
     type(string_t) :: type
   end function prc_core_driver_type_name
end interface
```

13.3.4 Process driver for intrinsic process

This is an abstract extension for the driver type. It has one additional method, namely a subroutine that fills the record of constant process data. For an external process, this task is performed by the external library driver instead.

⟨*Prc core def: public*⟩+≡

```
public :: process_driver_internal_t
```

⟨*Prc core def: types*⟩+≡

```
type, extends (prc_core_driver_t), abstract :: process_driver_internal_t
 contains
 ⟨Prc core def: process driver internal: TBP⟩
end type process_driver_internal_t
```

⟨*Prc core def: process driver internal: TBP*⟩≡

```
procedure(process_driver_fill_constants), deferred :: fill_constants
```

⟨*Prc core def: interfaces*⟩+≡

```
abstract interface
   subroutine process_driver_fill_constants (driver, data)
     import
     class(process_driver_internal_t), intent(in) :: driver
     type(process_constants_t), intent(out) :: data
   end subroutine process_driver_fill_constants
end interface
```

13.3.5 Process driver for user-defined processes

This is another abstract extension of the driver type. It links user-defined matrix element methods like BLHA, which also allow the dircet calculation of squared matrix elements.

⟨*Prc core def: public*⟩+≡

```
public :: prc_user_defined_base_driver_t
```

⟨*Prc core def: types*⟩+≡

```
type, abstract, extends (prc_core_driver_t) :: prc_user_defined_base_driver_t
end type prc_user_defined_base_driver_t
```

13.4 Process library access

Processes (the code and data that are necessary for evaluating matrix elements of a particular process or process component) are organized in process libraries. In full form, process libraries contain generated and dynamically compiled and linked code, so they are actual libraries on the OS level. Alternatively, there may be simple processes that can be generated without referring to external libraries, and external libraries that are just linked in.

This module interfaces the OS to create, build, and use process libraries.

We work with two related data structures. There is the list of process configurations that stores the user input and data derived from it. A given process configuration list is scanned for creating a process library, which consists of both data and code. The creation step involves calling external programs and incorporating external code.

For the subsequent integration and event generation steps, we read the process library. We also support partial (re)creation of the process library. To this end, we should be able to reconstruct the configuration data records from the process library.

⟨`process_libraries.f90`⟩≡

⟨*File header*⟩

```
module process_libraries

  use, intrinsic :: iso_c_binding !NODEP!
```

⟨*Use strings*⟩

```
  use io_units
  use diagnostics
  use md5
  use physics_defs
  use os_interface
  use model_data
  use particle_specifiers
  use process_constants
  use prclib_interfaces
  use prc_core_def
```

⟨*Standard module head*⟩

⟨*Process libraries: public*⟩

⟨*Process libraries: parameters*⟩

⟨*Process libraries: types*⟩

```
contains
```

⟨*Process libraries: procedures*⟩

```
end module process_libraries
```

13.4.1 Auxiliary stuff

Here is a small subroutine that strips the left-hand side and the equals sign off an equation.

⟨*Process libraries: public*⟩≡

```
    public :: strip_equation_lhs
```

⟨*Process libraries: procedures*⟩≡

```
    subroutine strip_equation_lhs (buffer)
      character(*), intent(inout) :: buffer
      type(string_t) :: string, prefix
      string = buffer
      call split (string, prefix, "=")
      buffer = string
    end subroutine strip_equation_lhs
```

13.4.2 Process definition objects

We collect process configuration data in a derived type, `process_def_t`. A process can be a collection of several components which are treated as a sin-

gle entity for the purpose of observables and event generation. Multiple process components may initially be defined by the user. The system may add additional components, e.g., subtraction terms. The common data type is `process_component_def_t`. Within each component, there are several universal data items, and a part which depend on the particular process variant. The latter is covered by an abstract type `prc_core_def_t` and its extensions.

Wrapper for components

We define a wrapper type for the configuration of individual components.

The string `basename` is used for building file, module, and function names for the current process component. Initially, it will be built from the corresponding process basename by appending an alphanumeric suffix.

The logical `initial` tells whether this is a user-defined (true) or system-generated (false) configuration.

The numbers `n_in`, `n_out`, and `n_tot` denote the incoming, outgoing and total number of particles (partons) participating in the process component, respectively. These are the nominal particles, as input by the user (recombination may change the particle content, for the output events).

The string arrays `prt_in` and `prt_out` hold the particle specifications as provided by the user. For a system-generated process component, they remain deallocated.

The `method` string is used to determine the type of process matrix element and how it is obtained.

The `description` string collects the information about particle content and method in a single human-readable string.

The pointer object `core_def` is allocated according to the actual process variant, which depends on the method. The subobject holds any additional configuration data that is relevant for the process component.

We assume that no finalizer is needed.

⟨*Process libraries: public*⟩+≡
```
public :: process_component_def_t
```

⟨*Process libraries: types*⟩≡
```
type :: process_component_def_t
   private
   type(string_t) :: basename
   logical :: initial = .false.
   integer :: n_in = 0
   integer :: n_out = 0
   integer :: n_tot = 0
   type(prt_spec_t), dimension(:), allocatable :: prt_in
   type(prt_spec_t), dimension(:), allocatable :: prt_out
   type(string_t) :: method
   type(string_t) :: description
   class(prc_core_def_t), allocatable :: core_def
   character(32) :: md5sum = ""
   integer :: nlo_type = BORN
   integer, dimension(N_ASSOCIATED_COMPONENTS) :: associated_components = 0
   logical :: active
   integer :: fixed_emitter = -1
   integer :: alpha_power = 0
```

```
    integer :: alphas_power = 0
  contains
  ⟨Process libraries: process component def: TBP⟩
end type process_component_def_t
```

Display the complete content.

⟨*Process libraries: process component def: TBP*⟩≡

```
procedure :: write => process_component_def_write
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_component_def_write (object, unit)
  class(process_component_def_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(3x,A,A)")  "Component ID        = ", char (object%basename)
  write (u, "(3x,A,L1)") "Initial component   = ", object%initial
  write (u, "(3x,A,I0,1x,I0,1x,I0)") "N (in, out, tot)    = ", &
       object%n_in, object%n_out, object%n_tot
  write (u, "(3x,A)", advance="no") "Particle content    = "
  if (allocated (object%prt_in)) then
     call prt_spec_write (object%prt_in, u, advance="no")
  else
     write (u, "(A)", advance="no")  "[undefined]"
  end if
  write (u, "(A)", advance="no") " => "
  if (allocated (object%prt_out)) then
     call prt_spec_write (object%prt_out, u, advance="no")
  else
     write (u, "(A)", advance="no")  "[undefined]"
  end if
  write (u, "(A)")
  if (object%method /= "") then
     write (u, "(3x,A,A)")  "Method              = ", &
          char (object%method)
  else
     write (u, "(3x,A)")  "Method              = [undefined]"
  end if
  if (allocated (object%core_def)) then
     write (u, "(3x,A,A)")  "Process variant     = ", &
          char (object%core_def%type_string ())
     call object%core_def%write (u)
  else
     write (u, "(3x,A)")  "Process variant     = [undefined]"
  end if
  write (u, "(3x,A,A,A)") "MD5 sum (def)       = '", object%md5sum, "'"
end subroutine process_component_def_write
```

Read the process component definition. Allocate the process variant definition with appropriate type, matching the type name on file with the provided templates.

⟨*Process libraries: process component def: TBP*⟩+≡

```
procedure :: read => process_component_def_read
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_component_def_read (component, unit, core_def_templates)
  class(process_component_def_t), intent(out) :: component
  integer, intent(in) :: unit
  type(prc_template_t), dimension(:), intent(in) :: core_def_templates
  character(80) :: buffer
  type(string_t) :: var_buffer, prefix, in_state, out_state
  type(string_t) :: variant_type

  read (unit, "(A)")  buffer
  call strip_equation_lhs (buffer)
  component%basename = trim (adjustl (buffer))

  read (unit, "(A)")  buffer
  call strip_equation_lhs (buffer)
  read (buffer, *)  component%initial

  read (unit, "(A)")  buffer
  call strip_equation_lhs (buffer)
  read (buffer, *)  component%n_in, component%n_out, component%n_tot

  call get (unit, var_buffer)
  call split (var_buffer, prefix, "=")   ! keeps 'in => out'
  call split (var_buffer, prefix, "=")   ! actually: separator is '=>'

  in_state = prefix
  if (component%n_in > 0) then
     call prt_spec_read (component%prt_in, in_state)
  end if

  out_state = extract (var_buffer, 2)
  if (component%n_out > 0) then
     call prt_spec_read (component%prt_out, out_state)
  end if

  read (unit, "(A)")  buffer
  call strip_equation_lhs (buffer)
  component%method = trim (adjustl (buffer))
  if (component%method == "[undefined]") &
       component%method = ""

  read (unit, "(A)")  buffer
  call strip_equation_lhs (buffer)
  variant_type = trim (adjustl (buffer))
  call allocate_core_def &
       (core_def_templates, variant_type, component%core_def)
  if (allocated (component%core_def)) then
     call component%core_def%read (unit)
  end if

  read (unit, "(A)")  buffer
  call strip_equation_lhs (buffer)
  read (buffer(3:34), "(A32)")  component%md5sum
```

```
end subroutine process_component_def_read
```

Short account.

⟨*Process libraries: process component def: TBP*⟩+≡

```
procedure :: show => process_component_def_show
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_component_def_show (object, unit)
  class(process_component_def_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(6x,A)", advance="no")  char (object%basename)
  if (.not. object%initial) &
       write (u, "('*')", advance="no")
  write (u, "(':',1x)", advance="no")
  if (allocated (object%prt_in)) then
     call prt_spec_write (object%prt_in, u, advance="no")
  else
     write (u, "(A)", advance="no")  "[undefined]"
  end if
  write (u, "(A)", advance="no") " => "
  if (allocated (object%prt_out)) then
     call prt_spec_write (object%prt_out, u, advance="no")
  else
     write (u, "(A)", advance="no")  "[undefined]"
  end if
  if (object%method /= "") then
     write (u, "(2x,'[',A,']')")  char (object%method)
  else
     write (u, *)
  end if
end subroutine process_component_def_show
```

Compute the MD5 sum of a process component. We reset the stored MD5 sum to the empty string (so a previous value is not included in the calculation), the write a temporary file and calculate the MD5 sum of that file.

This implies that all data that are displayed by the `write` method become part of the MD5 sum calculation.

The `model` is not part of the object, but must be included in the MD5 sum. Otherwise, modifying the model and nothing else would not trigger remaking the process-component source. Note that the model parameters may change later and therefore are not incorporated.

After the MD5 sum of the component has been computed, we communicate it to the `writer` subobject of the specific `core_def` component. Although these types are abstract, the MD5-related features are valid for the abstract types.

⟨*Process libraries: process component def: TBP*⟩+≡

```
procedure :: compute_md5sum => process_component_def_compute_md5sum
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_component_def_compute_md5sum (component, model)
  class(process_component_def_t), intent(inout) :: component
  class(model_data_t), intent(in), optional, target :: model
```

```
    integer :: u
    component%md5sum = ""
    u = free_unit ()
    open (u, status = "scratch", action = "readwrite")
    if (present (model))  write (u, "(A32)")  model%get_md5sum ()
    call component%write (u)
    rewind (u)
    component%md5sum = md5sum (u)
    close (u)
    if (allocated (component%core_def)) then
       call component%core_def%set_md5sum (component%md5sum)
    end if
  end subroutine process_component_def_compute_md5sum
```

⟨*Process libraries: process component def: TBP*⟩+≡

```
  procedure :: get_def_type_string => process_component_def_get_def_type_string
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_component_def_get_def_type_string (component) result (type_string)
    type(string_t) :: type_string
    class(process_component_def_t), intent(in) :: component
    type_string = component%core_def%type_string ()
  end function process_component_def_get_def_type_string
```

Allocate the process driver (with a suitable type) for a process component. For internal processes, we may set all data already at this stage.

⟨*Process libraries: process component def: TBP*⟩+≡

```
  procedure :: allocate_driver => process_component_def_allocate_driver
```

⟨*Process libraries: procedures*⟩+≡

```
  subroutine process_component_def_allocate_driver (component, driver)
    class(process_component_def_t), intent(in) :: component
    class(prc_core_driver_t), intent(out), allocatable :: driver
    if (allocated (component%core_def)) then
       call component%core_def%allocate_driver (driver, component%basename)
    end if
  end subroutine process_component_def_allocate_driver
```

Tell whether the process core needs external code.

⟨*Process libraries: process component def: TBP*⟩+≡

```
  procedure :: needs_code => process_component_def_needs_code
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_component_def_needs_code (component) result (flag)
    class(process_component_def_t), intent(in) :: component
    logical :: flag
    flag = component%core_def%needs_code ()
  end function process_component_def_needs_code
```

If there is external code, the `core_def` subobject should provide a writer object. This method returns a pointer to the writer.

⟨*Process libraries: process component def: TBP*⟩+≡

```
  procedure :: get_writer_ptr => process_component_def_get_writer_ptr
```

⟨*Process libraries: procedures*⟩+≡

```
function process_component_def_get_writer_ptr (component) result (writer)
  class(process_component_def_t), intent(in), target :: component
  class(prc_writer_t), pointer :: writer
  writer => component%core_def%writer
end function process_component_def_get_writer_ptr
```

Return an array which holds the names of all C functions that this process component implements.

⟨*Process libraries: process component def: TBP*⟩+≡

```
procedure :: get_features => process_component_def_get_features
```

⟨*Process libraries: procedures*⟩+≡

```
function process_component_def_get_features (component) result (features)
  class(process_component_def_t), intent(in) :: component
  type(string_t), dimension(:), allocatable :: features
  call component%core_def%get_features (features)
end function process_component_def_get_features
```

Assign procedure pointers in the `driver` component (external processes). For internal processes, this is meaningless.

⟨*Process libraries: process component def: TBP*⟩+≡

```
procedure :: connect => process_component_def_connect
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_component_def_connect &
     (component, lib_driver, i, proc_driver)
  class(process_component_def_t), intent(in) :: component
  class(prclib_driver_t), intent(in) :: lib_driver
  integer, intent(in) :: i
  class(prc_core_driver_t), intent(inout) :: proc_driver
  select type (proc_driver)
  class is (process_driver_internal_t)
     !!! Nothing to do
  class default
     call component%core_def%connect (lib_driver, i, proc_driver)
  end select
end subroutine process_component_def_connect
```

Return a pointer to the process core definition, which is of abstract type.

⟨*Process libraries: process component def: TBP*⟩+≡

```
procedure :: get_core_def_ptr => process_component_get_core_def_ptr
```

⟨*Process libraries: procedures*⟩+≡

```
function process_component_get_core_def_ptr (component) result (ptr)
  class(process_component_def_t), intent(in), target :: component
  class(prc_core_def_t), pointer :: ptr
  ptr => component%core_def
end function process_component_get_core_def_ptr
```

Return nominal particle counts, as input by the user.

⟨*Process libraries: process component def: TBP*⟩+≡

```
procedure :: get_n_in  => process_component_def_get_n_in
```

```
procedure :: get_n_out => process_component_def_get_n_out
procedure :: get_n_tot => process_component_def_get_n_tot
```

⟨*Process libraries: procedures*⟩+≡

```
function process_component_def_get_n_in (component) result (n_in)
  class(process_component_def_t), intent(in) :: component
  integer :: n_in
  n_in = component%n_in
end function process_component_def_get_n_in

function process_component_def_get_n_out (component) result (n_out)
  class(process_component_def_t), intent(in) :: component
  integer :: n_out
  n_out = component%n_out
end function process_component_def_get_n_out

function process_component_def_get_n_tot (component) result (n_tot)
  class(process_component_def_t), intent(in) :: component
  integer :: n_tot
  n_tot = component%n_tot
end function process_component_def_get_n_tot
```

Allocate and return string arrays for the incoming and outgoing particles.

⟨*Process libraries: process component def: TBP*⟩+≡

```
procedure :: get_prt_in => process_component_def_get_prt_in
procedure :: get_prt_out => process_component_def_get_prt_out
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_component_def_get_prt_in (component, prt)
  class(process_component_def_t), intent(in) :: component
  type(string_t), dimension(:), intent(out), allocatable :: prt
  integer :: i
  allocate (prt (component%n_in))
  do i = 1, component%n_in
     prt(i) = component%prt_in(i)%to_string ()
  end do
end subroutine process_component_def_get_prt_in

subroutine process_component_def_get_prt_out (component, prt)
  class(process_component_def_t), intent(in) :: component
  type(string_t), dimension(:), intent(out), allocatable :: prt
  integer :: i
  allocate (prt (component%n_out))
  do i = 1, component%n_out
     prt(i) = component%prt_out(i)%to_string ()
  end do
end subroutine process_component_def_get_prt_out
```

Return the combination of incoming particles as a PDG code

⟨*Process libraries: process component def: TBP*⟩+≡

```
procedure :: get_pdg_in => process_component_def_get_pdg_in
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_component_def_get_pdg_in (component, model, pdg)
```

```
    class(process_component_def_t), intent(in) :: component
    class(model_data_t), intent(in), target :: model
    integer, intent(out), dimension(:) :: pdg
    integer :: i
    do i = 1, size (pdg)
       pdg(i) = model%get_pdg (component%prt_in(i)%to_string ())
    end do
  end subroutine process_component_def_get_pdg_in
```

Return the MD5 sum.

⟨*Process libraries: process component def: TBP*⟩+≡

```
  procedure :: get_md5sum => process_component_def_get_md5sum
```

⟨*Process libraries: procedures*⟩+≡

```
  pure function process_component_def_get_md5sum (component) result (md5sum)
    class(process_component_def_t), intent(in) :: component
    character(32) :: md5sum
    md5sum = component%md5sum
  end function process_component_def_get_md5sum
```

Get NLO data

⟨*Process libraries: process component def: TBP*⟩+≡

```
  procedure :: get_nlo_type => process_component_def_get_nlo_type
  procedure :: get_associated_born &
               => process_component_def_get_associated_born
  procedure :: get_associated_real_fin &
               => process_component_def_get_associated_real_fin
  procedure :: get_associated_real_sing &
               => process_component_def_get_associated_real_sing
  procedure :: get_associated_subtraction &
               => process_component_def_get_associated_subtraction
  procedure :: get_association_list &
               => process_component_def_get_association_list
  procedure :: can_be_integrated &
               => process_component_def_can_be_integrated
  procedure :: get_associated_real => process_component_def_get_associated_real
```

⟨*Process libraries: procedures*⟩+≡

```
  elemental function process_component_def_get_nlo_type (component) result (nlo_type)
    integer :: nlo_type
    class(process_component_def_t), intent(in) :: component
    nlo_type = component%nlo_type
  end function process_component_def_get_nlo_type

  elemental function process_component_def_get_associated_born (component) result (i_born)
    integer :: i_born
    class(process_component_def_t), intent(in) :: component
    i_born = component%associated_components(ASSOCIATED_BORN)
  end function process_component_def_get_associated_born

  elemental function process_component_def_get_associated_real_fin (component) result (i_rfin)
    integer :: i_rfin
    class(process_component_def_t), intent(in) :: component
    i_rfin = component%associated_components(ASSOCIATED_REAL_FIN)
```

```
end function process_component_def_get_associated_real_fin

elemental function process_component_def_get_associated_real_sing (component) result (i_rsing)
  integer :: i_rsing
  class(process_component_def_t), intent(in) :: component
  i_rsing = component%associated_components(ASSOCIATED_REAL_SING)
end function process_component_def_get_associated_real_sing

elemental function process_component_def_get_associated_subtraction (component) result (i_sub)
  integer :: i_sub
  class(process_component_def_t), intent(in) :: component
  i_sub = component%associated_components(ASSOCIATED_SUB)
end function process_component_def_get_associated_subtraction

elemental function process_component_def_can_be_integrated (component) result (active)
  logical :: active
  class(process_component_def_t), intent(in) :: component
  active = component%active
end function process_component_def_can_be_integrated

function process_component_def_get_association_list (component, i_skip_in) result (list)
  integer, dimension(:), allocatable :: list
  class(process_component_def_t), intent(in) :: component
  integer, intent(in), optional :: i_skip_in
  integer :: i, j, n, i_skip
  logical :: valid
  i_skip = 0; if (present (i_skip_in)) i_skip = i_skip_in
  n = count (component%associated_components /= 0) - 1
  if (i_skip > 0) n = n - 1
  allocate (list (n))
  j = 1
  do i = 1, size(component%associated_components)
     valid = component%associated_components(i) /= 0 &
             .and. i /= ASSOCIATED_SUB .and. i /= i_skip
     if (valid) then
        list(j) = component%associated_components(i)
        j = j + 1
     end if
  end do
end function process_component_def_get_association_list

function process_component_def_get_associated_real (component) result (i_real)
  integer :: i_real
  class(process_component_def_t), intent(in) :: component
  i_real = component%associated_components(ASSOCIATED_REAL)
end function process_component_def_get_associated_real
```

⟨*Process libraries: process component def: TBP*⟩+≡

```
procedure :: get_me_method => process_component_def_get_me_method
```

⟨*Process libraries: procedures*⟩+≡

```
elemental function process_component_def_get_me_method (component) result (method)
  type(string_t) :: method
  class(process_component_def_t), intent(in) :: component
```

```
    method = component%method
  end function process_component_def_get_me_method
```

⟨*Process libraries: process component def: TBP*⟩+≡

```
  procedure :: get_fixed_emitter => process_component_def_get_fixed_emitter
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_component_def_get_fixed_emitter (component) result (emitter)
     integer :: emitter
     class(process_component_def_t), intent(in) :: component
     emitter = component%fixed_emitter
  end function process_component_def_get_fixed_emitter
```

⟨*Process libraries: process component def: TBP*⟩+≡

```
  procedure :: get_coupling_powers => process_component_def_get_coupling_powers
```

⟨*Process libraries: procedures*⟩+≡

```
  pure subroutine process_component_def_get_coupling_powers (component, alpha_power, alphas_power)
    class(process_component_def_t), intent(in) :: component
    integer, intent(out) :: alpha_power, alphas_power
    alpha_power = component%alpha_power
    alphas_power = component%alphas_power
  end subroutine process_component_def_get_coupling_powers
```

Process definition

The process component definitions are collected in a common process definition object.

The `id` is the ID string that the user has provided for identifying this process. It must be a string that is allowed as part of a Fortran variable name, since it may be used for generating code.

The number `n_in` is 1 or 2 for a decay or scattering process, respectively. This must be identical to `n_in` for all components.

The initial and extra component definitions (see above) are allocated as the `initial` and `extra` arrays, respectively. The latter are determined from the former.

The `md5sum` is used to verify the integrity of the configuration.

⟨*Process libraries: public*⟩+≡

```
  public :: process_def_t
```

⟨*Process libraries: types*⟩+≡

```
  type :: process_def_t
     private
     type(string_t) :: id
     integer :: num_id = 0
     class(model_data_t), pointer :: model => null ()
     type(string_t) :: model_name
     integer :: n_in  = 0
     integer :: n_initial = 0
     integer :: n_extra = 0
     type(process_component_def_t), dimension(:), allocatable :: initial
     type(process_component_def_t), dimension(:), allocatable :: extra
```

```
    character(32) :: md5sum = ""
    logical :: nlo_process = .false.
  contains
  ⟨Process libraries: process def: TBP⟩
end type process_def_t
```

Write the process definition including components:

⟨*Process libraries: process def: TBP*⟩≡

```
procedure :: write => process_def_write
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_write (object, unit)
  class(process_def_t), intent(in) :: object
  integer, intent(in) :: unit
  integer :: i
  write (unit, "(1x,A,A,A)") "ID = '", char (object%id), "'"
  if (object%num_id /= 0) &
       write (unit, "(1x,A,I0)")  "ID(num) = ", object%num_id
  select case (object%n_in)
  case (1);  write (unit, "(1x,A)")  "Decay"
  case (2);  write (unit, "(1x,A)")  "Scattering"
  case default
     write (unit, "(1x,A)")  "[Undefined process]"
     return
  end select
  if (object%model_name /= "") then
     write (unit, "(1x,A,A)")  "Model = ", char (object%model_name)
  else
     write (unit, "(1x,A)")  "Model = [undefined]"
  end if
  write (unit, "(1x,A,I0)")  "Initially defined component(s) = ", &
       object%n_initial
  write (unit, "(1x,A,I0)")  "Extra generated component(s)   = ", &
       object%n_extra
  write (unit, "(1x,A,A,A)") "MD5 sum   = '", object%md5sum, "'"
  if (allocated (object%initial)) then
     do i = 1, size (object%initial)
        write (unit, "(1x,A,I0)")  "Component #", i
        call object%initial(i)%write (unit)
     end do
  end if
  if (allocated (object%extra)) then
     do i = 1, size (object%extra)
        write (unit, "(1x,A,I0)")  "Component #", object%n_initial + i
        call object%extra(i)%write (unit)
     end do
  end if
end subroutine process_def_write
```

Read the process definition including components.

⟨*Process libraries: process def: TBP*⟩+≡

```
procedure :: read => process_def_read
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_read (object, unit, core_def_templates)
  class(process_def_t), intent(out) :: object
  integer, intent(in) :: unit
  type(prc_template_t), dimension(:), intent(in) :: core_def_templates
  integer :: i, i1, i2
  character(80) :: buffer, ref
  read (unit, "(A)")  buffer
  call strip_equation_lhs (buffer)
  i1 = scan (buffer, "'")
  i2 = scan (buffer, "'", back=.true.)
  if (i2 > i1) then
     object%id = buffer(i1+1:i2-1)
  else
     object%id = ""
  end if

  read (unit, "(A)")  buffer
  select case (buffer(2:11))
  case ("Decay     "); object%n_in = 1
  case ("Scattering"); object%n_in = 2
  case default
     return
  end select

  read (unit, "(A)")  buffer
  call strip_equation_lhs (buffer)
  object%model_name = trim (adjustl (buffer))
  if (object%model_name == "[undefined]")  object%model_name = ""

  read (unit, "(A)")  buffer
  call strip_equation_lhs (buffer)
  read (buffer, *)  object%n_initial

  read (unit, "(A)")  buffer
  call strip_equation_lhs (buffer)
  read (buffer, *)  object%n_extra

  read (unit, "(A)")  buffer
  call strip_equation_lhs (buffer)
  read (buffer(3:34), "(A32)")  object%md5sum

  if (object%n_initial > 0) then
     allocate (object%initial (object%n_initial))
     do i = 1, object%n_initial
        read (unit, "(A)")  buffer
        write (ref, "(1x,A,I0)")  "Component #", i
        if (buffer /= ref)  return                ! Wrong component header
        call object%initial(i)%read (unit, core_def_templates)
     end do
  end if

end subroutine process_def_read
```

Short account.

⟨*Process libraries: process def: TBP*⟩+≡

```
procedure :: show => process_def_show
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_show (object, unit)
  class(process_def_t), intent(in) :: object
  integer, intent(in) :: unit
  integer :: i
  write (unit, "(4x,A)", advance="no") char (object%id)
  if (object%num_id /= 0) &
       write (unit, "(1x,'(',I0,')')", advance="no")  object%num_id
  if (object%model_name /= "") &
       write (unit, "(1x,'[',A,']')")  char (object%model_name)
  if (allocated (object%initial)) then
     do i = 1, size (object%initial)
        call object%initial(i)%show (unit)
     end do
  end if
  if (allocated (object%extra)) then
     do i = 1, size (object%extra)
        call object%extra(i)%show (unit)
     end do
  end if
end subroutine process_def_show
```

Initialize an entry (initialize the process definition inside). We allocate the 'initial' set of components. Extra components remain unallocated.

The model should be present as a pointer. This allows us to retrieve the model's MD5 sum. However, for various tests it is sufficient to have the name.

We create the basenames for the process components by appending a suffix which we increment for each component.

⟨*Process libraries: process def: TBP*⟩+≡

```
procedure :: init => process_def_init
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_init (def, id, &
     model, model_name, n_in, n_components, num_id, nlo_process)
  class(process_def_t), intent(out) :: def
  type(string_t), intent(in), optional :: id
  class(model_data_t), intent(in), optional, target :: model
  type(string_t), intent(in), optional :: model_name
  integer, intent(in), optional :: n_in
  integer, intent(in), optional :: n_components
  integer, intent(in), optional :: num_id
  logical, intent(in), optional :: nlo_process
  character(16) :: suffix
  integer :: i
  if (present (id)) then
     def%id = id
  else
     def%id = ""
  end if
  if (present (num_id)) then
```

```
      def%num_id = num_id
   end if
   if (present (model)) then
      def%model => model
      def%model_name = model%get_name ()
   else
      def%model => null ()
      if (present (model_name)) then
         def%model_name = model_name
      else
         def%model_name = ""
      end if
   end if
   if (present (n_in))  def%n_in = n_in
   if (present (n_components)) then
      def%n_initial = n_components
      allocate (def%initial (n_components))
   end if
   if (present (nlo_process)) def%nlo_process = nlo_process
   def%initial%initial = .true.
   def%initial%method     = ""
   do i = 1, def%n_initial
      write (suffix, "(A,I0)")  "_i", i
      def%initial(i)%basename = def%id // trim (suffix)
   end do
   def%initial%description = ""
end subroutine process_def_init
```

Explicitly set the model name (for unit test).

⟨*Process libraries: process def: TBP*⟩+≡

```
procedure :: set_model_name => process_def_set_model_name
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_set_model_name (def, model_name)
  class(process_def_t), intent(inout) :: def
  type(string_t), intent(in) :: model_name
  def%model_name = model_name
end subroutine process_def_set_model_name
```

Initialize an initial component. The particle content must be specified. The process core block is not (yet) allocated.

We assume that the particle arrays match the `n_in` and `n_out` values in size. The model is referred to by name; it is identified as an existing model later. The index `i` must refer to an existing element of the component array.

Data specific for the process core of a component are imported as the `core_def` argument. We should allocate an object of class `prc_core_def_t` with the appropriate specific type, fill it, and transfer it to the process component definition here. The allocation is moved, so the original allocated object is returned empty.

⟨*Process libraries: process def: TBP*⟩+≡

```
procedure :: import_component => process_def_import_component
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_import_component (def, &
     i, n_out, prt_in, prt_out, method, variant, &
     nlo_type, can_be_integrated)
  class(process_def_t), intent(inout) :: def
  integer, intent(in) :: i
  integer, intent(in), optional :: n_out
  type(prt_spec_t), dimension(:), intent(in), optional :: prt_in
  type(prt_spec_t), dimension(:), intent(in), optional :: prt_out
  type(string_t), intent(in), optional :: method
  integer, intent(in), optional :: nlo_type
  logical, intent(in), optional :: can_be_integrated
  type(string_t) :: nlo_type_string
  class(prc_core_def_t), &
       intent(inout), allocatable, optional :: variant
  integer :: p
  associate (comp => def%initial(i))
    if (present (n_out)) then
       comp%n_in  = def%n_in
       comp%n_out = n_out
       comp%n_tot = def%n_in + n_out
    end if
    if (present (prt_in)) then
       allocate (comp%prt_in (size (prt_in)))
       comp%prt_in = prt_in
    end if
    if (present (prt_out)) then
       allocate (comp%prt_out (size (prt_out)))
       comp%prt_out = prt_out
    end if
    if (present (method))  comp%method = method
    if (present (variant)) then
       call move_alloc (variant, comp%core_def)
    end if
    if (present (nlo_type)) then
      comp%nlo_type = nlo_type
    end if
    if (present (can_be_integrated)) then
       comp%active = can_be_integrated
    else
       comp%active = .true.
    end if
    if (allocated (comp%prt_in) .and. allocated (comp%prt_out)) then
       associate (d => comp%description)
         d = ""
         do p = 1, size (prt_in)
            if (p > 1)  d = d // ", "
            d = d // comp%prt_in(p)%to_string ()
         end do
         d = d // " => "
         do p = 1, size (prt_out)
            if (p > 1)  d = d // ", "
            d = d // comp%prt_out(p)%to_string ()
         end do
```

```
            if (comp%method /= "") then
               !  TODO: (bcn 2016-09-16) better output for subtraction
               if ((def%nlo_process .and. .not. comp%active) .or. &
                    comp%nlo_type == NLO_SUBTRACTION) then
                  d = d // " [inactive]"
               else
                  d = d // " [" // comp%method // "]"
               end if
            end if
            nlo_type_string = component_status (comp%nlo_type)
            if (nlo_type_string /= "Born") then
              d = d // ", [" // nlo_type_string // "]"
            end if
          end associate
        end if
      end associate
    end subroutine process_def_import_component
```

⟨*Process libraries: process def: TBP*⟩+≡

```
    procedure :: get_n_components => process_def_get_n_components
```

⟨*Process libraries: procedures*⟩+≡

```
    function process_def_get_n_components (def) result (n)
      class(process_def_t), intent(in) :: def
      integer :: n
      n = size (def%initial)
    end function process_def_get_n_components
```

⟨*Process libraries: process def: TBP*⟩+≡

```
    procedure :: set_fixed_emitter => process_def_set_fixed_emitter
```

⟨*Process libraries: procedures*⟩+≡

```
    subroutine process_def_set_fixed_emitter (def, i, emitter)
      class(process_def_t), intent(inout) :: def
      integer, intent(in) :: i, emitter
      def%initial(i)%fixed_emitter = emitter
    end subroutine process_def_set_fixed_emitter
```

⟨*Process libraries: process def: TBP*⟩+≡

```
    procedure :: set_coupling_powers => process_def_set_coupling_powers
```

⟨*Process libraries: procedures*⟩+≡

```
    subroutine process_def_set_coupling_powers (def, alpha_power, alphas_power)
      class(process_def_t), intent(inout) :: def
      integer, intent(in) :: alpha_power, alphas_power
      def%initial(1)%alpha_power = alpha_power
      def%initial(1)%alphas_power = alphas_power
    end subroutine process_def_set_coupling_powers
```

⟨*Process libraries: process def: TBP*⟩+≡

```
    procedure :: set_associated_components => &
         process_def_set_associated_components
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_set_associated_components (def, i, &
       i_born, i_real, i_virt, i_sub, i_pdf, i_rfin)
  class(process_def_t), intent(inout) :: def
  integer, intent(in) :: i
  integer, intent(in) :: i_born, i_real
  integer, intent(in) :: i_virt, i_sub
  integer, intent(in), optional :: i_pdf, i_rfin
  associate (comp => def%initial(i)%associated_components)
     comp(ASSOCIATED_BORN) = i_born
     comp(ASSOCIATED_REAL) = i_real
     comp(ASSOCIATED_VIRT) = i_virt
     comp(ASSOCIATED_SUB) = i_sub
     if (present (i_pdf)) &
        comp(ASSOCIATED_PDF) = i_pdf
     if (present (i_rfin)) &
        comp(ASSOCIATED_REAL_FIN) = i_rfin
  end associate
end subroutine process_def_set_associated_components
```

Compute the MD5 sum for this process definition. We compute the MD5 sums for all components individually, than concatenate a string of those and compute the MD5 sum of this string. We also include the model name. All other data part of the component definitions.

⟨*Process libraries: process def: TBP*⟩+≡

```
procedure :: compute_md5sum => process_def_compute_md5sum
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_compute_md5sum (def, model)
  class(process_def_t), intent(inout) :: def
  class(model_data_t), intent(in), optional, target :: model
  integer :: i
  type(string_t) :: buffer
  buffer = def%model_name
  do i = 1, def%n_initial
     call def%initial(i)%compute_md5sum (model)
     buffer = buffer // def%initial(i)%md5sum
  end do
  do i = 1, def%n_extra
     call def%extra(i)%compute_md5sum (model)
     buffer = buffer // def%initial(i)%md5sum
  end do
  def%md5sum = md5sum (char (buffer))
end subroutine process_def_compute_md5sum
```

Return the MD5 sum of the process or of a process component.

⟨*Process libraries: process def: TBP*⟩+≡

```
procedure :: get_md5sum => process_def_get_md5sum
```

⟨*Process libraries: procedures*⟩+≡

```
function process_def_get_md5sum (def, i_component) result (md5sum)
  class(process_def_t), intent(in) :: def
  integer, intent(in), optional :: i_component
  character(32) :: md5sum
```

```
    if (present (i_component)) then
       md5sum = def%initial(i_component)%md5sum
    else
       md5sum = def%md5sum
    end if
  end function process_def_get_md5sum
```

Return a pointer to the definition of a particular component (for test purposes).

⟨*Process libraries: process def: TBP*⟩+≡

```
  procedure :: get_core_def_ptr => process_def_get_core_def_ptr
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_def_get_core_def_ptr (def, i_component) result (ptr)
    class(process_def_t), intent(in), target :: def
    integer, intent(in) :: i_component
    class(prc_core_def_t), pointer :: ptr
    ptr => def%initial(i_component)%get_core_def_ptr ()
  end function process_def_get_core_def_ptr
```

This query tells whether a specific process component relies on external code. This includes all traditional WHIZARD matrix elements which rely on `O'MEGA` for code generation. Other process components (trivial decays, subtraction terms) do not require external code.

NOTE: Implemented only for initial component.

The query is passed to the process component.

⟨*Process libraries: process def: TBP*⟩+≡

```
  procedure :: needs_code => process_def_needs_code
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_def_needs_code (def, i_component) result (flag)
    class(process_def_t), intent(in) :: def
    integer, intent(in) :: i_component
    logical :: flag
    flag = def%initial(i_component)%needs_code ()
  end function process_def_needs_code
```

Return the first entry for the incoming particle(s), PDG code, of this process.

⟨*Process libraries: process def: TBP*⟩+≡

```
  procedure :: get_pdg_in_1 => process_def_get_pdg_in_1
```

⟨*Process libraries: procedures*⟩+≡

```
  subroutine process_def_get_pdg_in_1 (def, pdg)
    class(process_def_t), intent(in), target :: def
    integer, dimension(:), intent(out) :: pdg
    call def%initial(1)%get_pdg_in (def%model, pdg)
  end subroutine process_def_get_pdg_in_1
```

⟨*Process libraries: process def: TBP*⟩+≡

```
  procedure :: get_nlo_type => process_def_get_nlo_type
```

⟨*Process libraries: procedures*⟩+≡
```
elemental function process_def_get_nlo_type (def, i_component) result (nlo_type)
  integer :: nlo_type
  class(process_def_t), intent(in) :: def
  integer, intent(in) :: i_component
  nlo_type = def%initial(i_component)%nlo_type
end function process_def_get_nlo_type
```

Process definition list

A list of process definitions is the starting point for creating a process library. The list is built when reading the user input. When reading an existing process library, the list is used for cross-checking and updating the configuration.

We need a type for the list entry. The simplest way is to extend the process definition type, so all methods apply to the process definition directly.

⟨*Process libraries: public*⟩+≡
```
public :: process_def_entry_t
```

⟨*Process libraries: types*⟩+≡
```
type, extends (process_def_t) :: process_def_entry_t
   private
   type(process_def_entry_t), pointer :: next => null ()
end type process_def_entry_t
```

This is the type for the list itself.

⟨*Process libraries: public*⟩+≡
```
public :: process_def_list_t
```

⟨*Process libraries: types*⟩+≡
```
type :: process_def_list_t
   private
   type(process_def_entry_t), pointer :: first => null ()
   type(process_def_entry_t), pointer :: last => null ()
 contains
 ⟨Process libraries: process def list: TBP⟩
end type process_def_list_t
```

The deallocates the list iteratively. We assume that the list entries do not need finalization themselves.

⟨*Process libraries: process def list: TBP*⟩≡
```
procedure :: final => process_def_list_final
```

⟨*Process libraries: procedures*⟩+≡
```
subroutine process_def_list_final (list)
  class(process_def_list_t), intent(inout) :: list
  type(process_def_entry_t), pointer :: current
  nullify (list%last)
  do while (associated (list%first))
     current => list%first
     list%first => current%next
     deallocate (current)
  end do
```

```
end subroutine process_def_list_final
```

Write the complete list.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: write => process_def_list_write
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_list_write (object, unit, libpath)
  class(process_def_list_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: libpath
  type(process_def_entry_t), pointer :: entry
  integer :: i, u
  u = given_output_unit (unit)
  if (associated (object%first)) then
     i = 1
     entry => object%first
     do while (associated (entry))
        write (u, "(1x,A,I0,A)")  "Process #", i, ":"
        call entry%write (u)
        i = i + 1
        entry => entry%next
        if (associated (entry))  write (u, *)
     end do
  else
     write (u, "(1x,A)")  "Process definition list: [empty]"
  end if
end subroutine process_def_list_write
```

Short account.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: show => process_def_list_show
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_list_show (object, unit)
  class(process_def_list_t), intent(in) :: object
  integer, intent(in), optional :: unit
  type(process_def_entry_t), pointer :: entry
  integer :: u
  u = given_output_unit (unit)
  if (associated (object%first)) then
     write (u, "(2x,A)")  "Processes:"
     entry => object%first
     do while (associated (entry))
        call entry%show (u)
        entry => entry%next
     end do
  else
     write (u, "(2x,A)")  "Processes: [empty]"
  end if
end subroutine process_def_list_show
```

Read the complete list. We need an array of templates for the component subobjects of abstract prc_core_t type, to allocate them with the correct specific type.

NOTE: Error handling is missing. Reading will just be aborted on error, or an I/O error occurs.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: read => process_def_list_read
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_list_read (object, unit, core_def_templates)
  class(process_def_list_t), intent(out) :: object
  integer, intent(in) :: unit
  type(prc_template_t), dimension(:), intent(in) :: core_def_templates
  type(process_def_entry_t), pointer :: entry
  character(80) :: buffer, ref
  integer :: i
  read (unit, "(A)")  buffer
  write (ref, "(1x,A)")  "Process definition list: [empty]"
  if (buffer == ref)  return         ! OK: empty library
  backspace (unit)
  READ_ENTRIES: do i = 1, huge (0)
     if (i > 1) read (unit, *, end=1)
     read (unit, "(A)")  buffer

     write (ref, "(1x,A,I0,A)")  "Process #", i, ":"
     if (buffer /= ref)  return      ! Wrong process header: done.
     allocate (entry)
     call entry%read (unit, core_def_templates)
     call object%append (entry)
  end do READ_ENTRIES
1 continue                           ! EOF: done
end subroutine process_def_list_read
```

Append an entry to the list. The entry should be allocated as a pointer, and the pointer allocation is transferred. The original pointer is returned null.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: append => process_def_list_append
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_list_append (list, entry)
  class(process_def_list_t), intent(inout) :: list
  type(process_def_entry_t), intent(inout), pointer :: entry
  if (list%contains (entry%id)) then
     call msg_fatal ("Recording process: '" // char (entry%id) &
          // "' has already been defined")
  end if
  if (associated (list%first)) then
     list%last%next => entry
  else
     list%first => entry
  end if
  list%last => entry
  entry => null ()
end subroutine process_def_list_append
```

Probe the process definition list

Return the number of processes supported by the library.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: get_n_processes => process_def_list_get_n_processes
```

⟨*Process libraries: procedures*⟩+≡

```
function process_def_list_get_n_processes (list) result (n)
  integer :: n
  class(process_def_list_t), intent(in) :: list
  type(process_def_entry_t), pointer :: current
  n = 0
  current => list%first
  do while (associated (current))
     n = n + 1
     current => current%next
  end do
end function process_def_list_get_n_processes
```

Allocate an array with the process IDs supported by the library.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: get_process_id_list => process_def_list_get_process_id_list
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_list_get_process_id_list (list, id)
  class(process_def_list_t), intent(in) :: list
  type(string_t), dimension(:), allocatable, intent(out) :: id
  type(process_def_entry_t), pointer :: current
  integer :: i
  allocate (id (list%get_n_processes ()))
  i = 0
  current => list%first
  do while (associated (current))
     i = i + 1
     id(i) = current%id
     current => current%next
  end do
end subroutine process_def_list_get_process_id_list
```

Return true if a given process is in the library.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: contains => process_def_list_contains
```

⟨*Process libraries: procedures*⟩+≡

```
function process_def_list_contains (list, id) result (flag)
  logical :: flag
  class(process_def_list_t), intent(in) :: list
  type(string_t), intent(in) :: id
  type(process_def_entry_t), pointer :: current
  current => list%first
  do while (associated (current))
     if (id == current%id) then
```

```
         flag = .true.;  return
      end if
      current => current%next
   end do
   flag = .false.
end function process_def_list_contains
```

Return the index of the entry that corresponds to a given process.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: get_entry_index => process_def_list_get_entry_index
```

⟨*Process libraries: procedures*⟩+≡

```
function process_def_list_get_entry_index (list, id) result (n)
  integer :: n
  class(process_def_list_t), intent(in) :: list
  type(string_t), intent(in) :: id
  type(process_def_entry_t), pointer :: current
  n = 0
  current => list%first
  do while (associated (current))
     n = n + 1
     if (id == current%id) then
        return
     end if
     current => current%next
  end do
  n = 0
end function process_def_list_get_entry_index
```

Return the numerical ID for a process.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: get_num_id => process_def_list_get_num_id
```

⟨*Process libraries: procedures*⟩+≡

```
function process_def_list_get_num_id (list, id) result (num_id)
  integer :: num_id
  class(process_def_list_t), intent(in) :: list
  type(string_t), intent(in) :: id
  type(process_def_entry_t), pointer :: current
  current => list%first
  do while (associated (current))
     if (id == current%id) then
        num_id = current%num_id
        return
     end if
     current => current%next
  end do
  num_id = 0
end function process_def_list_get_num_id
```

Return the model name for a given process in the library.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: get_model_name => process_def_list_get_model_name
```

⟨*Process libraries: procedures*⟩+≡

```
function process_def_list_get_model_name (list, id) result (model_name)
  type(string_t) :: model_name
  class(process_def_list_t), intent(in) :: list
  type(string_t), intent(in) :: id
  type(process_def_entry_t), pointer :: current
  current => list%first
  do while (associated (current))
     if (id == current%id) then
        model_name = current%model_name
        return
     end if
     current => current%next
  end do
  model_name = ""
end function process_def_list_get_model_name
```

Return the number of incoming particles of a given process in the library. This tells us whether the process is a decay or a scattering.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: get_n_in => process_def_list_get_n_in
```

⟨*Process libraries: procedures*⟩+≡

```
function process_def_list_get_n_in (list, id) result (n)
  integer :: n
  class(process_def_list_t), intent(in) :: list
  type(string_t), intent(in) :: id
  type(process_def_entry_t), pointer :: current
  current => list%first
  do while (associated (current))
     if (id == current%id) then
        n = current%n_in
        return
     end if
     current => current%next
  end do
end function process_def_list_get_n_in
```

Return the incoming particle pdg codesnumber of incoming particles of a given process in the library. If there is a PDG array, return only the first code for each beam. This serves as a quick way for (re)constructing beam properties.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: get_pdg_in_1 => process_def_list_get_pdg_in_1
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_list_get_pdg_in_1 (list, id, pdg)
  class(process_def_list_t), intent(in) :: list
  type(string_t), intent(in) :: id
  integer, dimension(:), intent(out) :: pdg
  type(process_def_entry_t), pointer :: current
  current => list%first
  do while (associated (current))
     if (id == current%id) then
        call current%get_pdg_in_1 (pdg)
```

```
         return
      end if
      current => current%next
   end do
end subroutine process_def_list_get_pdg_in_1
```

Return the number of components of a given process in the library.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: get_n_components => process_def_list_get_n_components
```

⟨*Process libraries: procedures*⟩+≡

```
function process_def_list_get_n_components (list, id) result (n)
  integer :: n
  class(process_def_list_t), intent(in) :: list
  type(string_t), intent(in) :: id
  type(process_def_entry_t), pointer :: current
  current => list%first
  do while (associated (current))
     if (id == current%id) then
        n = current%n_initial + current%n_extra
        return
     end if
     current => current%next
  end do
end function process_def_list_get_n_components
```

Return a pointer to a specific process component definition.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: get_component_def_ptr => process_def_list_get_component_def_ptr
```

⟨*Process libraries: procedures*⟩+≡

```
function process_def_list_get_component_def_ptr (list, id, i) result (ptr)
  class(process_def_list_t), intent(in) :: list
  type(string_t), intent(in) :: id
  integer, intent(in) :: i
  type(process_component_def_t), pointer :: ptr
  type(process_def_entry_t), pointer :: current
  ptr => null ()
  current => list%first
  do while (associated (current))
     if (id == current%id) then
        if (i <= current%n_initial) then
           ptr => current%initial(i)
        else if (i <= current%n_initial + current%n_extra) then
           ptr => current%extra(i-current%n_initial)
        end if
        return
     end if
     current => current%next
  end do
end function process_def_list_get_component_def_ptr
```

Return the list of component IDs of a given process in the library.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: get_component_list => process_def_list_get_component_list
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_list_get_component_list (list, id, cid)
  class(process_def_list_t), intent(in) :: list
  type(string_t), intent(in) :: id
  type(string_t), dimension(:), allocatable, intent(out) :: cid
  type(process_def_entry_t), pointer :: current
  integer :: i, n
  current => list%first
  do while (associated (current))
     if (id == current%id) then
        allocate (cid (current%n_initial + current%n_extra))
        do i = 1, current%n_initial
           cid(i) = current%initial(i)%basename
        end do
        n = current%n_initial
        do i = 1, current%n_extra
           cid(n + i) = current%extra(i)%basename
        end do
        return
     end if
     current => current%next
  end do
end subroutine process_def_list_get_component_list
```

Return the list of component description strings for a given process in the library.

⟨*Process libraries: process def list: TBP*⟩+≡

```
procedure :: get_component_description_list => &
     process_def_list_get_component_description_list
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_def_list_get_component_description_list &
     (list, id, description)
  class(process_def_list_t), intent(in) :: list
  type(string_t), intent(in) :: id
  type(string_t), dimension(:), allocatable, intent(out) :: description
  type(process_def_entry_t), pointer :: current
  integer :: i, n
  current => list%first
  do while (associated (current))
     if (id == current%id) then
        allocate (description (current%n_initial + current%n_extra))
        do i = 1, current%n_initial
           description(i) = current%initial(i)%description
        end do
        n = current%n_initial
        do i = 1, current%n_extra
           description(n + i) = current%extra(i)%description
        end do
        return
     end if
```

```
      current => current%next
    end do
  end subroutine process_def_list_get_component_description_list
```

Return whether the entry corresponds to an NLO-process

⟨*Process libraries: process def list: TBP*⟩+≡

```
  procedure :: get_nlo_process => process_def_list_get_nlo_process
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_def_list_get_nlo_process (list, id) result (nlo)
    class(process_def_list_t), intent(in) :: list
    type(string_t), intent(in) :: id
    logical :: nlo
    type(process_def_entry_t), pointer :: current
    current => list%first
    do while (associated (current))
      if (id == current%id) then
        nlo = current%nlo_process
        return
      end if
      current => current%next
    end do
  end function process_def_list_get_nlo_process
```

13.4.3 Process library

The process library object is the interface between the process definition data, as provided by the user, generated or linked process code on file, and the process run data that reference the process code.

Process library entry

For each process component that is part of the library, there is a separate library entry (`process_library_entry_t`. The library entry connects a process definition with the specific code (if any) in the compiled driver library.

The `status` indicates how far the process has been processed by the system (definition, code generation, compilation, linking). A process with status `STAT_LOADED` is accessible for computing matrix elements.

The `def` pointer identifies the corresponding process definition. The process component within that definition is identified by the `i_component` index.

The `i_external` index refers to the compiled library driver. If it is zero, there is no associated matrix-element code.

The `driver` component holds the pointers to the matrix-element specific functions, in particular the matrix element function itself.

⟨*Process libraries: types*⟩+≡

```
  type :: process_library_entry_t
     private
     integer :: status = STAT_UNKNOWN
     type(process_def_t), pointer :: def => null ()
     integer :: i_component = 0
     integer :: i_external = 0
```

```
    class(prc_core_driver_t), allocatable :: driver
  contains
  ⟨Process libraries: process library entry: TBP⟩
end type process_library_entry_t
```

Here are the available status codes. An entry starts with UNKNOWN status. Once the association with a valid process definition is established, the status becomes CONFIGURED. If matrix element source code is to be generated by the system or provided from elsewhere, CODE_GENERATED indicates that this is done. The COMPILED status is next, it also applies to processes which are accessed as pre-compiled binaries. Finally, the library is linked and process pointers are set; this is marked as LOADED.

For a process library, the initial status is OPEN, since process definitions may be added. After configuration, the process content is fixed and the status becomes CONFIGURED. The further states are as above, always referring to the lowest status among the process entries.

⟨*Process libraries: parameters*⟩≡

```
integer, parameter, public :: STAT_UNKNOWN = 0
integer, parameter, public :: STAT_OPEN = 1
integer, parameter, public :: STAT_CONFIGURED = 2
integer, parameter, public :: STAT_SOURCE = 3
integer, parameter, public :: STAT_COMPILED = 4
integer, parameter, public :: STAT_LINKED = 5
integer, parameter, public :: STAT_ACTIVE = 6

integer, parameter, public :: ASSOCIATED_BORN = 1
integer, parameter, public :: ASSOCIATED_REAL = 2
integer, parameter, public :: ASSOCIATED_VIRT = 3
integer, parameter, public :: ASSOCIATED_SUB = 4
integer, parameter, public :: ASSOCIATED_PDF = 5
integer, parameter, public :: ASSOCIATED_REAL_SING = 6
integer, parameter, public :: ASSOCIATED_REAL_FIN = 7
integer, parameter, public :: N_ASSOCIATED_COMPONENTS = 7
```

These are the associated code letters, for output:

⟨*Process libraries: parameters*⟩+≡

```
character, dimension(0:6), parameter :: STATUS_LETTER = &
     ["?", "o", "f", "s", "c", "l", "a"]
```

This produces a condensed account of the library entry. The status is indicated by a letter in brackets, then the ID and component index of the associated process definition, finally the library index, if available.

⟨*Process libraries: process library entry: TBP*⟩≡

```
procedure :: to_string => process_library_entry_to_string
```

⟨*Process libraries: procedures*⟩+≡

```
function process_library_entry_to_string (object) result (string)
  type(string_t) :: string
  class(process_library_entry_t), intent(in) :: object
  character(32) :: buffer
  string = "[" // STATUS_LETTER(object%status) // "]"
  select case (object%status)
```

```
    case (STAT_UNKNOWN)
    case default
       if (associated (object%def)) then
          write (buffer, "(I0)")  object%i_component
          string = string // " " // object%def%id // "." // trim (buffer)
       end if
       if (object%i_external /= 0) then
          write (buffer, "(I0)")  object%i_external
          string = string // " = ext:" // trim (buffer)
       else
          string = string // " = int"
       end if
       if (allocated (object%driver)) then
          string = string // " (" // object%driver%type_name () // ")"
       end if
    end select
  end function process_library_entry_to_string
```

Initialize with data. Used for the unit tests.

⟨*Process libraries: process library entry: TBP*⟩+≡

```
  procedure :: init => process_library_entry_init
```

⟨*Process libraries: procedures*⟩+≡

```
  subroutine process_library_entry_init (object, &
       status, def, i_component, i_external, driver_template)
    class(process_library_entry_t), intent(out) :: object
    integer, intent(in) :: status
    type(process_def_t), target, intent(in) :: def
    integer, intent(in) :: i_component
    integer, intent(in) :: i_external
    class(prc_core_driver_t), intent(inout), allocatable, optional &
         :: driver_template
    object%status = status
    object%def => def
    object%i_component = i_component
    object%i_external = i_external
    if (present (driver_template)) then
       call move_alloc (driver_template, object%driver)
    end if
  end subroutine process_library_entry_init
```

Assign pointers for all process-specific features. We have to combine the method from the `core_def` specification, the assigned pointers within the library driver, the index within that driver, and the process driver which should receive the links.

⟨*Process libraries: process library entry: TBP*⟩+≡

```
  procedure :: connect => process_library_entry_connect
```

⟨*Process libraries: procedures*⟩+≡

```
  subroutine process_library_entry_connect (entry, lib_driver, i)
    class(process_library_entry_t), intent(inout) :: entry
    class(prclib_driver_t), intent(in) :: lib_driver
    integer, intent(in) :: i
    call entry%def%initial(entry%i_component)%connect &
```

```
        (lib_driver, i, entry%driver)
  end subroutine process_library_entry_connect
```

The process library object

The `process_library_t` type is an extension of the `process_def_list_t` type. Thus, it automatically contains the process definition list.

The `basename` identifies the library generically.

The `external` flag is true if any process within the library needs external code, so the library must correspond to an actual code library (statically or dynamically linked).

The `entry` array contains all process components that can be handled by this library. Each entry refers to the process (component) definition and to the associated external matrix element code, if there is any.

The `driver` object is needed only if `external` is true. This object handles all interactions with external matrix-element code.

The `md5sum` summarizes the complete `process_def_list_t` base object. It can be used to check if the library configuration has changed.

⟨*Process libraries: public*⟩+≡

```
  public :: process_library_t
```

⟨*Process libraries: types*⟩+≡

```
  type, extends (process_def_list_t) :: process_library_t
     private
     type(string_t) :: basename
     integer :: n_entries = 0
     logical :: external = .false.
     integer :: status = STAT_UNKNOWN
     logical :: static = .false.
     logical :: driver_exists = .false.
     logical :: makefile_exists = .false.
     integer :: update_counter = 0
     type(process_library_entry_t), dimension(:), allocatable :: entry
     class(prclib_driver_t), allocatable :: driver
     character(32) :: md5sum = ""
   contains
   ⟨Process libraries: process library: TBP⟩
  end type process_library_t
```

For the output, we write first the metadata and the DL access record, then the library entries in short form, and finally the process definition list which is the base object.

Don't write the MD5 sum since this is used to generate it.

⟨*Process libraries: process library: TBP*⟩≡

```
  procedure :: write => process_library_write
```

⟨*Process libraries: procedures*⟩+≡

```
  subroutine process_library_write (object, unit, libpath)
    class(process_library_t), intent(in) :: object
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: libpath
```

```
    integer :: i, u
    u = given_output_unit (unit)
    write (u, "(1x,A,A)")  "Process library: ", char (object%basename)
    write (u, "(3x,A,L1)")   "external        = ", object%external
    write (u, "(3x,A,L1)")   "makefile exists = ", object%makefile_exists
    write (u, "(3x,A,L1)")   "driver exists   = ", object%driver_exists
    write (u, "(3x,A,A1)")   "code status     = ", &
         STATUS_LETTER (object%status)
    write (u, *)
    if (allocated (object%entry)) then
       write (u, "(1x,A)", advance="no")  "Process library entries:"
       write (u, "(1x,I0)")  object%n_entries
       do i = 1, size (object%entry)
          write (u, "(1x,A,I0,A,A)")  "Entry #", i, ": ", &
               char (object%entry(i)%to_string ())
       end do
       write (u, *)
    end if
    if (object%external) then
       call object%driver%write (u, libpath)
       write (u, *)
    end if
    call object%process_def_list_t%write (u)
  end subroutine process_library_write
```

Condensed version for screen output.

⟨*Process libraries: process library: TBP*⟩+≡

```
  procedure :: show => process_library_show
```

⟨*Process libraries: procedures*⟩+≡

```
  subroutine process_library_show (object, unit)
    class(process_library_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(A,A)")  "Process library: ", char (object%basename)
    write (u, "(2x,A,L1)")   "external        = ", object%external
    if (object%static) then
       write (u, "(2x,A,L1)")   "static          = ", .true.
    else
       write (u, "(2x,A,L1)")   "makefile exists = ", object%makefile_exists
       write (u, "(2x,A,L1)")   "driver exists   = ", object%driver_exists
    end if
    write (u, "(2x,A,A1)", advance="no")   "code status     = "
    select case (object%status)
    case (STAT_UNKNOWN);    write (u, "(A)")  "[unknown]"
    case (STAT_OPEN);       write (u, "(A)")  "open"
    case (STAT_CONFIGURED); write (u, "(A)")  "configured"
    case (STAT_SOURCE);     write (u, "(A)")  "source code exists"
    case (STAT_COMPILED);   write (u, "(A)")  "compiled"
    case (STAT_LINKED);     write (u, "(A)")  "linked"
    case (STAT_ACTIVE);     write (u, "(A)")  "active"
    end select
    call object%process_def_list_t%show (u)
```

```
end subroutine process_library_show
```

The initializer defines just the basename. We may now add process definitions to the library.

⟨*Process libraries: process library: TBP*⟩+≡
```
procedure :: init => process_library_init
```

⟨*Process libraries: procedures*⟩+≡
```
subroutine process_library_init (lib, basename)
  class(process_library_t), intent(out) :: lib
  type(string_t), intent(in) :: basename
  lib%basename = basename
  lib%status = STAT_OPEN
  call msg_message ("Process library '" // char (basename) &
       // "': initialized")
end subroutine process_library_init
```

This alternative initializer declares the library as static. We should now add process definitions to the library, but all external process code exists already. We need the driver object, and we should check the defined processes against the stored ones.

⟨*Process libraries: process library: TBP*⟩+≡
```
procedure :: init_static => process_library_init_static
```

⟨*Process libraries: procedures*⟩+≡
```
subroutine process_library_init_static (lib, basename)
  class(process_library_t), intent(out) :: lib
  type(string_t), intent(in) :: basename
  lib%basename = basename
  lib%status = STAT_OPEN
  lib%static = .true.
  call msg_message ("Static process library '" // char (basename) &
       // "': initialized")
end subroutine process_library_init_static
```

The `configure` procedure scans the allocated entries in the process definition list. The configuration proceeds in three passes.

In the first pass, we scan the process definition list and count the number of process components and the number of components which need external code. This is used to allocate the `entry` array.

In the second pass, we initialize the `entry` elements which connect process definitions, process driver objects, and external code.

In the third pass, we initialize the library driver object, allocating an entry for each external matrix element.

NOTE: Currently we handle only `initial` process components; `extra` components are ignored.

⟨*Process libraries: process library: TBP*⟩+≡
```
procedure :: configure => process_library_configure
```

⟨*Process libraries: procedures*⟩+≡
```
subroutine process_library_configure (lib, os_data)
  class(process_library_t), intent(inout) :: lib
```

```
type(os_data_t), intent(in) :: os_data
type(process_def_entry_t), pointer :: def_entry
integer :: n_entries, n_external, i_entry, i_external
type(string_t) :: model_name
integer :: i_component

n_entries = 0
n_external = 0
if (allocated (lib%entry))  deallocate (lib%entry)

def_entry => lib%first
do while (associated (def_entry))
   do i_component = 1, def_entry%n_initial
      n_entries = n_entries + 1
      if (def_entry%initial(i_component)%needs_code ()) then
         n_external = n_external + 1
         lib%external = .true.
      end if
   end do
   def_entry => def_entry%next
end do

call lib%allocate_entries (n_entries)

i_entry = 0
i_external = 0
def_entry => lib%first
do while (associated (def_entry))
   do i_component = 1, def_entry%n_initial
      i_entry = i_entry + 1
      associate (lib_entry => lib%entry(i_entry))
        lib_entry%status = STAT_CONFIGURED
        lib_entry%def => def_entry%process_def_t
        lib_entry%i_component = i_component
        if (def_entry%initial(i_component)%needs_code ()) then
           i_external = i_external + 1
           lib_entry%i_external = i_external
        end if
        call def_entry%initial(i_component)%allocate_driver &
             (lib_entry%driver)
      end associate
   end do
   def_entry => def_entry%next
end do

call dispatch_prclib_driver (lib%driver, &
     lib%basename, lib%get_modellibs_ldflags (os_data))
call lib%driver%init (n_external)
do i_entry = 1, n_entries
   associate (lib_entry => lib%entry(i_entry))
     i_component = lib_entry%i_component
     model_name = lib_entry%def%model_name
     associate (def => lib_entry%def%initial(i_component))
       if (def%needs_code ()) then
```

```
            call lib%driver%set_record (lib_entry%i_external, &
                 def%basename, &
                 model_name, &
                 def%get_features (), def%get_writer_ptr ())
         end if
       end associate
     end associate
   end do

   if (lib%static) then
      if (lib%n_entries /= 0)  lib%entry%status = STAT_LINKED
      lib%status = STAT_LINKED
   else if (lib%external) then
      where (lib%entry%i_external == 0)  lib%entry%status = STAT_LINKED
      lib%status = STAT_CONFIGURED
      lib%makefile_exists = .false.
      lib%driver_exists = .false.
   else
      if (lib%n_entries /= 0)  lib%entry%status = STAT_LINKED
      lib%status = STAT_LINKED
   end if
end subroutine process_library_configure
```

Basic setup: allocate the `entry` array.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: allocate_entries => process_library_allocate_entries
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_allocate_entries (lib, n_entries)
  class(process_library_t), intent(inout) :: lib
  integer, intent(in) :: n_entries
  lib%n_entries = n_entries
  allocate (lib%entry (n_entries))
end subroutine process_library_allocate_entries
```

Initialize an entry with data (used by unit tests).

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: init_entry => process_library_init_entry
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_init_entry (lib, i, &
     status, def, i_component, i_external, driver_template)
  class(process_library_t), intent(inout) :: lib
  integer, intent(in) :: i
  integer, intent(in) :: status
  type(process_def_t), target, intent(in) :: def
  integer, intent(in) :: i_component
  integer, intent(in) :: i_external
  class(prc_core_driver_t), intent(inout), allocatable, optional &
       :: driver_template
  call lib%entry(i)%init (status, def, i_component, i_external, &
       driver_template)
end subroutine process_library_init_entry
```

Compute the MD5 sum. We concatenate the individual MD5 sums of all processes (which, in turn, are derived from the MD5 sums of their components) and compute the MD5 sum of that.

This should be executed *after* configuration, where the driver was initialized, since otherwise the MD5 sum stored in the driver would be overwritten.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: compute_md5sum => process_library_compute_md5sum
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_compute_md5sum (lib, model)
  class(process_library_t), intent(inout) :: lib
  class(model_data_t), intent(in), optional, target :: model
  type(process_def_entry_t), pointer :: def_entry
  type(string_t) :: buffer
  buffer = lib%basename
  def_entry => lib%first
  do while (associated (def_entry))
     call def_entry%compute_md5sum (model)
     buffer = buffer // def_entry%md5sum
     def_entry => def_entry%next
  end do
  lib%md5sum = md5sum (char (buffer))
  call lib%driver%set_md5sum (lib%md5sum)
end subroutine process_library_compute_md5sum
```

Write an appropriate makefile, if there are external processes. Unless `force` is in effect, first check if there is already a makefile with the correct MD5 sum. If yes, do nothing.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: write_makefile => process_library_write_makefile
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_write_makefile (lib, os_data, force, testflag)
  class(process_library_t), intent(inout) :: lib
  type(os_data_t), intent(in) :: os_data
  logical, intent(in) :: force
  logical, intent(in), optional :: testflag
  character(32) :: md5sum_file
  logical :: generate
  integer :: unit
  if (lib%external .and. .not. lib%static) then
     generate = .true.
     if (.not. force) then
        md5sum_file = lib%driver%get_md5sum_makefile ()
        if (lib%md5sum == md5sum_file) then
           call msg_message ("Process library '" // char (lib%basename) &
                // "': keeping makefile")
           generate = .false.
        end if
     end if
     if (generate) then
        call msg_message ("Process library '" // char (lib%basename) &
             // "': writing makefile")
        unit = free_unit ()
```

```
         open (unit, file = char (lib%driver%basename // ".makefile"), &
              status="replace", action="write")
         call lib%driver%generate_makefile (unit, os_data, testflag)
         close (unit)
      end if
      lib%makefile_exists = .true.
   end if
end subroutine process_library_write_makefile
```

Write the driver source code for the library to file, if there are external processes.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: write_driver => process_library_write_driver
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_write_driver (lib, force)
  class(process_library_t), intent(inout) :: lib
  logical, intent(in) :: force
  character(32) :: md5sum_file
  logical :: generate
  integer :: unit
  if (lib%external .and. .not. lib%static) then
     generate = .true.
     if (.not. force) then
        md5sum_file = lib%driver%get_md5sum_driver ()
        if (lib%md5sum == md5sum_file) then
           call msg_message ("Process library '" // char (lib%basename) &
                // "': keeping driver")
           generate = .false.
        end if
     end if
     if (generate) then
        call msg_message ("Process library '" // char (lib%basename) &
             // "': writing driver")
        unit = free_unit ()
        open (unit, file = char (lib%driver%basename // ".f90"), &
             status="replace", action="write")
        call lib%driver%generate_driver_code (unit)
        close (unit)
     end if
     lib%driver_exists = .true.
  end if
end subroutine process_library_write_driver
```

Update the compilation status of an external library.

Strictly speaking, this is not necessary for a one-time run, since the individual library methods will update the status themselves. However, it allows us to identify compilation steps that we can skip because the file exists or is already loaded, for the whole library or for particular entries.

Independently, the building process is controlled by a makefile. Thus, previous files are reused if they are not modified by the current compilation.

1. If it is not already loaded, attempt to load the library. If successful, check the overall MD5 sum. If it matches, just keep it loaded and mark as

ACTIVE. If not, check the MD5 sum for all linked process components. Where it matches, mark the entry as COMPILED. Then, unload the library and mark as CONFIGURED.

Thus, we can identify compiled files for all matrix elements which are accessible via the previous compiled library, even if it is no longer up to date.

2. If the library is now in CONFIGURED state, look for valid source files. Each entry that is just in CONFIGURED state will advance to SOURCE if the MD5 sum matches. Finally, advance the whole library to SOURCE if all entries are at least in this condition.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: update_status => process_library_update_status
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_update_status (lib, os_data)
  class(process_library_t), intent(inout) :: lib
  type(os_data_t), intent(in) :: os_data
  character(32) :: md5sum_file
  integer :: i, i_external, i_component
  if (lib%external) then
     select case (lib%status)
     case (STAT_CONFIGURED:STAT_LINKED)
        call lib%driver%load (os_data, noerror=.true.)
     end select
     if (lib%driver%loaded) then
        md5sum_file = lib%driver%get_md5sum (0)
        if (lib%md5sum == md5sum_file) then
           call lib%load_entries ()
           lib%entry%status = STAT_ACTIVE
           lib%status = STAT_ACTIVE
           call msg_message ("Process library '" // char (lib%basename) &
                // "': active")
        else
           do i = 1, lib%n_entries
              associate (entry => lib%entry(i))
                i_external = entry%i_external
                i_component = entry%i_component
                if (i_external /= 0) then
                   md5sum_file = lib%driver%get_md5sum (i_external)
                   if (entry%def%get_md5sum (i_component) == md5sum_file) then
                      entry%status = STAT_COMPILED
                   else
                      entry%status = STAT_CONFIGURED
                   end if
                end if
              end associate
           end do
           call lib%driver%unload ()
           lib%status = STAT_CONFIGURED
        end if
     end if
     select case (lib%status)
```

```
      case (STAT_CONFIGURED)
         do i = 1, lib%n_entries
            associate (entry => lib%entry(i))
              i_external = entry%i_external
              i_component = entry%i_component
              if (i_external /= 0) then
                 select case (entry%status)
                 case (STAT_CONFIGURED)
                    md5sum_file = lib%driver%get_md5sum_source (i_external)
                    if (entry%def%get_md5sum (i_component) == md5sum_file) then
                       entry%status = STAT_SOURCE
                    end if
                 end select
              end if
            end associate
         end do
         if (all (lib%entry%status >= STAT_SOURCE)) then
            md5sum_file = lib%driver%get_md5sum_driver ()
            if (lib%md5sum == md5sum_file) then
               lib%status = STAT_SOURCE
            end if
         end if
      end select
   end if
end subroutine process_library_update_status
```

This procedure triggers code generation for all processes where this is possible.

We generate code only for external processes of status `STAT_CONFIGURED`, which then advance to `STAT_SOURCE`. If, for a particular process, the status is already advanced, we do not remove previous files, so `make` will consider them as up to date if they exist. Otherwise, we remove those files to force a fresh `make`.

Finally, if any source code has been generated, we need a driver file.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: make_source => process_library_make_source
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_make_source (lib, os_data, keep_old_source)
  class(process_library_t), intent(inout) :: lib
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: keep_old_source
  logical :: keep_old
  integer :: i, i_external
  keep_old = .false.
  if (present (keep_old_source))  keep_old = keep_old_source
  if (lib%external .and. .not. lib%static) then
     select case (lib%status)
     case (STAT_CONFIGURED)
        if (keep_old) then
           call msg_message ("Process library '" // char (lib%basename) &
                // "': keeping source code")
        else
           call msg_message ("Process library '" // char (lib%basename) &
```

```
            // "': creating source code")
         do i = 1, size (lib%entry)
            associate (entry => lib%entry(i))
              i_external = entry%i_external
              if (i_external /= 0 &
                   .and. lib%entry(i)%status == STAT_CONFIGURED) then
                 call lib%driver%clean_proc (i_external, os_data)
              end if
            end associate
            if (signal_is_pending ())  return
         end do
         call lib%driver%make_source (os_data)
      end if
      lib%status = STAT_SOURCE
      where (lib%entry%i_external /= 0 &
           .and. lib%entry%status == STAT_CONFIGURED)
         lib%entry%status = STAT_SOURCE
      end where
      lib%status = STAT_SOURCE
   end select
  end if
end subroutine process_library_make_source
```

Compile the generated code and update the status codes. Try to make the sources first, just in case. This includes compiling possible LaTeXFeynman diagram files.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: make_compile => process_library_make_compile
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_make_compile (lib, os_data, keep_old_source)
  class(process_library_t), intent(inout) :: lib
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: keep_old_source
  if (lib%external .and. .not. lib%static) then
     select case (lib%status)
     case (STAT_CONFIGURED)
        call lib%make_source (os_data, keep_old_source)
     end select
     if (signal_is_pending ())  return
     select case (lib%status)
     case (STAT_SOURCE)
        call msg_message ("Process library '" // char (lib%basename) &
             // "': compiling sources")
        call lib%driver%make_compile (os_data)
        where (lib%entry%i_external /= 0 &
             .and. lib%entry%status == STAT_SOURCE)
           lib%entry%status = STAT_COMPILED
        end where
        lib%status = STAT_COMPILED
     end select
  end if
end subroutine process_library_make_compile
```

Link the process library. Try to compile first, just in case.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: make_link => process_library_make_link
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_make_link (lib, os_data, keep_old_source)
  class(process_library_t), intent(inout) :: lib
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: keep_old_source
  if (lib%external .and. .not. lib%static) then
     select case (lib%status)
     case (STAT_CONFIGURED:STAT_SOURCE)
        call lib%make_compile (os_data, keep_old_source)
     end select
     if (signal_is_pending ())  return
     select case (lib%status)
     case (STAT_COMPILED)
        call msg_message ("Process library '" // char (lib%basename) &
             // "': linking")
        call lib%driver%make_link (os_data)
        lib%entry%status = STAT_LINKED
        lib%status = STAT_LINKED
     end select
  end if
end subroutine process_library_make_link
```

Load the process library, i.e., assign pointers to the library functions.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: load => process_library_load
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_load (lib, os_data, keep_old_source)
  class(process_library_t), intent(inout) :: lib
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: keep_old_source
  select case (lib%status)
  case (STAT_CONFIGURED:STAT_COMPILED)
     call lib%make_link (os_data, keep_old_source)
  end select
  if (signal_is_pending ())  return
  select case (lib%status)
  case (STAT_LINKED)
     if (lib%external) then
        call msg_message ("Process library '" // char (lib%basename) &
             // "': loading")
        call lib%driver%load (os_data)
        call lib%load_entries ()
     end if
     lib%entry%status = STAT_ACTIVE
     lib%status = STAT_ACTIVE
  end select
end subroutine process_library_load
```

This is the actual loading part for the process methods.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: load_entries => process_library_load_entries
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_load_entries (lib)
  class(process_library_t), intent(inout) :: lib
  integer :: i
  do i = 1, size (lib%entry)
     associate (entry => lib%entry(i))
       if (entry%i_external /= 0) then
          call entry%connect (lib%driver, entry%i_external)
       end if
     end associate
  end do
end subroutine process_library_load_entries
```

Unload the library, if possible. This reverts the status to "linked".

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: unload => process_library_unload
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_unload (lib)
  class(process_library_t), intent(inout) :: lib
  select case (lib%status)
  case (STAT_ACTIVE)
     if (lib%external) then
        call msg_message ("Process library '" // char (lib%basename) &
             // "': unloading")
        call lib%driver%unload ()
     end if
     lib%entry%status = STAT_LINKED
     lib%status = STAT_LINKED
  end select
end subroutine process_library_unload
```

Unload, clean all generated files and revert the library status. If `distclean` is set, also remove the makefile and the driver source.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: clean => process_library_clean
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_clean (lib, os_data, distclean)
  class(process_library_t), intent(inout) :: lib
  type(os_data_t), intent(in) :: os_data
  logical, intent(in) :: distclean
  call lib%unload ()
  if (lib%external .and. .not. lib%static) then
     call msg_message ("Process library '" // char (lib%basename) &
          // "': removing old files")
     if (distclean) then
        call lib%driver%distclean (os_data)
     else
        call lib%driver%clean (os_data)
```

```
      end if
    end if
    where (lib%entry%i_external /= 0)
      lib%entry%status = STAT_CONFIGURED
    elsewhere
      lib%entry%status = STAT_LINKED
    end where
    if (lib%external) then
      lib%status = STAT_CONFIGURED
    else
      lib%status = STAT_LINKED
    end if
  end subroutine process_library_clean
```

Unload and clean the compiled library and all generated object files, so force a recompilation.

⟨*XXX Process libraries: process library: TBP*⟩≡

```
  procedure :: clean_objects => process_library_clean_objects
```

⟨*XXX Process libraries: procedures*⟩≡

```
  subroutine process_library_clean_objects (lib, os_data)
    class(process_library_t), intent(inout) :: lib
    type(os_data_t), intent(in) :: os_data
    call lib%unload ()
    if (lib%external .and. .not. lib%static) then
      call msg_message ("Process library '" // char (lib%basename) &
          // "': removing compiled objects")
      call lib%driver%clean_library (os_data)
      call lib%driver%clean_objects (os_data)
    end if
    where (lib%entry%i_external /= 0)
      lib%entry%status = STAT_CONFIGURED
    elsewhere
      lib%entry%status = STAT_LINKED
    end where
    if (lib%external) then
      lib%status = STAT_CONFIGURED
    else
      lib%status = STAT_LINKED
    end if
  end subroutine process_library_clean_objects
```

Unload and revert the library status to INITIAL. This allows for appending new processes. No files are deleted.

⟨*Process libraries: process library: TBP*⟩+≡

```
  procedure :: open => process_library_open
```

⟨*Process libraries: procedures*⟩+≡

```
  subroutine process_library_open (lib)
    class(process_library_t), intent(inout) :: lib
    select case (lib%status)
    case (STAT_OPEN)
    case default
      call lib%unload ()
```

```
      if (.not. lib%static) then
         lib%entry%status = STAT_OPEN
         lib%status = STAT_OPEN
         if (lib%external)  lib%update_counter = lib%update_counter + 1
         call msg_message ("Process library '" // char (lib%basename) &
              // "': open")
      else
         call msg_error ("Static process library '" // char (lib%basename) &
              // "': processes can't be appended")
      end if
    end select
  end subroutine process_library_open
```

13.4.4 Use the library

Return the base name of the library

⟨*Process libraries: process library: TBP*⟩+≡

```
  procedure :: get_name => process_library_get_name
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_library_get_name (lib) result (name)
    class(process_library_t), intent(in) :: lib
    type(string_t) :: name
    name = lib%basename
  end function process_library_get_name
```

Once activated, we view the process library object as an interface for accessing the matrix elements.

⟨*Process libraries: process library: TBP*⟩+≡

```
  procedure :: is_active => process_library_is_active
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_library_is_active (lib) result (flag)
    logical :: flag
    class(process_library_t), intent(in) :: lib
    flag = lib%status == STAT_ACTIVE
  end function process_library_is_active
```

Return the current status code of the library. If an index is provided, return the status of that entry.

⟨*Process libraries: process library: TBP*⟩+≡

```
  procedure :: get_status => process_library_get_status
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_library_get_status (lib, i) result (status)
    class(process_library_t), intent(in) :: lib
    integer, intent(in), optional :: i
    integer :: status
    if (present (i)) then
       status = lib%entry(i)%status
    else
       status = lib%status
```

```
    end if
  end function process_library_get_status
```

Return the update counter. Since this is incremented each time the library is re-opened, we can use this to check if existing pointers to matrix element code are still valid.

⟨*Process libraries: process library: TBP*⟩+≡

```
  procedure :: get_update_counter => process_library_get_update_counter
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_library_get_update_counter (lib) result (counter)
    class(process_library_t), intent(in) :: lib
    integer :: counter
    counter = lib%update_counter
  end function process_library_get_update_counter
```

Manually set the current status code of the library. If the optional flag is set, set also the entry status codes. This is used for unit tests.

⟨*Process libraries: process library: TBP*⟩+≡

```
  procedure :: set_status => process_library_set_status
```

⟨*Process libraries: procedures*⟩+≡

```
  subroutine process_library_set_status (lib, status, entries)
    class(process_library_t), intent(inout) :: lib
    integer, intent(in) :: status
    logical, intent(in), optional :: entries
    lib%status = status
    if (present (entries)) then
       if (entries)  lib%entry%status = status
    end if
  end subroutine process_library_set_status
```

Return the load status of the associated driver.

⟨*Process libraries: process library: TBP*⟩+≡

```
  procedure :: is_loaded => process_library_is_loaded
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_library_is_loaded (lib) result (flag)
    class(process_library_t), intent(in) :: lib
    logical :: flag
    flag = lib%driver%loaded
  end function process_library_is_loaded
```

Retrieve constants using the process library driver. We assume that the process code has been loaded, if external.

⟨*Process libraries: process library entry: TBP*⟩+≡

```
  procedure :: fill_constants => process_library_entry_fill_constants
```

⟨*Process libraries: procedures*⟩+≡

```
  subroutine process_library_entry_fill_constants (entry, driver, data)
    class(process_library_entry_t), intent(in) :: entry
    class(prclib_driver_t), intent(in) :: driver
    type(process_constants_t), intent(out) :: data
```

```
    integer :: i
    if (entry%i_external /= 0) then
       i = entry%i_external
       data%id         = driver%get_process_id (i)
       data%model_name = driver%get_model_name (i)
       data%md5sum     = driver%get_md5sum (i)
       data%openmp_supported = driver%get_openmp_status (i)
       data%n_in  = driver%get_n_in  (i)
       data%n_out = driver%get_n_out (i)
       data%n_flv = driver%get_n_flv (i)
       data%n_hel = driver%get_n_hel (i)
       data%n_col = driver%get_n_col (i)
       data%n_cin = driver%get_n_cin (i)
       data%n_cf  = driver%get_n_cf  (i)
       call driver%set_flv_state (i, data%flv_state)
       call driver%set_hel_state (i, data%hel_state)
       call driver%set_col_state (i, data%col_state, data%ghost_flag)
       call driver%set_color_factors (i, data%color_factors, data%cf_index)
    else
       select type (proc_driver => entry%driver)
       class is (process_driver_internal_t)
          call proc_driver%fill_constants (data)
       end select
    end if
  end subroutine process_library_entry_fill_constants
```

Retrieve the constants for a process

⟨*Process libraries: process library: TBP*⟩+≡

```
  procedure :: fill_constants => process_library_fill_constants
```

⟨*Process libraries: procedures*⟩+≡

```
  subroutine process_library_fill_constants (lib, id, i_component, data)
    class(process_library_t), intent(in) :: lib
    type(string_t), intent(in) :: id
    integer, intent(in) :: i_component
    type(process_constants_t), intent(out) :: data
    integer :: i
    do i = 1, size (lib%entry)
       associate (entry => lib%entry(i))
          if (entry%def%id == id .and. entry%i_component == i_component) then
             call entry%fill_constants (lib%driver, data)
             return
          end if
       end associate
    end do
  end subroutine process_library_fill_constants
```

Retrieve the constants and a connected driver for a process, identified by a process ID and a subprocess index. We scan the process entries until we have found a match.

⟨*Process libraries: process library: TBP*⟩+≡

```
  procedure :: connect_process => process_library_connect_process
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_connect_process &
     (lib, id, i_component, data, proc_driver)
  class(process_library_t), intent(in) :: lib
  type(string_t), intent(in) :: id
  integer, intent(in) :: i_component
  type(process_constants_t), intent(out) :: data
  class(prc_core_driver_t), allocatable, intent(out) :: proc_driver
  integer :: i
  do i = 1, size (lib%entry)
     associate (entry => lib%entry(i))
       if (entry%def%id == id .and. entry%i_component == i_component) then
          call entry%fill_constants (lib%driver, data)
          allocate (proc_driver, source = entry%driver)
          return
       end if
     end associate
  end do
  call msg_fatal ("Process library '" // char (lib%basename) &
             // "': process '" // char (id) // "' not found")
end subroutine process_library_connect_process
```

Shortcut for use in unit tests: fetch the MD5sum from a specific library entry and inject it into the writer of a specific record.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: test_transfer_md5sum => process_library_test_transfer_md5sum
```

⟨*Process libraries: procedures*⟩+≡

```
subroutine process_library_test_transfer_md5sum (lib, r, e, c)
  class(process_library_t), intent(inout) :: lib
  integer, intent(in) :: r, e, c
  associate (writer => lib%driver%record(r)%writer)
     writer%md5sum = lib%entry(e)%def%get_md5sum (c)
  end associate
end subroutine process_library_test_transfer_md5sum
```

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: get_nlo_type => process_library_get_nlo_type
```

⟨*Process libraries: procedures*⟩+≡

```
function process_library_get_nlo_type (lib, id, i_component) result (nlo_type)
  integer :: nlo_type
  class(process_library_t), intent(in) :: lib
  type(string_t), intent(in) :: id
  integer, intent(in) :: i_component
  integer :: i
  do i = 1, size (lib%entry)
     if (lib%entry(i)%def%id == id .and. lib%entry(i)%i_component == i_component) then
        nlo_type = lib%entry(i)%def%get_nlo_type (i_component)
        exit
     end if
  end do
end function process_library_get_nlo_type
```

13.4.5 Collect model-specific libraries

This returns appropriate linker flags for the model parameter libraries that are used by the generated matrix element. At the end, the main libwhizard is appended (again), because functions from that may be reqired.

Extra models in the local user space need to be treated individually.

⟨*Process libraries: process library: TBP*⟩+≡

```
procedure :: get_modellibs_ldflags => process_library_get_modellibs_ldflags
```

⟨*Process libraries: procedures*⟩+≡

```
function process_library_get_modellibs_ldflags (prc_lib, os_data) result (flags)
  class(process_library_t), intent(in) :: prc_lib
  type(os_data_t), intent(in) :: os_data
  type(string_t) :: flags
  type(string_t), dimension(:), allocatable :: models
  type(string_t) :: modelname, modellib, modellib_full
  logical :: exist
  integer :: i, j, mi
  flags = " -lomega"
  if ((.not. os_data%use_testfiles) .and. &
            os_dir_exist (os_data%whizard_models_libpath_local)) &
              flags = flags // " -L" // os_data%whizard_models_libpath_local
  flags = flags // " -L" // os_data%whizard_models_libpath
  allocate (models(prc_lib%n_entries + 1))
  models = ""
  mi = 1
  if (allocated (prc_lib%entry)) then
     SCAN: do i = 1, prc_lib%n_entries
        if (associated (prc_lib%entry(i)%def)) then
           if (prc_lib%entry(i)%def%model_name /= "") then
              modelname = prc_lib%entry(i)%def%model_name
           else
              cycle SCAN
           end if
        else
           cycle SCAN
        end if
        do j = 1, mi
           if (models(mi) == modelname) cycle SCAN
        end do
        models(mi) = modelname
        mi = mi + 1
        if (os_data%use_libtool) then
           modellib = "libparameters_" // modelname // ".la"
        else
           modellib = "libparameters_" // modelname // ".a"
        end if
        exist = .false.
        if (.not. os_data%use_testfiles) then
           modellib_full = os_data%whizard_models_libpath_local &
                // "/" // modellib
           inquire (file=char (modellib_full), exist=exist)
        end if
        if (.not. exist) then
           modellib_full = os_data%whizard_models_libpath &
```

```
                 // "/" // modellib
              inquire (file=char (modellib_full), exist=exist)
           end if
           if (exist) flags = flags // " -lparameters_" // modelname
        end do SCAN
     end if
     deallocate (models)
     flags = flags // " -lwhizard"
  end function process_library_get_modellibs_ldflags
```

⟨*Process libraries: process library: TBP*⟩+≡

```
  procedure :: get_static_modelname => process_library_get_static_modelname
```

⟨*Process libraries: procedures*⟩+≡

```
  function process_library_get_static_modelname (prc_lib, os_data) result (name)
    class(process_library_t), intent(in) :: prc_lib
    type(os_data_t), intent(in) :: os_data
    type(string_t) :: name
    type(string_t), dimension(:), allocatable :: models
    type(string_t) :: modelname, modellib, modellib_full
    logical :: exist
    integer :: i, j, mi
    name = ""
    allocate (models(prc_lib%n_entries + 1))
    models = ""
    mi = 1
    if (allocated (prc_lib%entry)) then
       SCAN: do i = 1, prc_lib%n_entries
          if (associated (prc_lib%entry(i)%def)) then
             if (prc_lib%entry(i)%def%model_name /= "") then
                modelname = prc_lib%entry(i)%def%model_name
             else
                cycle SCAN
             end if
          else
             cycle SCAN
          end if
          do j = 1, mi
             if (models(mi) == modelname) cycle SCAN
          end do
          models(mi) = modelname
          mi = mi + 1
          modellib = "libparameters_" // modelname // ".a"
          exist = .false.
          if (.not. os_data%use_testfiles) then
             modellib_full = os_data%whizard_models_libpath_local &
                  // "/" // modellib
             inquire (file=char (modellib_full), exist=exist)
          end if
          if (.not. exist) then
             modellib_full = os_data%whizard_models_libpath &
                  // "/" // modellib
             inquire (file=char (modellib_full), exist=exist)
          end if
```

```
          if (exist) name = name // " " // modellib_full
        end do SCAN
      end if
      deallocate (models)
    end function process_library_get_static_modelname
```

13.4.6 Unit Test

Test module, followed by the corresponding implementation module.

⟨process_libraries_ut.f90⟩≡

```
  ⟨File header⟩

  module process_libraries_ut
    use unit_tests
    use process_libraries_uti

  ⟨Standard module head⟩

  ⟨Process libraries: public test⟩

  contains

  ⟨Process libraries: test driver⟩

  end module process_libraries_ut
```

⟨process_libraries_uti.f90⟩≡

```
  ⟨File header⟩

  module process_libraries_uti

    use, intrinsic :: iso_c_binding !NODEP!

  ⟨Use strings⟩
    use io_units
    use os_interface
    use particle_specifiers, only: new_prt_spec
    use process_constants
    use prclib_interfaces
    use prc_core_def

    use process_libraries

    use prclib_interfaces_ut, only: test_writer_4_t

  ⟨Standard module head⟩

  ⟨Process libraries: test declarations⟩

  ⟨Process libraries: test types⟩

  contains
```

⟨*Process libraries: tests*⟩

⟨*Process libraries: test auxiliary*⟩

```
end module process_libraries_uti
```

API: driver for the unit tests below.

⟨*Process libraries: public test*⟩≡
```
public :: process_libraries_test
```

⟨*Process libraries: test driver*⟩≡
```
subroutine process_libraries_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```
⟨*Process libraries: execute tests*⟩
```
end subroutine process_libraries_test
```

Empty process list

Test 1: Write an empty process list.

⟨*Process libraries: execute tests*⟩≡
```
call test (process_libraries_1, "process_libraries_1", &
     "empty process list", &
     u, results)
```

⟨*Process libraries: test declarations*⟩≡
```
public :: process_libraries_1
```

⟨*Process libraries: tests*⟩≡
```
subroutine process_libraries_1 (u)
  integer, intent(in) :: u
  type(process_def_list_t) :: process_def_list

  write (u, "(A)")  "* Test output: process_libraries_1"
  write (u, "(A)")  "*   Purpose: Display an empty process definition list"
  write (u, "(A)")

  call process_def_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: process_libraries_1"
end subroutine process_libraries_1
```

Process definition list

Test 2: Process definition list with processes and components. Construct the list, write to file, read it in again, and display. Finalize and delete the list after use.

We define a trivial 'test' type for the process variant. The test type contains just one (meaningless) data item, which is an integer.

⟨*Process libraries: test types*⟩≡
```
type, extends (prc_core_def_t) :: prcdef_2_t
```

```
    integer :: data = 0
    logical :: file = .false.
  contains
  ⟨Process libraries: prcdef 2: TBP⟩
end type prcdef_2_t
```

The process variant is named 'test'.

⟨*Process libraries: prcdef 2: TBP*⟩≡

```
procedure, nopass :: type_string => prcdef_2_type_string
```

⟨*Process libraries: test auxiliary*⟩≡

```
function prcdef_2_type_string () result (string)
  type(string_t) :: string
  string = "test"
end function prcdef_2_type_string
```

Write the contents (the integer value).

⟨*Process libraries: prcdef 2: TBP*⟩+≡

```
procedure :: write => prcdef_2_write
```

⟨*Process libraries: test auxiliary*⟩+≡

```
subroutine prcdef_2_write (object, unit)
  class(prcdef_2_t), intent(in) :: object
  integer, intent(in) :: unit
  write (unit, "(3x,A,I0)")  "Test data         = ", object%data
end subroutine prcdef_2_write
```

Recover the integer value.

⟨*Process libraries: prcdef 2: TBP*⟩+≡

```
procedure :: read => prcdef_2_read
```

⟨*Process libraries: test auxiliary*⟩+≡

```
subroutine prcdef_2_read (object, unit)
  class(prcdef_2_t), intent(out) :: object
  integer, intent(in) :: unit
  character(80) :: buffer
  read (unit, "(A)")  buffer
  call strip_equation_lhs (buffer)
  read (buffer, *)  object%data
end subroutine prcdef_2_read
```

No external procedures.

⟨*Process libraries: prcdef 2: TBP*⟩+≡

```
procedure, nopass :: get_features => prcdef_2_get_features
```

⟨*Process libraries: test auxiliary*⟩+≡

```
subroutine prcdef_2_get_features (features)
  type(string_t), dimension(:), allocatable, intent(out) :: features
  allocate (features (0))
end subroutine prcdef_2_get_features
```

No code generated.

⟨*Process libraries: prcdef 2: TBP*⟩+≡

```
procedure :: generate_code => prcdef_2_generate_code
```

⟨*Process libraries: test auxiliary*⟩+≡

```
subroutine prcdef_2_generate_code (object, &
     basename, model_name, prt_in, prt_out)
  class(prcdef_2_t), intent(in) :: object
  type(string_t), intent(in) :: basename
  type(string_t), intent(in) :: model_name
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
end subroutine prcdef_2_generate_code
```

Allocate the driver with the appropriate type.

⟨*Process libraries: prcdef 2: TBP*⟩+≡

```
procedure :: allocate_driver => prcdef_2_allocate_driver
```

⟨*Process libraries: test auxiliary*⟩+≡

```
subroutine prcdef_2_allocate_driver (object, driver, basename)
  class(prcdef_2_t), intent(in) :: object
  class(prc_core_driver_t), intent(out), allocatable :: driver
  type(string_t), intent(in) :: basename
  allocate (prctest_2_t :: driver)
end subroutine prcdef_2_allocate_driver
```

Nothing to connect.

⟨*Process libraries: prcdef 2: TBP*⟩+≡

```
procedure :: connect => prcdef_2_connect
```

⟨*Process libraries: test auxiliary*⟩+≡

```
subroutine prcdef_2_connect (def, lib_driver, i, proc_driver)
  class(prcdef_2_t), intent(in) :: def
  class(prclib_driver_t), intent(in) :: lib_driver
  integer, intent(in) :: i
  class(prc_core_driver_t), intent(inout) :: proc_driver
end subroutine prcdef_2_connect
```

The associated driver type.

⟨*Process libraries: test types*⟩+≡

```
type, extends (process_driver_internal_t) :: prctest_2_t
 contains
 ⟨Process libraries: prctest 2: TBP⟩
end type prctest_2_t
```

Return the type name.

⟨*Process libraries: prctest 2: TBP*⟩≡

```
procedure, nopass :: type_name => prctest_2_type_name
```

⟨*Process libraries: test auxiliary*⟩+≡

```
function prctest_2_type_name () result (type)
  type(string_t) :: type
  type = "test"
end function prctest_2_type_name
```

This should fill constant process data. We do not check those here, however, therefore nothing done.

⟨*Process libraries: prctest 2: TBP*⟩+≡
```
procedure :: fill_constants => prctest_2_fill_constants
```

⟨*Process libraries: test auxiliary*⟩+≡
```
subroutine prctest_2_fill_constants (driver, data)
  class(prctest_2_t), intent(in) :: driver
  type(process_constants_t), intent(out) :: data
end subroutine prctest_2_fill_constants
```

Here is the actual test.

For reading, we need a list of templates, i.e., an array containing allocated objects for all available process variants. This is the purpose of `process_core_templates`. Here, we have only a single template for the 'test' variant.

⟨*Process libraries: execute tests*⟩+≡
```
call test (process_libraries_2, "process_libraries_2", &
     "process definition list", &
     u, results)
```

⟨*Process libraries: test declarations*⟩+≡
```
public :: process_libraries_2
```

⟨*Process libraries: tests*⟩+≡
```
subroutine process_libraries_2 (u)
  integer, intent(in) :: u
  type(prc_template_t), dimension(:), allocatable :: process_core_templates
  type(process_def_list_t) :: process_def_list
  type(process_def_entry_t), pointer :: entry => null ()
  class(prc_core_def_t), allocatable :: test_def
  integer :: scratch_unit

  write (u, "(A)")  "* Test output: process_libraries_2"
  write (u, "(A)")  "* Purpose: Construct a process definition list,"
  write (u, "(A)")  "*          write it to file and reread it"
  write (u, "(A)")  ""
  write (u, "(A)")  "* Construct a process definition list"
  write (u, "(A)")  "*   First process definition: empty"
  write (u, "(A)")  "*   Second process definition: two components"
  write (u, "(A)")  "*     First component: empty"
  write (u, "(A)")  "*     Second component: test data"
  write (u, "(A)")  "*   Third process definition:"
  write (u, "(A)")  "*     Embedded decays and polarization"
  write (u, "(A)")

  allocate (process_core_templates (1))
  allocate (prcdef_2_t :: process_core_templates(1)%core_def)

  allocate (entry)
  call entry%init (var_str ("first"), n_in = 0, n_components = 0)
  call entry%compute_md5sum ()
  call process_def_list%append (entry)

  allocate (entry)
  call entry%init (var_str ("second"), model_name = var_str ("Test"), &
```

```
     n_in = 1, n_components = 2)
  allocate (prcdef_2_t :: test_def)
  select type (test_def)
  type is (prcdef_2_t);  test_def%data = 42
  end select
  call entry%import_component (2, n_out = 2, &
     prt_in  = new_prt_spec ([var_str ("a")]), &
     prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
     method  = var_str ("test"), &
     variant = test_def)
  call entry%compute_md5sum ()
  call process_def_list%append (entry)

  allocate (entry)
  call entry%init (var_str ("third"), model_name = var_str ("Test"), &
     n_in = 2, n_components = 1)
  allocate (prcdef_2_t :: test_def)
  call entry%import_component (1, n_out = 3, &
     prt_in  = &
       new_prt_spec ([var_str ("a"), var_str ("b")]), &
     prt_out = &
       [new_prt_spec (var_str ("c")), &
        new_prt_spec (var_str ("d"), .true.), &
        new_prt_spec (var_str ("e"), [var_str ("e_decay")])], &
     method  = var_str ("test"), &
     variant = test_def)
  call entry%compute_md5sum ()
  call process_def_list%append (entry)
  call process_def_list%write (u)

  write (u, "(A)")  ""
  write (u, "(A)")  "* Write the process definition list to (scratch) file"

  scratch_unit = free_unit ()
  open (unit = scratch_unit, status="scratch", action = "readwrite")
  call process_def_list%write (scratch_unit)
  call process_def_list%final ()

  write (u, "(A)")  "* Reread it"
  write (u, "(A)")  ""

  rewind (scratch_unit)
  call process_def_list%read (scratch_unit, process_core_templates)
  close (scratch_unit)

  call process_def_list%write (u)
  call process_def_list%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: process_libraries_2"
end subroutine process_libraries_2
```

Process library object

Test 3: Process library object with several process definitions and library entries. Just construct the object, modify some initial content, and write the result. The modifications are mostly applied directly, so we do not test anything but the contents and the output routine.

⟨*Process libraries: execute tests*⟩+≡

```
call test (process_libraries_3, "process_libraries_3", &
     "recover process definition list from file", &
     u, results)
```

⟨*Process libraries: test declarations*⟩+≡

```
public :: process_libraries_3
```

⟨*Process libraries: tests*⟩+≡

```
subroutine process_libraries_3 (u)
  integer, intent(in) :: u
  type(process_library_t) :: lib
  type(process_def_entry_t), pointer :: entry
  class(prc_core_driver_t), allocatable :: driver_template

  write (u, "(A)")  "* Test output: process_libraries_3"
  write (u, "(A)")  "* Purpose: Construct a process library object &
       &with entries"
  write (u, "(A)")  ""
  write (u, "(A)")  "* Construct and display a process library object"
  write (u, "(A)")  "*   with 5 entries"
  write (u, "(A)")  "*   associated with 3 matrix element codes"
  write (u, "(A)")  "*   corresponding to 3 process definitions"
  write (u, "(A)")  "*   with 2, 1, 1 components, respectively"
  write (u, "(A)")

  call lib%init (var_str ("testlib"))

  call lib%set_status (STAT_ACTIVE)
  call lib%allocate_entries (5)

  allocate (entry)
  call entry%init (var_str ("test_a"), n_in = 2, n_components = 2)
  allocate (prctest_2_t :: driver_template)
  call lib%init_entry (3, STAT_SOURCE, entry%process_def_t, 2, 2, &
       driver_template)
  call lib%init_entry (4, STAT_COMPILED, entry%process_def_t, 1, 0)
  call lib%append (entry)

  allocate (entry)
  call entry%init (var_str ("test_b"), n_in = 2, n_components = 1)
  call lib%init_entry (2, STAT_CONFIGURED, entry%process_def_t, 1, 1)
  call lib%append (entry)

  allocate (entry)
  call entry%init (var_str ("test_c"), n_in = 2, n_components = 1)
  allocate (prctest_2_t :: driver_template)
  call lib%init_entry (5, STAT_LINKED, entry%process_def_t, 1, 3, &
       driver_template)
```

```
    call lib%append (entry)

    call lib%write (u)
    call lib%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: process_libraries_3"
  end subroutine process_libraries_3
```

Process library for test matrix element (no file)

Test 4: We proceed through the library generation and loading phases with a test matrix element type that needs no code written on file.

⟨*Process libraries: execute tests*⟩+≡

```
  call test (process_libraries_4, "process_libraries_4", &
      "build and load internal process library", &
      u, results)
```

⟨*Process libraries: test declarations*⟩+≡

```
  public :: process_libraries_4
```

⟨*Process libraries: tests*⟩+≡

```
  subroutine process_libraries_4 (u)
    integer, intent(in) :: u
    type(process_library_t) :: lib
    type(process_def_entry_t), pointer :: entry
    class(prc_core_def_t), allocatable :: core_def
    type(os_data_t) :: os_data

    write (u, "(A)")  "* Test output: process_libraries_4"
    write (u, "(A)")  "* Purpose: build a process library with an &
        &internal (pseudo) matrix element"
    write (u, "(A)")  "*          No Makefile or code should be generated"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize a process library with one entry &
        &(no external code)"
    write (u, "(A)")
    call os_data_init (os_data)
    call lib%init (var_str ("proclibs4"))

    allocate (prcdef_2_t :: core_def)

    allocate (entry)
    call entry%init (var_str ("proclibs4_a"), n_in = 1, n_components = 1)
    call entry%import_component (1, n_out = 2, variant = core_def)
    call lib%append (entry)

    write (u, "(A)")  "* Configure library"
    write (u, "(A)")
    call lib%configure (os_data)

    write (u, "(A)")  "* Compute MD5 sum"
```

```
  write (u, "(A)")
  call lib%compute_md5sum ()

  write (u, "(A)")  "* Write makefile (no-op)"
  write (u, "(A)")
  call lib%write_makefile (os_data, force = .true.)

  write (u, "(A)")  "* Write driver source code (no-op)"
  write (u, "(A)")
  call lib%write_driver (force = .true.)

  write (u, "(A)")  "* Write process source code (no-op)"
  write (u, "(A)")
  call lib%make_source (os_data)

  write (u, "(A)")  "* Compile (no-op)"
  write (u, "(A)")
  call lib%make_compile (os_data)

  write (u, "(A)")  "* Link (no-op)"
  write (u, "(A)")
  call lib%make_link (os_data)

  write (u, "(A)")  "* Load (no-op)"
  write (u, "(A)")
  call lib%load (os_data)

  call lib%write (u)
  call lib%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: process_libraries_4"
end subroutine process_libraries_4
```

Build workflow for test matrix element

Test 5: We write source code for a dummy process.

We define another trivial type for the process variant. The test type contains just no variable data, but produces code on file.

⟨*Process libraries: test types*⟩+≡

```
type, extends (prc_core_def_t) :: prcdef_5_t
 contains
 ⟨Process libraries: prcdef 5: TBP⟩
end type prcdef_5_t
```

The process variant is named `test_file`.

⟨*Process libraries: prcdef 5: TBP*⟩≡

```
procedure, nopass :: type_string => prcdef_5_type_string
```

⟨*Process libraries: test auxiliary*⟩+≡

```
function prcdef_5_type_string () result (string)
  type(string_t) :: string
```

```
    string = "test_file"
  end function prcdef_5_type_string
```

We reuse the writer test_writer_4 from the previous module.

⟨*Process libraries: prcdef 5: TBP*⟩+≡

```
  procedure :: init => prcdef_5_init
```

⟨*Process libraries: test auxiliary*⟩+≡

```
  subroutine prcdef_5_init (object)
    class(prcdef_5_t), intent(out) :: object
    allocate (test_writer_4_t :: object%writer)
  end subroutine prcdef_5_init
```

Nothing to write.

⟨*Process libraries: prcdef 5: TBP*⟩+≡

```
  procedure :: write => prcdef_5_write
```

⟨*Process libraries: test auxiliary*⟩+≡

```
  subroutine prcdef_5_write (object, unit)
    class(prcdef_5_t), intent(in) :: object
    integer, intent(in) :: unit
  end subroutine prcdef_5_write
```

Nothing to read.

⟨*Process libraries: prcdef 5: TBP*⟩+≡

```
  procedure :: read => prcdef_5_read
```

⟨*Process libraries: test auxiliary*⟩+≡

```
  subroutine prcdef_5_read (object, unit)
    class(prcdef_5_t), intent(out) :: object
    integer, intent(in) :: unit
  end subroutine prcdef_5_read
```

Allocate the driver with the appropriate type.

⟨*Process libraries: prcdef 5: TBP*⟩+≡

```
  procedure :: allocate_driver => prcdef_5_allocate_driver
```

⟨*Process libraries: test auxiliary*⟩+≡

```
  subroutine prcdef_5_allocate_driver (object, driver, basename)
    class(prcdef_5_t), intent(in) :: object
    class(prc_core_driver_t), intent(out), allocatable :: driver
    type(string_t), intent(in) :: basename
    allocate (prctest_5_t :: driver)
  end subroutine prcdef_5_allocate_driver
```

This time we need code:

⟨*Process libraries: prcdef 5: TBP*⟩+≡

```
  procedure, nopass :: needs_code => prcdef_5_needs_code
```

⟨*Process libraries: test auxiliary*⟩+≡

```
  function prcdef_5_needs_code () result (flag)
    logical :: flag
    flag = .true.
  end function prcdef_5_needs_code
```

For the test case, we implement a single feature `proc1`.

⟨*Process libraries: prcdef 5: TBP*⟩+≡
```
procedure, nopass :: get_features => prcdef_5_get_features
```

⟨*Process libraries: test auxiliary*⟩+≡
```
subroutine prcdef_5_get_features (features)
  type(string_t), dimension(:), allocatable, intent(out) :: features
  allocate (features (1))
  features = [ var_str ("proc1") ]
end subroutine prcdef_5_get_features
```

Nothing to connect.

⟨*Process libraries: prcdef 5: TBP*⟩+≡
```
procedure :: connect => prcdef_5_connect
```

⟨*Process libraries: test auxiliary*⟩+≡
```
subroutine prcdef_5_connect (def, lib_driver, i, proc_driver)
  class(prcdef_5_t), intent(in) :: def
  class(prclib_driver_t), intent(in) :: lib_driver
  integer, intent(in) :: i
  class(prc_core_driver_t), intent(inout) :: proc_driver
end subroutine prcdef_5_connect
```

The driver type.

⟨*Process libraries: test types*⟩+≡
```
type, extends (prc_core_driver_t) :: prctest_5_t
 contains
 ⟨Process libraries: prctest 5: TBP⟩
end type prctest_5_t
```

Return the type name.

⟨*Process libraries: prctest 5: TBP*⟩≡
```
procedure, nopass :: type_name => prctest_5_type_name
```

⟨*Process libraries: test auxiliary*⟩+≡
```
function prctest_5_type_name () result (type)
  type(string_t) :: type
  type = "test_file"
end function prctest_5_type_name
```

Here is the actual test:

⟨*Process libraries: execute tests*⟩+≡
```
call test (process_libraries_5, "process_libraries_5", &
     "build external process library", &
     u, results)
```

⟨*Process libraries: test declarations*⟩+≡
```
public :: process_libraries_5
```

⟨*Process libraries: tests*⟩+≡
```
subroutine process_libraries_5 (u)
  integer, intent(in) :: u
  type(process_library_t) :: lib
  type(process_def_entry_t), pointer :: entry
```

```
class(prc_core_def_t), allocatable :: core_def
type(os_data_t) :: os_data

write (u, "(A)")  "* Test output: process_libraries_5"
write (u, "(A)")  "* Purpose: build a process library with an &
     &external (pseudo) matrix element"
write (u, "(A)")

write (u, "(A)")  "* Initialize a process library with one entry"
write (u, "(A)")
call lib%init (var_str ("proclibs5"))
call os_data_init (os_data)

allocate (prcdef_5_t :: core_def)
select type (core_def)
type is (prcdef_5_t)
   call core_def%init ()
end select

allocate (entry)
call entry%init (var_str ("proclibs5_a"), &
     model_name = var_str ("Test_Model"), &
     n_in = 1, n_components = 1)
call entry%import_component (1, n_out = 2, &
     prt_in  = new_prt_spec ([var_str ("a")]), &
     prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
     method  = var_str ("test"), &
     variant = core_def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")
call lib%configure (os_data)

write (u, "(A)")  "* Compute MD5 sum"
write (u, "(A)")
call lib%compute_md5sum ()

write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Write driver source code"
write (u, "(A)")
call lib%write_driver (force = .true.)

write (u, "(A)")  "* Write process source code"
write (u, "(A)")
call lib%make_source (os_data)

write (u, "(A)")  "* Compile"
write (u, "(A)")
call lib%make_compile (os_data)
```

```
    write (u, "(A)")  "* Link"
    write (u, "(A)")
    call lib%make_link (os_data)

    call lib%write (u, libpath = .false.)

    call lib%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: process_libraries_5"
  end subroutine process_libraries_5
```

Build and load library with test matrix element

Test 6: We write source code for a dummy process.

This process variant is identical to the previous case, but it supports a driver for the test procedure 'proc1'.

⟨*Process libraries: test types*⟩+≡

```
  type, extends (prc_core_def_t) :: prcdef_6_t
   contains
   ⟨Process libraries: prcdef 6: TBP⟩
  end type prcdef_6_t
```

The process variant is named `test_file`.

⟨*Process libraries: prcdef 6: TBP*⟩≡

```
  procedure, nopass :: type_string => prcdef_6_type_string
```

⟨*Process libraries: test auxiliary*⟩+≡

```
  function prcdef_6_type_string () result (string)
    type(string_t) :: string
    string = "test_file"
  end function prcdef_6_type_string
```

We reuse the writer `test_writer_4` from the previous module.

⟨*Process libraries: prcdef 6: TBP*⟩+≡

```
  procedure :: init => prcdef_6_init
```

⟨*Process libraries: test auxiliary*⟩+≡

```
  subroutine prcdef_6_init (object)
    class(prcdef_6_t), intent(out) :: object
    allocate (test_writer_4_t :: object%writer)
    call object%writer%init_test ()
  end subroutine prcdef_6_init
```

Nothing to write.

⟨*Process libraries: prcdef 6: TBP*⟩+≡

```
  procedure :: write => prcdef_6_write
```

⟨*Process libraries: test auxiliary*⟩+≡

```
subroutine prcdef_6_write (object, unit)
  class(prcdef_6_t), intent(in) :: object
  integer, intent(in) :: unit
end subroutine prcdef_6_write
```

Nothing to read.

⟨*Process libraries: prcdef 6: TBP*⟩+≡

```
procedure :: read => prcdef_6_read
```

⟨*Process libraries: test auxiliary*⟩+≡

```
subroutine prcdef_6_read (object, unit)
  class(prcdef_6_t), intent(out) :: object
  integer, intent(in) :: unit
end subroutine prcdef_6_read
```

Allocate the driver with the appropriate type.

⟨*Process libraries: prcdef 6: TBP*⟩+≡

```
procedure :: allocate_driver => prcdef_6_allocate_driver
```

⟨*Process libraries: test auxiliary*⟩+≡

```
subroutine prcdef_6_allocate_driver (object, driver, basename)
  class(prcdef_6_t), intent(in) :: object
  class(prc_core_driver_t), intent(out), allocatable :: driver
  type(string_t), intent(in) :: basename
  allocate (prctest_6_t :: driver)
end subroutine prcdef_6_allocate_driver
```

This time we need code:

⟨*Process libraries: prcdef 6: TBP*⟩+≡

```
procedure, nopass :: needs_code => prcdef_6_needs_code
```

⟨*Process libraries: test auxiliary*⟩+≡

```
function prcdef_6_needs_code () result (flag)
  logical :: flag
  flag = .true.
end function prcdef_6_needs_code
```

For the test case, we implement a single feature `proc1`.

⟨*Process libraries: prcdef 6: TBP*⟩+≡

```
procedure, nopass :: get_features => prcdef_6_get_features
```

⟨*Process libraries: test auxiliary*⟩+≡

```
subroutine prcdef_6_get_features (features)
  type(string_t), dimension(:), allocatable, intent(out) :: features
  allocate (features (1))
  features = [ var_str ("proc1") ]
end subroutine prcdef_6_get_features
```

The interface of the only specific feature.

⟨*Process libraries: test types*⟩+≡

```
abstract interface
   subroutine proc1_t (n) bind(C)
     import
     integer(c_int), intent(out) :: n
   end subroutine proc1_t
end interface
```

Connect the feature `proc1` with the process driver.

⟨*Process libraries: prcdef 6: TBP*⟩+≡

```
procedure :: connect => prcdef_6_connect
```

⟨*Process libraries: test auxiliary*⟩+≡

```
subroutine prcdef_6_connect (def, lib_driver, i, proc_driver)
  class(prcdef_6_t), intent(in) :: def
  class(prclib_driver_t), intent(in) :: lib_driver
  integer, intent(in) :: i
  class(prc_core_driver_t), intent(inout) :: proc_driver
  integer(c_int) :: pid, fid
  type(c_funptr) :: fptr
  select type (proc_driver)
  type is  (prctest_6_t)
     pid = i
     fid = 1
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%proc1)
  end select
end subroutine prcdef_6_connect
```

The driver type.

⟨*Process libraries: test types*⟩+≡

```
type, extends (prc_core_driver_t) :: prctest_6_t
   procedure(proc1_t), nopass, pointer :: proc1 => null ()
 contains
 ⟨Process libraries: prctest 6: TBP⟩
end type prctest_6_t
```

Return the type name.

⟨*Process libraries: prctest 6: TBP*⟩≡

```
procedure, nopass :: type_name => prctest_6_type_name
```

⟨*Process libraries: test auxiliary*⟩+≡

```
function prctest_6_type_name () result (type)
  type(string_t) :: type
  type = "test_file"
end function prctest_6_type_name
```

Here is the actual test:

⟨*Process libraries: execute tests*⟩+≡

```
call test (process_libraries_6, "process_libraries_6", &
     "build and load external process library", &
     u, results)
```

⟨*Process libraries: test declarations*⟩+≡

```
public :: process_libraries_6
```

⟨*Process libraries: tests*⟩+≡

```
subroutine process_libraries_6 (u)
  integer, intent(in) :: u
  type(process_library_t) :: lib
  type(process_def_entry_t), pointer :: entry
  class(prc_core_def_t), allocatable :: core_def
  type(os_data_t) :: os_data
  type(string_t), dimension(:), allocatable :: name_list
  type(process_constants_t) :: data
  class(prc_core_driver_t), allocatable :: proc_driver
  integer :: i
  integer(c_int) :: n

  write (u, "(A)")  "* Test output: process_libraries_6"
  write (u, "(A)")  "* Purpose: build and load a process library"
  write (u, "(A)")  "*          with an external (pseudo) matrix element"
  write (u, "(A)")  "*          Check single-call linking"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize a process library with one entry"
  write (u, "(A)")
  call lib%init (var_str ("proclibs6"))
  call os_data_init (os_data)

  allocate (prcdef_6_t :: core_def)
  select type (core_def)
  type is (prcdef_6_t)
     call core_def%init ()
  end select

  allocate (entry)
  call entry%init (var_str ("proclibs6_a"), &
       model_name = var_str ("Test_model"), &
       n_in = 1, n_components = 1)
  call entry%import_component (1, n_out = 2, &
       prt_in  = new_prt_spec ([var_str ("a")]), &
       prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
       method  = var_str ("test"), &
       variant = core_def)
  call lib%append (entry)

  write (u, "(A)")  "* Configure library"
  write (u, "(A)")
  call lib%configure (os_data)

  write (u, "(A)")  "* Write makefile"
  write (u, "(A)")
  call lib%write_makefile (os_data, force = .true.)

  write (u, "(A)")  "* Write driver source code"
  write (u, "(A)")
  call lib%write_driver (force = .true.)
```

```
write (u, "(A)")  "* Write process source code, compile, link, load"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u, libpath = .false.)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,A,A)")  "name                      = '", &
     char (lib%get_name ()), "'"
write (u, "(1x,A,L1)")  "is active                 = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes               = ", &
     lib%get_n_processes ()
write (u, "(1x,A)", advance="no")  "processes                  ="
call lib%get_process_id_list (name_list)
do i = 1, size (name_list)
   write (u, "(1x,A)", advance="no")  char (name_list(i))
end do
write (u, *)
write (u, "(1x,A,L1)")  "proclibs6_a is process    = ", &
     lib%contains (var_str ("proclibs6_a"))
write (u, "(1x,A,I0)")  "proclibs6_a has index     = ", &
     lib%get_entry_index (var_str ("proclibs6_a"))
write (u, "(1x,A,L1)")  "foobar is process         = ", &
     lib%contains (var_str ("foobar"))
write (u, "(1x,A,I0)")  "foobar has index          = ", &
     lib%get_entry_index (var_str ("foobar"))
write (u, "(1x,A,I0)")  "n_in(proclibs6_a)         = ", &
     lib%get_n_in (var_str ("proclibs6_a"))
write (u, "(1x,A,A)")   "model_name(proclibs6_a)   = ", &
     char (lib%get_model_name (var_str ("proclibs6_a")))
write (u, "(1x,A,I0)")  "n_components(proclibs6_a) = ", &
     lib%get_n_components (var_str ("proclibs6_a"))
write (u, "(1x,A)", advance="no")  "components(proclibs6_a)   ="
call lib%get_component_list (var_str ("proclibs6_a"), name_list)
do i = 1, size (name_list)
   write (u, "(1x,A)", advance="no")  char (name_list(i))
end do
write (u, *)

write (u, "(A)")
write (u, "(A)")  "* Constants of proclibs6_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("proclibs6_a"), 1, data, proc_driver)

write (u, "(1x,A,A)")  "component ID     = ", char (data%id)
write (u, "(1x,A,A)")  "model name       = ", char (data%model_name)
write (u, "(1x,A,A,A)")  "md5sum           = '", data%md5sum, "'"
write (u, "(1x,A,L1)") "openmp supported = ", data%openmp_supported
```

```
    write (u, "(1x,A,I0)") "n_in  = ", data%n_in
    write (u, "(1x,A,I0)") "n_out = ", data%n_out
    write (u, "(1x,A,I0)") "n_flv = ", data%n_flv
    write (u, "(1x,A,I0)") "n_hel = ", data%n_hel
    write (u, "(1x,A,I0)") "n_col = ", data%n_col
    write (u, "(1x,A,I0)") "n_cin = ", data%n_cin
    write (u, "(1x,A,I0)") "n_cf  = ", data%n_cf
    write (u, "(1x,A,10(1x,I0))") "flv state =", data%flv_state
    write (u, "(1x,A,10(1x,I0))") "hel state =", data%hel_state
    write (u, "(1x,A,10(1x,I0))") "col state =", data%col_state
    write (u, "(1x,A,10(1x,L1))") "ghost flag =", data%ghost_flag
    write (u, "(1x,A,10(1x,F5.3))") "color factors =", data%color_factors
    write (u, "(1x,A,10(1x,I0))") "cf index =", data%cf_index

    write (u, "(A)")
    write (u, "(A)")  "* Call feature of proclibs6_a:"
    write (u, "(A)")

    select type (proc_driver)
    type is (prctest_6_t)
       call proc_driver%proc1 (n)
       write (u, "(1x,A,I0)") "proc1 = ", n
    end select

    call lib%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: process_libraries_6"
  end subroutine process_libraries_6
```

MD5 sums

Check MD5 sum calculation.

⟨*Process libraries: execute tests*⟩+≡

```
  call test (process_libraries_7, "process_libraries_7", &
       "process definition list", &
       u, results)
```

⟨*Process libraries: test declarations*⟩+≡

```
  public :: process_libraries_7
```

⟨*Process libraries: tests*⟩+≡

```
  subroutine process_libraries_7 (u)
    integer, intent(in) :: u
    type(prc_template_t), dimension(:), allocatable :: process_core_templates
    type(process_def_entry_t), target :: entry
    class(prc_core_def_t), allocatable :: test_def
    class(prc_core_def_t), pointer :: def

    write (u, "(A)")  "* Test output: process_libraries_7"
    write (u, "(A)")  "* Purpose: Construct a process definition list &
         &and check MD5 sums"
    write (u, "(A)")
    write (u, "(A)")  "* Construct a process definition list"
```

```
write (u, "(A)")  "*   Process: two components"
write (u, "(A)")

allocate (process_core_templates (1))
allocate (prcdef_2_t :: process_core_templates(1)%core_def)

call entry%init (var_str ("first"), model_name = var_str ("Test"), &
     n_in = 1, n_components = 2)
allocate (prcdef_2_t :: test_def)
select type (test_def)
type is (prcdef_2_t);  test_def%data = 31
end select
call entry%import_component (1, n_out = 3, &
     prt_in  = new_prt_spec ([var_str ("a")]), &
     prt_out = new_prt_spec ([var_str ("b"), var_str ("c"), &
                              var_str ("e")]), &
     method  = var_str ("test"), &
     variant = test_def)
allocate (prcdef_2_t :: test_def)
select type (test_def)
type is (prcdef_2_t);  test_def%data = 42
end select
call entry%import_component (2, n_out = 2, &
     prt_in  = new_prt_spec ([var_str ("a")]), &
     prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
     method  = var_str ("test"), &
     variant = test_def)
call entry%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute MD5 sums"
write (u, "(A)")

call entry%compute_md5sum ()
call entry%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recalculate MD5 sums (should be identical)"
write (u, "(A)")

call entry%compute_md5sum ()
call entry%write (u)

write (u, "(A)")
write (u, "(A)")  "* Modify a component and recalculate MD5 sums"
write (u, "(A)")

def => entry%get_core_def_ptr (2)
select type (def)
type is (prcdef_2_t)
   def%data = 54
end select
call entry%compute_md5sum ()
call entry%write (u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Modify the model and recalculate MD5 sums"
    write (u, "(A)")

    call entry%set_model_name (var_str ("foo"))
    call entry%compute_md5sum ()
    call entry%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: process_libraries_7"
  end subroutine process_libraries_7
```

Here is the actual test:

⟨*Process libraries: execute tests*⟩+≡

```
  call test (process_libraries_8, "process_libraries_8", &
       "library status checks", &
       u, results)
```

⟨*Process libraries: test declarations*⟩+≡

```
  public :: process_libraries_8
```

⟨*Process libraries: tests*⟩+≡

```
  subroutine process_libraries_8 (u)
    integer, intent(in) :: u
    type(process_library_t) :: lib
    type(process_def_entry_t), pointer :: entry
    class(prc_core_def_t), allocatable :: core_def
    type(os_data_t) :: os_data

    write (u, "(A)")  "* Test output: process_libraries_8"
    write (u, "(A)")  "* Purpose: build and load a process library"
    write (u, "(A)")  "*          with an external (pseudo) matrix element"
    write (u, "(A)")  "*          Check status updates"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize a process library with one entry"
    write (u, "(A)")
    call lib%init (var_str ("proclibs8"))
    call os_data_init (os_data)

    allocate (prcdef_6_t :: core_def)
    select type (core_def)
    type is (prcdef_6_t)
       call core_def%init ()
    end select

    allocate (entry)
    call entry%init (var_str ("proclibs8_a"), &
         model_name = var_str ("Test_model"), &
         n_in = 1, n_components = 1)
    call entry%import_component (1, n_out = 2, &
         prt_in  = new_prt_spec ([var_str ("a")]), &
         prt_out = new_prt_spec ([var_str ("b"), var_str ("c")]), &
         method  = var_str ("test"), &
```

```
     variant = core_def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")

call lib%configure (os_data)
call lib%compute_md5sum ()

call lib%test_transfer_md5sum (1, 1, 1)

write (u, "(1x,A,L1)")  "library loaded = ", lib%is_loaded ()
write (u, "(1x,A,I0)")  "lib status   = ", lib%get_status ()
write (u, "(1x,A,I0)")  "proc1 status = ", lib%get_status (1)

write (u, "(A)")
write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Update status"
write (u, "(A)")

call lib%update_status (os_data)
write (u, "(1x,A,L1)")  "library loaded = ", lib%is_loaded ()
write (u, "(1x,A,I0)")  "lib status   = ", lib%get_status ()
write (u, "(1x,A,I0)")  "proc1 status = ", lib%get_status (1)

write (u, "(A)")
write (u, "(A)")  "* Write driver source code"
write (u, "(A)")
call lib%write_driver (force = .false.)

write (u, "(A)")  "* Write process source code"
write (u, "(A)")
call lib%make_source (os_data)

write (u, "(1x,A,L1)")  "library loaded = ", lib%is_loaded ()
write (u, "(1x,A,I0)")  "lib status   = ", lib%get_status ()
write (u, "(1x,A,I0)")  "proc1 status = ", lib%get_status (1)

write (u, "(A)")
write (u, "(A)")  "* Compile and load"
write (u, "(A)")

call lib%load (os_data)
write (u, "(1x,A,L1)")  "library loaded = ", lib%is_loaded ()
write (u, "(1x,A,I0)")  "lib status   = ", lib%get_status ()
write (u, "(1x,A,I0)")  "proc1 status = ", lib%get_status (1)

write (u, "(A)")
write (u, "(A)")  "* Append process and reconfigure"
write (u, "(A)")
```

```
allocate (prcdef_6_t :: core_def)
select type (core_def)
type is (prcdef_6_t)
   call core_def%init ()
end select

allocate (entry)
call entry%init (var_str ("proclibs8_b"), &
     model_name = var_str ("Test_model"), &
     n_in = 1, n_components = 1)
call entry%import_component (1, n_out = 2, &
     prt_in  = new_prt_spec ([var_str ("a")]), &
     prt_out = new_prt_spec ([var_str ("b"), var_str ("d")]), &
     method  = var_str ("test"), &
     variant = core_def)
call lib%append (entry)

call lib%configure (os_data)
call lib%compute_md5sum ()
call lib%test_transfer_md5sum (2, 2, 1)
call lib%write_makefile (os_data, force = .false.)
call lib%write_driver (force = .false.)

write (u, "(1x,A,L1)")  "library loaded = ", lib%is_loaded ()
write (u, "(1x,A,I0)")  "lib status   = ", lib%get_status ()
write (u, "(1x,A,I0)")  "proc1 status = ", lib%get_status (1)
write (u, "(1x,A,I0)")  "proc2 status = ", lib%get_status (2)

write (u, "(A)")
write (u, "(A)")  "* Update status"
write (u, "(A)")

call lib%update_status (os_data)
write (u, "(1x,A,L1)")  "library loaded = ", lib%is_loaded ()
write (u, "(1x,A,I0)")  "lib status   = ", lib%get_status ()
write (u, "(1x,A,I0)")  "proc1 status = ", lib%get_status (1)
write (u, "(1x,A,I0)")  "proc2 status = ", lib%get_status (2)

write (u, "(A)")
write (u, "(A)")  "* Write source code"
write (u, "(A)")

call lib%make_source (os_data)
write (u, "(1x,A,L1)")  "library loaded = ", lib%is_loaded ()
write (u, "(1x,A,I0)")  "lib status   = ", lib%get_status ()
write (u, "(1x,A,I0)")  "proc1 status = ", lib%get_status (1)
write (u, "(1x,A,I0)")  "proc2 status = ", lib%get_status (2)

write (u, "(A)")
write (u, "(A)")  "* Reset status"
write (u, "(A)")

call lib%set_status (STAT_CONFIGURED, entries=.true.)
write (u, "(1x,A,L1)")  "library loaded = ", lib%is_loaded ()
```

```
  write (u, "(1x,A,I0)")  "lib status   = ", lib%get_status ()
  write (u, "(1x,A,I0)")  "proc1 status = ", lib%get_status (1)
  write (u, "(1x,A,I0)")  "proc2 status = ", lib%get_status (2)

  write (u, "(A)")
  write (u, "(A)")  "* Update status"
  write (u, "(A)")

  call lib%update_status (os_data)
  write (u, "(1x,A,L1)")  "library loaded = ", lib%is_loaded ()
  write (u, "(1x,A,I0)")  "lib status   = ", lib%get_status ()
  write (u, "(1x,A,I0)")  "proc1 status = ", lib%get_status (1)
  write (u, "(1x,A,I0)")  "proc2 status = ", lib%get_status (2)

  write (u, "(A)")
  write (u, "(A)")  "* Partial cleanup"
  write (u, "(A)")

  call lib%clean (os_data, distclean = .false.)
  write (u, "(1x,A,L1)")  "library loaded = ", lib%is_loaded ()
  write (u, "(1x,A,I0)")  "lib status   = ", lib%get_status ()
  write (u, "(1x,A,I0)")  "proc1 status = ", lib%get_status (1)
  write (u, "(1x,A,I0)")  "proc2 status = ", lib%get_status (2)


  write (u, "(A)")
  write (u, "(A)")  "* Update status"
  write (u, "(A)")

  call lib%update_status (os_data)
  write (u, "(1x,A,L1)")  "library loaded = ", lib%is_loaded ()
  write (u, "(1x,A,I0)")  "lib status   = ", lib%get_status ()
  write (u, "(1x,A,I0)")  "proc1 status = ", lib%get_status (1)
  write (u, "(1x,A,I0)")  "proc2 status = ", lib%get_status (2)

  write (u, "(A)")
  write (u, "(A)")  "* Complete cleanup"

  call lib%clean (os_data, distclean = .true.)
  call lib%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: process_libraries_8"
end subroutine process_libraries_8
```

13.5 Process Library Stacks

For storing and handling multiple libraries, we define process library stacks. These are ordinary stacks where new entries are pushed onto the top.

⟨prclib_stacks.f90⟩≡
  ⟨*File header*⟩

```
module prclib_stacks

⟨Use strings⟩
  use io_units
  use format_utils, only: write_separator
  use process_libraries

⟨Standard module head⟩

⟨Prclib stacks: public⟩

⟨Prclib stacks: types⟩

contains

⟨Prclib stacks: procedures⟩

end module prclib_stacks
```

13.5.1 The stack entry type

A stack entry is a process library object, augmented by a pointer to the next entry. We do not need specific methods, all relevant methods are inherited.

On higher level, process libraries should be prepared as process entry objects.

⟨*Prclib stacks: public*⟩≡

```
public :: prclib_entry_t
```

⟨*Prclib stacks: types*⟩≡

```
type, extends (process_library_t) :: prclib_entry_t
   type(prclib_entry_t), pointer :: next => null ()
end type prclib_entry_t
```

13.5.2 The prclib stack type

For easy conversion and lookup it is useful to store the filling number in the object. The content is stored as a linked list.

⟨*Prclib stacks: public*⟩+≡

```
public :: prclib_stack_t
```

⟨*Prclib stacks: types*⟩+≡

```
type :: prclib_stack_t
   integer :: n = 0
   type(prclib_entry_t), pointer :: first => null ()
 contains
 ⟨Prclib stacks: prclib stack: TBP⟩
end type prclib_stack_t
```

Finalizer. Iteratively deallocate the stack entries. The resulting empty stack can be immediately recycled, if necessary.

⟨*Prclib stacks: prclib stack: TBP*⟩≡

```
procedure :: final => prclib_stack_final
```

⟨*Prclib stacks: procedures*⟩≡

```
subroutine prclib_stack_final (object)
  class(prclib_stack_t), intent(inout) :: object
  type(prclib_entry_t), pointer :: lib
  do while (associated (object%first))
     lib => object%first
     object%first => lib%next
     call lib%final ()
     deallocate (lib)
  end do
  object%n = 0
end subroutine prclib_stack_final
```

Output. The entries on the stack will be ordered LIFO, i.e., backwards.

⟨*Prclib stacks: prclib stack: TBP*⟩+≡

```
procedure :: write => prclib_stack_write
```

⟨*Prclib stacks: procedures*⟩+≡

```
subroutine prclib_stack_write (object, unit, libpath)
  class(prclib_stack_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: libpath
  type(prclib_entry_t), pointer :: lib
  integer :: u
  u = given_output_unit (unit)
  call write_separator (u, 2)
  select case (object%n)
  case (0)
     write (u, "(1x,A)")  "Process library stack: [empty]"
  case default
     write (u, "(1x,A)")  "Process library stack:"
     lib => object%first
     do while (associated (lib))
        call write_separator (u)
        call lib%write (u, libpath)
        lib => lib%next
     end do
  end select
  call write_separator (u, 2)
end subroutine prclib_stack_write
```

13.5.3 Operating on Stacks

We take a library entry pointer and push it onto the stack. The previous pointer is nullified. Subsequently, the library entry is 'owned' by the stack and will be finalized when the stack is deleted.

⟨*Prclib stacks: prclib stack: TBP*⟩+≡

```
procedure :: push => prclib_stack_push
```

⟨*Prclib stacks: procedures*⟩+≡

```
subroutine prclib_stack_push (stack, lib)
  class(prclib_stack_t), intent(inout) :: stack
```

```
    type(prclib_entry_t), intent(inout), pointer :: lib
    lib%next => stack%first
    stack%first => lib
    lib => null ()
    stack%n = stack%n + 1
  end subroutine prclib_stack_push
```

13.5.4 Accessing Contents

Return a pointer to the topmost stack element. The result type is just the bare `process_library_t`. There is no `target` attribute required since the stack elements are allocated via pointers.

⟨*Prclib stacks: prclib stack: TBP*⟩+≡

```
  procedure :: get_first_ptr => prclib_stack_get_first_ptr
```

⟨*Prclib stacks: procedures*⟩+≡

```
  function prclib_stack_get_first_ptr (stack) result (ptr)
    class(prclib_stack_t), intent(in) :: stack
    type(process_library_t), pointer :: ptr
    if (associated (stack%first)) then
       ptr => stack%first%process_library_t
    else
       ptr => null ()
    end if
  end function prclib_stack_get_first_ptr
```

Return a complete list of the libraries (names) in the stack. The list is in the order in which the elements got pushed onto the stack, so the 'first' entry is listed last.

⟨*Prclib stacks: prclib stack: TBP*⟩+≡

```
  procedure :: get_names => prclib_stack_get_names
```

⟨*Prclib stacks: procedures*⟩+≡

```
  subroutine prclib_stack_get_names (stack, libname)
    class(prclib_stack_t), intent(in) :: stack
    type(string_t), dimension(:), allocatable, intent(out) :: libname
    type(prclib_entry_t), pointer :: lib
    integer :: i
    allocate (libname (stack%n))
    i = stack%n
    lib => stack%first
    do while (associated (lib))
       libname(i) = lib%get_name ()
       i = i - 1
       lib => lib%next
    end do
  end subroutine prclib_stack_get_names
```

Return a pointer to the library with given name.

⟨*Prclib stacks: prclib stack: TBP*⟩+≡

```
  procedure :: get_library_ptr => prclib_stack_get_library_ptr
```

⟨*Prclib stacks: procedures*⟩+≡

```
function prclib_stack_get_library_ptr (stack, libname) result (ptr)
  class(prclib_stack_t), intent(in) :: stack
  type(string_t), intent(in) :: libname
  type(process_library_t), pointer :: ptr
  type(prclib_entry_t), pointer :: current
  current => stack%first
  do while (associated (current))
     if (current%get_name () == libname) then
        ptr => current%process_library_t
        return
     end if
     current => current%next
  end do
  ptr => null ()
end function prclib_stack_get_library_ptr
```

13.5.5 Unit tests

Test module, followed by the corresponding implementation module.

⟨prclib_stacks_ut.f90⟩≡

```
⟨File header⟩

module prclib_stacks_ut
  use unit_tests
  use prclib_stacks_uti

⟨Standard module head⟩

⟨Prclib stacks: public test⟩

contains

⟨Prclib stacks: test driver⟩

end module prclib_stacks_ut
```

⟨prclib_stacks_uti.f90⟩≡

```
⟨File header⟩

module prclib_stacks_uti

⟨Use strings⟩

  use prclib_stacks

⟨Standard module head⟩

⟨Prclib stacks: test declarations⟩

contains

⟨Prclib stacks: tests⟩
```

```
  end module prclib_stacks_uti
```

API: driver for the unit tests below.

⟨*Prclib stacks: public test*⟩≡

```
    public :: prclib_stacks_test
```

⟨*Prclib stacks: test driver*⟩≡

```
    subroutine prclib_stacks_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Prclib stacks: execute tests⟩
    end subroutine prclib_stacks_test
```

Write an empty process library stack

The most trivial test is to write an uninitialized process library stack.

⟨*Prclib stacks: execute tests*⟩≡

```
    call test (prclib_stacks_1, "prclib_stacks_1", &
         "write an empty process library stack", &
         u, results)
```

⟨*Prclib stacks: test declarations*⟩≡

```
    public :: prclib_stacks_1
```

⟨*Prclib stacks: tests*⟩≡

```
    subroutine prclib_stacks_1 (u)
      integer, intent(in) :: u
      type(prclib_stack_t) :: stack

      write (u, "(A)")  "* Test output: prclib_stacks_1"
      write (u, "(A)")  "*   Purpose: display an empty process library stack"
      write (u, "(A)")

      call stack%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: prclib_stacks_1"

    end subroutine prclib_stacks_1
```

Fill a process library stack

Fill a process library stack with two (identical) processes.

⟨*Prclib stacks: execute tests*⟩+≡

```
    call test (prclib_stacks_2, "prclib_stacks_2", &
         "fill a process library stack", &
         u, results)
```

⟨*Prclib stacks: test declarations*⟩+≡

```
    public :: prclib_stacks_2
```

⟨*Prclib stacks: tests*⟩+≡

```
subroutine prclib_stacks_2 (u)
  integer, intent(in) :: u
  type(prclib_stack_t) :: stack
  type(prclib_entry_t), pointer :: lib

  write (u, "(A)")  "* Test output: prclib_stacks_2"
  write (u, "(A)")  "*   Purpose: fill a process library stack"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize two (empty) libraries &
       &and push them on the stack"
  write (u, "(A)")

  allocate (lib)
  call lib%init (var_str ("lib1"))
  call stack%push (lib)

  allocate (lib)
  call lib%init (var_str ("lib2"))
  call stack%push (lib)

  call stack%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call stack%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: prclib_stacks_2"

end subroutine prclib_stacks_2
```

13.6 Trivial matrix element for tests

For the purpose of testing the workflow, we implement here two matrix elements with the simplest possible structure.

This matrix element generator can only generate a single scattering process and a single decay process. The scattering process is a quartic interaction of a massless, neutral and colorless scalar `s` with unit coupling results in a trivial $2 \to 2$ scattering process. The matrix element is implemented internally, so we do not need the machinery of external process libraries. The decay process is a decay of `s` into a pair of colored fermions `f`.

⟨`prc_test.f90`⟩≡

⟨*File header*⟩

```
module prc_test

  use, intrinsic :: iso_c_binding !NODEP!
```

```
⟨Use kinds⟩
⟨Use strings⟩
  use os_interface
  use process_constants
  use prclib_interfaces
  use prc_core_def
  use particle_specifiers, only: new_prt_spec
  use process_libraries

⟨Standard module head⟩

⟨Test ME: public⟩

⟨Test ME: types⟩

contains

⟨Test ME: procedures⟩

end module prc_test
```

13.6.1 Process definition

For the process definition we implement an extension of the `prc_core_def_t` abstract type.

```
⟨Test ME: public⟩≡
    public :: prc_test_def_t
```

```
⟨Test ME: types⟩≡
    type, extends (prc_core_def_t) :: prc_test_def_t
       type(string_t) :: model_name
       type(string_t), dimension(:), allocatable :: prt_in
       type(string_t), dimension(:), allocatable :: prt_out
     contains
     ⟨Test ME: test me def: TBP⟩
    end type prc_test_def_t
```

```
⟨Test ME: test me def: TBP⟩≡
    procedure, nopass :: type_string => prc_test_def_type_string
```

```
⟨Test ME: procedures⟩≡
    function prc_test_def_type_string () result (string)
      type(string_t) :: string
      string = "test_me"
    end function prc_test_def_type_string
```

There is no 'feature' here since there is no external code.

```
⟨Test ME: test me def: TBP⟩+≡
    procedure, nopass :: get_features => prc_test_def_get_features
```

```
⟨Test ME: procedures⟩+≡
    subroutine prc_test_def_get_features (features)
      type(string_t), dimension(:), allocatable, intent(out) :: features
      allocate (features (0))
```

```
end subroutine prc_test_def_get_features
```

Initialization: set some data (not really useful).

⟨*Test ME: test me def: TBP*⟩+≡

```
procedure :: init => prc_test_def_init
```

⟨*Test ME: procedures*⟩+≡

```
subroutine prc_test_def_init (object, model_name, prt_in, prt_out)
  class(prc_test_def_t), intent(out) :: object
  type(string_t), intent(in) :: model_name
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
  object%model_name = model_name
  allocate (object%prt_in (size (prt_in)))
  object%prt_in = prt_in
  allocate (object%prt_out (size (prt_out)))
  object%prt_out = prt_out
end subroutine prc_test_def_init
```

Write/read process- and method-specific data. (No-op)

⟨*Test ME: test me def: TBP*⟩+≡

```
procedure :: write => prc_test_def_write
```

⟨*Test ME: procedures*⟩+≡

```
subroutine prc_test_def_write (object, unit)
  class(prc_test_def_t), intent(in) :: object
  integer, intent(in) :: unit
end subroutine prc_test_def_write
```

⟨*Test ME: test me def: TBP*⟩+≡

```
procedure :: read => prc_test_def_read
```

⟨*Test ME: procedures*⟩+≡

```
subroutine prc_test_def_read (object, unit)
  class(prc_test_def_t), intent(out) :: object
  integer, intent(in) :: unit
end subroutine prc_test_def_read
```

Allocate the driver for test ME matrix elements. We get the actual component ID (basename), and we can transfer all process-specific data from the process definition.

⟨*Test ME: test me def: TBP*⟩+≡

```
procedure :: allocate_driver => prc_test_def_allocate_driver
```

⟨*Test ME: procedures*⟩+≡

```
subroutine prc_test_def_allocate_driver (object, driver, basename)
  class(prc_test_def_t), intent(in) :: object
  class(prc_core_driver_t), intent(out), allocatable :: driver
  type(string_t), intent(in) :: basename
  allocate (prc_test_t :: driver)
  select type (driver)
  type is (prc_test_t)
     driver%id = basename
```

```
      driver%model_name = object%model_name
      select case (size (object%prt_in))
      case (1);  driver%scattering = .false.
      case (2);  driver%scattering = .true.
      end select
    end select
  end subroutine prc_test_def_allocate_driver
```

Nothing to connect. This subroutine will not be called.

⟨*Test ME: test me def: TBP*⟩+≡

```
procedure :: connect => prc_test_def_connect
```

⟨*Test ME: procedures*⟩+≡

```
subroutine prc_test_def_connect (def, lib_driver, i, proc_driver)
  class(prc_test_def_t), intent(in) :: def
  class(prclib_driver_t), intent(in) :: lib_driver
  integer, intent(in) :: i
  class(prc_core_driver_t), intent(inout) :: proc_driver
end subroutine prc_test_def_connect
```

13.6.2 Driver

⟨*Test ME: public*⟩+≡

```
public :: prc_test_t
```

⟨*Test ME: types*⟩+≡

```
type, extends (process_driver_internal_t) :: prc_test_t
   type(string_t) :: id
   type(string_t) :: model_name
   logical :: scattering = .true.
 contains
 ⟨Test ME: test me driver: TBP⟩
end type prc_test_t
```

In contrast to generic matrix-element implementations, we can hard-wire the amplitude method as a type-bound procedure.

⟨*Test ME: test me driver: TBP*⟩≡

```
procedure, nopass :: get_amplitude => prc_test_get_amplitude
```

⟨*Test ME: procedures*⟩+≡

```
function prc_test_get_amplitude (p) result (amp)
  complex(default) :: amp
  real(default), dimension(:,:), intent(in) :: p
  amp = 1
end function prc_test_get_amplitude
```

The reported type is the same as for the `prc_test_def_t` type.

⟨*Test ME: test me driver: TBP*⟩+≡

```
procedure, nopass :: type_name => prc_test_type_name
```

⟨*Test ME: procedures*⟩+≡

```
function prc_test_type_name () result (string)
  type(string_t) :: string
  string = "test_me"
end function prc_test_type_name
```

Fill process constants.

⟨*Test ME: test me driver: TBP*⟩+≡

```
procedure :: fill_constants => prc_test_fill_constants
```

⟨*Test ME: procedures*⟩+≡

```
subroutine prc_test_fill_constants (driver, data)
  class(prc_test_t), intent(in) :: driver
  type(process_constants_t), intent(out) :: data
  data%id = driver%id
  data%model_name = driver%model_name
  if (driver%scattering) then
     data%n_in  = 2
     data%n_out = 2
     data%n_flv = 1
     data%n_hel = 1
     data%n_col = 1
     data%n_cin = 2
     data%n_cf  = 1
     allocate (data%flv_state (4, 1))
     data%flv_state = 25
     allocate (data%hel_state (4, 1))
     data%hel_state = 0
     allocate (data%col_state (2, 4, 1))
     data%col_state = 0
     allocate (data%ghost_flag (4, 1))
     data%ghost_flag = .false.
     allocate (data%color_factors (1))
     data%color_factors = 1
     allocate (data%cf_index (2, 1))
     data%cf_index = 1
  else
     data%n_in  = 1
     data%n_out = 2
     data%n_flv = 1
     data%n_hel = 2
     data%n_col = 1
     data%n_cin = 2
     data%n_cf  = 1
     allocate (data%flv_state (3, 1))
     data%flv_state(:,1) = [25, 6, -6]
     allocate (data%hel_state (3, 2))
     data%hel_state(:,1) = [0, 1,-1]
     data%hel_state(:,2) = [0,-1, 1]
     allocate (data%col_state (2, 3, 1))
     data%col_state = reshape ([0,0, 1,0, 0,-1], [2,3,1])
     allocate (data%ghost_flag (3, 1))
     data%ghost_flag = .false.
     allocate (data%color_factors (1))
```

```
      data%color_factors = 3
      allocate (data%cf_index (2, 1))
      data%cf_index = 1
   end if
end subroutine prc_test_fill_constants
```

13.6.3 Shortcut

Since this module is there for testing purposes, we set up a subroutine that does all the work at once: create a library with the two processes (scattering and decay), configure and load, and set up the driver.

⟨*Test ME: public*⟩+≡

```
public :: prc_test_create_library
```

⟨*Test ME: procedures*⟩+≡

```
subroutine prc_test_create_library &
     (libname, lib, scattering, decay, procname1, procname2)
  type(string_t), intent(in) :: libname
  type(process_library_t), intent(out) :: lib
  logical, intent(in), optional :: scattering, decay
  type(string_t), intent(in), optional :: procname1, procname2
  type(string_t) :: model_name, procname
  type(string_t), dimension(:), allocatable :: prt_in, prt_out
  class(prc_core_def_t), allocatable :: def
  type(process_def_entry_t), pointer :: entry
  type(os_data_t) :: os_data
  logical :: sca, dec
  sca = .true.;   if (present (scattering))  sca = scattering
  dec = .false.;  if (present (decay))       dec = decay

  call os_data_init (os_data)
  call lib%init (libname)
  model_name = "Test"

  if (sca) then
     if (present (procname1)) then
        procname = procname1
     else
        procname = libname
     end if
     allocate (prt_in (2), prt_out (2))
     prt_in  = [var_str ("s"), var_str ("s")]
     prt_out = [var_str ("s"), var_str ("s")]
     allocate (prc_test_def_t :: def)
     select type (def)
     type is (prc_test_def_t)
        call def%init (model_name, prt_in, prt_out)
     end select
     allocate (entry)
     call entry%init (procname, model_name = model_name, &
          n_in = 2, n_components = 1)
     call entry%import_component (1, n_out = size (prt_out), &
          prt_in  = new_prt_spec (prt_in), &
```

```
            prt_out = new_prt_spec (prt_out), &
            method  = var_str ("test_me"), &
            variant = def)
         call lib%append (entry)
      end if

      if (dec) then
         if (present (procname2)) then
            procname = procname2
         else
            procname = libname
         end if
         if (allocated (prt_in))  deallocate (prt_in, prt_out)
         allocate (prt_in (1), prt_out (2))
         prt_in  = [var_str ("s")]
         prt_out = [var_str ("f"), var_str ("fbar")]
         allocate (prc_test_def_t :: def)
         select type (def)
         type is (prc_test_def_t)
            call def%init (model_name, prt_in, prt_out)
         end select
         allocate (entry)
         call entry%init (procname, model_name = model_name, &
              n_in = 1, n_components = 1)
         call entry%import_component (1, n_out = size (prt_out), &
              prt_in  = new_prt_spec (prt_in), &
              prt_out = new_prt_spec (prt_out), &
              method  = var_str ("test_decay"), &
              variant = def)
         call lib%append (entry)
      end if

      call lib%configure (os_data)
      call lib%load (os_data)
   end subroutine prc_test_create_library
```

13.6.4 Unit Test

Test module, followed by the corresponding implementation module.

⟨prc_test_ut.f90⟩≡

```
  ⟨File header⟩

  module prc_test_ut
    use unit_tests
    use prc_test_uti

  ⟨Standard module head⟩

  ⟨Test ME: public test⟩

  contains
```

```
  ⟨Test ME: test driver⟩

  end module prc_test_ut
```

⟨prc_test_uti.f90⟩≡

```
  ⟨File header⟩

  module prc_test_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use os_interface
    use particle_specifiers, only: new_prt_spec
    use process_constants
    use prc_core_def
    use process_libraries

    use prc_test

  ⟨Standard module head⟩

  ⟨Test ME: test declarations⟩

  contains

  ⟨Test ME: tests⟩

  end module prc_test_uti
```

API: driver for the unit tests below.

⟨*Test ME: public test*⟩≡

```
    public :: prc_test_test
```

⟨*Test ME: test driver*⟩≡

```
    subroutine prc_test_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Test ME: execute tests⟩
  end subroutine prc_test_test
```

Generate and load the scattering process

The process is $ss \to ss$, where s is a trivial scalar particle, for vanishing mass and unit coupling. We initialize the process, build the library, and compute the particular matrix element for momenta of unit energy and right-angle scattering. (The scattering is independent of angle.) The matrix element is equal to unity.

⟨*Test ME: execute tests*⟩≡

```
    call test (prc_test_1, "prc_test_1", &
         "build and load trivial process", &
         u, results)
```

⟨*Test ME: test declarations*⟩≡

```
    public :: prc_test_1
```

⟨*Test ME: tests*⟩≡

```
subroutine prc_test_1 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(process_library_t) :: lib
  class(prc_core_def_t), allocatable :: def
  type(process_def_entry_t), pointer :: entry
  type(string_t) :: model_name
  type(string_t), dimension(:), allocatable :: prt_in, prt_out
  type(process_constants_t) :: data
  class(prc_core_driver_t), allocatable :: driver
  real(default), dimension(0:3,4) :: p
  integer :: i

  write (u, "(A)")  "* Test output: prc_test_1"
  write (u, "(A)")  "*   Purpose: create a trivial process"
  write (u, "(A)")  "*            build a library and &
       &access the matrix element"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize a process library with one entry"
  write (u, "(A)")
  call os_data_init (os_data)
  call lib%init (var_str ("prc_test1"))

  model_name = "Test"
  allocate (prt_in (2), prt_out (2))
  prt_in  = [var_str ("s"), var_str ("s")]
  prt_out = [var_str ("s"), var_str ("s")]

  allocate (prc_test_def_t :: def)
  select type (def)
  type is (prc_test_def_t)
     call def%init (model_name, prt_in, prt_out)
  end select
  allocate (entry)
  call entry%init (var_str ("prc_test1_a"), model_name = model_name, &
       n_in = 2, n_components = 1)
  call entry%import_component (1, n_out = size (prt_out), &
       prt_in  = new_prt_spec (prt_in), &
       prt_out = new_prt_spec (prt_out), &
       method  = var_str ("test_me"), &
       variant = def)
  call lib%append (entry)

  write (u, "(A)")  "* Configure library"
  write (u, "(A)")
  call lib%configure (os_data)

  write (u, "(A)")  "* Load library"
  write (u, "(A)")
  call lib%load (os_data)

  call lib%write (u)
```

```
write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                 = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes               = ", &
     lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Constants of prc_test1_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("prc_test1_a"), 1, data, driver)

write (u, "(1x,A,A)")  "component ID     = ", char (data%id)
write (u, "(1x,A,A)")  "model name       = ", char (data%model_name)
write (u, "(1x,A,A,A)")  "md5sum           = '", data%md5sum, "'"
write (u, "(1x,A,L1)") "openmp supported = ", data%openmp_supported
write (u, "(1x,A,I0)") "n_in  = ", data%n_in
write (u, "(1x,A,I0)") "n_out = ", data%n_out
write (u, "(1x,A,I0)") "n_flv = ", data%n_flv
write (u, "(1x,A,I0)") "n_hel = ", data%n_hel
write (u, "(1x,A,I0)") "n_col = ", data%n_col
write (u, "(1x,A,I0)") "n_cin = ", data%n_cin
write (u, "(1x,A,I0)") "n_cf  = ", data%n_cf
write (u, "(1x,A,10(1x,I0))") "flv state =", data%flv_state
write (u, "(1x,A,10(1x,I2))") "hel state =", data%hel_state(:,1)
write (u, "(1x,A,10(1x,I0))") "col state =", data%col_state
write (u, "(1x,A,10(1x,L1))") "ghost flag =", data%ghost_flag
write (u, "(1x,A,10(1x,F5.3))") "color factors =", data%color_factors
write (u, "(1x,A,10(1x,I0))") "cf index =", data%cf_index

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
     1.0_default, 0.0_default, 0.0_default, 1.0_default, &
     1.0_default, 0.0_default, 0.0_default,-1.0_default, &
     1.0_default, 1.0_default, 0.0_default, 0.0_default, &
     1.0_default,-1.0_default, 0.0_default, 0.0_default &
     ], [4,4])
do i = 1, 4
   write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
end do

write (u, "(A)")
write (u, "(A)")  "* Compute matrix element:"
write (u, "(A)")

select type (driver)
type is (prc_test_t)
```

```
      write (u, "(1x,A,1x,E11.4)") "|amp| =", abs (driver%get_amplitude (p))
    end select

    call lib%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prc_test_1"

  end subroutine prc_test_1
```

Shortcut

This is identical to the previous test, but we create the library be a single command. This is handy for other modules which use the test process.

⟨*Test ME: execute tests*⟩+≡

```
  call test (prc_test_2, "prc_test_2", &
      "build and load trivial process using shortcut", &
      u, results)
```

⟨*Test ME: test declarations*⟩+≡

```
  public :: prc_test_2
```

⟨*Test ME: tests*⟩+≡

```
  subroutine prc_test_2 (u)
    integer, intent(in) :: u
    type(process_library_t) :: lib
    class(prc_core_driver_t), allocatable :: driver
    type(process_constants_t) :: data
    real(default), dimension(0:3,4) :: p

    write (u, "(A)")  "* Test output: prc_test_2"
    write (u, "(A)")  "*   Purpose: create a trivial process"
    write (u, "(A)")  "*            build a library and &
         &access the matrix element"
    write (u, "(A)")

    write (u, "(A)")  "* Build and load a process library with one entry"

    call prc_test_create_library (var_str ("prc_test2"), lib)
    call lib%connect_process (var_str ("prc_test2"), 1, data, driver)

    p = reshape ([ &
         1.0_default, 0.0_default, 0.0_default, 1.0_default, &
         1.0_default, 0.0_default, 0.0_default,-1.0_default, &
         1.0_default, 1.0_default, 0.0_default, 0.0_default, &
         1.0_default,-1.0_default, 0.0_default, 0.0_default &
         ], [4,4])

    write (u, "(A)")
    write (u, "(A)")  "* Compute matrix element:"
    write (u, "(A)")

    select type (driver)
```

```
    type is (prc_test_t)
       write (u, "(1x,A,1x,E11.4)") "|amp| =", abs (driver%get_amplitude (p))
    end select

    call lib%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prc_test_2"

  end subroutine prc_test_2
```

Generate and load the decay process

The process is $s \to f\bar{f}$, where s is a trivial scalar particle and f is a colored fermion. We initialize the process, build the library, and compute the particular matrix element for a fixed momentum configuration. (The decay is independent of angle.) The matrix element is equal to unity.

⟨*Test ME: execute tests*⟩+≡

```
  call test (prc_test_3, "prc_test_3", &
       "build and load trivial decay", &
       u, results)
```

⟨*Test ME: test declarations*⟩+≡

```
  public :: prc_test_3
```

⟨*Test ME: tests*⟩+≡

```
  subroutine prc_test_3 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(process_library_t) :: lib
    class(prc_core_def_t), allocatable :: def
    type(process_def_entry_t), pointer :: entry
    type(string_t) :: model_name
    type(string_t), dimension(:), allocatable :: prt_in, prt_out
    type(process_constants_t) :: data
    class(prc_core_driver_t), allocatable :: driver
    real(default), dimension(0:3,3) :: p
    integer :: i

    write (u, "(A)")  "* Test output: prc_test_3"
    write (u, "(A)")  "*   Purpose: create a trivial decay process"
    write (u, "(A)")  "*            build a library and &
         &access the matrix element"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize a process library with one entry"
    write (u, "(A)")
    call os_data_init (os_data)
    call lib%init (var_str ("prc_test3"))

    model_name = "Test"
    allocate (prt_in (1), prt_out (2))
    prt_in  = [var_str ("s")]
```

```
prt_out = [var_str ("f"), var_str ("F")]

allocate (prc_test_def_t :: def)
select type (def)
type is (prc_test_def_t)
   call def%init (model_name, prt_in, prt_out)
end select
allocate (entry)
call entry%init (var_str ("prc_test3_a"), model_name = model_name, &
     n_in = 1, n_components = 1)
call entry%import_component (1, n_out = size (prt_out), &
     prt_in  = new_prt_spec (prt_in), &
     prt_out = new_prt_spec (prt_out), &
     method  = var_str ("test_me"), &
     variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")
call lib%configure (os_data)

write (u, "(A)")  "* Load library"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                 = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes               = ", &
     lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Constants of prc_test3_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("prc_test3_a"), 1, data, driver)

write (u, "(1x,A,A)")  "component ID     = ", char (data%id)
write (u, "(1x,A,A)")  "model name       = ", char (data%model_name)
write (u, "(1x,A,A,A)")  "md5sum           = '", data%md5sum, "'"
write (u, "(1x,A,L1)") "openmp supported = ", data%openmp_supported
write (u, "(1x,A,I0)") "n_in  = ", data%n_in
write (u, "(1x,A,I0)") "n_out = ", data%n_out
write (u, "(1x,A,I0)") "n_flv = ", data%n_flv
write (u, "(1x,A,I0)") "n_hel = ", data%n_hel
write (u, "(1x,A,I0)") "n_col = ", data%n_col
write (u, "(1x,A,I0)") "n_cin = ", data%n_cin
write (u, "(1x,A,I0)") "n_cf  = ", data%n_cf
write (u, "(1x,A,10(1x,I0))") "flv state =", data%flv_state
```

```
    write (u, "(1x,A,10(1x,I2))") "hel state =", data%hel_state(:,1)
    write (u, "(1x,A,10(1x,I2))") "hel state =", data%hel_state(:,2)
    write (u, "(1x,A,10(1x,I0))") "col state =", data%col_state
    write (u, "(1x,A,10(1x,L1))") "ghost flag =", data%ghost_flag
    write (u, "(1x,A,10(1x,F5.3))") "color factors =", data%color_factors
    write (u, "(1x,A,10(1x,I0))") "cf index =", data%cf_index

    write (u, "(A)")
    write (u, "(A)")  "* Set kinematics:"
    write (u, "(A)")

    p = reshape ([ &
         125._default, 0.0_default, 0.0_default, 0.0_default, &
         62.5_default, 0.0_default, 0.0_default, 62.5_default, &
         62.5_default, 0.0_default, 0.0_default,-62.5_default &
         ], [4,3])
    do i = 1, 3
       write (u, "(2x,A,I0,A,4(1x,F8.4))")  "p", i, " =", p(:,i)
    end do

    write (u, "(A)")
    write (u, "(A)")  "* Compute matrix element:"
    write (u, "(A)")

    select type (driver)
    type is (prc_test_t)
       write (u, "(1x,A,1x,E11.4)") "|amp| =", abs (driver%get_amplitude (p))
    end select

    call lib%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prc_test_3"

  end subroutine prc_test_3
```

Shortcut

This is identical to the previous test, but we create the library be a single command. This is handy for other modules which use the test process.

⟨*Test ME: execute tests*⟩+≡

```
  call test (prc_test_4, "prc_test_4", &
       "build and load trivial decay using shortcut", &
       u, results)
```

⟨*Test ME: test declarations*⟩+≡

```
  public :: prc_test_4
```

⟨*Test ME: tests*⟩+≡

```
  subroutine prc_test_4 (u)
    integer, intent(in) :: u
    type(process_library_t) :: lib
    class(prc_core_driver_t), allocatable :: driver
```

```
  type(process_constants_t) :: data
  real(default), dimension(0:3,3) :: p

  write (u, "(A)")  "* Test output: prc_test_4"
  write (u, "(A)")  "*   Purpose: create a trivial decay process"
  write (u, "(A)")  "*            build a library and &
       &access the matrix element"
  write (u, "(A)")

  write (u, "(A)")  "* Build and load a process library with one entry"

  call prc_test_create_library (var_str ("prc_test4"), lib, &
       scattering=.false., decay=.true.)
  call lib%connect_process (var_str ("prc_test4"), 1, data, driver)

  p = reshape ([ &
       125._default, 0.0_default, 0.0_default, 0.0_default, &
       62.5_default, 0.0_default, 0.0_default, 62.5_default, &
       62.5_default, 0.0_default, 0.0_default,-62.5_default &
       ], [4,3])

  write (u, "(A)")
  write (u, "(A)")  "* Compute matrix element:"
  write (u, "(A)")

  select type (driver)
  type is (prc_test_t)
     write (u, "(1x,A,1x,E11.4)") "|amp| =", abs (driver%get_amplitude (p))
  end select

  call lib%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: prc_test_4"

end subroutine prc_test_4
```

Chapter 14

Particles

This chapter collects modules that implement particle objects, for use in event records.

While within interactions, all correlations are manifest, a particle array is derived by selecting a particular quantum number set. This involves tracing over all other particles, as far as polarization is concerned. Thus, a particle has definite flavor, color, and a single-particle density matrix for polarization.

su_algebra We make use of $su(N)$ generators as the basis for representing polarization matrices. This module defines the basis and provides the necessary transformation routines.

bloch_vectors This defines polarization objects in Bloch representation. The object describes the spin density matrix of a particle, currently restricted to spin $0 \ldots 2$.

polarizations This extends the basic polarization object such that it supports properties of physical particles and appropriate constructors.

particles Particle objects and particle lists, as the base of event records.

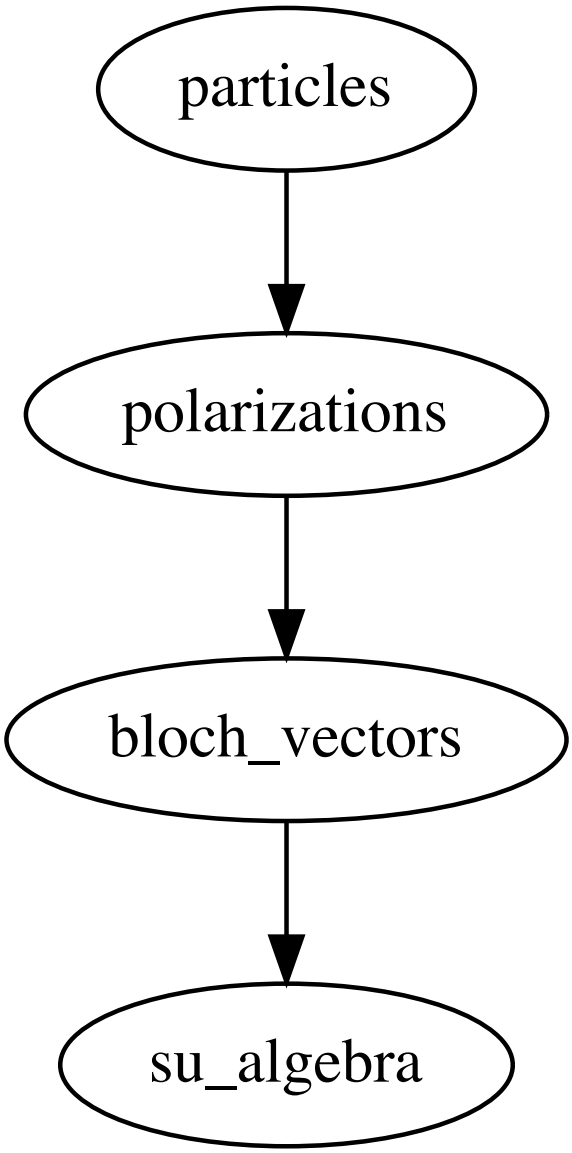

Figure 14.1: Module dependencies in `src/particles`.

14.1 $su(N)$ Algebra

We need a specific choice of basis for a well-defined component representation. The matrix elements of T^a are ordered as $m = \ell, \ell-1, \ldots -\ell$, i.e., from highest down to lowest weight, for both row and column.

We list first the generators of the $su(2)$ subalgebras which leave $|m|$ invariant ($|m| \neq 0$):

$$T^{b+1,b+2,b+3} \equiv \sigma^{1,2,3} \tag{14.1}$$

acting on the respective subspace $|m| = \ell, \ell-1, \ldots$ for $b = 0, 1, \ldots$. This defines generators T^a for $a = 1, \ldots 3N/2$ ($\ldots 3(N-1)/2$) for N even (odd), respectively.

The following generators successively extend this to $su(4)$, $su(6)$, ... until $su(N)$ by adding first the missing off-diagonal and then diagonal generators. The phase conventions are analogous.

(It should be possible to code these conventions for generic spin, but in the current implementation we restrict ourselves to $s \leq 2$, i.e., $N \leq 5$.)

⟨`su_algebra.f90`⟩≡

```
⟨File header⟩

module su_algebra

⟨Use kinds⟩
  use physics_defs, only: SCALAR, SPINOR, VECTOR, VECTORSPINOR, TENSOR

⟨Standard module head⟩

⟨su algebra: public⟩

contains

⟨su algebra: procedures⟩

end module su_algebra
```

14.1.1 $su(N)$ fundamental representation

The dimension of the basis for a given spin type. consecutively, starting at `SCALAR=1`.

⟨*su algebra: public*⟩≡

```
public :: algebra_dimension
```

⟨*su algebra: procedures*⟩≡

```
function algebra_dimension (s) result (n)
  integer :: n
  integer, intent(in) :: s
  n = fundamental_dimension (s) ** 2 - 1
end function algebra_dimension
```

The dimension of the fundamental (defining) representation that we use. This implementation assumes that the spin type is numerically equal to the fundamental dimension.

⟨*su algebra: public*⟩+≡

```
public :: fundamental_dimension
```

⟨*su algebra: procedures*⟩+≡

```
function fundamental_dimension (s) result (d)
  integer :: d
  integer, intent(in) :: s
  d = s
end function fundamental_dimension
```

14.1.2 Mapping between helicity and matrix index

Return the helicity that corresponds to a particular entry in the polarization matrix representation. Helicities are counted downwards, in integers, and zero helicity is included (omitted) for odd (even) spin, respectively.

⟨*su algebra: public*⟩+≡

```
public :: helicity_value
```

⟨*su algebra: procedures*⟩+≡

```
function helicity_value (s, i) result (h)
  integer :: h
  integer, intent(in) :: s, i
  integer, dimension(1), parameter :: hh1 = [0]
  integer, dimension(2), parameter :: hh2 = [1, -1]
  integer, dimension(3), parameter :: hh3 = [1,  0, -1]
  integer, dimension(4), parameter :: hh4 = [2,  1, -1, -2]
  integer, dimension(5), parameter :: hh5 = [2,  1,  0, -1, -2]
  h = 0
  select case (s)
  case (SCALAR)
     select case (i)
     case (1:1);  h = hh1(i)
     end select
  case (SPINOR)
     select case (i)
     case (1:2);  h = hh2(i)
     end select
  case (VECTOR)
     select case (i)
     case (1:3);  h = hh3(i)
     end select
  case (VECTORSPINOR)
     select case (i)
     case (1:4);  h = hh4(i)
     end select
  case (TENSOR)
     select case (i)
     case (1:5);  h = hh5(i)
     end select
  end select
end function helicity_value
```

Inverse: return the index that corresponds to a certain helicity value in the chosen representation.

⟨*su algebra: public*⟩+≡

```
public :: helicity_index
```

⟨*su algebra: procedures*⟩+≡

```
function helicity_index (s, h) result (i)
  integer, intent(in) :: s, h
  integer :: i
  integer, dimension(0:0), parameter :: hi1 = [1]
  integer, dimension(-1:1), parameter :: hi2 = [2, 0, 1]
  integer, dimension(-1:1), parameter :: hi3 = [3, 2, 1]
  integer, dimension(-2:2), parameter :: hi4 = [4, 3, 0, 2, 1]
  integer, dimension(-2:2), parameter :: hi5 = [5, 4, 3, 2, 1]
  select case (s)
  case (SCALAR)
     i = hi1(h)
  case (SPINOR)
     i = hi2(h)
  case (VECTOR)
     i = hi3(h)
  case (VECTORSPINOR)
     i = hi4(h)
  case (TENSOR)
     i = hi5(h)
  end select
end function helicity_index
```

14.1.3 Generator Basis: Cartan Generators

For each supported spin type, we return specific properties of the set of generators via inquiry functions. This is equivalent to using explicit representations of the generators.

For easy access, the properties are hard-coded and selected via case expressions.

Return true if the generator #`i` is in the Cartan subalgebra, i.e., a diagonal matrix for spin type `s`.

⟨*su algebra: public*⟩+≡

```
public :: is_cartan_generator
```

⟨*su algebra: procedures*⟩+≡

```
elemental function is_cartan_generator (s, i) result (cartan)
  logical :: cartan
  integer, intent(in) :: s, i
  select case (s)
  case (SCALAR)
  case (SPINOR)
     select case (i)
     case (3);  cartan = .true.
     case default
        cartan = .false.
     end select
  case (VECTOR)
     select case (i)
     case (3,8);  cartan = .true.
     case default
```

```
         cartan = .false.
      end select
   case (VECTORSPINOR)
      select case (i)
      case (3,6,15);  cartan = .true.
      case default
         cartan = .false.
      end select
   case (TENSOR)
      select case (i)
      case (3,6,15,24);  cartan = .true.
      case default
         cartan = .false.
      end select
   case default
      cartan = .false.
   end select
end function is_cartan_generator
```

Return the index of Cartan generator #k in the chosen representation. This has to conform to cartan above.

⟨*su algebra: public*⟩+≡

```
public :: cartan_index
```

⟨*su algebra: procedures*⟩+≡

```
elemental function cartan_index (s, k) result (ci)
  integer :: ci
  integer, intent(in) :: s, k
  integer, dimension(1), parameter :: ci2 = [3]
  integer, dimension(2), parameter :: ci3 = [3,8]
  integer, dimension(3), parameter :: ci4 = [3,6,15]
  integer, dimension(4), parameter :: ci5 = [3,6,15,24]
  select case (s)
  case (SPINOR)
     ci = ci2(k)
  case (VECTOR)
     ci = ci3(k)
  case (VECTORSPINOR)
     ci = ci4(k)
  case (TENSOR)
     ci = ci5(k)
  case default
     ci = 0
  end select
end function cartan_index
```

The element #k of the result vector a is equal to the (h, h) diagonal entry of the generator matrix T^k. That is, evaluating this for all allowed values of h, we recover the set of Cartan generator matrices.

⟨*su algebra: public*⟩+≡

```
public :: cartan_element
```

⟨*su algebra: procedures*⟩+≡

```
function cartan_element (s, h) result (a)
```

```
real(default), dimension(:), allocatable :: a
integer, intent(in) :: s, h
real(default), parameter :: sqrt2 = sqrt (2._default)
real(default), parameter :: sqrt3 = sqrt (3._default)
real(default), parameter :: sqrt10 = sqrt (10._default)
allocate (a (algebra_dimension (s)), source = 0._default)
select case (s)
case (SCALAR)
case (SPINOR)
   select case (h)
   case (1)
      a(3) =  1._default / 2
   case (-1)
      a(3) = -1._default / 2
   end select
case (VECTOR)
   select case (h)
   case (1)
      a(3) =  1._default / 2
      a(8) =  1._default / (2 * sqrt3)
   case (-1)
      a(3) = -1._default / 2
      a(8) =  1._default / (2 * sqrt3)
   case (0)
      a(8) = -1._default / sqrt3
   end select
case (VECTORSPINOR)
   select case (h)
   case (2)
      a(3)  =  1._default / 2
      a(15) =  1._default / (2 * sqrt2)
   case (-2)
      a(3)  = -1._default / 2
      a(15) =  1._default / (2 * sqrt2)
   case (1)
      a(6)  =  1._default / 2
      a(15) = -1._default / (2 * sqrt2)
   case (-1)
      a(6)  = -1._default / 2
      a(15) = -1._default / (2 * sqrt2)
   end select
case (TENSOR)
   select case (h)
   case (2)
      a(3)  =  1._default / 2
      a(15) =  1._default / (2 * sqrt2)
      a(24) =  1._default / (2 * sqrt10)
   case (-2)
      a(3)  = -1._default / 2
      a(15) =  1._default / (2 * sqrt2)
      a(24) =  1._default / (2 * sqrt10)
   case (1)
      a(6)  =  1._default / 2
      a(15) = -1._default / (2 * sqrt2)
```

```
         a(24) =  1._default / (2 * sqrt10)
      case (-1)
         a(6)  = -1._default / 2
         a(15) = -1._default / (2 * sqrt2)
         a(24) =  1._default / (2 * sqrt10)
      case (0)
         a(24) = -4._default / (2 * sqrt10)
      end select
    end select
  end function cartan_element
```

Given an array of diagonal matrix elements `rd` of a generator, compute the array `a` of basis coefficients. The array must be ordered as defined by `helicity_value`, i.e., highest weight first. The calculation is organized such that the trace of the generator, i.e., the sum of `rd` values, drops out. The result array `a` has coefficients for all basis generators, but only Cartan generators can get a nonzero coefficient.

⟨*su algebra: public*⟩+≡

```
  public :: cartan_coeff
```

⟨*su algebra: procedures*⟩+≡

```
  function cartan_coeff (s, rd) result (a)
    real(default), dimension(:), allocatable :: a
    integer, intent(in) :: s
    real(default), dimension(:), intent(in) :: rd
    real(default), parameter :: sqrt2 = sqrt (2._default)
    real(default), parameter :: sqrt3 = sqrt (3._default)
    real(default), parameter :: sqrt10 = sqrt (10._default)
    integer :: n
    n = algebra_dimension (s)
    allocate (a (n), source = 0._default)
    select case (s)
    case (SPINOR)
       a(3) = rd(1) - rd(2)
    case (VECTOR)
       a(3) = rd(1) - rd(3)
       a(8) = (rd(1) - 2 * rd(2) + rd(3)) / sqrt3
    case (VECTORSPINOR)
       a(3) = rd(1) - rd(4)
       a(6) = rd(2) - rd(3)
       a(15) = (rd(1) - rd(2) - rd(3) + rd(4)) / sqrt2
    case (TENSOR)
       a(3) = rd(1) - rd(5)
       a(6) = rd(2) - rd(4)
       a(15) = (rd(1) - rd(2) - rd(4) + rd(5)) / sqrt2
       a(24) = (rd(1) + rd(2) - 4 * rd(3) + rd(4) + rd(5)) / sqrt10
    end select
  end function cartan_coeff
```

14.1.4 Roots (Off-Diagonal Generators)

Return the appropriate generator index for a given off-diagonal helicity combination. We require $h_1 > h_2$. We return the index of the appropriate real-valued generator if `r` is true, else the complex-valued one.

This is separate from the `cartan_coeff` function above. The reason is that the off-diagonal generators have only a single nonzero matrix element, so there is a one-to-one correspondence of helicity and index.

⟨*su algebra: public*⟩+≡

```
public :: root_index
```

⟨*su algebra: procedures*⟩+≡

```
function root_index (s, h1, h2, r) result (ai)
  integer :: ai
  integer, intent(in) :: s, h1, h2
  logical :: r
  ai = 0
  select case (s)
  case (SCALAR)
  case (SPINOR)
     select case (h1)
     case (1)
        select case (h2)
        case (-1);  ai = 1
        end select
     end select
  case (VECTOR)
     select case (h1)
     case (1)
        select case (h2)
        case (-1);  ai = 1
        case (0);   ai = 4
        end select
     case (0)
        select case (h2)
        case (-1);  ai = 6
        end select
     end select
  case (VECTORSPINOR)
     select case (h1)
     case (2)
        select case (h2)
        case (-2);  ai = 1
        case (1);   ai = 7
        case (-1);  ai = 11
        end select
     case (1)
        select case (h2)
        case (-1);  ai = 4
        case (-2);  ai = 13
        end select
     case (-1)
        select case (h2)
        case (-2);  ai = 9
```

```
         end select
      end select
   case (TENSOR)
      select case (h1)
      case (2)
         select case (h2)
         case (-2);  ai = 1
         case (1);   ai = 7
         case (-1);  ai = 11
         case (0);   ai = 16
         end select
      case (1)
         select case (h2)
         case (-1);  ai = 4
         case (-2);  ai = 13
         case (0);   ai = 20
         end select
      case (-1)
         select case (h2)
         case (-2);  ai = 9
         end select
      case (0)
         select case (h2)
         case (-2);  ai = 18
         case (-1);  ai = 22
         end select
      end select
   end select
   if (ai /= 0 .and. .not. r)  ai = ai + 1
end function root_index
```

Inverse: return the helicity values ($h_2 > h_1$) for an off-diagonal generator. The flag `r` tells whether this is a real or diagonal generator. The others are Cartan generators.

⟨*su algebra: public*⟩+≡

```
public :: root_helicity
```

⟨*su algebra: procedures*⟩+≡

```
subroutine root_helicity (s, i, h1, h2, r)
  integer, intent(in) :: s, i
  integer, intent(out) :: h1, h2
  logical, intent(out) :: r
  h1 = 0
  h2 = 0
  r  = .false.
  select case (s)
  case (SCALAR)
  case (SPINOR)
     select case (i)
     case ( 1, 2);  h1 =  1;  h2 = -1;  r = i == 1
     end select
  case (VECTOR)
     select case (i)
     case ( 1, 2);  h1 =  1;  h2 = -1;  r = i == 1
```

```
         case ( 4, 5);  h1 =  1;  h2 =  0;  r = i == 4
         case ( 6, 7);  h1 =  0;  h2 = -1;  r = i == 6
         end select
      case (VECTORSPINOR)
         select case (i)
         case ( 1, 2);  h1 =  2;  h2 = -2;  r = i == 1
         case ( 4, 5);  h1 =  1;  h2 = -1;  r = i == 4
         case ( 7, 8);  h1 =  2;  h2 =  1;  r = i == 7
         case ( 9,10);  h1 = -1;  h2 = -2;  r = i == 9
         case (11,12);  h1 =  2;  h2 = -1;  r = i ==11
         case (13,14);  h1 =  1;  h2 = -2;  r = i ==13
         end select
      case (TENSOR)
         select case (i)
         case ( 1, 2);  h1 =  2;  h2 = -2;  r = i == 1
         case ( 4, 5);  h1 =  1;  h2 = -1;  r = i == 4
         case ( 7, 8);  h1 =  2;  h2 =  1;  r = i == 7
         case ( 9,10);  h1 = -1;  h2 = -2;  r = i == 9
         case (11,12);  h1 =  2;  h2 = -1;  r = i ==11
         case (13,14);  h1 =  1;  h2 = -2;  r = i ==13
         case (16,17);  h1 =  2;  h2 =  0;  r = i ==16
         case (18,19);  h1 =  0;  h2 = -2;  r = i ==18
         case (20,21);  h1 =  1;  h2 =  0;  r = i ==20
         case (22,23);  h1 =  0;  h2 = -1;  r = i ==22
         end select
      end select
    end subroutine root_helicity
```

14.1.5 Unit tests

Test module, followed by the corresponding implementation module.

⟨su_algebra_ut.f90⟩≡
  ⟨*File header*⟩

```
module su_algebra_ut
  use unit_tests
  use su_algebra_uti
```

⟨*Standard module head*⟩

⟨*su algebra: public test*⟩

```
contains
```

⟨*su algebra: test driver*⟩

```
end module su_algebra_ut
```

⟨su_algebra_uti.f90⟩≡
  ⟨*File header*⟩

```
module su_algebra_uti
```

```
⟨Use kinds⟩
  use physics_defs, only: SCALAR, SPINOR, VECTOR, VECTORSPINOR, TENSOR
  use su_algebra

⟨Standard module head⟩

⟨su algebra: test declarations⟩

contains

⟨su algebra: tests⟩

end module su_algebra_uti
```

API: driver for the unit tests below.

```
⟨su algebra: public test⟩≡
    public :: su_algebra_test
```

```
⟨su algebra: test driver⟩≡
    subroutine su_algebra_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨su algebra: execute tests⟩
    end subroutine su_algebra_test
```

Generator Ordering

Show the position of Cartan generators in the sequence of basis generators.

```
⟨su algebra: execute tests⟩≡
    call test (su_algebra_1, "su_algebra_1", &
         "generator ordering", &
         u, results)
```

```
⟨su algebra: test declarations⟩≡
    public :: su_algebra_1
```

```
⟨su algebra: tests⟩≡
    subroutine su_algebra_1 (u)
      integer, intent(in) :: u

      write (u, "(A)")  "* Test output: su_algebra_1"
      write (u, "(A)")  "*   Purpose: test su(N) algebra implementation"
      write (u, "(A)")

      write (u, "(A)")  "* su(N) generators: &
           &list and mark Cartan subalgebra"

      write (u, "(A)")
      write (u, "(A)")  "* s = 0"
      call cartan_check (SCALAR)

      write (u, "(A)")
      write (u, "(A)")  "* s = 1/2"
```

```
    call cartan_check (SPINOR)

    write (u, "(A)")
    write (u, "(A)")  "* s = 1"
    call cartan_check (VECTOR)

    write (u, "(A)")
    write (u, "(A)")  "* s = 3/2"
    call cartan_check (VECTORSPINOR)

    write (u, "(A)")
    write (u, "(A)")  "* s = 2"
    call cartan_check (TENSOR)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: su_algebra_1"

  contains

    subroutine cartan_check (s)
      integer, intent(in) :: s
      integer :: i
      write (u, *)
      do i = 1, algebra_dimension (s)
         write (u, "(1x,L1)", advance="no")  is_cartan_generator (s, i)
      end do
      write (u, *)

    end subroutine cartan_check

  end subroutine su_algebra_1
```

Cartan Generator Basis

Show the explicit matrix representation for all Cartan generators and check their traces and Killing products.

Also test helicity index mappings.

⟨*su algebra: execute tests*⟩+≡

```
  call test (su_algebra_2, "su_algebra_2", &
       "Cartan generator representation", &
       u, results)
```

⟨*su algebra: test declarations*⟩+≡

```
  public :: su_algebra_2
```

⟨*su algebra: tests*⟩+≡

```
  subroutine su_algebra_2 (u)
    integer, intent(in) :: u

    write (u, "(A)")  "* Test output: su_algebra_2"
    write (u, "(A)")  "*   Purpose: test su(N) algebra implementation"
    write (u, "(A)")
```

```
  write (u, "(A)")  "* diagonal su(N) generators: &
       &show explicit representation"
  write (u, "(A)")  "* and check trace and Killing form"

  write (u, "(A)")
  write (u, "(A)")  "* s = 1/2"
  call cartan_show (SPINOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 1"
  call cartan_show (VECTOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 3/2"
  call cartan_show (VECTORSPINOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 2"
  call cartan_show (TENSOR)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: su_algebra_2"

contains

  subroutine cartan_show (s)
    integer, intent(in) :: s
    real(default), dimension(:,:), allocatable :: rd
    integer, dimension(:), allocatable :: ci
    integer :: n, d, h, i, j, k, l

    n = algebra_dimension (s)
    d = fundamental_dimension (s)

    write (u, *)
    write (u, "(A2,5X)", advance="no")  "h:"
    do i = 1, d
       j = helicity_index (s, helicity_value (s, i))
       write (u, "(1x,I2,5X)", advance="no")  helicity_value (s, j)
    end do
    write (u, "(8X)", advance="no")
    write (u, "(1X,A)")  "tr"

    allocate (rd (n,d), source = 0._default)
    do i = 1, d
       h = helicity_value (s, i)
       rd(:,i) = cartan_element (s, h)
    end do

    allocate (ci (d-1), source = 0)
    do k = 1, d-1
       ci(k) = cartan_index (s, k)
    end do
```

```
      write (u, *)
      do k = 1, d-1
         write (u, "('T',I2,':',1X)", advance="no")  ci(k)
         do i = 1, d
            write (u, 1, advance="no")  rd(ci(k),i)
         end do
         write (u, "(8X)", advance="no")
         write (u, 1)  sum (rd(ci(k),:))
      end do

      write (u, *)
      write (u, "(6X)", advance="no")
      do k = 1, d-1
         write (u, "(2X,'T',I2,3X)", advance="no")  ci(k)
      end do
      write (u, *)

      do k = 1, d-1
         write (u, "('T',I2,2X)", advance="no")  ci(k)
         do l = 1, d-1
            write (u, 1, advance="no")  dot_product (rd(ci(k),:), rd(ci(l),:))
         end do
         write (u, *)
      end do

1     format (1x,F7.4)

    end subroutine cartan_show

  end subroutine su_algebra_2
```

Bloch Representation: Cartan Generators

Transform from Bloch vectors to matrix and back, considering Cartan generators only.

⟨*su algebra: execute tests*⟩+≡

```
    call test (su_algebra_3, "su_algebra_3", &
         "Cartan generator mapping", &
         u, results)
```

⟨*su algebra: test declarations*⟩+≡

```
    public :: su_algebra_3
```

⟨*su algebra: tests*⟩+≡

```
    subroutine su_algebra_3 (u)
      integer, intent(in) :: u

      write (u, "(A)")  "* Test output: su_algebra_3"
      write (u, "(A)")  "*   Purpose: test su(N) algebra implementation"
      write (u, "(A)")

      write (u, "(A)")  "* diagonal su(N) generators: &
         &transform to matrix and back"
```

```
  write (u, "(A)")
  write (u, "(A)")  "* s = 1/2"
  call cartan_expand (SPINOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 1"
  call cartan_expand (VECTOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 3/2"
  call cartan_expand (VECTORSPINOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 2"
  call cartan_expand (TENSOR)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: su_algebra_3"

contains

  subroutine cartan_expand (s)
    integer, intent(in) :: s
    real(default), dimension(:,:), allocatable :: rd
    integer, dimension(:), allocatable :: ci
    real(default), dimension(:), allocatable :: a
    logical, dimension(:), allocatable :: mask
    integer :: n, d, h, i, k, l

    n = algebra_dimension (s)
    d = fundamental_dimension (s)

    allocate (rd (n,d), source = 0._default)
    do i = 1, d
       h = helicity_value (s, i)
       rd(:,i) = cartan_element (s, h)
    end do

    allocate (ci (d-1), source = 0)
    do k = 1, d-1
       ci(k) = cartan_index (s, k)
    end do

    allocate (a (n))

    write (u, *)
    do k = 1, d-1
       a(:) = cartan_coeff (s, rd(ci(k),:))
       write (u, "('T',I2,':',1X)", advance="no")  ci(k)
       do i = 1, n
          if (is_cartan_generator (s, i)) then
             write (u, 1, advance="no")  a(i)
          else if (a(i) /= 0) then
```

```
              ! this should not happen (nonzero non-Cartan entry)
              write (u, "(1X,':',I2,':',3X)", advance="no")  i
           end if
        end do
        write (u, *)
     end do

1    format (1X,F7.4)

   end subroutine cartan_expand

 end subroutine su_algebra_3
```

Bloch Representation: Roots

List the mapping between helicity transitions and (real) off-diagonal generators.

⟨*su algebra: execute tests*⟩+≡

```
call test (su_algebra_4, "su_algebra_4", &
     "Root-helicity mapping", &
     u, results)
```

⟨*su algebra: test declarations*⟩+≡

```
public :: su_algebra_4
```

⟨*su algebra: tests*⟩+≡

```
subroutine su_algebra_4 (u)
  integer, intent(in) :: u

  write (u, "(A)")  "* Test output: su_algebra_4"
  write (u, "(A)")  "*   Purpose: test su(N) algebra implementation"
  write (u, "(A)")

  write (u, "(A)")  "* off-diagonal su(N) generators: &
       &mapping from/to helicity pair"

  write (u, "(A)")
  write (u, "(A)")  "* s = 1/2"
  call root_expand (SPINOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 1"
  call root_expand (VECTOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 3/2"
  call root_expand (VECTORSPINOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 2"
  call root_expand (TENSOR)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: su_algebra_4"
```

```
contains

  subroutine root_expand (s)
    integer, intent(in) :: s
    integer :: n, d, i, j, h1, h2
    logical :: r

    n = algebra_dimension (s)

    write (u, *)
    do i = 1, n
       if (is_cartan_generator (s, i))  cycle
       call root_helicity (s, i, h1, h2, r)
       j = root_index (s, h1, h2, r)
       write (u, "('T',I2,':')", advance="no")  j
       write (u, "(2(1x,I2))", advance="no")  h1, h2
       if (r) then
          write (u, *)
       else
          write (u, "('*')")
       end if
    end do

  end subroutine root_expand

end subroutine su_algebra_4
```

14.2 Bloch Representation

Particle polarization is determined by a particular quantum state which has just helicity information. Physically, this is the spin density matrix ρ, where we do not restrict ourselves to pure states.

We adopt the phase convention for a spin-1/2 particle that

$$\rho = \tfrac{1}{2}(1 + \vec{\alpha} \cdot \vec{\sigma}) \tag{14.2}$$

with the polarization axis $\vec{\alpha}$. For a particle with arbitrary spin s, and thus $N = 2s + 1$ spin states, we extend the above definition to generalized Bloch form

$$\rho = \frac{1}{N}\left(1 + \sqrt{2N(N-1)}\alpha^a T^a\right) \tag{14.3}$$

where the T^a $(a = 1, \ldots N^2 - 1)$ are a basis of $su(N)$ algebra generators. These $N \times N$ matrices are hermitean, traceless, and orthogonal via

$$\mathrm{Tr}\, T^a T^b = \frac{1}{2}\delta^{ab} \tag{14.4}$$

In the spin-1/2 case, this reduces to the above (standard Bloch) representation since $T^a = \sigma^a/2$, $a = 1, 2, 3$. For the spin-1 case, we could use $T^a = \lambda^a/2$ with the Gell-Mann matrices,

$$\rho = \frac{1}{3}\left(1 + \sqrt{3}\alpha^a \lambda^a\right), \tag{14.5}$$

The normalization is chosen that $|alpha| \leq 1$ for allowed density matrix, where $|\alpha| = 1$ is a necessary, but not sufficient, condition for a pure state.

We need a specific choice of basis for a well-defined component representation. The matrix elements of T^a are ordered as $m = \ell, \ell - 1, \ldots - \ell$, i.e., from highest down to lowest weight, for both row and column.

We list first the generators of the $su(2)$ subalgebras which leave $|m|$ invariant $(|m| \neq 0)$:

$$T^{b+1,b+2,b+3} \equiv \sigma^{1,2,3} \tag{14.6}$$

acting on the respective subspace $|m| = \ell, \ell - 1, \ldots$ for $b = 0, 1, \ldots$. This defines generators T^a for $a = 1, \ldots 3N/2$ $(\ldots 3(N-1)/2)$ for N even (odd), respectively.

The following generators successively extend this to $su(4)$, $su(6)$, $\ldots$ until $su(N)$ by adding first the missing off-diagonal and then diagonal generators. The phase conventions are analogous.

(It should be possible to code these conventions for generic spin, but in the current implementation we restrict ourselves to $s \leq 2$, i.e., $N \leq 5$.)

Particle polarization is determined by a particular quantum state which has just helicity information. Physically, this is the spin density matrix ρ, where we do not restrict ourselves to pure states.

We adopt the phase convention for a spin-1/2 particle that

$$\rho = \tfrac{1}{2}(1 + \vec{\alpha} \cdot \vec{\sigma}) \tag{14.7}$$

with the polarization axis $\vec{\alpha}$. For a particle with arbitrary spin s, and thus $N = 2s + 1$ spin states, we extend the above definition to generalized Bloch form

$$\rho = \frac{1}{N}\left(1 + \sqrt{2N(N-1)}\alpha^a T^a\right) \tag{14.8}$$

where the T^a $(a = 1, \ldots N^2 - 1)$ are a basis of $su(N)$ algebra generators. These $N \times N$ matrices are hermitean, traceless, and orthogonal via

$$\mathrm{Tr}\, T^a T^b = \frac{1}{2}\delta^{ab} \tag{14.9}$$

In the spin-1/2 case, this reduces to the above (standard Bloch) representation since $T^a = \sigma^a/2$, $a = 1, 2, 3$. For the spin-1 case, we could use $T^a = \lambda^a/2$ with the Gell-Mann matrices,

$$\rho = \frac{1}{3}\left(1 + \sqrt{3}\alpha^a \lambda^a\right), \tag{14.10}$$

The normalization is chosen that $|alpha| \leq 1$ for allowed density matrix, where $|\alpha| = 1$ is a necessary, but not sufficient, condition for a pure state.

⟨bloch_vectors.f90⟩≡

```
⟨File header⟩

module bloch_vectors

⟨Use kinds⟩
  use physics_defs, only: UNKNOWN, SCALAR, SPINOR, VECTOR, VECTORSPINOR, TENSOR
  use su_algebra

⟨Standard module head⟩

⟨Bloch vectors: public⟩

⟨Bloch vectors: types⟩

contains

⟨Bloch vectors: procedures⟩

end module bloch_vectors
```

14.2.1 Preliminaries

The normalization factor $\sqrt{2N(N-1)}/N$ that enters the Bloch representation.

⟨Bloch vectors: procedures⟩≡

```
function bloch_factor (s) result (f)
  real(default) :: f
  integer, intent(in) :: s
  select case (s)
  case (SCALAR)
     f = 0
  case (SPINOR)
     f = 1
  case (VECTOR)
     f = 2 * sqrt (3._default) / 3
  case (VECTORSPINOR)
     f = 2 * sqrt (6._default) / 4
  case (TENSOR)
     f = 2 * sqrt (10._default) / 5
```

```
      case default
         f = 0
      end select
    end function bloch_factor
```

14.2.2 The basic polarization type

The basic polarization object holds just the entries of the Bloch vector as an allocatable array.

Bloch is active whenever the coefficient array is allocated. For convenience, we store the spin type ($2s$) and the multiplicity (N) together with the coefficient array (α). We have to allow for the massless case where s is arbitrary > 0 but $N = 2$, and furthermore the chiral massless case where $N = 1$. In the latter case, the array remains deallocated but the chirality is set to ± 1.

In the Bloch vector implementation, we do not distinguish between particle and antiparticle. If the distinction applies, it must be made by the caller when transforming between density matrix and Bloch vector.

⟨*Bloch vectors: public*⟩≡

```
    public :: bloch_vector_t
```

⟨*Bloch vectors: types*⟩≡

```
    type :: bloch_vector_t
       private
       integer :: spin_type = UNKNOWN
       real(default), dimension(:), allocatable :: a
     contains
     ⟨Bloch vectors: bloch vector: TBP⟩
  end type bloch_vector_t
```

14.2.3 Direct Access

This basic initializer just sets the spin type, leaving the Bloch vector unallocated. The object therefore does not support nonzero polarization.

⟨*Bloch vectors: bloch vector: TBP*⟩≡

```
    procedure :: init_unpolarized => bloch_vector_init_unpolarized
```

⟨*Bloch vectors: procedures*⟩+≡

```
    subroutine bloch_vector_init_unpolarized (pol, spin_type)
      class(bloch_vector_t), intent(out) :: pol
      integer, intent(in) :: spin_type
      pol%spin_type = spin_type
    end subroutine bloch_vector_init_unpolarized
```

The standard initializer allocates the Bloch vector and initializes with zeros, so we can define a polarization later. We make sure that this works only for the supported spin type. Initializing with `UNKNOWN` spin type resets the Bloch vector to undefined, i.e., unpolarized state.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡

```
    generic :: init => bloch_vector_init
    procedure, private :: bloch_vector_init
```

⟨*Bloch vectors: procedures*⟩+≡

```
subroutine bloch_vector_init (pol, spin_type)
  class(bloch_vector_t), intent(out) :: pol
  integer, intent(in) :: spin_type
  pol%spin_type = spin_type
  select case (spin_type)
  case (SCALAR,SPINOR,VECTOR,VECTORSPINOR,TENSOR)
     allocate (pol%a (algebra_dimension (spin_type)), source = 0._default)
  end select
end subroutine bloch_vector_init
```

Fill the Bloch vector from an array, no change of normalization. No initialization and no check, we assume that the shapes do match.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡

```
procedure :: from_array => bloch_vector_from_array
```

⟨*Bloch vectors: procedures*⟩+≡

```
subroutine bloch_vector_from_array (pol, a)
  class(bloch_vector_t), intent(inout) :: pol
  real(default), dimension(:), allocatable, intent(in) :: a
  pol%a(:) = a
end subroutine bloch_vector_from_array
```

Transform to an array of reals, i.e., extract the Bloch vector as-is.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡

```
procedure :: to_array => bloch_vector_to_array
```

⟨*Bloch vectors: procedures*⟩+≡

```
subroutine bloch_vector_to_array (pol, a)
  class(bloch_vector_t), intent(in) :: pol
  real(default), dimension(:), allocatable, intent(out) :: a
  if (pol%is_defined ())  allocate (a (size (pol%a)), source = pol%a)
end subroutine bloch_vector_to_array
```

14.2.4 Raw I/O

⟨*Bloch vectors: bloch vector: TBP*⟩+≡

```
procedure :: write_raw => bloch_vector_write_raw
procedure :: read_raw => bloch_vector_read_raw
```

⟨*Bloch vectors: procedures*⟩+≡

```
subroutine bloch_vector_write_raw (pol, u)
  class(bloch_vector_t), intent(in) :: pol
  integer, intent(in) :: u
  write (u) pol%spin_type
  write (u) allocated (pol%a)
  if (allocated (pol%a)) then
     write (u) pol%a
  end if
end subroutine bloch_vector_write_raw

subroutine bloch_vector_read_raw (pol, u, iostat)
  class(bloch_vector_t), intent(out) :: pol
```

```
    integer, intent(in) :: u
    integer, intent(out) :: iostat
    integer :: s
    logical :: polarized
    read (u, iostat=iostat) s
    read (u, iostat=iostat) polarized
    if (iostat /= 0)  return
    if (polarized) then
       call pol%init (s)
       read (u, iostat=iostat) pol%a
    else
       call pol%init_unpolarized (s)
    end if
  end subroutine bloch_vector_read_raw
```

14.2.5 Properties

Re-export algebra functions that depend on the spin type. These functions do not depend on the Bloch vector being allocated.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡

```
  procedure :: get_n_states
  procedure :: get_length
  procedure :: hel_index => bv_helicity_index
  procedure :: hel_value => bv_helicity_value
  procedure :: bloch_factor => bv_factor
```

⟨*Bloch vectors: procedures*⟩+≡

```
  function get_n_states (pol) result (n)
    class(bloch_vector_t), intent(in) :: pol
    integer :: n
    n = fundamental_dimension (pol%spin_type)
  end function get_n_states

  function get_length (pol) result (n)
    class(bloch_vector_t), intent(in) :: pol
    integer :: n
    n = algebra_dimension (pol%spin_type)
  end function get_length

  function bv_helicity_index (pol, h) result (i)
    class(bloch_vector_t), intent(in) :: pol
    integer, intent(in) :: h
    integer :: i
    i = helicity_index (pol%spin_type, h)
  end function bv_helicity_index

  function bv_helicity_value (pol, i) result (h)
    class(bloch_vector_t), intent(in) :: pol
    integer, intent(in) :: i
    integer :: h
    h = helicity_value (pol%spin_type, i)
  end function bv_helicity_value
```

```
function bv_factor (pol) result (f)
  class(bloch_vector_t), intent(in) :: pol
  real(default) :: f
  f = bloch_factor (pol%spin_type)
end function bv_factor
```

If the Bloch vector object is defined, the spin type is anything else but `UNKNOWN`. This allows us the provide the representation-specific functions above.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡

```
procedure :: is_defined => bloch_vector_is_defined
```

⟨*Bloch vectors: procedures*⟩+≡

```
function bloch_vector_is_defined (pol) result (flag)
  class(bloch_vector_t), intent(in) :: pol
  logical :: flag
  flag = pol%spin_type /= UNKNOWN
end function bloch_vector_is_defined
```

If the Bloch vector object is (technically) polarized, it is defined, and the vector coefficient array has been allocated. However, the vector value may be zero.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡

```
procedure :: is_polarized => bloch_vector_is_polarized
```

⟨*Bloch vectors: procedures*⟩+≡

```
function bloch_vector_is_polarized (pol) result (flag)
  class(bloch_vector_t), intent(in) :: pol
  logical :: flag
  flag = allocated (pol%a)
end function bloch_vector_is_polarized
```

Return true if the polarization is diagonal, i.e., all entries in the density matrix are on the diagonal. This is equivalent to requiring that only Cartan generator coefficients are nonzero in the Bloch vector.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡

```
procedure :: is_diagonal => bloch_vector_is_diagonal
```

⟨*Bloch vectors: procedures*⟩+≡

```
function bloch_vector_is_diagonal (pol) result (diagonal)
  class(bloch_vector_t), intent(in) :: pol
  logical :: diagonal
  integer :: s, i
  s = pol%spin_type
  diagonal = .true.
  if (pol%is_polarized ()) then
     do i = 1, size (pol%a)
        if (is_cartan_generator (s, i))  cycle
        if (pol%a(i) /= 0) then
           diagonal = .false.
           return
        end if
     end do
  end if
end function bloch_vector_is_diagonal
```

Return the Euclidean norm of the Bloch vector. This is equal to the Killing form value of the corresponding algebra generator. We assume that the polarization object has been initialized.

For a pure state, the norm is unity. All other allowed states have a norm less than unity. (For $s \geq 1$, this is a necessary but not sufficient condition.)

⟨*Bloch vectors: bloch vector: TBP*⟩+≡
```
procedure :: get_norm => bloch_vector_get_norm
```

⟨*Bloch vectors: procedures*⟩+≡
```
function bloch_vector_get_norm (pol) result (norm)
  class(bloch_vector_t), intent(in) :: pol
  real(default) :: norm
  select case (pol%spin_type)
  case (SPINOR,VECTOR,VECTORSPINOR,TENSOR)
     norm = sqrt (dot_product (pol%a, pol%a))
  case default
     norm = 1
  end select
end function bloch_vector_get_norm
```

14.2.6 Diagonal density matrix

This initializer takes a diagonal density matrix, represented by a real-valued array. We assume that the trace is unity, and that the array has the correct shape for the given `spin_type`.

The `bloch_factor` renormalization is necessary such that a pure state maps to a Bloch vector with unit norm.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡
```
generic :: init => bloch_vector_init_diagonal
procedure, private :: bloch_vector_init_diagonal
```

⟨*Bloch vectors: procedures*⟩+≡
```
subroutine bloch_vector_init_diagonal (pol, spin_type, rd)
  class(bloch_vector_t), intent(out) :: pol
  integer, intent(in) :: spin_type
  real(default), dimension(:), intent(in) :: rd
  call pol%init (spin_type)
  call pol%set (rd)
end subroutine bloch_vector_init_diagonal
```

Set a Bloch vector, given a diagonal density matrix as a real array. The Bloch vector must be initialized with correct characteristics.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡
```
generic :: set => bloch_vector_set_diagonal
procedure, private :: bloch_vector_set_diagonal
```

⟨*Bloch vectors: procedures*⟩+≡
```
subroutine bloch_vector_set_diagonal (pol, rd)
  class(bloch_vector_t), intent(inout) :: pol
  real(default), dimension(:), intent(in) :: rd
  integer :: s
  s = pol%spin_type
```

```
    select case (s)
    case (SCALAR,SPINOR,VECTOR,VECTORSPINOR,TENSOR)
       pol%a(:) = cartan_coeff (s, rd) / bloch_factor (s)
    end select
  end subroutine bloch_vector_set_diagonal
```

14.2.7 Massless density matrix

This is a specific variant which initializes an equipartition for the maximum helicity, corresponding to an unpolarized massless particle.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡

```
  procedure :: init_max_weight => bloch_vector_init_max_weight
```

⟨*Bloch vectors: procedures*⟩+≡

```
  subroutine bloch_vector_init_max_weight (pol, spin_type)
    class(bloch_vector_t), intent(out) :: pol
    integer, intent(in) :: spin_type
    call pol%init (spin_type)
    select case (spin_type)
    case (VECTOR)
       call pol%set ([0.5_default, 0._default, 0.5_default])
    case (VECTORSPINOR)
       call pol%set ([0.5_default, 0._default, 0._default, 0.5_default])
    case (TENSOR)
       call pol%set ([0.5_default, 0._default, 0._default, 0._default, 0.5_default])
    end select
  end subroutine bloch_vector_init_max_weight
```

Initialize the maximum-weight submatrix with a three-component Bloch vector. This is not as trivial as it seems because we need the above initialization for the generalized Bloch in order to remove the lower weights from the density matrix.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡

```
  procedure :: init_vector => bloch_vector_init_vector
  procedure :: to_vector => bloch_vector_to_vector
```

⟨*Bloch vectors: procedures*⟩+≡

```
  subroutine bloch_vector_init_vector (pol, s, a)
    class(bloch_vector_t), intent(out) :: pol
    integer, intent(in) :: s
    real(default), dimension(3), intent(in) :: a
    call pol%init_max_weight (s)
    select case (s)
    case (SPINOR, VECTOR, VECTORSPINOR, TENSOR)
       pol%a(1:3) = a / bloch_factor (s)
    end select
  end subroutine bloch_vector_init_vector

  subroutine bloch_vector_to_vector (pol, a)
    class(bloch_vector_t), intent(in) :: pol
    real(default), dimension(3), intent(out) :: a
    integer :: s
    s = pol%spin_type
```

```
    select case (s)
    case (SPINOR, VECTOR, VECTORSPINOR, TENSOR)
       a = pol%a(1:3) * bloch_factor (s)
    case default
       a = 0
    end select
  end subroutine bloch_vector_to_vector
```

14.2.8 Arbitrary density matrix

Initialize the Bloch vector from a density matrix. We assume that the density is valid. In particular, the shape should match, the matrix should be hermitian, and the trace should be unity.

We first fill the diagonal, then add the off-diagonal parts.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡

```
  generic :: init => bloch_vector_init_matrix
  procedure, private :: bloch_vector_init_matrix
```

⟨*Bloch vectors: procedures*⟩+≡

```
  subroutine bloch_vector_init_matrix (pol, spin_type, r)
    class(bloch_vector_t), intent(out) :: pol
    integer, intent(in) :: spin_type
    complex(default), dimension(:,:), intent(in) :: r
    select case (spin_type)
    case (SCALAR,SPINOR,VECTOR,VECTORSPINOR,TENSOR)
       call pol%init (spin_type)
       call pol%set (r)
    case default
       call pol%init (UNKNOWN)
    end select
  end subroutine bloch_vector_init_matrix
```

Set a Bloch vector, given an arbitrary density matrix as a real array. The Bloch vector must be initialized with correct characteristics.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡

```
  generic :: set => bloch_vector_set_matrix
  procedure, private :: bloch_vector_set_matrix
```

⟨*Bloch vectors: procedures*⟩+≡

```
  subroutine bloch_vector_set_matrix (pol, r)
    class(bloch_vector_t), intent(inout) :: pol
    complex(default), dimension(:,:), intent(in) :: r
    real(default), dimension(:), allocatable :: rd
    integer :: s, d, i, j, h1, h2, ir, ii
    s = pol%spin_type
    select case (s)
    case (SCALAR,SPINOR,VECTOR,VECTORSPINOR,TENSOR)
       d = fundamental_dimension (s)
       allocate (rd (d))
       do i = 1, d
          rd(i) = r(i,i)
       end do
```

```
      call pol%set (rd)
      do i = 1, d
         h1 = helicity_value (s, i)
         do j = i+1, d
            h2 = helicity_value (s, j)
            ir = root_index (s, h1, h2, .true.)
            ii = root_index (s, h1, h2, .false.)
            pol%a(ir) = real (r(j,i) + r(i,j)) / bloch_factor (s)
            pol%a(ii) = aimag (r(j,i) - r(i,j)) / bloch_factor (s)
         end do
      end do
   end select
end subroutine bloch_vector_set_matrix
```

Allocate and fill the density matrix `r` (with the index ordering as defined in `su_algebra`) that corresponds to a given Bloch vector.

If the optional `only_max_weight` is set, the resulting matrix has entries only for $\pm h_{\max}$, as appropriate for a massless particle (for spin ≥ 1). Note that we always add the unit matrix, as this is part of the Bloch-vector definition.

⟨*Bloch vectors: bloch vector: TBP*⟩+≡

```
procedure :: to_matrix => bloch_vector_to_matrix
```

⟨*Bloch vectors: procedures*⟩+≡

```
subroutine bloch_vector_to_matrix (pol, r, only_max_weight)
  class(bloch_vector_t), intent(in) :: pol
  complex(default), dimension(:,:), intent(out), allocatable :: r
  logical, intent(in), optional :: only_max_weight
  integer :: d, s, h0, ng, ai, h, h1, h2, i, j
  logical :: is_real, only_max
  complex(default) :: val
  if (.not. pol%is_polarized ())  return
  s = pol%spin_type
  only_max = .false.
  select case (s)
  case (VECTOR, VECTORSPINOR, TENSOR)
     if (present (only_max_weight))  only_max = only_max_weight
  end select
  if (only_max) then
     ng = 2
     h0 = helicity_value (s, 1)
  else
     ng = algebra_dimension (s)
     h0 = 0
  end if
  d = fundamental_dimension (s)
  allocate (r (d, d), source = (0._default, 0._default))
  do i = 1, d
     h = helicity_value (s, i)
     if (abs (h) < h0)  cycle
     r(i,i) = 1._default / d &
          + dot_product (cartan_element (s, h), pol%a) * bloch_factor (s)
  end do
  do ai = 1, ng
     if (is_cartan_generator (s, ai))  cycle
```

```
      call root_helicity (s, ai, h1, h2, is_real)
      i = helicity_index (s, h1)
      j = helicity_index (s, h2)
      if (is_real) then
         val = cmplx (pol%a(ai) / 2 * bloch_factor (s), 0._default, &
              kind=default)
         r(i,j) = r(i,j) + val
         r(j,i) = r(j,i) + val
      else
         val = cmplx (0._default, pol%a(ai) / 2 * bloch_factor (s), &
              kind=default)
         r(i,j) = r(i,j) - val
         r(j,i) = r(j,i) + val
      end if
    end do
  end subroutine bloch_vector_to_matrix
```

14.2.9 Unit tests

Test module, followed by the corresponding implementation module.

⟨bloch_vectors_ut.f90⟩≡

```
  ⟨File header⟩

  module bloch_vectors_ut
    use unit_tests
    use bloch_vectors_uti

  ⟨Standard module head⟩

  ⟨Bloch vectors: public test⟩

  contains

  ⟨Bloch vectors: test driver⟩

  end module bloch_vectors_ut
```

⟨bloch_vectors_uti.f90⟩≡

```
  ⟨File header⟩

  module bloch_vectors_uti

  ⟨Use kinds⟩
    use physics_defs, only: UNKNOWN, SCALAR, SPINOR, VECTOR, VECTORSPINOR, TENSOR
    use su_algebra, only: algebra_dimension, fundamental_dimension, helicity_value

    use bloch_vectors

  ⟨Standard module head⟩

  ⟨Bloch vectors: test declarations⟩

  contains
```

⟨*Bloch vectors: tests*⟩

```
end module bloch_vectors_uti
```

API: driver for the unit tests below.

⟨*Bloch vectors: public test*⟩≡

```
public :: bloch_vectors_test
```

⟨*Bloch vectors: test driver*⟩≡

```
subroutine bloch_vectors_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```

⟨*Bloch vectors: execute tests*⟩

```
end subroutine bloch_vectors_test
```

Initialization

Initialize the Bloch vector for any spin type. First as unpolarized (no array), then as polarized but with zero polarization.

⟨*Bloch vectors: execute tests*⟩≡

```
call test (bloch_vectors_1, "bloch_vectors_1", &
     "initialization", &
     u, results)
```

⟨*Bloch vectors: test declarations*⟩≡

```
public :: bloch_vectors_1
```

⟨*Bloch vectors: tests*⟩≡

```
subroutine bloch_vectors_1 (u)
  integer, intent(in) :: u

  write (u, "(A)")  "* Test output: bloch_vectors_1"
  write (u, "(A)")  "*   Purpose: test Bloch-vector &
       &polarization implementation"
  write (u, "(A)")

  write (u, "(A)")  "* Initialization (unpolarized)"

  write (u, "(A)")
  write (u, "(A)")  "* unknown"
  call bloch_init (UNKNOWN)

  write (u, "(A)")
  write (u, "(A)")  "* s = 0"
  call bloch_init (SCALAR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 1/2"
  call bloch_init (SPINOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 1"
```

```
  call bloch_init (VECTOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 3/2"
  call bloch_init (VECTORSPINOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 2"
  call bloch_init (TENSOR)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: bloch_vectors_1"

contains

  subroutine bloch_init (s)
    integer, intent(in) :: s
    type(bloch_vector_t) :: pol
    real(default), dimension(:), allocatable :: a
    integer :: i
    write (u, *)
    write (u, "(1X,L1,L1)", advance="no") &
         pol%is_defined (), pol%is_polarized ()
    call pol%init_unpolarized (s)
    write (u, "(1X,L1,L1)", advance="no") &
         pol%is_defined (), pol%is_polarized ()
    call pol%init (s)
    write (u, "(1X,L1,L1)", advance="no") &
         pol%is_defined (), pol%is_polarized ()
    write (u, *)
    call pol%to_array (a)
    if (allocated (a)) then
       write (u, "(*(F7.4))")  a
       a(:) = [(real (mod (i, 10), kind=default), i = 1, size (a))]
       call pol%from_array (a)
       call pol%to_array (a)
       write (u, "(*(F7.4))")  a
    else
       write (u, *)
       write (u, *)
    end if
  end subroutine bloch_init

end subroutine bloch_vectors_1
```

Pure state (diagonal)

Initialize the Bloch vector with a pure state of definite helicity and check the normalization.

⟨*Bloch vectors: execute tests*⟩+≡

```
call test (bloch_vectors_2, "bloch_vectors_2", &
     "pure state (diagonal)", &
```

```
        u, results)
```

⟨*Bloch vectors: test declarations*⟩+≡

```
    public :: bloch_vectors_2
```

⟨*Bloch vectors: tests*⟩+≡

```
    subroutine bloch_vectors_2 (u)
      integer, intent(in) :: u

      write (u, "(A)")  "* Test output: bloch_vectors_2"
      write (u, "(A)")  "*   Purpose: test Bloch-vector &
           &polarization implementation"
      write (u, "(A)")

      write (u, "(A)")  "* Initialization (polarized, diagonal): &
           &display vector and norm"
      write (u, "(A)")  "*   transform back"

      write (u, "(A)")
      write (u, "(A)")  "* s = 0"
      call bloch_diagonal (SCALAR)

      write (u, "(A)")
      write (u, "(A)")  "* s = 1/2"
      call bloch_diagonal (SPINOR)

      write (u, "(A)")
      write (u, "(A)")  "* s = 1"
      call bloch_diagonal (VECTOR)

      write (u, "(A)")
      write (u, "(A)")  "* s = 3/2"
      call bloch_diagonal (VECTORSPINOR)

      write (u, "(A)")
      write (u, "(A)")  "* s = 2"
      call bloch_diagonal (TENSOR)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: bloch_vectors_2"

    contains

      subroutine bloch_diagonal (s)
        integer, intent(in) :: s
        type(bloch_vector_t) :: pol
        real(default), dimension(:), allocatable :: a
        real(default), dimension(:), allocatable :: rd
        complex(default), dimension(:,:), allocatable :: r
        integer :: i, j, d
        real(default) :: rj
        real, parameter :: tolerance = 1.E-14_default
        d = fundamental_dimension (s)
        do i = 1, d
           allocate (rd (d), source = 0._default)
```

```
         rd(i) = 1
         call pol%init (s, rd)
         call pol%to_array (a)
         write (u, *)
         write (u, "(A,1X,I2)")  "h:", helicity_value (s, i)
         write (u, 1, advance="no")  a
         write (u, "(1X,L1)")  pol%is_diagonal ()
         write (u, 1)  pol%get_norm ()
         call pol%to_matrix (r)
         do j = 1, d
            rj = real (r(j,j))
            if (abs (rj) < tolerance)  rj = 0
            write (u, 1, advance="no")  rj
         end do
         write (u, "(1X,L1)")  matrix_is_diagonal (r)
         deallocate (a, rd, r)
      end do
1     format (99(1X,F7.4,:))
    end subroutine bloch_diagonal

    function matrix_is_diagonal (r) result (diagonal)
      complex(default), dimension(:,:), intent(in) :: r
      logical :: diagonal
      integer :: i, j
      diagonal = .true.
      do j = 1, size (r, 2)
         do i = 1, size (r, 1)
            if (i == j)  cycle
            if (r(i,j) /= 0) then
               diagonal = .false.
               return
            end if
         end do
      end do
    end function matrix_is_diagonal

  end subroutine bloch_vectors_2
```

Pure state (arbitrary)

Initialize the Bloch vector with an arbitrarily chosen pure state, check the normalization, and transform back to the density matrix.

⟨*Bloch vectors: execute tests*⟩+≡

```
  call test (bloch_vectors_3, "bloch_vectors_3", &
       "pure state (arbitrary)", &
       u, results)
```

⟨*Bloch vectors: test declarations*⟩+≡

```
  public :: bloch_vectors_3
```

⟨*Bloch vectors: tests*⟩+≡

```
  subroutine bloch_vectors_3 (u)
    integer, intent(in) :: u
```

```
    write (u, "(A)")  "* Test output: bloch_vectors_3"
    write (u, "(A)")  "*   Purpose: test Bloch-vector &
         &polarization implementation"
    write (u, "(A)")

    write (u, "(A)")  "* Initialization (pure polarized, arbitrary):"
    write (u, "(A)")  "*   input matrix, transform, display norm, transform back"

    write (u, "(A)")
    write (u, "(A)")  "* s = 0"
    call bloch_arbitrary (SCALAR)

    write (u, "(A)")
    write (u, "(A)")  "* s = 1/2"
    call bloch_arbitrary (SPINOR)

    write (u, "(A)")
    write (u, "(A)")  "* s = 1"
    call bloch_arbitrary (VECTOR)

    write (u, "(A)")
    write (u, "(A)")  "* s = 3/2"
    call bloch_arbitrary (VECTORSPINOR)

    write (u, "(A)")
    write (u, "(A)")  "* s = 2"
    call bloch_arbitrary (TENSOR)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: bloch_vectors_3"

  contains

    subroutine bloch_arbitrary (s)
      integer, intent(in) :: s
      type(bloch_vector_t) :: pol
      complex(default), dimension(:,:), allocatable :: r
      integer :: d
      d = fundamental_dimension (s)
      write (u, *)
      call init_matrix (d, r)
      call write_matrix (d, r)
      call pol%init (s, r)
      write (u, *)
      write (u, 2)  pol%get_norm (), pol%is_diagonal ()
      write (u, *)
      call pol%to_matrix (r)
      call write_matrix (d, r)
2     format (1X,F7.4,1X,L1)
    end subroutine bloch_arbitrary

    subroutine init_matrix (d, r)
      integer, intent(in) :: d
```

```
      complex(default), dimension(:,:), allocatable, intent(out) :: r
      complex(default), dimension(:), allocatable :: a
      real(default) :: norm
      integer :: i, j
      allocate (a (d))
      norm = 0
      do i = 1, d
         a(i) = cmplx (2*i-1, 2*i, kind=default)
         norm = norm + conjg (a(i)) * a(i)
      end do
      a = a / sqrt (norm)
      allocate (r (d,d))
      do i = 1, d
         do j = 1, d
            r(i,j) = conjg (a(i)) * a(j)
         end do
      end do
    end subroutine init_matrix

    subroutine write_matrix (d, r)
      integer, intent(in) :: d
      complex(default), dimension(:,:), intent(in) :: r
      integer :: i, j
      do i = 1, d
         do j = 1, d
            write (u, 1, advance="no")  r(i,j)
         end do
         write (u, *)
      end do
1     format (99(1X,'(',F7.4,',',F7.4,')',:))
    end subroutine write_matrix

  end subroutine bloch_vectors_3
```

Raw I/O

Check correct input/output in raw format.

⟨*Bloch vectors: execute tests*⟩+≡

```
  call test (bloch_vectors_4, "bloch_vectors_4", &
       "raw I/O", &
       u, results)
```

⟨*Bloch vectors: test declarations*⟩+≡

```
  public :: bloch_vectors_4
```

⟨*Bloch vectors: tests*⟩+≡

```
  subroutine bloch_vectors_4 (u)
    integer, intent(in) :: u

    write (u, "(A)")  "* Test output: bloch_vectors_4"
    write (u, "(A)")  "*   Purpose: test Bloch-vector &
         &polarization implementation"
    write (u, "(A)")
```

```
  write (u, "(A)")  "* Raw I/O"

  write (u, "(A)")
  write (u, "(A)")  "* s = 0"
  call bloch_io (SCALAR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 1/2"
  call bloch_io (SPINOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 1"
  call bloch_io (VECTOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 3/2"
  call bloch_io (VECTORSPINOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 2"
  call bloch_io (TENSOR)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: bloch_vectors_4"

contains

  subroutine bloch_io (s)
    integer, intent(in) :: s
    type(bloch_vector_t) :: pol
    real(default), dimension(:), allocatable :: a
    integer :: n, i, utmp, iostat
    n = algebra_dimension (s)
    allocate (a (n))
    a(:) = [(real (mod (i, 10), kind=default), i = 1, size (a))]
    write (u, *)
    write (u, "(*(F7.4))")  a
    call pol%init (s)
    call pol%from_array (a)
    open (newunit = utmp, status = "scratch", action = "readwrite", &
         form = "unformatted")
    call pol%write_raw (utmp)
    rewind (utmp)
    call pol%read_raw (utmp, iostat=iostat)
    close (utmp)
    call pol%to_array (a)
    write (u, "(*(F7.4))")  a
  end subroutine bloch_io

end subroutine bloch_vectors_4
```

Convenience Methods

Check some further TBP that are called by the `polarizations` module.

⟨*Bloch vectors: execute tests*⟩+≡

```
call test (bloch_vectors_5, "bloch_vectors_5", &
     "massless state (unpolarized)", &
     u, results)
```

⟨*Bloch vectors: test declarations*⟩+≡

```
public :: bloch_vectors_5
```

⟨*Bloch vectors: tests*⟩+≡

```
subroutine bloch_vectors_5 (u)
  integer, intent(in) :: u

  write (u, "(A)")  "* Test output: bloch_vectors_5"
  write (u, "(A)")  "*   Purpose: test Bloch-vector &
       &polarization implementation"
  write (u, "(A)")

  write (u, "(A)")  "* Massless states: equipartition"

  write (u, "(A)")
  write (u, "(A)")  "* s = 0"
  call bloch_massless_unpol (SCALAR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 1/2"
  call bloch_massless_unpol (SPINOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 1"
  call bloch_massless_unpol (VECTOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 3/2"
  call bloch_massless_unpol (VECTORSPINOR)

  write (u, "(A)")
  write (u, "(A)")  "* s = 2"
  call bloch_massless_unpol (TENSOR)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: bloch_vectors_5"

contains

  subroutine bloch_massless_unpol (s)
    integer, intent(in) :: s
    type(bloch_vector_t) :: pol
    complex(default), dimension(:,:), allocatable :: r
    real(default), dimension(:), allocatable :: a
    integer :: d
    d = fundamental_dimension (s)
    call pol%init_max_weight (s)
```

```
      call pol%to_matrix (r, only_max_weight = .false.)
      write (u, *)
      where (abs (r) < 1.e-14_default)  r = 0
      call write_matrix (d, r)
      call pol%to_matrix (r, only_max_weight = .true.)
      write (u, *)
      call write_matrix (d, r)
    end subroutine bloch_massless_unpol

    subroutine write_matrix (d, r)
      integer, intent(in) :: d
      complex(default), dimension(:,:), intent(in) :: r
      integer :: i, j
      do i = 1, d
         do j = 1, d
            write (u, 1, advance="no")  r(i,j)
         end do
         write (u, *)
      end do
1     format (99(1X,'(',F7.4,',',F7.4,')',:))
    end subroutine write_matrix

  end subroutine bloch_vectors_5
```

Massless state (arbitrary)

Initialize the Bloch vector with an arbitrarily chosen pure state which consists only of highest-weight components. Transform back to the density matrix.

⟨*Bloch vectors: execute tests*⟩+≡

```
  call test (bloch_vectors_6, "bloch_vectors_6", &
       "massless state (arbitrary)", &
       u, results)
```

⟨*Bloch vectors: test declarations*⟩+≡

```
  public :: bloch_vectors_6
```

⟨*Bloch vectors: tests*⟩+≡

```
  subroutine bloch_vectors_6 (u)
    integer, intent(in) :: u

    write (u, "(A)")  "* Test output: bloch_vectors_6"
    write (u, "(A)")  "*   Purpose: test Bloch-vector &
         &polarization implementation"
    write (u, "(A)")

    write (u, "(A)")  "* Initialization (pure polarized massless, arbitrary):"
    write (u, "(A)")  "*   input matrix, transform, display norm, transform back"

    write (u, "(A)")
    write (u, "(A)")  "* s = 0"
    call bloch_massless (SCALAR)

    write (u, "(A)")
```

```
    write (u, "(A)")  "* s = 1/2"
    call bloch_massless (SPINOR)

    write (u, "(A)")
    write (u, "(A)")  "* s = 1"
    call bloch_massless (VECTOR)

    write (u, "(A)")
    write (u, "(A)")  "* s = 3/2"
    call bloch_massless (VECTORSPINOR)

    write (u, "(A)")
    write (u, "(A)")  "* s = 2"
    call bloch_massless (TENSOR)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: bloch_vectors_6"

  contains

    subroutine bloch_massless (s)
      integer, intent(in) :: s
      type(bloch_vector_t) :: pol
      complex(default), dimension(:,:), allocatable :: r
      integer :: d
      d = fundamental_dimension (s)
      write (u, *)
      call init_matrix (d, r)
      call write_matrix (d, r)
      call pol%init (s, r)
      write (u, *)
      write (u, 2)  pol%get_norm (), pol%is_diagonal ()
      write (u, *)
      call pol%to_matrix (r, only_max_weight = .true.)
      call write_matrix (d, r)
2     format (1X,F7.4,1X,L1)
    end subroutine bloch_massless

    subroutine init_matrix (d, r)
      integer, intent(in) :: d
      complex(default), dimension(:,:), allocatable, intent(out) :: r
      complex(default), dimension(:), allocatable :: a
      real(default) :: norm
      integer :: i, j
      allocate (a (d), source = (0._default, 0._default))
      norm = 0
      do i = 1, d, max (d-1, 1)
         a(i) = cmplx (2*i-1, 2*i, kind=default)
         norm = norm + conjg (a(i)) * a(i)
      end do
      a = a / sqrt (norm)
      allocate (r (d,d), source = (0._default, 0._default))
      do i = 1, d, max (d-1, 1)
         do j = 1, d, max (d-1, 1)
```

```
            r(i,j) = conjg (a(i)) * a(j)
         end do
      end do
    end subroutine init_matrix

    subroutine write_matrix (d, r)
      integer, intent(in) :: d
      complex(default), dimension(:,:), intent(in) :: r
      integer :: i, j
      do i = 1, d
         do j = 1, d
            write (u, 1, advance="no")  r(i,j)
         end do
         write (u, *)
      end do
1     format (99(1X,'(',F7.4,',',F7.4,')',:))
    end subroutine write_matrix

  end subroutine bloch_vectors_6
```

Massless state (Bloch vector)

Initialize the (generalized) Bloch vector with an ordinary three-component Bloch vector that applies to the highest-weight part only.

⟨*Bloch vectors: execute tests*⟩+≡

```
  call test (bloch_vectors_7, "bloch_vectors_7", &
       "massless state (vector)", &
       u, results)
```

⟨*Bloch vectors: test declarations*⟩+≡

```
  public :: bloch_vectors_7
```

⟨*Bloch vectors: tests*⟩+≡

```
  subroutine bloch_vectors_7 (u)
    integer, intent(in) :: u

    write (u, "(A)")  "* Test output: bloch_vectors_7"
    write (u, "(A)")  "*   Purpose: test Bloch-vector &
         &polarization implementation"
    write (u, "(A)")

    write (u, "(A)")  "* Initialization &
         &(pure polarized massless, arbitrary Bloch vector):"
    write (u, "(A)")  "*   input vector, transform, display norm, &
         &transform back"

    write (u, "(A)")
    write (u, "(A)")  "* s = 0"
    call bloch_massless_vector (SCALAR)

    write (u, "(A)")
    write (u, "(A)")  "* s = 1/2"
    call bloch_massless_vector (SPINOR)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* s = 1"
    call bloch_massless_vector (VECTOR)

    write (u, "(A)")
    write (u, "(A)")  "* s = 3/2"
    call bloch_massless_vector (VECTORSPINOR)

    write (u, "(A)")
    write (u, "(A)")  "* s = 2"
    call bloch_massless_vector (TENSOR)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: bloch_vectors_7"

  contains

    subroutine bloch_massless_vector (s)
      integer, intent(in) :: s
      type(bloch_vector_t) :: pol
      real(default), dimension(3) :: a
      complex(default), dimension(:,:), allocatable :: r
      write (u, *)
      a =  [1._default, 2._default, 4._default]
      a = a / sqrt (sum (a ** 2))
      write (u, 2)  a
      call pol%init_vector (s, a)
      write (u, 2)  pol%get_norm ()
      call pol%to_vector (a)
      write (u, 2)  a
      call pol%to_matrix (r, only_max_weight = .false.)
      write (u, *)
      where (abs (r) < 1.e-14_default)  r = 0
      call write_matrix (r)
      call pol%to_matrix (r, only_max_weight = .true.)
      write (u, *)
      call write_matrix (r)
2     format (99(1X,F7.4,:))
    end subroutine bloch_massless_vector

    subroutine write_matrix (r)
      complex(default), dimension(:,:), intent(in) :: r
      integer :: i, j
      do i = 1, size (r, 1)
         do j = 1, size (r, 2)
            write (u, 1, advance="no")  r(i,j)
         end do
         write (u, *)
      end do
1     format (99(1X,'(',F7.4,',',F7.4,')',:))
    end subroutine write_matrix

  end subroutine bloch_vectors_7
```

14.3 Polarization

Using generalized Bloch vectors and the $su(N)$ algebra (see above) for the internal representation, we can define various modes of polarization. For spin-1/2, and analogously for massless spin-s particles, we introduce

1. Trivial polarization: $\vec{\alpha} = 0$. [This is unpolarized, but distinct from the particular undefined polarization matrix which has the same meaning.]
2. Circular polarization: $\vec{\alpha}$ points in $\pm z$ direction.
3. Transversal polarization: $\vec{\alpha}$ points orthogonal to the z direction, with a phase ϕ that is 0 for the x axis, and $\pi/2 = 90°$ for the y axis. For antiparticles, the phase switches sign, corresponding to complex conjugation.
4. Axis polarization, where we explicitly give $\vec{\alpha}$.

For higher spin, we retain this definition, but apply it to the two components with maximum and minimum weight. In effect, we concentrate on the first three entries in the α^a array. For massless particles, this is sufficient. For massive particles, we then add the possibilities:

5. Longitudinal polarization: Only the 0-component is set. This is possible only for bosons.
6. Diagonal polarization: Explicitly specify all components in the helicity basis. The $su(N)$ representation consists of diagonal generators only, the Cartan subalgebra.

Obviously, this does not exhaust the possible density matrices for higher spin, but it should cover practical applications.

⟨`polarizations.f90`⟩≡

```
⟨File header⟩

module polarizations

⟨Use kinds⟩
  use io_units
  use format_defs, only: FMT_19
  use diagnostics
  use physics_defs, only: SCALAR, SPINOR, VECTOR, VECTORSPINOR, TENSOR
  use flavors
  use helicities
  use quantum_numbers
  use state_matrices
  use bloch_vectors

⟨Standard module head⟩

⟨Polarizations: public⟩

⟨Polarizations: types⟩

⟨Polarizations: interfaces⟩
```

```
contains
```

⟨*Polarizations: procedures*⟩

```
end module polarizations
```

14.3.1 The polarization type

Polarization is active whenever the coefficient array is allocated. For convenience, we store the spin type ($2s$) and the multiplicity (N) together with the coefficient array (α). We have to allow for the massless case where s is arbitrary > 0 but $N = 2$, and furthermore the chiral massless case where $N = 1$. In the latter case, the array remains deallocated but the chirality is set to ± 1.

There is a convention that an antiparticle transforms according to the complex conjugate representation. We apply this only when transforming from/to polarization defined by a three-vector. For antiparticles, the two-component flips sign in that case. When transforming from/to a state matrix or `pmatrix` representation, we do not apply this sign flip.

TODO: Check these conventions for consistency.

⟨*Polarizations: public*⟩≡

```
public :: polarization_t
```

⟨*Polarizations: types*⟩≡

```
type :: polarization_t
   private
   integer :: spin_type = SCALAR
   integer :: multiplicity = 1
   integer :: chirality = 0
   logical :: anti = .false.
   type(bloch_vector_t) :: bv
 contains
 ⟨Polarizations: polarization: TBP⟩
end type polarization_t
```

14.3.2 Basic initializer and finalizer

We need the particle flavor for determining the allowed helicity values. The Bloch vector is left undefined, so this initializer (in two versions) creates an unpolarized particle. Exception: a chiral particle is always polarized with definite helicity, it doesn't need a Bloch vector.

This is private.

⟨*Polarizations: polarization: TBP*⟩≡

```
generic, private :: init => polarization_init, polarization_init_flv
procedure, private :: polarization_init
procedure, private :: polarization_init_flv
```

⟨*Polarizations: procedures*⟩≡

```
subroutine polarization_init (pol, spin_type, multiplicity, &
     anti, left_handed, right_handed)
  class(polarization_t), intent(out) :: pol
  integer, intent(in) :: spin_type
```

```
    integer, intent(in) :: multiplicity
    logical, intent(in) :: anti
    logical, intent(in) :: left_handed
    logical, intent(in) :: right_handed
    pol%spin_type = spin_type
    pol%multiplicity = multiplicity
    pol%anti = anti
    select case (pol%multiplicity)
    case (1)
       if (left_handed) then
          pol%chirality = -1
       else if (right_handed) then
          pol%chirality = 1
       end if
    end select
    select case (pol%chirality)
    case (0)
       call pol%bv%init_unpolarized (spin_type)
    end select
  end subroutine polarization_init

  subroutine polarization_init_flv (pol, flv)
    class(polarization_t), intent(out) :: pol
    type(flavor_t), intent(in) :: flv
    call pol%init ( &
         spin_type = flv%get_spin_type (), &
         multiplicity = flv%get_multiplicity (), &
         anti = flv%is_antiparticle (), &
         left_handed = flv%is_left_handed (), &
         right_handed = flv%is_right_handed ())
  end subroutine polarization_init_flv
```

Generic polarization: as before, but create a polarized particle (Bloch vector defined) with initial polarization zero.

⟨*Polarizations: polarization: TBP*⟩+≡

```
  generic :: init_generic => &
       polarization_init_generic, &
       polarization_init_generic_flv
  procedure, private :: polarization_init_generic
  procedure, private :: polarization_init_generic_flv
```

⟨*Polarizations: procedures*⟩+≡

```
  subroutine polarization_init_generic (pol, spin_type, multiplicity, &
       anti, left_handed, right_handed)
    class(polarization_t), intent(out) :: pol
    integer, intent(in) :: spin_type
    integer, intent(in) :: multiplicity
    logical, intent(in) :: anti
    logical, intent(in) :: left_handed
    logical, intent(in) :: right_handed
    call pol%init (spin_type, multiplicity, &
         anti, left_handed, right_handed)
    select case (pol%chirality)
    case (0)
```

```
      if (pol%multiplicity == pol%bv%get_n_states ()) then
         call pol%bv%init (spin_type)
      else
         call pol%bv%init_max_weight (spin_type)
      end if
   end select
end subroutine polarization_init_generic

subroutine polarization_init_generic_flv (pol, flv)
   class(polarization_t), intent(out) :: pol
   type(flavor_t), intent(in) :: flv
   call pol%init_generic ( &
        spin_type = flv%get_spin_type (), &
        multiplicity = flv%get_multiplicity (), &
        anti = flv%is_antiparticle (), &
        left_handed = flv%is_left_handed (), &
        right_handed = flv%is_right_handed ())
end subroutine polarization_init_generic_flv
```

A finalizer is no longer necessary.

14.3.3 I/O

The default setting produces a tabular output of the polarization vector entries. Optionally, we can create a state matrix and write its contents, emulating the obsolete original implementation.

If `all_states` is true (default), we generate all helity combinations regardless of the matrix-element value. Otherwise, skip helicities with zero entry, or absolute value less than `tolerance`, if also given.

⟨*Polarizations: polarization: TBP*⟩+≡

```
procedure :: write => polarization_write
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine polarization_write (pol, unit, state_matrix, all_states, tolerance)
   class(polarization_t), intent(in) :: pol
   integer, intent(in), optional :: unit
   logical, intent(in), optional :: state_matrix, all_states
   real(default), intent(in), optional :: tolerance
   logical :: state_m
   type(state_matrix_t) :: state
   real(default), dimension(:), allocatable :: a
   integer :: u, i
   u = given_output_unit (unit);  if (u < 0)  return
   state_m = .false.;  if (present (state_matrix))  state_m = state_matrix
   if (pol%anti) then
      write (u, "(1x,A,I1,A,I1,A,L1,A)")  &
           "Polarization: [spin_type = ", pol%spin_type, &
           ", mult = ", pol%multiplicity, ", anti = ", pol%anti, "]"
   else
      write (u, "(1x,A,I1,A,I1,A)")  &
           "Polarization: [spin_type = ", pol%spin_type, &
           ", mult = ", pol%multiplicity, "]"
   end if
```

```
    if (state_m) then
       call pol%to_state (state, all_states, tolerance)
       call state%write (unit=unit)
       call state%final ()
    else if (pol%chirality == 1) then
       write (u, "(1x,A)")  "chirality = +"
    else if (pol%chirality == -1) then
       write (u, "(1x,A)")  "chirality = -"
    else if (pol%bv%is_polarized ()) then
       call pol%bv%to_array (a)
       do i = 1, size (a)
          write (u, "(1x,I2,':',1x,F10.7)")  i, a(i)
       end do
    else
       write (u, "(1x,A)")  "[unpolarized]"
    end if
  end subroutine polarization_write
```

Binary I/O.

⟨*Polarizations: polarization: TBP*⟩+≡

```
  procedure :: write_raw => polarization_write_raw
  procedure :: read_raw => polarization_read_raw
```

⟨*Polarizations: procedures*⟩+≡

```
  subroutine polarization_write_raw (pol, u)
    class(polarization_t), intent(in) :: pol
    integer, intent(in) :: u
    write (u) pol%spin_type
    write (u) pol%multiplicity
    write (u) pol%chirality
    write (u) pol%anti
    call pol%bv%write_raw (u)
  end subroutine polarization_write_raw

  subroutine polarization_read_raw (pol, u, iostat)
    class(polarization_t), intent(out) :: pol
    integer, intent(in) :: u
    integer, intent(out), optional :: iostat
    read (u, iostat=iostat) pol%spin_type
    read (u, iostat=iostat) pol%multiplicity
    read (u, iostat=iostat) pol%chirality
    read (u, iostat=iostat) pol%anti
    call pol%bv%read_raw (u, iostat)
  end subroutine polarization_read_raw
```

14.3.4 Accessing contents

Return true if the particle is technically polarized. The particle is either chiral, or its Bloch vector has been defined. The function returns true even if the Bloch vector is zero or the particle is scalar.

⟨*Polarizations: polarization: TBP*⟩+≡

```
  procedure :: is_polarized => polarization_is_polarized
```

⟨*Polarizations: procedures*⟩+≡

```
function polarization_is_polarized (pol) result (polarized)
  class(polarization_t), intent(in) :: pol
  logical :: polarized
  polarized = pol%chirality /= 0 .or. pol%bv%is_polarized ()
end function polarization_is_polarized
```

Return true if the polarization is diagonal, i.e., all entries in the density matrix are diagonal. For an unpolarized particle, we also return `.true.` since the density matrix is proportional to the unit matrix.

⟨*Polarizations: polarization: TBP*⟩+≡

```
procedure :: is_diagonal => polarization_is_diagonal
```

⟨*Polarizations: procedures*⟩+≡

```
function polarization_is_diagonal (pol) result (diagonal)
  class(polarization_t), intent(in) :: pol
  logical :: diagonal
  select case (pol%chirality)
  case (0)
     diagonal = pol%bv%is_diagonal ()
  case default
     diagonal = .true.
  end select
end function polarization_is_diagonal
```

14.3.5 Mapping between polarization and state matrix

Create the polarization object that corresponds to a state matrix. The state matrix is not necessarily normalized. The result will be either unpolarized, or a generalized Bloch vector that we compute in terms of the appropriate spin generator basis. To this end, we first construct the complete density matrix, then set the Bloch vector with this input.

For a naturally chiral particle (i.e., neutrino), we do not set the polarization vector, it is implied.

TODO: The state matrix does not support an antiparticle flag. Therefore, we cannot account for any sign flip and transform as-is.

⟨*Polarizations: polarization: TBP*⟩+≡

```
procedure :: init_state_matrix => polarization_init_state_matrix
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine polarization_init_state_matrix (pol, state)
  class(polarization_t), intent(out) :: pol
  type(state_matrix_t), intent(in), target :: state
  type(state_iterator_t) :: it
  type(flavor_t) :: flv
  type(helicity_t) :: hel
  integer :: d, h1, h2, i, j
  complex(default), dimension(:,:), allocatable :: r
  complex(default) :: me
  real(default) :: trace
  call it%init (state)
  flv = it%get_flavor (1)
```

```
  hel = it%get_helicity (1)
  if (hel%is_defined ()) then
     call pol%init_generic (flv)
     select case (pol%chirality)
     case (0)
        trace = 0
        d = pol%bv%get_n_states ()
        allocate (r (d, d), source = (0._default, 0._default))
        do while (it%is_valid ())
           hel = it%get_helicity (1)
           call hel%get_indices (h1, h2)
           i = pol%bv%hel_index (h1)
           j = pol%bv%hel_index (h2)
           me = it%get_matrix_element ()
           r(i,j) = me
           if (i == j)  trace = trace + real (me)
           call it%advance ()
        end do
        if (trace /= 0)  call pol%bv%set (r / trace)
     end select
  else
     call pol%init (flv)
  end if
end subroutine polarization_init_state_matrix
```

Create the state matrix that corresponds to a given polarization. We make use of the polarization iterator as defined below, which should iterate according to the canonical helicity ordering.

⟨*Polarizations: polarization: TBP*⟩+≡

```
procedure :: to_state => polarization_to_state_matrix
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine polarization_to_state_matrix (pol, state, all_states, tolerance)
  class(polarization_t), intent(in), target :: pol
  type(state_matrix_t), intent(out) :: state
  logical, intent(in), optional :: all_states
  real(default), intent(in), optional :: tolerance
  type(polarization_iterator_t) :: it
  type(quantum_numbers_t), dimension(1) :: qn
  complex(default) :: value
  call it%init (pol, all_states, tolerance)
  call state%init (store_values = .true.)
  do while (it%is_valid ())
     value = it%get_value ()
     qn(1) = it%get_quantum_numbers ()
     call state%add_state (qn, value = value)
     call it%advance ()
  end do
  call state%freeze ()
end subroutine polarization_to_state_matrix
```

14.3.6 Specific initializers

Unpolarized particle, no nontrivial entries in the density matrix. This is the default initialization mode.

⟨*Polarizations: polarization: TBP*⟩+≡

```
procedure :: init_unpolarized => polarization_init_unpolarized
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine polarization_init_unpolarized (pol, flv)
  class(polarization_t), intent(out) :: pol
  type(flavor_t), intent(in) :: flv
  call pol%init (flv)
end subroutine polarization_init_unpolarized
```

The following three modes are useful mainly for spin-1/2 particle and massless particles of any nonzero spin. Only the highest-weight components are filled.

Circular polarization: The density matrix of the two highest-weight states is

$$\rho(f) = \frac{1-|f|}{2}\mathbf{1} + |f| \times \begin{cases} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, & f > 0; \\ \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, & f < 0, \end{cases}$$

In the generalized Bloch representation, this is an entry for the T^3 generator only, regardless of the spin representation.

A chiral particle is not affected.

⟨*Polarizations: polarization: TBP*⟩+≡

```
procedure :: init_circular => polarization_init_circular
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine polarization_init_circular (pol, flv, f)
  class(polarization_t), intent(out) :: pol
  type(flavor_t), intent(in) :: flv
  real(default), intent(in) :: f
  call pol%init (flv)
  select case (pol%chirality)
  case (0)
     call pol%bv%init_vector (pol%spin_type, &
          [0._default, 0._default, f])
  end select
end subroutine polarization_init_circular
```

Transversal polarization is analogous to circular, but we get a density matrix

$$\rho(f,\phi) = \frac{1-|f|}{2}\mathbf{1} + \frac{|f|}{2}\begin{pmatrix} 1 & e^{-i\phi} \\ e^{i\phi} & 1 \end{pmatrix}.$$

for the highest-weight subspace. The lower weights are unaffected. The phase is $\phi = 0$ for the x-axis, $\phi = 90°$ for the y axis as polarization vector.

For an antiparticle, the phase switches sign, and for $f < 0$, the off-diagonal elements switch sign.

A chiral particle is not affected.

⟨*Polarizations: polarization: TBP*⟩+≡

```
procedure :: init_transversal => polarization_init_transversal
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine polarization_init_transversal (pol, flv, phi, f)
  class(polarization_t), intent(out) :: pol
  type(flavor_t), intent(in) :: flv
  real(default), intent(in) :: phi, f
  call pol%init (flv)
  select case (pol%chirality)
  case (0)
     if (pol%anti) then
        call pol%bv%init_vector (pol%spin_type, &
             [f * cos (phi), f * sin (phi), 0._default])
     else
        call pol%bv%init_vector (pol%spin_type, &
             [f * cos (phi),-f * sin (phi), 0._default])
     end if
  end select
end subroutine polarization_init_transversal
```

For axis polarization, we again set only the entries with maximum weight, which for spin 1/2 means

$$\rho(f,\phi) = \frac{1}{2}\begin{pmatrix} 1+\alpha_3 & \alpha_1 - i\alpha_2 \\ \alpha_1 + i\alpha_2 & 1-\alpha_3 \end{pmatrix}.$$

For an antiparticle, the imaginary part proportional to α_2 switches sign (complex conjugate). A chiral particle is not affected.

In the generalized Bloch representation, this translates into coefficients for $T^{1,2,3}$, all others stay zero.

⟨*Polarizations: polarization: TBP*⟩+≡

```
procedure :: init_axis => polarization_init_axis
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine polarization_init_axis (pol, flv, alpha)
  class(polarization_t), intent(out) :: pol
  type(flavor_t), intent(in) :: flv
  real(default), dimension(3), intent(in) :: alpha
  call pol%init (flv)
  select case (pol%chirality)
  case (0)
     if (pol%anti) then
        call pol%bv%init_vector (pol%spin_type, &
             [alpha(1), alpha(2), alpha(3)])
     else
        call pol%bv%init_vector (pol%spin_type, &
             [alpha(1),-alpha(2), alpha(3)])
     end if
  end select
end subroutine polarization_init_axis
```

This version specifies the polarization axis in terms of r (polarization degree) and θ, ϕ (polar and azimuthal angles).

If one of the angles is a nonzero multiple of π, roundoff errors typically will result in tiny contributions to unwanted components. Therefore, include a catch

for small numbers.

⟨*Polarizations: polarization: TBP*⟩+≡

```
procedure :: init_angles => polarization_init_angles
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine polarization_init_angles (pol, flv, r, theta, phi)
  class(polarization_t), intent(out) :: pol
  type(flavor_t), intent(in) :: flv
  real(default), intent(in) :: r, theta, phi
  real(default), dimension(3) :: alpha
  real(default), parameter :: eps = 10 * epsilon (1._default)

  alpha(1) = r * sin (theta) * cos (phi)
  alpha(2) = r * sin (theta) * sin (phi)
  alpha(3) = r * cos (theta)
  where (abs (alpha) < eps)  alpha = 0
  call pol%init_axis (flv, alpha)
end subroutine polarization_init_angles
```

Longitudinal polarization is defined only for massive bosons. Only the zero component is filled. Otherwise, unpolarized.

In the generalized Bloch representation, the zero component corresponds to a linear combination of all diagonal (Cartan) generators.

⟨*Polarizations: polarization: TBP*⟩+≡

```
procedure :: init_longitudinal => polarization_init_longitudinal
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine polarization_init_longitudinal (pol, flv, f)
  class(polarization_t), intent(out) :: pol
  type(flavor_t), intent(in) :: flv
  real(default), intent(in) :: f
  real(default), dimension(:), allocatable :: rd
  integer :: s, d
  s = flv%get_spin_type ()
  select case (s)
  case (VECTOR, TENSOR)
     call pol%init_generic (flv)
     if (pol%bv%is_polarized ()) then
        d = pol%bv%get_n_states ()
        allocate (rd (d), source = 0._default)
        rd(pol%bv%hel_index (0)) = f
        call pol%bv%set (rd)
     end if
  case default
     call pol%init_unpolarized (flv)
  end select
end subroutine polarization_init_longitudinal
```

This is diagonal polarization: we specify all components explicitly. `rd` is the array of diagonal elements of the density matrix. We assume that the length of `rd` is equal to the particle multiplicity.

⟨*Polarizations: polarization: TBP*⟩+≡

```
procedure :: init_diagonal => polarization_init_diagonal
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine polarization_init_diagonal (pol, flv, rd)
  class(polarization_t), intent(out) :: pol
  type(flavor_t), intent(in) :: flv
  real(default), dimension(:), intent(in) :: rd
  real(default) :: trace
  call pol%init_generic (flv)
  if (pol%bv%is_polarized ()) then
     trace = sum (rd)
     if (trace /= 0)  call pol%bv%set (rd / trace)
  end if
end subroutine polarization_init_diagonal
```

14.3.7 Operations

Combine polarization states by computing the outer product of the state matrices.

⟨*Polarizations: public*⟩+≡

```
public :: combine_polarization_states
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine combine_polarization_states (pol, state)
  type(polarization_t), dimension(:), intent(in), target :: pol
  type(state_matrix_t), intent(out) :: state
  type(state_matrix_t), dimension(size(pol)), target :: pol_state
  integer :: i
  do i = 1, size (pol)
     call pol(i)%to_state (pol_state(i))
  end do
  call outer_multiply (pol_state, state)
  do i = 1, size (pol)
     call pol_state(i)%final ()
  end do
end subroutine combine_polarization_states
```

Transform a polarization density matrix into a polarization vector. This is possible without information loss only for spin-1/2 and for massless particles. To get a unique answer in all cases, we consider only the components with highest weight. Obviously, this loses the longitudinal component of a massive vector, for instance. The norm of the returned axis is the polarization fraction for the highest-weight subspace. For a scalar particle, we return a zero vector. The same result applies if the highest-weight component vanishes.

This is the inverse operation of `polarization_init_axis` above, where the polarization fraction is set to unity.

For an antiparticle, the `alpha(2)` coefficient flips sign.

⟨*Polarizations: polarization: TBP*⟩+≡

```
procedure :: get_axis => polarization_get_axis
```

⟨*Polarizations: procedures*⟩+≡

```
function polarization_get_axis (pol) result (alpha)
  class(polarization_t), intent(in), target :: pol
  real(default), dimension(3) :: alpha
```

```
    select case (pol%chirality)
    case (0)
       call pol%bv%to_vector (alpha)
       if (.not. pol%anti)  alpha(2) = - alpha(2)
    case (-1)
       alpha = [0._default, 0._default, -1._default]
    case (1)
       alpha = [0._default, 0._default, 1._default]
    end select
  end function polarization_get_axis
```

This function returns polarization degree and polar and azimuthal angles (θ, ϕ) of the polarization axis. The same restrictions apply as above.

Since we call the `get_axis` method, the phase flips sign for an antiparticle.

⟨*Polarizations: polarization: TBP*⟩+≡

```
  procedure :: to_angles => polarization_to_angles
```

⟨*Polarizations: procedures*⟩+≡

```
  subroutine polarization_to_angles (pol, r, theta, phi)
    class(polarization_t), intent(in) :: pol
    real(default), intent(out) :: r, theta, phi
    real(default), dimension(3) :: alpha
    real(default) :: norm, r12
    alpha = pol%get_axis ()
    norm = sum (alpha**2)
    r = sqrt (norm)
    if (norm > 0) then
       r12 = sqrt (alpha(1)**2 + alpha(2)**2)
       theta = atan2 (r12, alpha(3))
       if (any (alpha(1:2) /= 0)) then
          phi = atan2 (alpha(2), alpha(1))
       else
          phi = 0
       end if
    else
       theta = 0
       phi = 0
    end if
  end subroutine polarization_to_angles
```

14.3.8 Polarization Iterator

The iterator acts like a state matrix iterator, i.e., it points to one helicity combination at a time and can return the corresponding helicity object and matrix-element value.

Since the polarization is stored as a Bloch vector, we recover the whole density matrix explicitly upon initialization, store it inside the iterator object, and then just return its elements one at a time.

For an unpolarized particle, the iterator returns a single state with undefined helicity. The value is the value of any diagonal density matrix element, $1/n$ where n is the multiplicity.

⟨*Polarizations: public*⟩+≡

```
public :: polarization_iterator_t
```

⟨*Polarizations: types*⟩+≡

```
type :: polarization_iterator_t
   private
   type(polarization_t), pointer :: pol => null ()
   logical :: polarized = .false.
   integer :: h1 = 0
   integer :: h2 = 0
   integer :: i = 0
   integer :: j = 0
   complex(default), dimension(:,:), allocatable :: r
   complex(default) :: value = 1._default
   real(default) :: tolerance = -1._default
   logical :: valid = .false.
 contains
 ⟨Polarizations: polarization iterator: TBP⟩
end type polarization_iterator_t
```

Output for debugging purposes only, therefore no format for real/complex.

⟨*Polarizations: polarization iterator: TBP*⟩≡

```
procedure :: write => polarization_iterator_write
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine polarization_iterator_write (it, unit)
  class(polarization_iterator_t), intent(in) :: it
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1X,A)")  "Polarization iterator:"
  write (u, "(3X,A,L1)")  "assigned = ", associated (it%pol)
  write (u, "(3X,A,L1)")  "valid    = ", it%valid
  if (it%valid) then
     write (u, "(3X,A,2(1X,I2))")  "i, j     = ", it%i, it%j
     write (u, "(3X,A,2(1X,I2))")  "h1, h2   = ", it%h1, it%h2
     write (u, "(3X,A)", advance="no")  "value    = "
     write (u, *)  it%value
     if (allocated (it%r)) then
        do i = 1, size (it%r, 2)
           write (u, *)  it%r(i,:)
        end do
     end if
  end if
end subroutine polarization_iterator_write
```

Initialize, i.e., (virtually) point to the first helicity state supported by the polarization object. If the density matrix is nontrivial, we calculate it here.

Following the older state-matrix conventions, the iterator sequence starts at the lowest helicity value. In the current internal representation, this corresponds to the highest index value.

If the current matrix-element value is zero, advance the iterator. Advancing will stop at a nonzero value or if the iterator becomes invalid.

If `tolerance` is given, any state matrix entry less or equal will be treated as zero, causing the iterator to skip an entry. By default, the value is negative, so no entry is skipped.

⟨*Polarizations: polarization iterator: TBP*⟩+≡

```
procedure :: init => polarization_iterator_init
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine polarization_iterator_init (it, pol, all_states, tolerance)
  class(polarization_iterator_t), intent(out) :: it
  type(polarization_t), intent(in), target :: pol
  logical, intent(in), optional :: all_states
  real(default), intent(in), optional :: tolerance
  integer :: d
  logical :: only_max_weight
  it%pol => pol
  if (present (all_states)) then
     if (.not. all_states) then
        if (present (tolerance)) then
           it%tolerance = tolerance
        else
           it%tolerance = 0
        end if
     end if
  end if
  select case (pol%chirality)
  case (0)
     d = pol%bv%get_n_states ()
     only_max_weight = pol%multiplicity < d
     it%polarized = pol%bv%is_polarized ()
     if (it%polarized) then
        it%i = d
        it%j = it%i
        it%h1 = pol%bv%hel_value (it%i)
        it%h2 = it%h1
        call pol%bv%to_matrix (it%r, only_max_weight)
        it%value = it%r(it%i, it%j)
     else
        it%value = 1._default / d
     end if
     it%valid = .true.
  case (1,-1)
     it%polarized = .true.
     select case (pol%spin_type)
     case (SPINOR)
        it%h1 = pol%chirality
     case (VECTORSPINOR)
        it%h1 = 2 * pol%chirality
     end select
     it%h2 = it%h1
     it%valid = .true.
  end select
  if (it%valid .and. abs (it%value) <= it%tolerance)  call it%advance ()
end subroutine polarization_iterator_init
```

Advance to the next valid helicity state. Repeat if the returned value is zero.

For an unpolarized object, we iterate through the diagonal helicity states with a constant value.

⟨*Polarizations: polarization iterator: TBP*⟩+≡

```
procedure :: advance => polarization_iterator_advance
```

⟨*Polarizations: procedures*⟩+≡

```
recursive subroutine polarization_iterator_advance (it)
  class(polarization_iterator_t), intent(inout) :: it
  if (it%valid) then
     select case (it%pol%chirality)
     case (0)
        if (it%polarized) then
           if (it%j > 1) then
              it%j = it%j - 1
              it%h2 = it%pol%bv%hel_value (it%j)
              it%value = it%r(it%i, it%j)
           else if (it%i > 1) then
              it%j = it%pol%bv%get_n_states ()
              it%h2 = it%pol%bv%hel_value (it%j)
              it%i = it%i - 1
              it%h1 = it%pol%bv%hel_value (it%i)
              it%value = it%r(it%i, it%j)
           else
              it%valid = .false.
           end if
        else
           it%valid = .false.
        end if
     case default
        it%valid = .false.
     end select
     if (it%valid .and. abs (it%value) <= it%tolerance)  call it%advance ()
  end if
end subroutine polarization_iterator_advance
```

This is true as long as the iterator points to a valid helicity state.

⟨*Polarizations: polarization iterator: TBP*⟩+≡

```
procedure :: is_valid => polarization_iterator_is_valid
```

⟨*Polarizations: procedures*⟩+≡

```
function polarization_iterator_is_valid (it) result (is_valid)
  logical :: is_valid
  class(polarization_iterator_t), intent(in) :: it
  is_valid = it%valid
end function polarization_iterator_is_valid
```

Return the matrix element value for the helicity that we are currently pointing at.

⟨*Polarizations: polarization iterator: TBP*⟩+≡

```
procedure :: get_value => polarization_iterator_get_value
```

⟨*Polarizations: procedures*⟩+≡

```
function polarization_iterator_get_value (it) result (value)
  complex(default) :: value
  class(polarization_iterator_t), intent(in) :: it
  if (it%valid) then
     value = it%value
  else
     value = 0
  end if
end function polarization_iterator_get_value
```

Return a quantum number object for the helicity that we are currently pointing at. This is a single quantum number object, not an array.

Note that the `init` method of the helicity object has the order reversed.

⟨*Polarizations: polarization iterator: TBP*⟩+≡

```
procedure :: get_quantum_numbers => polarization_iterator_get_quantum_numbers
```

⟨*Polarizations: procedures*⟩+≡

```
function polarization_iterator_get_quantum_numbers (it) result (qn)
  class(polarization_iterator_t), intent(in) :: it
  type(helicity_t) :: hel
  type(quantum_numbers_t) :: qn
  if (it%polarized) then
     call hel%init (it%h2, it%h1)
  end if
  call qn%init (hel)
end function polarization_iterator_get_quantum_numbers
```

14.3.9 Sparse Matrix

We introduce a simple implementation of a sparse matrix that can represent polarization (or similar concepts) for transfer to I/O within the program. It consists of an integer array that represents the index values, and a complex array that represents the nonvanishing entries. The number of nonvanishing entries must be known for initialization, but the entries are filled one at a time.

Here is a base type without the special properties of a spin-density matrix.

⟨*Polarizations: public*⟩+≡

```
public :: smatrix_t
```

⟨*Polarizations: types*⟩+≡

```
type :: smatrix_t
   private
   integer :: dim = 0
   integer :: n_entry = 0
   integer, dimension(:,:), allocatable :: index
   complex(default), dimension(:), allocatable :: value
 contains
 ⟨Polarizations: smatrix: TBP⟩
end type smatrix_t
```

Output.

⟨*Polarizations: smatrix: TBP*⟩≡

```
procedure :: write => smatrix_write
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine smatrix_write (object, unit, indent)
  class(smatrix_t), intent(in) :: object
  integer, intent(in), optional :: unit, indent
  integer :: u, i, ind
  u = given_output_unit (unit)
  ind = 0;  if (present (indent))  ind = indent
  if (allocated (object%value)) then
     if (size (object%value) > 0) then
        do i = 1, object%n_entry
           write (u, "(1x,A,'@(')", advance="no")  repeat ("  ", ind)
           write (u, "(SP,9999(I2.1,':',1x))", advance="no") &
                object%index(:,i)
           write (u, "('(',," // FMT_19 // ",',',," // FMT_19 // &
                ",'))')")  object%value(i)
        end do
     else
        write (u, "(1x,A)", advance="no")  repeat ("  ", ind)
        write (u, "(A)")  "[empty matrix]"
     end if
  else
     write (u, "(1x,A)", advance="no")  repeat ("  ", ind)
     write (u, "(A)")  "[undefined matrix]"
  end if
end subroutine smatrix_write
```

Initialization: allocate arrays to the correct size. We specify both the dimension of the matrix (if different from two, this is rather a generic tensor) and the number of nonvanishing entries.

⟨*Polarizations: smatrix: TBP*⟩+≡

```
procedure :: init => smatrix_init
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine smatrix_init (smatrix, dim, n_entry)
  class(smatrix_t), intent(out) :: smatrix
  integer, intent(in) :: dim
  integer, intent(in) :: n_entry
  smatrix%dim = dim
  smatrix%n_entry = n_entry
  allocate (smatrix%index (dim, n_entry))
  allocate (smatrix%value (n_entry))
end subroutine smatrix_init
```

Fill: one entry at a time.

⟨*Polarizations: smatrix: TBP*⟩+≡

```
procedure :: set_entry => smatrix_set_entry
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine smatrix_set_entry (smatrix, i, index, value)
  class(smatrix_t), intent(inout) :: smatrix
```

```
    integer, intent(in) :: i
    integer, dimension(:), intent(in) :: index
    complex(default), intent(in) :: value
    smatrix%index(:,i) = index
    smatrix%value(i) = value
  end subroutine smatrix_set_entry
```

⟨*Polarizations: smatrix: TBP*⟩+≡

```
  procedure :: exists => smatrix_exists
```

⟨*Polarizations: procedures*⟩+≡

```
  elemental function smatrix_exists (smatrix) result (exist)
    logical :: exist
    class(smatrix_t), intent(in) :: smatrix
    exist = .not. all (smatrix%value == 0)
  end function smatrix_exists
```

14.3.10 Polarization Matrix

As an extension of the more generic `smatrix` type, we implement a proper spin-density matrix. After the matrix has been filled, we can fix spin type and multiplicity for a particle, check the matrix for consistency, and normalize it if necessary.

TODO: This implementation does not have an antiparticle flag, just like the state matrix object. We therefore cannot account for sign flips when using this object.

TODO: The `pure` flag is for informational purposes only, and it only represents a necessary condition if spin is greater than 1/2. We may either check purity for all spins or drop this.

⟨*Polarizations: public*⟩+≡

```
  public :: pmatrix_t
```

⟨*Polarizations: types*⟩+≡

```
  type, extends (smatrix_t) :: pmatrix_t
     private
     integer :: spin_type = 0
     integer :: multiplicity = 0
     logical :: massive = .true.
     integer :: chirality = 0
     real(default) :: degree = 1
     logical :: pure = .false.
   contains
   ⟨Polarizations: pmatrix: TBP⟩
  end type pmatrix_t
```

Output, including extra data. (The `indent` argument is ignored.)

⟨*Polarizations: pmatrix: TBP*⟩≡

```
  procedure :: write => pmatrix_write
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine pmatrix_write (object, unit, indent)
  class(pmatrix_t), intent(in) :: object
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Polarization: spin density matrix"
  write (u, "(3x,A,I0)")  "spin type     = ", object%spin_type
  write (u, "(3x,A,I0)")  "multiplicity  = ", object%multiplicity
  write (u, "(3x,A,L1)")  "massive       = ", object%massive
  write (u, "(3x,A,I0)")  "chirality     = ", object%chirality
  write (u, "(3x,A,F10.7)")  "pol.degree    =", object%degree
  write (u, "(3x,A,L1)")  "pure state    = ", object%pure
  call object%smatrix_t%write (u, 1)
end subroutine pmatrix_write
```

This assignment is trivial, but must be coded explicitly.

⟨*Polarizations: pmatrix: TBP*⟩+≡

```
generic :: assignment(=) => pmatrix_assign_from_smatrix
procedure, private :: pmatrix_assign_from_smatrix
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine pmatrix_assign_from_smatrix (pmatrix, smatrix)
  class(pmatrix_t), intent(out) :: pmatrix
  type(smatrix_t), intent(in) :: smatrix
  pmatrix%smatrix_t = smatrix
end subroutine pmatrix_assign_from_smatrix
```

Declare spin, multiplicity, and polarization degree. Check whether all entries fit, and whether this is a valid matrix.

The required properties are:

1. all entries apply to the given spin and mass type
2. the diagonal is real
3. only the upper of corresponding off-diagonal elements is specified, i.e., the row index is less than the column index
4. the trace is nonnegative and equal to the polarization degree (the remainder, proportional to the unit matrix, is understood to be present)
5. the trace of the matrix square is positive and less or equal to the trace of the matrix itself, which is the polarization degree.
6. If the trace of the matrix square and the trace of the matrix are unity, we may have a pure state. (For spin up to 1/2, this is actually sufficient.)

⟨*Polarizations: pmatrix: TBP*⟩+≡

```
procedure :: normalize => pmatrix_normalize
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine pmatrix_normalize (pmatrix, flv, degree, tolerance)
  class(pmatrix_t), intent(inout) :: pmatrix
  type(flavor_t), intent(in) :: flv
  real(default), intent(in), optional :: degree
  real(default), intent(in), optional :: tolerance
  integer :: i, hmax
  logical :: fermion, ok
  real(default) :: trace, trace_sq
  real(default) :: tol
  tol = 0;  if (present (tolerance))  tol = tolerance
  pmatrix%spin_type = flv%get_spin_type ()
  pmatrix%massive = flv%get_mass () /= 0
  if (.not. pmatrix%massive) then
     if (flv%is_left_handed ()) then
        pmatrix%chirality = -1
     else if (flv%is_right_handed ()) then
        pmatrix%chirality = +1
     end if
  end if
  if (pmatrix%spin_type == SCALAR) then
     pmatrix%multiplicity = 1
  else if (pmatrix%massive) then
     pmatrix%multiplicity = pmatrix%spin_type
  else if (pmatrix%chirality == 0) then
     pmatrix%multiplicity = 2
  else
     pmatrix%multiplicity = 1
  end if
  if (present (degree)) then
     if (degree < 0 .or. degree > 1) &
          call msg_error ("polarization degree must be between 0 and 1")
     pmatrix%degree = degree
  end if
  if (size (pmatrix%index, 1) /= 2)  call error ("wrong array rank")
  fermion = mod (pmatrix%spin_type, 2) == 0
  hmax = pmatrix%spin_type / 2
  if (pmatrix%n_entry > 0) then
     if (fermion) then
        if (pmatrix%massive) then
           ok = all (pmatrix%index /= 0) &
                .and. all (abs (pmatrix%index) <= hmax)
        else if (pmatrix%chirality == -1) then
           ok = all (pmatrix%index == -hmax)
        else if (pmatrix%chirality == +1) then
           ok = all (pmatrix%index == +hmax)
        else
           ok = all (abs (pmatrix%index) == hmax)
        end if
     else
        if (pmatrix%massive) then
           ok = all (abs (pmatrix%index) <= hmax)
        else
           ok = all (abs (pmatrix%index) == hmax)
```

```
        end if
      end if
      if (.not. ok)  call error ("illegal index value")
   else
      pmatrix%degree = 0
      pmatrix%pure = pmatrix%multiplicity == 1
      return
   end if
   trace = 0
   do i = 1, pmatrix%n_entry
      associate (index => pmatrix%index(:,i), value => pmatrix%value(i))
        if (index(1) == index(2)) then
           if (abs (aimag (value)) > tol)  call error ("diagonal must be real")
           value = real (value, kind=default)
           trace = trace + value

        else if (any (pmatrix%index(1,:) == index(2) &
             .and.    pmatrix%index(2,:) == index(1))) then
           call error ("redundant off-diagonal entry")
        else if (index(2) < index (1)) then
           index = index([2,1])
           value = conjg (value)
        end if
      end associate
   end do
   if (abs (trace) <= tol)  call error ("trace must not vanish")
   trace = real (trace, kind=default)
   pmatrix%value = pmatrix%value / trace * pmatrix%degree
   trace_sq = (1 - pmatrix%degree ** 2) / pmatrix%multiplicity
   do i = 1, pmatrix%n_entry
      associate (index => pmatrix%index(:,i), value => pmatrix%value(i))
        if (index(1) == index(2)) then
           trace_sq = trace_sq + abs (value) ** 2
        else
           trace_sq = trace_sq + 2 * abs (value) ** 2
        end if
      end associate
   end do
   if (pmatrix%multiplicity == 1) then
      pmatrix%pure = .true.
   else if (abs (trace_sq - 1) <= tol) then
      pmatrix%pure = .true.
   else if (trace_sq - 1 > tol .or. trace_sq < -tol) then
      print *, "Trace of matrix square = ", trace_sq
      call error ("not permissible as density matrix")
   end if
contains
   subroutine error (msg)
     character(*), intent(in) :: msg
     call pmatrix%write ()
     call msg_fatal ("Spin density matrix: " // msg)
   end subroutine error
end subroutine pmatrix_normalize
```

A polarized matrix is defined as one with a positive polarization degree, even if the actual matrix is trivial.

⟨*Polarizations: pmatrix: TBP*⟩+≡
```
procedure :: is_polarized => pmatrix_is_polarized
```

⟨*Polarizations: procedures*⟩+≡
```
elemental function pmatrix_is_polarized (pmatrix) result (flag)
  class(pmatrix_t), intent(in) :: pmatrix
  logical :: flag
  flag = pmatrix%degree > 0
end function pmatrix_is_polarized
```

Check if there are only diagonal entries.

⟨*Polarizations: pmatrix: TBP*⟩+≡
```
procedure :: is_diagonal => pmatrix_is_diagonal
```

⟨*Polarizations: procedures*⟩+≡
```
elemental function pmatrix_is_diagonal (pmatrix) result (flag)
  class(pmatrix_t), intent(in) :: pmatrix
  logical :: flag
  flag = all (pmatrix%index(1,:) == pmatrix%index(2,:))
end function pmatrix_is_diagonal
```

Check if there are only diagonal entries.

⟨*Polarizations: pmatrix: TBP*⟩+≡
```
procedure :: get_simple_pol => pmatrix_get_simple_pol
```

⟨*Polarizations: procedures*⟩+≡
```
elemental function pmatrix_get_simple_pol (pmatrix) result (pol)
  class(pmatrix_t), intent(in) :: pmatrix
  real(default) :: pol
  if (pmatrix%is_polarized ()) then
     select case (size (pmatrix%value))
     case (0)
        pol = 0
     case (1)
        pol = pmatrix%index (1,1) * pmatrix%degree
     case (2)
        pol = 42
     end select
  else
     pol = 0
  end if
end function pmatrix_get_simple_pol
```

14.3.11 Data Transformation

Create a `polarization_t` object from the contents of a normalized `pmatrix_t` object. We scan the entries as present in `pmatrix` and transform them into a density matrix, if necessary. The density matrix then initializes the Bloch vector. This is analogous to `polarization_init_state_matrix`.

There is a subtlety associated with massless particles. Since the `pmatrix` doesn't contain the full density matrix but just the nontrivial part, we have to initialize the polarization object with the massless equipartion, which contains nonzero entries for the Cartan generators. The `set` method therefore should not erase those initial contents. This is a constraint for the implementation of `set`, as applied to the Bloch vector.

TODO: As mentioned above, `pmatrix_t` does not support an antiparticle flag.

⟨*Polarizations: polarization: TBP*⟩+≡

```
procedure :: init_pmatrix => polarization_init_pmatrix
```

⟨*Polarizations: procedures*⟩+≡

```
subroutine polarization_init_pmatrix (pol, pmatrix)
  class(polarization_t), intent(out) :: pol
  type(pmatrix_t), intent(in) :: pmatrix
  integer :: d, i, j, k, h1, h2
  complex(default), dimension(:,:), allocatable :: r
  call pol%init_generic ( &
       spin_type = pmatrix%spin_type, &
       multiplicity = pmatrix%multiplicity, &
       anti = .false., &                    !!! SUFFICIENT?
       left_handed = pmatrix%chirality < 0, &
       right_handed = pmatrix%chirality > 0)
  if (pol%bv%is_polarized ()) then
     d = pol%bv%get_n_states ()
     allocate (r (d, d), source = (0._default, 0._default))
     if (d == pmatrix%multiplicity) then
        do i = 1, d
           r(i,i) = (1 - pmatrix%degree) / d
        end do
     else if (d > pmatrix%multiplicity) then
        r(1,1) = (1 - pmatrix%degree) / 2
        r(d,d) = r(1,1)
     end if
     do k = 1, size (pmatrix%value)
        h1 = pmatrix%index(1,k)
        h2 = pmatrix%index(2,k)
        i = pol%bv%hel_index (h1)
        j = pol%bv%hel_index (h2)
        r(i,j) = r(i,j) + pmatrix%value(k)
        r(j,i) = conjg (r(i,j))
     end do
     call pol%bv%set (r)
  end if
end subroutine polarization_init_pmatrix
```

14.3.12 Unit tests

Test module, followed by the corresponding implementation module.

⟨`polarizations_ut.f90`⟩≡

⟨*File header*⟩

```
module polarizations_ut
  use unit_tests
  use polarizations_uti

⟨Standard module head⟩

⟨Polarizations: public test⟩

contains

⟨Polarizations: test driver⟩

end module polarizations_ut
```

⟨polarizations_uti.f90⟩≡

```
⟨File header⟩

module polarizations_uti

⟨Use kinds⟩
  use flavors
  use model_data

  use polarizations

⟨Standard module head⟩

⟨Polarizations: test declarations⟩

contains

⟨Polarizations: tests⟩

end module polarizations_uti
```

API: driver for the unit tests below.

⟨*Polarizations: public test*⟩≡

```
  public :: polarizations_test
```

⟨*Polarizations: test driver*⟩≡

```
  subroutine polarizations_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨Polarizations: execute tests⟩
  end subroutine polarizations_test
```

Polarization type

Checking the setup for polarization.

⟨*Polarizations: execute tests*⟩≡

```
  call test (polarization_1, "polarization_1", &
       "check polarization setup", &
       u, results)
```

⟨*Polarizations: test declarations*⟩≡

```
public :: polarization_1
```

⟨*Polarizations: tests*⟩≡

```
subroutine polarization_1 (u)
  use os_interface
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(polarization_t) :: pol
  type(flavor_t) :: flv
  real(default), dimension(3) :: alpha
  real(default) :: r, theta, phi
  real(default), parameter :: tolerance = 1.E-14_default

  write (u, "(A)")  "* Test output: polarization_1"
  write (u, "(A)")  "*   Purpose: test polarization setup"
  write (u, "(A)")

  write (u, "(A)")  "* Reading model file"
  write (u, "(A)")

  call model%init_sm_test ()

  write (u, "(A)") "* Unpolarized fermion"
  write (u, "(A)")

  call flv%init (1, model)
  call pol%init_unpolarized (flv)
  call pol%write (u, state_matrix = .true.)
  write (u, "(A,L1)")  "   diagonal =", pol%is_diagonal ()

  write (u, "(A)")
  write (u, "(A)")  "* Unpolarized fermion"
  write (u, "(A)")

  call pol%init_circular (flv, 0._default)
  call pol%write (u, state_matrix = .true., all_states = .false.)

  write (u, "(A)")
  write (u, "(A)")  "* Transversally polarized fermion, phi=0"
  write (u, "(A)")

  call pol%init_transversal (flv, 0._default, 1._default)
  call pol%write (u, state_matrix = .true.)
  write (u, "(A,L1)")  "   diagonal =", pol%is_diagonal ()

  write (u, "(A)")
  write (u, "(A)")  "* Transversally polarized fermion, phi=0.9, frac=0.8"
  write (u, "(A)")

  call pol%init_transversal (flv, 0.9_default, 0.8_default)
  call pol%write (u, state_matrix = .true.)
  write (u, "(A,L1)")  "   diagonal =", pol%is_diagonal ()

  write (u, "(A)")
```

```
write (u, "(A)") "* All polarization directions of a fermion"
write (u, "(A)")

call pol%init_generic (flv)
call pol%write (u, state_matrix = .true.)


call flv%init (21, model)

write (u, "(A)")
write (u, "(A)")  "* Circularly polarized gluon, frac=0.3"
write (u, "(A)")

call pol%init_circular (flv, 0.3_default)
call pol%write (u, state_matrix = .true., &
     all_states = .false., tolerance = tolerance)


call flv%init (23, model)

write (u, "(A)")
write (u, "(A)") "* Circularly polarized massive vector, frac=-0.7"
write (u, "(A)")

call pol%init_circular (flv,  -0.7_default)
call pol%write (u, state_matrix = .true., &
     all_states = .false., tolerance = tolerance)

write (u, "(A)")
write (u, "(A)")  "* Circularly polarized massive vector"
write (u, "(A)")

call pol%init_circular (flv, 1._default)
call pol%write (u, state_matrix = .true., &
     all_states = .false., tolerance = tolerance)

write (u, "(A)")
write (u, "(A)")  "* Longitudinally polarized massive vector, frac=0.4"
write (u, "(A)")

call pol%init_longitudinal (flv, 0.4_default)
call pol%write (u, state_matrix = .true., &
     all_states = .false., tolerance = tolerance)

write (u, "(A)")
write (u, "(A)")  "* Longitudinally polarized massive vector"
write (u, "(A)")

call pol%init_longitudinal (flv, 1._default)
call pol%write (u, state_matrix = .true., &
     all_states = .false., tolerance = tolerance)

write (u, "(A)")
write (u, "(A)")  "* Diagonally polarized massive vector"
```

```
  write (u, "(A)")

  call pol%init_diagonal &
       (flv, [2._default, 1._default, 0._default])
  call pol%write (u, state_matrix = .true., &
       all_states = .false., tolerance = tolerance)

  write (u, "(A)")
  write (u, "(A)")  "* All polarization directions of a massive vector"
  write (u, "(A)")

  call pol%init_generic (flv)
  call pol%write (u, state_matrix = .true.)
  call flv%init (21, model)

  write (u, "(A)")
  write (u, "(A)")  "* Axis polarization (0.2, 0.4, 0.6)"
  write (u, "(A)")

  alpha = [0.2_default, 0.4_default, 0.6_default]
  call pol%init_axis (flv, alpha)
  call pol%write (u, state_matrix = .true., &
       all_states = .false., tolerance = tolerance)

  write (u, "(A)")
  write (u, "(1X,A)")  "Recovered axis:"
  alpha = pol%get_axis ()
  write (u, "(3(1X,F10.7))")  alpha

  write (u, "(A)")
  write (u, "(A)")  "* Angle polarization (0.5, 0.6, -1)"
  r = 0.5_default
  theta = 0.6_default
  phi = -1._default
  call pol%init_angles (flv, r, theta, phi)
  write (u, "(A)")
  call pol%write (u, state_matrix = .true., &
       all_states = .false., tolerance = tolerance)

  write (u, "(A)")
  write (u, "(1X,A)")  "Recovered parameters (r, theta, phi):"
  call pol%to_angles (r, theta, phi)
  write (u, "(3(1x,F10.7))")  r, theta, phi

  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: polarization_1"

end subroutine polarization_1
```

Sparse-Matrix type

Use a sparse density matrix universally as the input for setting up polarization.

⟨*Polarizations: execute tests*⟩+≡

```
call test (polarization_2, "polarization_2", &
     "matrix polarization setup", &
     u, results)
```

⟨*Polarizations: test declarations*⟩+≡

```
public :: polarization_2
```

⟨*Polarizations: tests*⟩+≡

```
subroutine polarization_2 (u)
  use os_interface
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t) :: flv
  type(polarization_t) :: pol
  real(default), dimension(3) :: alpha
  type(pmatrix_t) :: pmatrix
  real(default), parameter :: tolerance = 1e-8_default

  write (u, "(A)")  "* Test output: polarization_2"
  write (u, "(A)")  "*   Purpose: matrix polarization setup"
  write (u, "(A)")

  write (u, "(A)")  "* Reading model file"
  write (u, "(A)")

  call model%init_sm_test ()

  write (u, "(A)") "* Unpolarized fermion"
  write (u, "(A)")

  call flv%init (1, model)
  call pmatrix%init (2, 0)
  call pmatrix%normalize (flv, 0._default, tolerance)
  call pmatrix%write (u)
  write (u, *)
  write (u, "(1x,A,L1)")  "polarized = ", pmatrix%is_polarized ()
  write (u, "(1x,A,L1)")  "diagonal = ", pmatrix%is_diagonal ()
  write (u, *)

  call pol%init_pmatrix (pmatrix)
  call pol%write (u, state_matrix = .true., &
       all_states = .false., tolerance = tolerance)
  ! call pol%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Transversally polarized fermion, phi=0"
  write (u, "(A)")

  call pmatrix%init (2, 3)
  call pmatrix%set_entry (1, [-1,-1], (1._default, 0._default))
  call pmatrix%set_entry (2, [+1,+1], (1._default, 0._default))
```

```
call pmatrix%set_entry (3, [-1,+1], (1._default, 0._default))
call pmatrix%normalize (flv, 1._default, tolerance)
call pmatrix%write (u)
write (u, *)
write (u, "(1x,A,L1)")  "polarized = ", pmatrix%is_polarized ()
write (u, "(1x,A,L1)")  "diagonal = ", pmatrix%is_diagonal ()
write (u, *)

call pol%init_pmatrix (pmatrix)
call pol%write (u, state_matrix = .true., &
     all_states = .false., tolerance = tolerance)
! call pol%final ()

write (u, "(A)")
write (u, "(A)")  "* Transversally polarized fermion, phi=0.9, frac=0.8"
write (u, "(A)")

call pmatrix%init (2, 3)
call pmatrix%set_entry (1, [-1,-1], (1._default, 0._default))
call pmatrix%set_entry (2, [+1,+1], (1._default, 0._default))
call pmatrix%set_entry (3, [-1,+1], exp ((0._default, -0.9_default)))
call pmatrix%normalize (flv, 0.8_default, tolerance)
call pmatrix%write (u)
write (u, *)

call pol%init_pmatrix (pmatrix)
call pol%write (u, state_matrix = .true.)
! call pol%final ()

write (u, "(A)")
write (u, "(A)")  "* Left-handed massive fermion, frac=1"
write (u, "(A)")

call flv%init (11, model)
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [-1,-1], (1._default, 0._default))
call pmatrix%normalize (flv, 1._default, tolerance)
call pmatrix%write (u)
write (u, *)
write (u, "(1x,A,L1)")  "polarized = ", pmatrix%is_polarized ()
write (u, "(1x,A,L1)")  "diagonal = ", pmatrix%is_diagonal ()
write (u, *)

call pol%init_pmatrix (pmatrix)
call pol%write (u, state_matrix = .true., &
     all_states = .false., tolerance = tolerance)
! call pol%final ()

write (u, "(A)")
write (u, "(A)")  "* Left-handed massive fermion, frac=0.8"
write (u, "(A)")

call flv%init (11, model)
call pmatrix%init (2, 1)
```

```
call pmatrix%set_entry (1, [-1,-1], (1._default, 0._default))
call pmatrix%normalize (flv, 0.8_default, tolerance)
call pmatrix%write (u)
write (u, *)

call pol%init_pmatrix (pmatrix)
call pol%write (u, state_matrix = .true., &
     all_states = .false., tolerance = tolerance)
! call pol%final ()

write (u, "(A)")
write (u, "(A)")  "* Left-handed massless fermion"
write (u, "(A)")

call flv%init (12, model)
call pmatrix%init (2, 0)
call pmatrix%normalize (flv, 1._default, tolerance)
call pmatrix%write (u)
write (u, *)

call pol%init_pmatrix (pmatrix)
call pol%write (u, state_matrix = .true.)
! call pol%final ()

write (u, "(A)")
write (u, "(A)")  "* Right-handed massless fermion, frac=0.5"
write (u, "(A)")

call flv%init (-12, model)
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [1,1], (1._default, 0._default))
call pmatrix%normalize (flv, 0.5_default, tolerance)
call pmatrix%write (u)
write (u, *)

call pol%init_pmatrix (pmatrix)
call pol%write (u, state_matrix = .true.)
! call pol%final ()

write (u, "(A)")
write (u, "(A)")  "* Circularly polarized gluon, frac=0.3"
write (u, "(A)")

call flv%init (21, model)
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [1,1], (1._default, 0._default))
call pmatrix%normalize (flv, 0.3_default, tolerance)
call pmatrix%write (u)
write (u, *)

call pol%init_pmatrix (pmatrix)
call pol%write (u, state_matrix = .true., &
     all_states = .false., tolerance = tolerance)
! call pol%final ()
```

```
write (u, "(A)")
write (u, "(A)") "* Circularly polarized massive vector, frac=0.7"
write (u, "(A)")

call flv%init (23, model)
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [1,1], (1._default, 0._default))
call pmatrix%normalize (flv, 0.7_default, tolerance)
call pmatrix%write (u)
write (u, *)

call pol%init_pmatrix (pmatrix)
call pol%write (u, state_matrix = .true., &
     all_states = .false., tolerance = tolerance)
! call pol%final ()

write (u, "(A)")
write (u, "(A)")  "* Circularly polarized massive vector"
write (u, "(A)")

call flv%init (23, model)
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [1,1], (1._default, 0._default))
call pmatrix%normalize (flv, 1._default, tolerance)
call pmatrix%write (u)
write (u, *)

call pol%init_pmatrix (pmatrix)
call pol%write (u, state_matrix = .true., &
     all_states = .false., tolerance = tolerance)
! call pol%final ()

write (u, "(A)")
write (u, "(A)")  "* Longitudinally polarized massive vector, frac=0.4"
write (u, "(A)")

call flv%init (23, model)
call pmatrix%init (2, 1)
call pmatrix%set_entry (1, [0,0], (1._default, 0._default))
call pmatrix%normalize (flv, 0.4_default, tolerance)
call pmatrix%write (u)
write (u, *)
write (u, "(1x,A,L1)")  "polarized = ", pmatrix%is_polarized ()
write (u, "(1x,A,L1)")  "diagonal = ", pmatrix%is_diagonal ()
write (u, *)

call pol%init_pmatrix (pmatrix)
call pol%write (u, state_matrix = .true., &
     all_states = .false., tolerance = tolerance)
! call pol%final ()

write (u, "(A)")
write (u, "(A)")  "* Longitudinally polarized massive vector"
```

```
  write (u, "(A)")

  call flv%init (23, model)
  call pmatrix%init (2, 1)
  call pmatrix%set_entry (1, [0,0], (1._default, 0._default))
  call pmatrix%normalize (flv, 1._default, tolerance)
  call pmatrix%write (u)
  write (u, *)
  write (u, "(1x,A,L1)")  "polarized = ", pmatrix%is_polarized ()
  write (u, "(1x,A,L1)")  "diagonal = ", pmatrix%is_diagonal ()
  write (u, *)

  call pol%init_pmatrix (pmatrix)
  call pol%write (u, state_matrix = .true., &
       all_states = .false., tolerance = tolerance)
  ! call pol%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Axis polarization (0.2, 0.4, 0.6)"
  write (u, "(A)")

  call flv%init (11, model)
  alpha = [0.2_default, 0.4_default, 0.6_default]
  alpha = alpha / sqrt (sum (alpha**2))
  call pmatrix%init (2, 3)
  call pmatrix%set_entry (1, [-1,-1], cmplx (1 - alpha(3), kind=default))
  call pmatrix%set_entry (2, [1,-1], &
       cmplx (alpha(1),-alpha(2), kind=default))
  call pmatrix%set_entry (3, [1,1], cmplx (1 + alpha(3), kind=default))
  call pmatrix%normalize (flv, 1._default, tolerance)
  call pmatrix%write (u)
  write (u, *)

  call pol%init_pmatrix (pmatrix)
  call pol%write (u, state_matrix = .true.)
  ! call pol%final ()

  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: polarization_2"

end subroutine polarization_2
```

14.4 Particles

This module defines the `particle_t` object type, and the methods and operations that deal with it.

⟨`particles.f90`⟩≡

```
⟨File header⟩

module particles

⟨Use kinds with double⟩
⟨Use strings⟩
  use io_units
  use format_utils, only: write_compressed_integer_array, write_separator
  use format_utils, only: pac_fmt
  use format_defs, only: FMT_16, FMT_19
  use numeric_utils
  use diagnostics
  use lorentz
  use model_data
  use flavors
  use colors
  use helicities
  use quantum_numbers
  use state_matrices
  use interactions
  use subevents
  use polarizations
  use pdg_arrays, only: is_quark, is_gluon

⟨Standard module head⟩

⟨Particles: public⟩

⟨Particles: parameters⟩

⟨Particles: types⟩

⟨Particles: interfaces⟩

contains

⟨Particles: procedures⟩

end module particles
```

14.4.1 The particle type

Particle status codes

The overall status codes (incoming/outgoing etc.) are inherited from the module `subevents`.

Polarization status:

⟨*Particles: parameters*⟩≡

```
integer, parameter, public :: PRT_UNPOLARIZED = 0
integer, parameter, public :: PRT_DEFINITE_HELICITY = 1
integer, parameter, public :: PRT_GENERIC_POLARIZATION = 2
```

Definition

The quantum numbers are flavor (from which invariant particle properties can be derived), color, and polarization. The particle may be unpolarized. In this case, `hel` and `pol` are unspecified. If it has a definite helicity, the `hel` component is defined. If it has a generic polarization, the `pol` component is defined. For each particle we store the four-momentum and the invariant mass squared, i.e., the squared norm of the four-momentum. There is also an optional list of parent and child particles, for bookkeeping in physical events. The `vertex` is an optional component that consists of a Lorentz 4-vector, denoting the position and time of the vertex (displaced vertex/time). `lifetime` is an optional component that accounts for the finite lifetime τ of a decaying particle. In case there is no magnetic field etc., the true decay vertex of a particle in the detector would be $\vec{v}' = \vec{v} + \tau \times \vec{p}/p^0$, where p^0 and $\vec{p}$ are the energy and 3-momentum of the particle.

⟨*Particles: public*⟩≡
```
public :: particle_t
```

⟨*Particles: types*⟩≡
```
type :: particle_t
   !private
   integer :: status = PRT_UNDEFINED
   integer :: polarization = PRT_UNPOLARIZED
   type(flavor_t) :: flv
   type(color_t) :: col
   type(helicity_t) :: hel
   type(polarization_t) :: pol
   type(vector4_t) :: p = vector4_null
   real(default) :: p2 = 0
   type(vector4_t), allocatable :: vertex
   real(default), allocatable :: lifetime
   integer, dimension(:), allocatable :: parent
   integer, dimension(:), allocatable :: child
 contains
⟨Particles: particle: TBP⟩
end type particle_t
```

Copy a particle. (Deep copy) This excludes the parent-child relations.

⟨*Particles: particle: TBP*⟩≡
```
generic :: init => init_particle
procedure :: init_particle => particle_init_particle
```

⟨*Particles: procedures*⟩≡
```
subroutine particle_init_particle (prt_out, prt_in)
  class(particle_t), intent(out) :: prt_out
  type(particle_t), intent(in) :: prt_in
  prt_out%status = prt_in%status
  prt_out%polarization = prt_in%polarization
```

```
    prt_out%flv = prt_in%flv
    prt_out%col = prt_in%col
    prt_out%hel = prt_in%hel
    prt_out%pol = prt_in%pol
    prt_out%p = prt_in%p
    prt_out%p2 = prt_in%p2
    if (allocated (prt_in%vertex))  &
       allocate (prt_out%vertex, source=prt_in%vertex)
    if (allocated (prt_in%lifetime))  &
         allocate (prt_out%lifetime, source=prt_in%lifetime)
  end subroutine particle_init_particle
```

Initialize a particle using external information.

⟨*Particles: particle: TBP*⟩+≡

```
  generic :: init => init_external
  procedure :: init_external => particle_init_external
```

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_init_external &
         (particle, status, pdg, model, col, anti_col, mom)
    class(particle_t), intent(out) :: particle
    integer, intent(in) :: status, pdg, col, anti_col
    class(model_data_t), pointer, intent(in) :: model
    type(vector4_t), intent(in) :: mom
    type(flavor_t) :: flavor
    type(color_t) :: color
    call flavor%init (pdg, model)
    call particle%set_flavor (flavor)
    call color%init_col_acl (col, anti_col)
    call particle%set_color (color)
    call particle%set_status (status)
    call particle%set_momentum (mom)
  end subroutine particle_init_external
```

Initialize a particle using a single-particle state matrix which determines flavor, color, and polarization. The state matrix must have unique flavor and color. The factorization mode determines whether the particle is unpolarized, has definite helicity, or generic polarization. This mode is translated into the polarization status.

⟨*Particles: particle: TBP*⟩+≡

```
  generic :: init => init_state
  procedure :: init_state => particle_init_state
```

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_init_state (prt, state, status, mode)
    class(particle_t), intent(out) :: prt
    type(state_matrix_t), intent(in), target :: state
    integer, intent(in) :: status, mode
    type(state_iterator_t) :: it
    prt%status = status
    call it%init (state)
    prt%flv = it%get_flavor (1)
    if (prt%flv%is_radiated ())  prt%status = PRT_BEAM_REMNANT
```

```
    prt%col = it%get_color (1)
    select case (mode)
    case (FM_SELECT_HELICITY)
       prt%hel = it%get_helicity (1)
       if (prt%hel%is_defined ()) then
          prt%polarization = PRT_DEFINITE_HELICITY
       end if
    case (FM_FACTOR_HELICITY)
       call prt%pol%init_state_matrix (state)
       prt%polarization = PRT_GENERIC_POLARIZATION
    end select
  end subroutine particle_init_state
```

Finalizer.

⟨*Particles: particle: TBP*⟩+≡

```
  procedure :: final => particle_final
```

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_final (prt)
    class(particle_t), intent(inout) :: prt
    if (allocated (prt%vertex))  deallocate (prt%vertex)
    if (allocated (prt%lifetime))  deallocate (prt%lifetime)
  end subroutine particle_final
```

I/O

⟨*Particles: particle: TBP*⟩+≡

```
  procedure :: write => particle_write
```

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_write (prt, unit, testflag, compressed, polarization)
    class(particle_t), intent(in) :: prt
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: testflag, compressed, polarization
    logical :: comp, pacified, pol
    integer :: u, h1, h2
    real(default) :: pp2
    character(len=7) :: fmt
    character(len=20) :: buffer
    comp = .false.; if (present (compressed))  comp = compressed
    pacified = .false.;  if (present (testflag))  pacified = testflag
    pol = .true.;  if (present (polarization))  pol = polarization
    call pac_fmt (fmt, FMT_19, FMT_16, testflag)
    u = given_output_unit (unit);  if (u < 0)  return
    pp2 = prt%p2
    if (pacified)  call pacify (pp2, tolerance = 1E-10_default)
    select case (prt%status)
    case (PRT_UNDEFINED);    write (u, "(1x, A)", advance="no")  "[-]"
    case (PRT_BEAM);         write (u, "(1x, A)", advance="no")  "[b]"
    case (PRT_INCOMING);     write (u, "(1x, A)", advance="no")  "[i]"
    case (PRT_OUTGOING);     write (u, "(1x, A)", advance="no")  "[o]"
    case (PRT_VIRTUAL);      write (u, "(1x, A)", advance="no")  "[v]"
    case (PRT_RESONANT);     write (u, "(1x, A)", advance="no")  "[r]"
```

```
case (PRT_BEAM_REMNANT); write (u, "(1x, A)", advance="no")  "[x]"
end select
write (u, "(1x)", advance="no")
if (comp) then
   write (u, "(A7,1X)", advance="no") char (prt%flv%get_name ())
   if (pol) then
      select case (prt%polarization)
      case (PRT_DEFINITE_HELICITY)
         ! Integer helicity, assumed diagonal
         call prt%hel%get_indices (h1, h2)
         write (u, "(I2,1X)", advance="no") h1
      case (PRT_GENERIC_POLARIZATION)
         ! No space for full density matrix here
         write (u, "(A2,1X)", advance="no") "*"
      case default
         ! Blank entry if helicity is undefined
         write (u, "(A2,1X)", advance="no") " "
      end select
   end if
   write (u, "(2(I4,1X))", advance="no") &
        prt%col%get_col (), prt%col%get_acl ()
   call write_compressed_integer_array (buffer, prt%parent)
   write (u, "(A,1X)", advance="no") buffer
   call write_compressed_integer_array (buffer, prt%child)
   write (u, "(A,1X)", advance="no") buffer
   call prt%p%write(u, testflag = testflag, compressed = comp)
   write (u, "(F12.3)") pp2
else
   call prt%flv%write (unit)
   if (prt%col%is_nonzero ()) then
      call color_write (prt%col, unit)
   end if
   if (pol) then
      select case (prt%polarization)
      case (PRT_DEFINITE_HELICITY)
         call prt%hel%write (unit)
         write (u, *)
      case (PRT_GENERIC_POLARIZATION)
         write (u, *)
         call prt%pol%write (unit, state_matrix = .true.)
      case default
         write (u, *)
      end select
   else
      write (u, *)
   end if
   call prt%p%write (unit, testflag = testflag)
   write (u, "(1x,A,1x," // fmt // ")")  "T = ", pp2
   if (allocated (prt%parent)) then
      if (size (prt%parent) /= 0) then
         write (u, "(1x,A,40(1x,I0))")  "Parents: ", prt%parent
      end if
   end if
   if (allocated (prt%child)) then
```

```
         if (size (prt%child) /= 0) then
            write (u, "(1x,A,40(1x,I0))")  "Children:", prt%child
         end if
      end if
      if (allocated (prt%vertex)) then
         write (u, "(1x,A,1x," // fmt // ")")  "Vtx t = ", prt%vertex%p(0)
         write (u, "(1x,A,1x," // fmt // ")")  "Vtx x = ", prt%vertex%p(1)
         write (u, "(1x,A,1x," // fmt // ")")  "Vtx y = ", prt%vertex%p(2)
         write (u, "(1x,A,1x," // fmt // ")")  "Vtx z = ", prt%vertex%p(3)
      end if
      if (allocated (prt%lifetime)) then
         write (u, "(1x,A,1x," // fmt // ")")  "Lifetime = ", &
              prt%lifetime
      end if
   end if
end subroutine particle_write
```

Binary I/O:

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: write_raw => particle_write_raw
procedure :: read_raw => particle_read_raw
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_write_raw (prt, u)
  class(particle_t), intent(in) :: prt
  integer, intent(in) :: u
  write (u) prt%status, prt%polarization
  call prt%flv%write_raw (u)
  call prt%col%write_raw (u)
  select case (prt%polarization)
  case (PRT_DEFINITE_HELICITY)
     call prt%hel%write_raw (u)
  case (PRT_GENERIC_POLARIZATION)
     call prt%pol%write_raw (u)
  end select
  call vector4_write_raw (prt%p, u)
  write (u) prt%p2
  write (u) allocated (prt%parent)
  if (allocated (prt%parent)) then
     write (u) size (prt%parent)
     write (u) prt%parent
  end if
  write (u) allocated (prt%child)
  if (allocated (prt%child)) then
     write (u) size (prt%child)
     write (u) prt%child
  end if
  write (u) allocated (prt%vertex)
  if (allocated (prt%vertex)) then
     call vector4_write_raw (prt%vertex, u)
  end if
  write (u) allocated (prt%lifetime)
  if (allocated (prt%lifetime)) then
     write (u) prt%lifetime
```

```
  end if
end subroutine particle_write_raw

subroutine particle_read_raw (prt, u, iostat)
  class(particle_t), intent(out) :: prt
  integer, intent(in) :: u
  integer, intent(out) :: iostat
  logical :: allocated_parent, allocated_child
  logical :: allocated_vertex, allocated_lifetime
  integer :: size_parent, size_child
  read (u, iostat=iostat) prt%status, prt%polarization
  call prt%flv%read_raw (u, iostat=iostat)
  call prt%col%read_raw (u, iostat=iostat)
  select case (prt%polarization)
  case (PRT_DEFINITE_HELICITY)
     call prt%hel%read_raw (u, iostat=iostat)
  case (PRT_GENERIC_POLARIZATION)
     call prt%pol%read_raw (u, iostat=iostat)
  end select
  call vector4_read_raw (prt%p, u, iostat=iostat)
  read (u, iostat=iostat) prt%p2
  read (u, iostat=iostat) allocated_parent
  if (allocated_parent) then
     read (u, iostat=iostat) size_parent
     allocate (prt%parent (size_parent))
     read (u, iostat=iostat) prt%parent
  end if
  read (u, iostat=iostat) allocated_child
  if (allocated_child) then
     read (u, iostat=iostat) size_child
     allocate (prt%child (size_child))
     read (u, iostat=iostat) prt%child
  end if
  read (u, iostat=iostat) allocated_vertex
  if (allocated_vertex) then
     allocate (prt%vertex)
     read (u, iostat=iostat) prt%vertex%p
  end if
  read (u, iostat=iostat) allocated_lifetime
  if (allocated_lifetime) then
     allocate (prt%lifetime)
     read (u, iostat=iostat) prt%lifetime
  end if
end subroutine particle_read_raw
```

Setting contents

Reset the status code. Where applicable, set p^2 assuming that the particle is on-shell.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: reset_status => particle_reset_status
```

⟨*Particles: procedures*⟩+≡

```
elemental subroutine particle_reset_status (prt, status)
  class(particle_t), intent(inout) :: prt
  integer, intent(in) :: status
  prt%status = status
  select case (status)
  case (PRT_BEAM, PRT_INCOMING, PRT_OUTGOING)
     prt%p2 = prt%flv%get_mass () ** 2
  end select
end subroutine particle_reset_status
```

The color can be given explicitly.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: set_color => particle_set_color
```

⟨*Particles: procedures*⟩+≡

```
elemental subroutine particle_set_color (prt, col)
  class(particle_t), intent(inout) :: prt
  type(color_t), intent(in) :: col
  prt%col = col
end subroutine particle_set_color
```

The flavor can be given explicitly.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: set_flavor => particle_set_flavor
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_flavor (prt, flv)
  class(particle_t), intent(inout) :: prt
  type(flavor_t), intent(in) :: flv
  prt%flv = flv
end subroutine particle_set_flavor
```

As can the helicity.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: set_helicity => particle_set_helicity
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_helicity (prt, hel)
  class(particle_t), intent(inout) :: prt
  type(helicity_t), intent(in) :: hel
  prt%hel = hel
end subroutine particle_set_helicity
```

And the polarization.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: set_pol => particle_set_pol
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_pol (prt, pol)
  class(particle_t), intent(inout) :: prt
  type(polarization_t), intent(in) :: pol
  prt%pol = pol
end subroutine particle_set_pol
```

Manually set the model for the particle flavor. This is required, e.g., if the particle has been read from file.

⟨*Particles: particle: TBP*⟩+≡
```
procedure :: set_model => particle_set_model
```

⟨*Particles: procedures*⟩+≡
```
subroutine particle_set_model (prt, model)
  class(particle_t), intent(inout) :: prt
  class(model_data_t), intent(in), target :: model
  call prt%flv%set_model (model)
end subroutine particle_set_model
```

The momentum is set independent of the quantum numbers.

⟨*Particles: particle: TBP*⟩+≡
```
procedure :: set_momentum => particle_set_momentum
```

⟨*Particles: procedures*⟩+≡
```
elemental subroutine particle_set_momentum (prt, p, p2, on_shell)
  class(particle_t), intent(inout) :: prt
  type(vector4_t), intent(in) :: p
  real(default), intent(in), optional :: p2
  logical, intent(in), optional :: on_shell
  prt%p = p
  if (present (on_shell)) then
     if (on_shell) then
        if (prt%flv%is_associated ()) then
           prt%p2 = prt%flv%get_mass () ** 2
           return
        end if
     end if
  end if
  if (present (p2)) then
     prt%p2 = p2
  else
     prt%p2 = p ** 2
  end if
end subroutine particle_set_momentum
```

Set resonance information. This should be done after momentum assignment, because we need to know wheter the particle is spacelike or timelike. The resonance flag is defined only for virtual particles.

⟨*Particle: particle: TBP*⟩≡
```
procedure :: set_resonance_flag => particle_set_resonance_flag
```

⟨*Particles: procedures*⟩+≡
```
elemental subroutine particle_set_resonance_flag (prt, resonant)
  class(particle_t), intent(inout) :: prt
  logical, intent(in) :: resonant
  select case (prt%status)
  case (PRT_VIRTUAL)
     if (resonant)  prt%status = PRT_RESONANT
  end select
end subroutine particle_set_resonance_flag
```

Set children and parents information.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: set_children => particle_set_children
procedure :: set_parents => particle_set_parents
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_children (prt, idx)
  class(particle_t), intent(inout) :: prt
  integer, dimension(:), intent(in) :: idx
  if (allocated (prt%child))  deallocate (prt%child)
  allocate (prt%child (count (idx /= 0)))
  prt%child = pack (idx, idx /= 0)
end subroutine particle_set_children

subroutine particle_set_parents (prt, idx)
  class(particle_t), intent(inout) :: prt
  integer, dimension(:), intent(in) :: idx
  if (allocated (prt%parent))  deallocate (prt%parent)
  allocate (prt%parent (count (idx /= 0)))
  prt%parent = pack (idx, idx /= 0)
end subroutine particle_set_parents
```

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: add_child => particle_add_child
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_add_child (prt, new_child)
  class(particle_t), intent(inout) :: prt
  integer, intent(in) :: new_child
  integer, dimension(:), allocatable :: idx
  integer :: n, i
  n = prt%get_n_children()
  if (n == 0) then
     call prt%set_children ([new_child])
  else
     do i = 1, n
        if (prt%child(i) == new_child) then
           return
        end if
     end do
     allocate (idx (1:n+1))
     idx(1:n) = prt%get_children ()
     idx(n+1) = new_child
     call prt%set_children (idx)
  end if
end subroutine particle_add_child
```

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: add_children => particle_add_children
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_add_children (prt, new_child)
  class(particle_t), intent(inout) :: prt
  integer, dimension(:), intent(in) :: new_child
```

```
    integer, dimension(:), allocatable :: idx
    integer :: n
    n = prt%get_n_children()
    if (n == 0) then
       call prt%set_children (new_child)
    else
       allocate (idx (1:n+size(new_child)))
       idx(1:n) = prt%get_children ()
       idx(n+1:n+size(new_child)) = new_child
       call prt%set_children (idx)
    end if
  end subroutine particle_add_children
```

⟨*Particles: particle: TBP*⟩+≡

```
  procedure :: set_status => particle_set_status
```

⟨*Particles: procedures*⟩+≡

```
  elemental subroutine particle_set_status (prt, status)
    class(particle_t), intent(inout) :: prt
    integer, intent(in) :: status
    prt%status = status
  end subroutine particle_set_status
```

⟨*Particles: particle: TBP*⟩+≡

```
  procedure :: set_polarization => particle_set_polarization
```

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_set_polarization (prt, polarization)
    class(particle_t), intent(inout) :: prt
    integer, intent(in) :: polarization
    prt%polarization = polarization
  end subroutine particle_set_polarization
```

⟨*Particles: particle: TBP*⟩+≡

```
  generic :: set_vertex => set_vertex_from_vector3, set_vertex_from_xyz, &
       set_vertex_from_vector4, set_vertex_from_xyzt
  procedure :: set_vertex_from_vector4 => particle_set_vertex_from_vector4
  procedure :: set_vertex_from_vector3 => particle_set_vertex_from_vector3
  procedure :: set_vertex_from_xyzt => particle_set_vertex_from_xyzt
  procedure :: set_vertex_from_xyz => particle_set_vertex_from_xyz
```

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_set_vertex_from_vector4 (prt, vertex)
    class(particle_t), intent(inout) :: prt
    type(vector4_t), intent(in) :: vertex
    if (allocated (prt%vertex)) deallocate (prt%vertex)
    allocate (prt%vertex, source=vertex)
  end subroutine particle_set_vertex_from_vector4

  subroutine particle_set_vertex_from_vector3 (prt, vertex)
    class(particle_t), intent(inout) :: prt
    type(vector3_t), intent(in) :: vertex
    type(vector4_t) :: vtx
    vtx = vector4_moving (0._default, vertex)
```

```
    if (allocated (prt%vertex)) deallocate (prt%vertex)
    allocate (prt%vertex, source=vtx)
  end subroutine particle_set_vertex_from_vector3

  subroutine particle_set_vertex_from_xyzt (prt, vx, vy, vz, t)
    class(particle_t), intent(inout) :: prt
    real(default), intent(in) :: vx, vy, vz, t
    type(vector4_t) :: vertex
    if (allocated (prt%vertex)) deallocate (prt%vertex)
    vertex = vector4_moving (t, vector3_moving ([vx, vy, vz]))
    allocate (prt%vertex, source=vertex)
  end subroutine particle_set_vertex_from_xyzt

  subroutine particle_set_vertex_from_xyz (prt, vx, vy, vz)
    class(particle_t), intent(inout) :: prt
    real(default), intent(in) :: vx, vy, vz
    type(vector4_t) :: vertex
    if (allocated (prt%vertex)) deallocate (prt%vertex)
    vertex = vector4_moving (0._default, vector3_moving ([vx, vy, vz]))
    allocate (prt%vertex, source=vertex)
  end subroutine particle_set_vertex_from_xyz
```

Set the lifetime of a particle.

⟨*Particles: particle: TBP*⟩+≡

```
  procedure :: set_lifetime => particle_set_lifetime
```

⟨*Particles: procedures*⟩+≡

```
  elemental subroutine particle_set_lifetime (prt, lifetime)
    class(particle_t), intent(inout) :: prt
    real(default), intent(in) :: lifetime
    if (allocated (prt%lifetime))  deallocate (prt%lifetime)
    allocate (prt%lifetime, source=lifetime)
  end subroutine particle_set_lifetime
```

Accessing contents

The status code.

⟨*Particles: particle: TBP*⟩+≡

```
  procedure :: get_status => particle_get_status
```

⟨*Particles: procedures*⟩+≡

```
  elemental function particle_get_status (prt) result (status)
    integer :: status
    class(particle_t), intent(in) :: prt
    status = prt%status
  end function particle_get_status
```

Return true if the status is either `INCOMING`, `OUTGOING` or `RESONANT`. `BEAM` is kept, if `keep_beams` is set true.

⟨*Particles: particle: TBP*⟩+≡

```
  procedure :: is_real => particle_is_real
```

⟨*Particles: procedures*⟩+≡
```
elemental function particle_is_real (prt, keep_beams) result (flag)
  logical :: flag, kb
  class(particle_t), intent(in) :: prt
  logical, intent(in), optional :: keep_beams
  kb = .false.
  if (present (keep_beams)) kb = keep_beams
  select case (prt%status)
  case (PRT_INCOMING, PRT_OUTGOING, PRT_RESONANT)
     flag = .true.
  case (PRT_BEAM)
     flag = kb
  case default
     flag = .false.
  end select
end function particle_is_real
```

⟨*Particles: particle: TBP*⟩+≡
```
procedure :: is_colored => particle_is_colored
```
⟨*Particles: procedures*⟩+≡
```
elemental function particle_is_colored (particle) result (flag)
  logical :: flag
  class(particle_t), intent(in) :: particle
  flag = particle%col%is_nonzero ()
end function particle_is_colored
```

[90, 100] hopefully catches all of them and not too many.

⟨*Particles: particle: TBP*⟩+≡
```
procedure :: is_hadronic_beam_remnant => particle_is_hadronic_beam_remnant
```
⟨*Particles: procedures*⟩+≡
```
elemental function particle_is_hadronic_beam_remnant (particle) result (flag)
  class(particle_t), intent(in) :: particle
  logical :: flag
  integer :: pdg
  pdg = particle%flv%get_pdg ()
  flag = particle%status == PRT_BEAM_REMNANT .and. &
       abs(pdg) >= 90 .and. abs(pdg) <= 100
end function particle_is_hadronic_beam_remnant
```

⟨*Particles: particle: TBP*⟩+≡
```
procedure :: is_beam_remnant => particle_is_beam_remnant
```
⟨*Particles: procedures*⟩+≡
```
elemental function particle_is_beam_remnant (particle) result (flag)
  class(particle_t), intent(in) :: particle
  logical :: flag
  flag = particle%status == PRT_BEAM_REMNANT
end function particle_is_beam_remnant
```

Polarization status.

⟨*Particles: particle: TBP*⟩+≡
```
procedure :: get_polarization_status => particle_get_polarization_status
```

⟨*Particles: procedures*⟩+≡

```
elemental function particle_get_polarization_status (prt) result (status)
  integer :: status
  class(particle_t), intent(in) :: prt
  status = prt%polarization
end function particle_get_polarization_status
```

Return the PDG code from the flavor component directly.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: get_pdg => particle_get_pdg
```

⟨*Particles: procedures*⟩+≡

```
elemental function particle_get_pdg (prt) result (pdg)
  integer :: pdg
  class(particle_t), intent(in) :: prt
  pdg = prt%flv%get_pdg ()
end function particle_get_pdg
```

Return the color and anticolor quantum numbers.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: get_color => particle_get_color
```

⟨*Particles: procedures*⟩+≡

```
pure function particle_get_color (prt) result (col)
  integer, dimension(2) :: col
  class(particle_t), intent(in) :: prt
  col(1) = prt%col%get_col ()
  col(2) = prt%col%get_acl ()
end function particle_get_color
```

Return a copy of the polarization density matrix.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: get_polarization => particle_get_polarization
```

⟨*Particles: procedures*⟩+≡

```
function particle_get_polarization (prt) result (pol)
  class(particle_t), intent(in) :: prt
  type(polarization_t) :: pol
  pol = prt%pol
end function particle_get_polarization
```

Return the flavor, color and helicity.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: get_flv => particle_get_flv
procedure :: get_col => particle_get_col
procedure :: get_hel => particle_get_hel
```

⟨*Particles: procedures*⟩+≡

```
function particle_get_flv (prt) result (flv)
  class(particle_t), intent(in) :: prt
  type(flavor_t) :: flv
  flv = prt%flv
end function particle_get_flv
```

```
function particle_get_col (prt) result (col)
  class(particle_t), intent(in) :: prt
  type(color_t) :: col
  col = prt%col
end function particle_get_col

function particle_get_hel (prt) result (hel)
  class(particle_t), intent(in) :: prt
  type(helicity_t) :: hel
  hel = prt%hel
end function particle_get_hel
```

Return the helicity (if defined and diagonal).

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: get_helicity => particle_get_helicity
```

⟨*Particles: procedures*⟩+≡

```
elemental function particle_get_helicity (prt) result (hel)
  integer :: hel
  integer, dimension(2) :: hel_arr
  class(particle_t), intent(in) :: prt
  hel = 0
  if (prt%hel%is_defined () .and. prt%hel%is_diagonal ()) then
     hel_arr = prt%hel%to_pair ()
     hel = hel_arr (1)
  end if
end function particle_get_helicity
```

Return the number of children/parents

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: get_n_parents => particle_get_n_parents
procedure :: get_n_children => particle_get_n_children
```

⟨*Particles: procedures*⟩+≡

```
elemental function particle_get_n_parents (prt) result (n)
  integer :: n
  class(particle_t), intent(in) :: prt
  if (allocated (prt%parent)) then
     n = size (prt%parent)
  else
     n = 0
  end if
end function particle_get_n_parents

elemental function particle_get_n_children (prt) result (n)
  integer :: n
  class(particle_t), intent(in) :: prt
  if (allocated (prt%child)) then
     n = size (prt%child)
  else
     n = 0
  end if
end function particle_get_n_children
```

Return the array of parents/children.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: get_parents => particle_get_parents
procedure :: get_children => particle_get_children
```

⟨*Particles: procedures*⟩+≡

```
function particle_get_parents (prt) result (parent)
  class(particle_t), intent(in) :: prt
  integer, dimension(:), allocatable :: parent
  if (allocated (prt%parent)) then
     allocate (parent (size (prt%parent)))
     parent = prt%parent
  else
     allocate (parent (0))
  end if
end function particle_get_parents

function particle_get_children (prt) result (child)
  class(particle_t), intent(in) :: prt
  integer, dimension(:), allocatable :: child
  if (allocated (prt%child)) then
     allocate (child (size (prt%child)))
     child = prt%child
  else
     allocate (child (0))
  end if
end function particle_get_children
```

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: has_children => particle_has_children
```

⟨*Particles: procedures*⟩+≡

```
elemental function particle_has_children (prt) result (has_children)
  logical :: has_children
  class(particle_t), intent(in) :: prt
  has_children = .false.
  if (allocated (prt%child)) then
     has_children = size (prt%child) > 0
  end if
end function particle_has_children
```

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: has_parents => particle_has_parents
```

⟨*Particles: procedures*⟩+≡

```
elemental function particle_has_parents (prt) result (has_parents)
  logical :: has_parents
  class(particle_t), intent(in) :: prt
  has_parents = .false.
  if (allocated (prt%parent)) then
     has_parents = size (prt%parent) > 0
  end if
end function particle_has_parents
```

Return momentum and momentum squared.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: get_momentum => particle_get_momentum
procedure :: get_p2 => particle_get_p2
```

⟨*Particles: procedures*⟩+≡

```
elemental function particle_get_momentum (prt) result (p)
  type(vector4_t) :: p
  class(particle_t), intent(in) :: prt
  p = prt%p
end function particle_get_momentum

elemental function particle_get_p2 (prt) result (p2)
  real(default) :: p2
  class(particle_t), intent(in) :: prt
  p2 = prt%p2
end function particle_get_p2
```

Return the particle vertex, if allocated.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: get_vertex => particle_get_vertex
```

⟨*Particles: procedures*⟩+≡

```
elemental function particle_get_vertex (prt) result (vtx)
  type(vector4_t) :: vtx
  class(particle_t), intent(in) :: prt
  if (allocated (prt%vertex)) then
     vtx = prt%vertex
  else
     vtx = vector4_null
  end if
end function particle_get_vertex
```

Return the lifetime of a particle.

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: get_lifetime => particle_get_lifetime
```

⟨*Particles: procedures*⟩+≡

```
elemental function particle_get_lifetime (prt) result (lifetime)
  real(default) :: lifetime
  class(particle_t), intent(in) :: prt
  if (allocated (prt%lifetime)) then
     lifetime = prt%lifetime
  else
     lifetime = 0
  end if
end function particle_get_lifetime
```

⟨*Particles: particle: TBP*⟩+≡

```
procedure :: momentum_to_pythia6 => particle_momentum_to_pythia6
```

⟨*Particles: procedures*⟩+≡

```
pure function particle_momentum_to_pythia6 (prt) result (p)
  real(double), dimension(1:5) :: p
  class(particle_t), intent(in) :: prt
  p = prt%p%to_pythia6 (sqrt (prt%p2))
end function particle_momentum_to_pythia6
```

14.4.2 Particle sets

A particle set is what is usually called an event: an array of particles. The individual particle entries carry momentum, quantum numbers, polarization, and optionally connections. There is (also optionally) a correlated state-density matrix that maintains spin correlations that are lost in the individual particle entries.

⟨*Particles: public*⟩+≡

```
public :: particle_set_t
```

⟨*Particles: types*⟩+≡

```
type :: particle_set_t
   integer :: n_beam = 0
   integer :: n_in  = 0
   integer :: n_vir = 0
   integer :: n_out = 0
   integer :: n_tot = 0
   integer :: factorization_mode = FM_IGNORE_HELICITY
   type(particle_t), dimension(:), allocatable :: prt
   type(state_matrix_t) :: correlated_state
 contains
 ⟨Particles: particle set: TBP⟩
end type particle_set_t
```

A particle set can be initialized from an interaction or from a HepMC event record.

⟨*Particles: particle set: TBP*⟩≡

```
generic :: init => init_interaction
procedure :: init_interaction => particle_set_init_interaction
```

When a particle set is initialized from a given interaction, we have to determine the branch within the original state matrix that fixes the particle quantum numbers. This is done with the appropriate probabilities, based on a random number `x`. The `mode` determines whether the individual particles become unpolarized, or take a definite (diagonal) helicity, or acquire single-particle polarization matrices. The flag `keep_correlations` tells whether the spin-correlation matrix is to be calculated and stored in addition to the particles. The flag `keep_virtual` tells whether virtual particles should be dropped. Note that if virtual particles are dropped, the spin-correlation matrix makes no sense, and parent-child relations are not set.

For a correct disentangling of color and flavor (in the presence of helicity), we consider two interactions. `int` has no color information, and is used to select a flavor state. Consequently, we trace over helicities here. `int_flows` contains color-flow and potentially helicity information, but is useful only after the flavor

combination has been chosen. So this interaction is used to select helicity and color, but restricted to the selected flavor combination.

`int` and `int_flows` may be identical if there is only a single (or no) color flow. If there is just a single flavor combination, `x(1)` can be set to zero.

The current algorithm of evaluator convolution requires that the beam particles are assumed outgoing (in the beam interaction) and become virtual in all derived interactions. In the particle set they should be re-identified as incoming. The optional integer `n_incoming` can be used to perform this correction.

The flag `is_valid` is false if factorization of the state is not possible, in particular if the squared matrix element is zero.

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_init_interaction &
     (particle_set, is_valid, int, int_flows, mode, x, &
      keep_correlations, keep_virtual, n_incoming, qn_select)
  class(particle_set_t), intent(out) :: particle_set
  logical, intent(out) :: is_valid
  type(interaction_t), intent(in), target :: int, int_flows
  integer, intent(in) :: mode
  real(default), dimension(2), intent(in) :: x
  logical, intent(in) :: keep_correlations, keep_virtual
  integer, intent(in), optional :: n_incoming
  type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_select
  type(state_matrix_t), dimension(:), allocatable, target :: flavor_state
  type(state_matrix_t), dimension(:), allocatable, target :: single_state
  integer :: n_in, n_vir, n_out, n_tot
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn
  logical :: ok
  integer :: i, j
  if (present (n_incoming)) then
     n_in  = n_incoming
     n_vir = int%get_n_vir () - n_incoming
  else
     n_in  = int%get_n_in  ()
     n_vir = int%get_n_vir ()
  end if
  n_out = int%get_n_out ()
  n_tot = int%get_n_tot ()
  particle_set%n_in  = n_in
  particle_set%n_out = n_out
  if (keep_virtual) then
     particle_set%n_vir = n_vir
     particle_set%n_tot = n_tot
  else
     particle_set%n_vir = 0
     particle_set%n_tot = n_in + n_out
  end if
  particle_set%factorization_mode = mode
  allocate (qn (n_tot, 1))
  if (.not. present (qn_select)) then
     call int%factorize &
          (FM_IGNORE_HELICITY, x(1), is_valid, flavor_state)
     do i = 1, n_tot
        qn(i,:) = flavor_state(i)%get_quantum_number (1)
```

```
      end do
   else
      do i = 1, n_tot
         qn(i,:) = qn_select(i)
      end do
   end if
   if (keep_correlations .and. keep_virtual) then
      call particle_set%correlated_state%final ()
      call int_flows%factorize (mode, x(2), ok, &
           single_state, particle_set%correlated_state, qn(:,1))
   else
      call int_flows%factorize (mode, x(2), ok, &
           single_state, qn_in=qn(:,1))
   end if
   is_valid = is_valid .and. ok
   allocate (particle_set%prt (particle_set%n_tot))
   j = 1
   do i = 1, n_tot
      if (i <= n_in) then
         call particle_set%prt(j)%init (single_state(i), PRT_INCOMING, mode)
         call particle_set%prt(j)%set_momentum (int%get_momentum (i))
      else if (i <= n_in + n_vir) then
         if (.not. keep_virtual)  cycle
         call particle_set%prt(j)%init &
              (single_state(i), PRT_VIRTUAL, mode)
         call particle_set%prt(j)%set_momentum (int%get_momentum (i))
      else
         call particle_set%prt(j)%init (single_state(i), PRT_OUTGOING, mode)
         call particle_set%prt(j)%set_momentum &
              (int%get_momentum (i), on_shell = .true.)
      end if
      if (keep_virtual) then
         call particle_set%prt(j)%set_children &
              (interaction_get_children (int, i))
         call particle_set%prt(j)%set_parents &
              (interaction_get_parents (int, i))
      end if
      j = j + 1
   end do
   if (keep_virtual) then
      call particle_set_resonance_flag &
           (particle_set%prt, int%get_resonance_flags ())
   end if
   if (allocated (flavor_state)) then
      do i = 1, size(flavor_state)
         call flavor_state(i)%final ()
      end do
   end if
   do i = 1, size(single_state)
      call single_state(i)%final ()
   end do
end subroutine particle_set_init_interaction
```

⟨*Particles: particle set: TBP*⟩+≡

```
generic :: init => init_particle_set
procedure :: init_particle_set => particle_set_init_particle_set
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_init_particle_set (pset_out, pset_in)
  class(particle_set_t), intent(out) :: pset_out
  type(particle_set_t), intent(in) :: pset_in
  integer :: i
  pset_out%n_beam = pset_in%n_beam
  pset_out%n_in   = pset_in%n_in
  pset_out%n_vir  = pset_in%n_vir
  pset_out%n_out  = pset_in%n_out
  pset_out%n_tot  = pset_in%n_tot
  pset_out%factorization_mode = pset_in%factorization_mode
  if (allocated (pset_in%prt)) then
     allocate (pset_out%prt (size (pset_in%prt)))
     do i = 1, size (pset_in%prt)
        pset_out%prt(i) = pset_in%prt(i)
     end do
  end if
  pset_out%correlated_state = pset_in%correlated_state
end subroutine particle_set_init_particle_set
```

Manually set the model for the stored particles.

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: set_model => particle_set_set_model
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_set_model (particle_set, model)
  class(particle_set_t), intent(inout) :: particle_set
  class(model_data_t), intent(in), target :: model
  integer :: i
  do i = 1, particle_set%n_tot
     call particle_set%prt(i)%set_model (model)
  end do
  call particle_set%correlated_state%set_model (model)
end subroutine particle_set_set_model
```

Duplicate implementation.

⟨*Particles: interfaces*⟩≡

```
interface assignment(=)
   module procedure particle_set_init_particle_set
end interface
```

Pointer components are hidden inside the particle polarization, and in the correlated state matrix.

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: final => particle_set_final
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_final (particle_set)
  class(particle_set_t), intent(inout) :: particle_set
  integer :: i
  if (allocated (particle_set%prt)) then
```

```
      do i = 1, size(particle_set%prt)
         call particle_set%prt(i)%final ()
      end do
      deallocate (particle_set%prt)
    end if
    call particle_set%correlated_state%final ()
  end subroutine particle_set_final
```

⟨*Particles: particle set: TBP*⟩+≡

```
  generic :: get_momenta => get_momenta_all
  generic :: get_momenta => get_momenta_indices
  procedure :: get_momenta_all => particle_set_get_momenta_all
  procedure :: get_momenta_indices => particle_set_get_momenta_indices
```

⟨*Particles: procedures*⟩+≡

```
  function particle_set_get_momenta_all (particle_set) result (p)
    class(particle_set_t), intent(in) :: particle_set
    type(vector4_t), dimension(:), allocatable :: p
    allocate (p (size (particle_set%prt)))
    p = particle_set%prt%p
  end function particle_set_get_momenta_all
```

⟨*Particles: procedures*⟩+≡

```
  function particle_set_get_momenta_indices (particle_set, indices) result (p)
     type(vector4_t), dimension(:), allocatable :: p
     class(particle_set_t), intent(in) :: particle_set
     integer, intent(in), dimension(:), allocatable :: indices
     integer :: i
     allocate (p (size (indices)))
     do i = 1, size (indices)
        p(i) = particle_set%prt(indices(i))%p
     end do
  end function particle_set_get_momenta_indices
```

⟨*Particles: particle set: TBP*⟩+≡

```
  procedure :: set_momenta => particle_set_set_momenta
```

⟨*Particles: procedures*⟩+≡

```
  pure subroutine particle_set_set_momenta (particle_set, p)
    class(particle_set_t), intent(inout) :: particle_set
    type(vector4_t), dimension(:), intent(in) :: p
    if (.not. allocated (particle_set%prt)) then
       allocate (particle_set%prt (size (p)))
    end if
    particle_set%prt%p = p
  end subroutine particle_set_set_momenta
```

⟨*Particles: particle set: TBP*⟩+≡

```
  procedure :: replace_incoming_momenta => particle_set_replace_incoming_momenta
```

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_set_replace_incoming_momenta (particle_set, p)
    class(particle_set_t), intent(inout) :: particle_set
```

```
    type(vector4_t), intent(in), dimension(:) :: p
    integer :: i, j
    i = 1
    do j = 1, particle_set%get_n_tot ()
       if (particle_set%prt(j)%get_status () == PRT_INCOMING) then
          particle_set%prt(j)%p = p(i)
          i = i + 1
          if (i > particle_set%n_in) exit
       end if
    end do
  end subroutine particle_set_replace_incoming_momenta
```

⟨*Particles: particle set: TBP*⟩+≡

```
  procedure :: replace_outgoing_momenta => particle_set_replace_outgoing_momenta
```

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_set_replace_outgoing_momenta (particle_set, p)
    class(particle_set_t), intent(inout) :: particle_set
    type(vector4_t), intent(in), dimension(:) :: p
    integer :: i, j
    i = particle_set%n_in + 1
    do j = 1, particle_set%n_tot
       if (particle_set%prt(j)%get_status () == PRT_OUTGOING) then
          particle_set%prt(j)%p = p(i)
          i = i + 1
       end if
    end do
  end subroutine particle_set_replace_outgoing_momenta
```

⟨*Particles: particle set: TBP*⟩+≡

```
  procedure :: parent_add_child => particle_set_parent_add_child
```

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_set_parent_add_child (particle_set, parent, child)
    class(particle_set_t), intent(inout) :: particle_set
    integer, intent(in) :: parent, child
    call particle_set%prt(child)%set_parents ([parent])
    call particle_set%prt(parent)%add_child (child)
  end subroutine particle_set_parent_add_child
```

Given the `particle_set` before radiation, the new momenta `p_radiated`, the `emitter` and the `flv_radiated` as well as the `model` and a random number `r_color` for chosing a color, we update the `particle_set`.

⟨*Particles: particle set: TBP*⟩+≡

```
  procedure :: build_radiation => particle_set_build_radiation
```

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_set_build_radiation (particle_set, p_radiated, &
     emitter, flv_radiated, model, r_color)
     class(particle_set_t), intent(inout) :: particle_set
     type(vector4_t), intent(in), dimension(:) :: p_radiated
     integer, intent(in) :: emitter
     integer, intent(in), dimension(:) :: flv_radiated
     class(model_data_t), intent(in), target :: model
```

```
real(default), intent(in) :: r_color
type(particle_set_t) :: new_particle_set
type(particle_t) :: new_particle
integer :: i
integer :: pdg_index_emitter, pdg_index_radiation
integer, dimension(:), allocatable :: parents, children
type(flavor_t) :: new_flv
logical, dimension(:), allocatable :: status_mask
integer, dimension(:), allocatable :: &
     i_in1, i_beam1, i_remnant1, i_virt1, i_out1
integer, dimension(:), allocatable :: &
     i_in2, i_beam2, i_remnant2, i_virt2, i_out2
integer :: n_in1, n_beam1, n_remnant1, n_virt1, n_out1
integer :: n_in2, n_beam2, n_remnant2, n_virt2, n_out2
integer :: n, n_tot
integer :: i_emitter

n = particle_set%get_n_tot ()
allocate (status_mask (n))
!!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
do i = 1, n
   status_mask(i) = particle_set%prt(i)%get_status () == PRT_INCOMING
end do
n_in1 = count (status_mask)
allocate (i_in1 (n_in1))
i_in1 = particle_set%get_indices (status_mask)
do i = 1, n
   status_mask(i) = particle_set%prt(i)%get_status () == PRT_BEAM
end do
n_beam1 = count (status_mask)
allocate (i_beam1 (n_beam1))
i_beam1 = particle_set%get_indices (status_mask)
do i = 1, n
   status_mask(i) = particle_set%prt(i)%get_status () == PRT_BEAM_REMNANT
end do
n_remnant1 = count (status_mask)
allocate (i_remnant1 (n_remnant1))
i_remnant1 = particle_set%get_indices (status_mask)
do i = 1, n
   status_mask(i) = particle_set%prt(i)%get_status () == PRT_VIRTUAL
end do
n_virt1 = count (status_mask)
allocate (i_virt1 (n_virt1))
i_virt1 = particle_set%get_indices (status_mask)
do i = 1, n
   status_mask(i) = particle_set%prt(i)%get_status () == PRT_OUTGOING
end do
n_out1 = count (status_mask)
allocate (i_out1 (n_out1))
i_out1 = particle_set%get_indices (status_mask)

n_in2 = n_in1; n_beam2 = n_beam1; n_remnant2 = n_remnant1
n_virt2 = n_virt1 + n_out1
n_out2 = n_out1 + 1
```

```
n_tot = n_in2 + n_beam2 + n_remnant2 + n_virt2 + n_out2

allocate (i_in2 (n_in2), i_beam2 (n_beam2), i_remnant2 (n_remnant2))
i_in2 = i_in1; i_beam2 = i_beam1; i_remnant2 = i_remnant1

allocate (i_virt2 (n_virt2))
i_virt2(1 : n_virt1) = i_virt1
i_virt2(n_virt1 + 1 : n_virt2) = i_out1

allocate (i_out2 (n_out2))
i_out2(1 : n_out1) = i_out1(1 : n_out1) + n_out1
i_out2(n_out2) = n_tot

new_particle_set%n_beam = n_beam2
new_particle_set%n_in = n_in2
new_particle_set%n_vir = n_virt2
new_particle_set%n_out = n_out2
new_particle_set%n_tot = n_tot
new_particle_set%correlated_state = particle_set%correlated_state
allocate (new_particle_set%prt (n_tot))
if (size (i_beam1) > 0) new_particle_set%prt(i_beam2) = particle_set%prt(i_beam1)
if (size (i_remnant1) > 0) new_particle_set%prt(i_remnant2) = particle_set%prt(i_remnant1)
do i = 1, n_virt1
   new_particle_set%prt(i_virt2(i)) = particle_set%prt(i_virt1(i))
end do

do i = n_virt1 + 1, n_virt2
   new_particle_set%prt(i_virt2(i)) = particle_set%prt(i_out1(i - n_virt1))
   call new_particle_set%prt(i_virt2(i))%reset_status (PRT_VIRTUAL)
end do

do i = 1, n_in2
   new_particle_set%prt(i_in2(i)) = particle_set%prt(i_in1(i))
   new_particle_set%prt(i_in2(i))%p = p_radiated (i)
end do

do i = 1, n_out2 - 1
   new_particle_set%prt(i_out2(i)) = particle_set%prt(i_out1(i))
   new_particle_set%prt(i_out2(i))%p = p_radiated(i + n_in2)
   call new_particle_set%prt(i_out2(i))%reset_status (PRT_OUTGOING)
end do

call new_particle%reset_status (PRT_OUTGOING)
call new_particle%set_momentum (p_radiated (n_in2 + n_out2))

!!! Helicity and polarization handling is missing at this point
!!! Also, no helicities or polarizations yet
pdg_index_emitter = flv_radiated (emitter)
pdg_index_radiation = flv_radiated (n_in2 + n_out2)
call new_flv%init (pdg_index_radiation, model)
i_emitter = emitter + n_virt2 + n_remnant2 + n_beam2

call reassign_colors (new_particle, new_particle_set%prt(i_emitter), &
   pdg_index_radiation, pdg_index_emitter, r_color)
```

```
   call new_particle%set_flavor (new_flv)
   new_particle_set%prt(n_tot) = new_particle

   allocate (children (n_out2))
   children = i_out2
   do i = n_in2 + n_beam2 + n_remnant2 + n_virt1 + 1, n_in2 + n_beam2 + n_remnant2 + n_virt2
      call new_particle_set%prt(i)%set_children (children)
   end do

   !!! Set proper parents for outgoing particles
   allocate (parents (n_out1))
   parents = i_out1
   do i = n_in2 + n_beam2 + n_remnant2 + n_virt2 + 1, n_tot
      call new_particle_set%prt(i)%set_parents (parents)
   end do
   call particle_set%init (new_particle_set)
contains
```

⟨*build radiation: set color offset*⟩

```
  subroutine reassign_colors (prt_radiated, prt_emitter, i_rad, i_em, r_col)
    type(particle_t), intent(inout) :: prt_radiated, prt_emitter
    integer, intent(in) :: i_rad, i_em
    real(default), intent(in) :: r_col
    type(color_t) :: col_rad, col_em
    if (is_quark (i_em) .and. is_gluon (i_rad)) then
       call reassign_colors_qg (prt_emitter, col_rad, col_em)
    else if (is_gluon (i_em) .and. is_gluon (i_rad)) then
       call reassign_colors_gg (prt_emitter, r_col, col_rad, col_em)
    else if (is_gluon (i_em) .and. is_quark (i_rad)) then
       call reassign_colors_qq (prt_emitter, i_em, col_rad, col_em)
    else
       call msg_fatal ("Invalid splitting")
    end if
    call prt_emitter%set_color (col_em)
    call prt_radiated%set_color (col_rad)
  end subroutine reassign_colors

  subroutine reassign_colors_qg (prt_emitter, col_rad, col_em)
    type(particle_t), intent(in) :: prt_emitter
    type(color_t), intent(out) :: col_rad, col_em
    integer, dimension(2) :: color_rad, color_em
    integer :: i1, i2
    integer :: new_color_index
    logical :: is_anti_quark

    color_em = prt_emitter%get_color ()
    i1 = 1; i2 = 2
    is_anti_quark = color_em(2) /= 0
    if (is_anti_quark) then
       i1 = 2; i2 = 1
    end if
    new_color_index = color_em(i1)+1
    color_rad(i1) = color_em(i1)
```

```
      color_rad(i2) = new_color_index
      color_em(i1) = new_color_index
      call col_em%init_col_acl (color_em(1), color_em(2))
      call col_rad%init_col_acl (color_rad(1), color_rad(2))
    end subroutine reassign_colors_qg

    subroutine reassign_colors_gg (prt_emitter, random, col_rad, col_em)
      !!! NOT TESTED YET
      type(particle_t), intent(in) :: prt_emitter
      real(default), intent(in) :: random
      type(color_t), intent(out) :: col_rad, col_em
      integer, dimension(2) :: color_rad, color_em
      integer :: i1, i2
      integer :: new_color_index

      color_em = prt_emitter%get_color ()
      new_color_index = maxval (abs (color_em))
      i1 = 1; i2 = 2
      if (random < 0.5) then
         i1 = 2; i2 = 1
      end if
      color_rad(i1) = new_color_index
      color_rad(i2) = color_em(i2)
      color_em(i2) = new_color_index
      call col_em%init_col_acl (color_em(1), color_em(2))
      call col_rad%init_col_acl (color_rad(1), color_rad(2))
    end subroutine reassign_colors_gg

    subroutine reassign_colors_qq (prt_emitter, pdg_emitter, col_rad, col_em)
      !!! NOT TESTED YET
      type(particle_t), intent(in) :: prt_emitter
      integer, intent(in) :: pdg_emitter
      type(color_t), intent(out) :: col_rad, col_em
      integer, dimension(2) :: color_rad, color_em
      integer :: i1, i2
      logical :: is_anti_quark

      color_em = prt_emitter%get_color ()
      i1 = 1; i2 = 2
      is_anti_quark = pdg_emitter < 0
      if (is_anti_quark) then
         i1 = 2; i1 = 1
      end if
      color_em(i2) = 0
      color_rad(i1) = 0
      color_rad(i2) = color_em(i1)
      call col_em%init_col_acl (color_em(1), color_em(2))
      call col_rad%init_col_acl (color_rad(1), color_rad(2))
    end subroutine reassign_colors_qq
  end subroutine particle_set_build_radiation
```

Increments the color indices of all particles by their maximal value to distinguish them from the record-keeping Born particles in the LHE-output if the virtual

entries are kept.

⟨*build radiation: set color offset*⟩≡

```
subroutine set_color_offset (particle_set)
  type(particle_set_t), intent(inout) :: particle_set
  integer, dimension(2) :: color
  integer :: i, i_color_max
  type(color_t) :: new_color

  i_color_max = 0
  do i = 1, size (particle_set%prt)
     associate (prt => particle_set%prt(i))
        if (prt%get_status () <= PRT_INCOMING) cycle
        color = prt%get_color ()
        i_color_max = maxval([i_color_max, color(1), color(2)])
     end associate
  end do


  do i = 1, size (particle_set%prt)
     associate (prt => particle_set%prt(i))
        if (prt%get_status () /= PRT_OUTGOING) cycle
        color = prt%get_color ()
        where (color /= 0) color = color + i_color_max
        call new_color%init_col_acl (color(1), color(2))
        call prt%set_color (new_color)
     end associate
  end do
end subroutine set_color_offset
```

Output (default format)

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: write => particle_set_write
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_write &
  (particle_set, unit, testflag, summary, compressed)
  class(particle_set_t), intent(in) :: particle_set
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag, summary, compressed
  logical :: summ, comp, pol
  type(vector4_t) :: sum_vec
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  summ = .false.; if (present (summary)) summ = summary
  comp = .false.; if (present (compressed)) comp = compressed
  pol = particle_set%factorization_mode /= FM_IGNORE_HELICITY
  write (u, "(1x,A)") "Particle set:"
  call write_separator (u)
  if (comp) then
     if (pol) then
        write (u, &
             "((A4,1X),(A6,1X),(A7,1X),(A3),2(A4,1X),2(A20,1X),5(A12,1X))") &
             "Nr", "Status", "Flavor", "Hel", "Col", "ACol", &
             "Parents", "Children", &
```

```
            "P(0)", "P(1)", "P(2)", "P(3)", "P^2"
      else
         write (u, &
            "((A4,1X),(A6,1X),(A7,1X),2(A4,1X),2(A20,1X),5(A12,1X))") &
            "Nr", "Status", "Flavor", "Col", "ACol", &
            "Parents", "Children", &
            "P(0)", "P(1)", "P(2)", "P(3)", "P^2"
      end if
   end if
   if (particle_set%n_tot /= 0) then
      do i = 1, particle_set%n_tot
         if (comp) then
            write (u, "(I4,1X,2X)", advance="no") i
         else
            write (u, "(1x,A,1x,I0)", advance="no") "Particle", i
         end if
         call particle_set%prt(i)%write (u, testflag = testflag, &
            compressed = comp, polarization = pol)
      end do
      if (particle_set%correlated_state%is_defined ()) then
         call write_separator (u)
         write (u, *) "Correlated state density matrix:"
         call particle_set%correlated_state%write (u)
      end if
      if (summ) then
         call write_separator (u)
         write (u, "(A)", advance="no") &
            "Sum of incoming momenta: p(0:3) =     "
         sum_vec = sum (particle_set%prt%p, &
            mask=particle_set%prt%get_status () == PRT_INCOMING)
         call pacify (sum_vec, tolerance = 1E-3_default)
         call sum_vec%write (u, compressed=.true.)
         write (u, *)
         write (u, "(A)", advance="no") &
            "Sum of beam remnant momenta: p(0:3) = "
         sum_vec = sum (particle_set%prt%p, &
            mask=particle_set%prt%get_status () == PRT_BEAM_REMNANT)
         call pacify (sum_vec, tolerance = 1E-3_default)
         call sum_vec%write (u, compressed=.true.)
         write (u, *)
         write (u, "(A)", advance="no") &
            "Sum of outgoing momenta: p(0:3) =     "
         sum_vec = sum (particle_set%prt%p, &
            mask=particle_set%prt%get_status () == PRT_OUTGOING)
         call pacify (sum_vec, tolerance = 1E-3_default)
         call sum_vec%write (u, compressed=.true.)
         write (u, "(A)") ""
      end if
   else
      write (u, "(3x,A)") "[empty]"
   end if
end subroutine particle_set_write
```

14.4.3 I/O formats

Here, we define input/output of particle sets in various formats. This is the right place since particle sets contain most of the event information.

All write/read routines take as first argument the object, as second argument the I/O unit which in this case is a mandatory argument. Then follow further event data.

Internal binary format

This format is supposed to contain the complete information, so the particle data set can be fully reconstructed. The exception is the model part of the particle flavors; this is unassigned for the flavor values read from file.

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: write_raw => particle_set_write_raw
procedure :: read_raw => particle_set_read_raw
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_write_raw (particle_set, u)
  class(particle_set_t), intent(in) :: particle_set
  integer, intent(in) :: u
  integer :: i
  write (u) &
       particle_set%n_beam, particle_set%n_in, &
       particle_set%n_vir, particle_set%n_out
  write (u) particle_set%factorization_mode
  write (u) particle_set%n_tot
  do i = 1, particle_set%n_tot
     call particle_set%prt(i)%write_raw (u)
  end do
  call particle_set%correlated_state%write_raw (u)
end subroutine particle_set_write_raw

subroutine particle_set_read_raw (particle_set, u, iostat)
  class(particle_set_t), intent(out) :: particle_set
  integer, intent(in) :: u
  integer, intent(out) :: iostat
  integer :: i
  read (u, iostat=iostat) &
       particle_set%n_beam, particle_set%n_in, &
       particle_set%n_vir, particle_set%n_out
  read (u, iostat=iostat) particle_set%factorization_mode
  read (u, iostat=iostat) particle_set%n_tot
  allocate (particle_set%prt (particle_set%n_tot))
  do i = 1, size (particle_set%prt)
     call particle_set%prt(i)%read_raw (u, iostat=iostat)
  end do
  call particle_set%correlated_state%read_raw (u, iostat=iostat)
end subroutine particle_set_read_raw
```

Get contents

Find parents/children of a particular particle recursively; the search terminates if a parent/child has status BEAM, INCOMING, OUTGOING or RESONANT.

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: get_real_parents => particle_set_get_real_parents
procedure :: get_real_children => particle_set_get_real_children
```

⟨*Particles: procedures*⟩+≡

```
function particle_set_get_real_parents (pset, i, keep_beams) result (parent)
  integer, dimension(:), allocatable :: parent
  class(particle_set_t), intent(in) :: pset
  integer, intent(in) :: i
  logical, intent(in), optional :: keep_beams
  logical, dimension(:), allocatable :: is_real
  logical, dimension(:), allocatable :: is_parent, is_real_parent
  logical :: kb
  integer :: j, k
  kb = .false.
  if (present (keep_beams)) kb = keep_beams
  allocate (is_real (pset%n_tot))
  !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
  do j = 1, pset%n_tot
     is_real(j) = pset%prt(j)%is_real (kb)
  end do
  !!! is_real = pset%prt%is_real (kb)
  allocate (is_parent (pset%n_tot), is_real_parent (pset%n_tot))
  is_real_parent = .false.
  is_parent = .false.
  is_parent(pset%prt(i)%get_parents()) = .true.
  do while (any (is_parent))
     where (is_real .and. is_parent)
        is_real_parent = .true.
        is_parent = .false.
     end where
     mark_next_parent: do j = size (is_parent), 1, -1
        if (is_parent(j)) then
           is_parent(pset%prt(j)%get_parents()) = .true.
           is_parent(j) = .false.
           exit mark_next_parent
        end if
     end do mark_next_parent
  end do
  allocate (parent (count (is_real_parent)))
  j = 0
  do k = 1, size (is_parent)
     if (is_real_parent(k)) then
        j = j + 1
        parent(j) = k
     end if
  end do
end function particle_set_get_real_parents

function particle_set_get_real_children (pset, i, keep_beams) result (child)
  integer, dimension(:), allocatable :: child
```

```
    class(particle_set_t), intent(in) :: pset
    integer, intent(in) :: i
    logical, dimension(:), allocatable :: is_real
    logical, dimension(:), allocatable :: is_child, is_real_child
    logical, intent(in), optional :: keep_beams
    integer :: j, k
    logical :: kb
    kb = .false.
    if (present (keep_beams)) kb = keep_beams
    allocate (is_real (pset%n_tot))
    !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
    do j = 1, pset%n_tot
       is_real(j) = pset%prt(j)%is_real (kb)
    end do
    is_real = pset%prt%is_real (kb)
    allocate (is_child (pset%n_tot), is_real_child (pset%n_tot))
    is_real_child = .false.
    is_child = .false.
    is_child(pset%prt(i)%get_children()) = .true.
    do while (any (is_child))
       where (is_real .and. is_child)
          is_real_child = .true.
          is_child = .false.
       end where
       mark_next_child: do j = 1, size (is_child)
          if (is_child(j)) then
             is_child(pset%prt(j)%get_children()) = .true.
             is_child(j) = .false.
             exit mark_next_child
          end if
       end do mark_next_child
    end do
    allocate (child (count (is_real_child)))
    j = 0
    do k = 1, size (is_child)
       if (is_real_child(k)) then
          j = j + 1
          child(j) = k
       end if
    end do
  end function particle_set_get_real_children
```

Get the n_tot, n_in, and n_out values out of the particle set.

⟨*Particles: particle set: TBP*⟩+≡

```
  procedure :: get_n_beam => particle_set_get_n_beam
  procedure :: get_n_in => particle_set_get_n_in
  procedure :: get_n_vir => particle_set_get_n_vir
  procedure :: get_n_out => particle_set_get_n_out
  procedure :: get_n_tot => particle_set_get_n_tot
  procedure :: get_n_remnants => particle_set_get_n_remnants
```

⟨*Particles: procedures*⟩+≡

```
  function particle_set_get_n_beam (pset) result (n_beam)
     class(particle_set_t), intent(in) :: pset
```

```
   integer :: n_beam
   n_beam = pset%n_beam
end function particle_set_get_n_beam

function particle_set_get_n_in (pset) result (n_in)
   class(particle_set_t), intent(in) :: pset
   integer :: n_in
   n_in = pset%n_in
end function particle_set_get_n_in

function particle_set_get_n_vir (pset) result (n_vir)
   class(particle_set_t), intent(in) :: pset
   integer :: n_vir
   n_vir = pset%n_vir
 end function particle_set_get_n_vir

function particle_set_get_n_out (pset) result (n_out)
   class(particle_set_t), intent(in) :: pset
   integer :: n_out
   n_out = pset%n_out
end function particle_set_get_n_out

function particle_set_get_n_tot (pset) result (n_tot)
   class(particle_set_t), intent(in) :: pset
   integer :: n_tot
   n_tot = pset%n_tot
end function particle_set_get_n_tot

function particle_set_get_n_remnants (pset) result (n_remn)
  class(particle_set_t), intent(in) :: pset
  integer :: n_remn
  if (allocated (pset%prt)) then
     n_remn = count (pset%prt%get_status () == PRT_BEAM_REMNANT)
  else
     n_remn = 0
  end if
end function particle_set_get_n_remnants
```

Return a pointer to the particle corresponding to the number

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: get_particle => particle_set_get_particle
```

⟨*Particles: procedures*⟩+≡

```
function particle_set_get_particle (pset, index) result (particle)
  class(particle_set_t), intent(in) :: pset
  integer, intent(in) :: index
  type(particle_t) :: particle
  particle = pset%prt(index)
end function particle_set_get_particle
```

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: get_indices => particle_set_get_indices
```

⟨*Particles: procedures*⟩+≡

```
pure function particle_set_get_indices (pset, mask) result (finals)
  integer, dimension(:), allocatable :: finals
  class(particle_set_t), intent(in) :: pset
  logical, dimension(:), intent(in) :: mask
  integer, dimension(size(mask)) :: indices
  integer :: i
  allocate (finals (count (mask)))
  indices = [(i, i=1, pset%n_tot)]
  finals = pack (indices, mask)
end function particle_set_get_indices
```

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: get_in_and_out_momenta => particle_set_get_in_and_out_momenta
```

⟨*Particles: procedures*⟩+≡

```
function particle_set_get_in_and_out_momenta (pset) result (phs_point)
  type(phs_point_t) :: phs_point
  class(particle_set_t), intent(in) :: pset
  logical, dimension(:), allocatable :: mask
  integer, dimension(:), allocatable :: indices
  type(vector4_t), dimension(:), allocatable :: p
  allocate (mask (pset%get_n_tot ()))
  allocate (p (size (pset%prt)))
  mask = pset%prt%status == PRT_INCOMING .or. &
         pset%prt%status == PRT_OUTGOING
  allocate (indices (count (mask)))
  indices = pset%get_indices (mask)
  phs_point = pset%get_momenta (indices)
end function particle_set_get_in_and_out_momenta
```

Tools

Build a new particles array without hadronic remnants but with `n_extra` additional spots. We also update the mother-daughter relations assuming the ordering `b`, `i`, `r`, `x`, `o`.

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: without_hadronic_remnants => &
     particle_set_without_hadronic_remnants
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_without_hadronic_remnants &
       (particle_set, particles, n_particles, n_extra)
  class(particle_set_t), intent(inout) :: particle_set
  type(particle_t), dimension(:), allocatable, intent(out) :: particles
  integer, intent(out) :: n_particles
  integer, intent(in) :: n_extra
  logical, dimension(:), allocatable :: no_hadronic_remnants, &
       no_hadronic_children
  integer, dimension(:), allocatable :: children, new_children
  integer :: i, j, k, first_remnant
  first_remnant = particle_set%n_tot
  do i = 1, particle_set%n_tot
```

```
      if (particle_set%prt(i)%is_hadronic_beam_remnant ()) then
         first_remnant = i
         exit
      end if
   end do
   n_particles = count (.not. particle_set%prt%is_hadronic_beam_remnant ())
   allocate (no_hadronic_remnants (particle_set%n_tot))
   !!! !!! !!! Workaround for intel 16.0 standard-semantics bug
   !!! no_hadronic_remnants = .not. particle_set%prt%is_hadronic_beam_remnant ()
   do i = 1, particle_set%n_tot
      no_hadronic_remnants(i) = .not. particle_set%prt(i)%is_hadronic_beam_remnant ()
   end do
   allocate (particles (n_particles + n_extra))
   k = 1
   do i = 1, particle_set%n_tot
      if (no_hadronic_remnants(i)) then
         particles(k) = particle_set%prt(i)
         k = k + 1
      end if
   end do
   if (n_particles /= particle_set%n_tot) then
      do i = 1, n_particles
         select case (particles(i)%get_status ())
         case (PRT_BEAM)
            if (allocated (children))  deallocate (children)
            allocate (children (particles(i)%get_n_children ()))
            children = particles(i)%get_children ()
            if (allocated (no_hadronic_children)) &
                 deallocate (no_hadronic_children)
            allocate (no_hadronic_children (particles(i)%get_n_children ()))
            no_hadronic_children = .not. &
                 particle_set%prt(children)%is_hadronic_beam_remnant ()
            if (allocated (new_children))  deallocate (new_children)
            allocate (new_children (count (no_hadronic_children)))
            new_children = pack (children, no_hadronic_children)
            call particles(i)%set_children (new_children)
         case (PRT_INCOMING, PRT_RESONANT)
            ⟨update children after remnant⟩
         case (PRT_OUTGOING, PRT_BEAM_REMNANT)
         case default
         end select
      end do
   end if
end subroutine particle_set_without_hadronic_remnants
```

⟨*update children after remnant*⟩≡

```
if (allocated (children))  deallocate (children)
allocate (children (particles(i)%get_n_children ()))
children = particles(i)%get_children ()
do j = 1, size (children)
   if (children(j) > first_remnant) then
      children(j) = children (j) - &
           (particle_set%n_tot - n_particles)
   end if
```

```
  end do
  call particles(i)%set_children (children)
```

Build a new particles array without remnants but with `n_extra` additional spots. We also update the mother-daughter relations assuming the ordering `b`, `i`, `r`, `x`, `o`.

⟨*Particles: particle set: TBP*⟩+≡

```
    procedure :: without_remnants => particle_set_without_remnants
```

⟨*Particles: procedures*⟩+≡

```
    subroutine particle_set_without_remnants &
           (particle_set, particles, n_particles, n_extra)
      class(particle_set_t), intent(inout) :: particle_set
      type(particle_t), dimension(:), allocatable, intent(out) :: particles
      integer, intent(in) :: n_extra
      integer, intent(out) :: n_particles
      logical, dimension(:), allocatable :: no_remnants, no_children
      integer, dimension(:), allocatable :: children, new_children
      integer :: i,j, k, first_remnant
      first_remnant = particle_set%n_tot
      do i = 1, particle_set%n_tot
         if (particle_set%prt(i)%is_beam_remnant ()) then
            first_remnant = i
            exit
         end if
      end do
      allocate (no_remnants (particle_set%n_tot))
      no_remnants = .not. (particle_set%prt%is_beam_remnant ())
      n_particles = count (no_remnants)
      allocate (particles (n_particles + n_extra))
      k = 1
      do i = 1, particle_set%n_tot
         if (no_remnants(i)) then
            particles(k) = particle_set%prt(i)
            k = k + 1
         end if
      end do
      if (n_particles /= particle_set%n_tot) then
         do i = 1, n_particles
            select case (particles(i)%get_status ())
            case (PRT_BEAM)
               if (allocated (children))  deallocate (children)
               allocate (children (particles(i)%get_n_children ()))
               children = particles(i)%get_children ()
               if (allocated (no_children))  deallocate (no_children)
               allocate (no_children (particles(i)%get_n_children ()))
               no_children = .not. (particle_set%prt(children)%is_beam_remnant ())
               if (allocated (new_children))  deallocate (new_children)
               allocate (new_children (count (no_children)))
               new_children = pack (children, no_children)
               call particles(i)%set_children (new_children)
            case (PRT_INCOMING, PRT_RESONANT)
               ⟨update children after remnant⟩
            case (PRT_OUTGOING, PRT_BEAM_REMNANT)
            case default
```

```
          end select
       end do
    end if
  end subroutine particle_set_without_remnants
```

⟨*Particles: particle set: TBP*⟩+≡

```
  procedure :: find_particle => particle_set_find_particle
```

⟨*Particles: procedures*⟩+≡

```
  pure function particle_set_find_particle &
         (particle_set, pdg, momentum, abs_smallness, rel_smallness) result (idx)
    integer :: idx
    class(particle_set_t), intent(in) :: particle_set
    integer, intent(in) :: pdg
    type(vector4_t), intent(in) :: momentum
    real(default), intent(in), optional :: abs_smallness, rel_smallness
    integer :: i, j
    logical, dimension(0:3) :: equals
    idx = 0
    do i = 1, size (particle_set%prt)
       if (particle_set%prt(i)%flv%get_pdg () == pdg) then
          !!! Workaround for gfortran 4.8.3 with overloaded elemental function
          do j = 0, 3
            equals(j) = nearly_equal (particle_set%prt(i)%p%p(j), momentum%p(j), &
                                      abs_smallness, rel_smallness)
          end do
          if (all (equals)) then
             idx = i
             return
          end if
       end if
    end do
  end function particle_set_find_particle
```

⟨*Particles: particle set: TBP*⟩+≡

```
  procedure :: reverse_find_particle => particle_set_reverse_find_particle
```

⟨*Particles: procedures*⟩+≡

```
  pure function particle_set_reverse_find_particle &
         (particle_set, pdg, momentum, abs_smallness, rel_smallness) result (idx)
    integer :: idx
    class(particle_set_t), intent(in) :: particle_set
    integer, intent(in) :: pdg
    type(vector4_t), intent(in) :: momentum
    real(default), intent(in), optional :: abs_smallness, rel_smallness
    integer :: i
    idx = 0
    do i = size (particle_set%prt), 1, -1
       if (particle_set%prt(i)%flv%get_pdg () == pdg) then
          if (all (nearly_equal (particle_set%prt(i)%p%p, momentum%p, &
               abs_smallness, rel_smallness))) then
             idx = i
             return
          end if
```

```
      end if
    end do
  end function particle_set_reverse_find_particle
```

This connects broken links of the form something $\to i \to$ none or$j$ and none $\to j \to$ something or none where the particles i and j are *identical*. It also works if $i \to j$, directly, and thus removes duplicates. We are removing j and connect the possible daughters to i.

⟨*Particles: particle set: TBP*⟩+≡

```
  procedure :: remove_duplicates => particle_set_remove_duplicates
```

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_set_remove_duplicates (particle_set, smallness)
    class(particle_set_t), intent(inout) :: particle_set
    real(default), intent(in) :: smallness
    integer :: n_removals
    integer, dimension(particle_set%n_tot) :: to_remove
    type(particle_t), dimension(:), allocatable :: particles
    type(vector4_t) :: p_i
    integer, dimension(:), allocatable :: map
    to_remove = 0
    call find_duplicates ()
    n_removals = count (to_remove > 0)
    if (n_removals > 0) then
       call strip_duplicates (particles)
       call particle_set%replace (particles)
    end if

  contains

  ⟨Particles: remove duplicates: procedures⟩

  end subroutine particle_set_remove_duplicates
```

This doesn't catch all cases. Missing are splittings of the type $i \to$ something and$j$.

⟨*Particles: remove duplicates: procedures*⟩≡

```
  subroutine find_duplicates ()
    integer :: pdg_i, child_i, i, j
    OUTER: do i = 1, particle_set%n_tot
       if (particle_set%prt(i)%status == PRT_OUTGOING .or. &
            particle_set%prt(i)%status == PRT_VIRTUAL .or. &
            particle_set%prt(i)%status == PRT_RESONANT) then
          if (allocated (particle_set%prt(i)%child)) then
             if (size (particle_set%prt(i)%child) > 1) cycle OUTER
             if (size (particle_set%prt(i)%child) == 1) then
                child_i = particle_set%prt(i)%child(1)
             else
                child_i = 0
             end if
          else
             child_i = 0
          end if
          pdg_i = particle_set%prt(i)%flv%get_pdg ()
```

```
         p_i = particle_set%prt(i)%p
         do j = i + 1, particle_set%n_tot
            if (pdg_i == particle_set%prt(j)%flv%get_pdg ()) then
               if (all (nearly_equal (particle_set%prt(j)%p%p, p_i%p, &
                    abs_smallness = smallness, &
                    rel_smallness = 1E4_default * smallness))) then
                  if (child_i == 0 .or. j == child_i) then
                     to_remove(j) = i
                     call msg_debug2 (D_PARTICLES, &
                          "Particles: Will remove duplicate of i", i)
                     call msg_debug2 (D_PARTICLES, &
                          "Particles: j", j)
                  end if
                  cycle OUTER
               end if
            end if
         end do
      end if
   end do OUTER
end subroutine find_duplicates
```

⟨*Particles: remove duplicates: procedures*⟩+≡

```
recursive function get_alive_index (try) result (alive)
  integer :: alive
  integer :: try
  if (map(try) > 0) then
     alive = map(try)
  else
     alive = get_alive_index (to_remove(try))
  end if
end function get_alive_index
```

⟨*Particles: remove duplicates: procedures*⟩+≡

```
subroutine strip_duplicates (particles)
  type(particle_t), dimension(:), allocatable, intent(out) :: particles
  integer :: kept, removed, i, j
  integer, dimension(:), allocatable :: old_children
  logical, dimension(:), allocatable :: parent_set
  call msg_debug (D_PARTICLES, "Particles: Removing duplicates")
  call msg_debug (D_PARTICLES, "Particles: n_removals", n_removals)
  if (debug2_active (D_PARTICLES)) then
     call msg_debug2 (D_PARTICLES, "Particles: Given set before removing:")
     call particle_set%write (summary=.true., compressed=.true.)
  end if
  allocate (particles (particle_set%n_tot - n_removals))
  allocate (map (particle_set%n_tot))
  allocate (parent_set (particle_set%n_tot))
  parent_set = .false.
  map = 0
  j = 0
  do i = 1, particle_set%n_tot
     if (to_remove(i) == 0) then
        j = j + 1
```

```
         map(i) = j
         call particles(j)%init (particle_set%prt(i))
      end if
   end do
   do i = 1, particle_set%n_tot
      if (map(i) /= 0) then
         if (.not. parent_set(map(i))) then
            call particles(map(i))%set_parents &
                 (map (particle_set%prt(i)%get_parents ()))
         end if
         call particles(map(i))%set_children &
              (map (particle_set%prt(i)%get_children ()))
      else
         removed = i
         kept = to_remove(i)
         if (particle_set%prt(removed)%has_children ()) then
            old_children = particle_set%prt(removed)%get_children ()
            do j = 1, size (old_children)
               if (map(old_children(j)) > 0) then
                  call particles(map(old_children(j)))%set_parents &
                       ([get_alive_index (kept)])
                  parent_set(map(old_children(j))) = .true.
                  call particles(get_alive_index (kept))%add_child &
                       (map(old_children(j)))
               end if
            end do
            particles(get_alive_index (kept))%status = PRT_RESONANT
         else
            particles(get_alive_index (kept))%status = PRT_OUTGOING
         end if
      end if
   end do
end subroutine strip_duplicates
```

Given a subevent, reset status codes. If the new status is beam, incoming, or outgoing, we also make sure that the stored p^2 value is equal to the on-shell mass squared.

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: reset_status => particle_set_reset_status
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_reset_status (particle_set, index, status)
  class(particle_set_t), intent(inout) :: particle_set
  integer, dimension(:), intent(in) :: index
  integer, intent(in) :: status
  integer :: i
  if (allocated (particle_set%prt)) then
     do i = 1, size (index)
        call particle_set%prt(index(i))%reset_status (status)
     end do
  end if
  particle_set%n_beam  = &
       count (particle_set%prt%get_status () == PRT_BEAM)
  particle_set%n_in  = &
```

```
        count (particle_set%prt%get_status () == PRT_INCOMING)
    particle_set%n_out = &
        count (particle_set%prt%get_status () == PRT_OUTGOING)
    particle_set%n_vir = particle_set%n_tot &
        - particle_set%n_beam - particle_set%n_in - particle_set%n_out
end subroutine particle_set_reset_status
```

Reduce a particle set to the essential entries. The entries kept are those with status INCOMING, OUTGOING or RESONANT. BEAM is kept if keep_beams is true. Other entries are skipped. The correlated state matrix, if any, is also ignored.

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: reduce => particle_set_reduce
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_reduce (pset_in, pset_out, keep_beams)
  class(particle_set_t), intent(in) :: pset_in
  type(particle_set_t), intent(out) :: pset_out
  logical, intent(in), optional :: keep_beams
  integer, dimension(:), allocatable :: status, map
  integer :: i, j
  logical :: kb
  kb = .false.;  if (present (keep_beams))  kb = keep_beams
  allocate (status (pset_in%n_tot))
  pset_out%factorization_mode = pset_in%factorization_mode
  !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
  do i = 1, pset_in%n_tot
     status(i) = pset_in%prt(i)%get_status ()
  end do
  !!! status = pset_in%prt%get_status ()
  if (kb)  pset_out%n_beam  = count (status == PRT_BEAM)
  pset_out%n_in  = count (status == PRT_INCOMING)
  pset_out%n_vir = count (status == PRT_RESONANT)
  pset_out%n_out = count (status == PRT_OUTGOING)
  pset_out%n_tot = &
       pset_out%n_beam + pset_out%n_in + pset_out%n_vir + pset_out%n_out
  allocate (pset_out%prt (pset_out%n_tot))
  allocate (map (pset_in%n_tot))
  map = 0
  j = 0
  if (kb) call copy_particles (PRT_BEAM)
  call copy_particles (PRT_INCOMING)
  call copy_particles (PRT_RESONANT)
  call copy_particles (PRT_OUTGOING)
  do i = 1, pset_in%n_tot
     if (map(i) == 0)  cycle
     !!! !!! triggers nagfor bug!
     !!!  call particle_set_parents (pset_out%prt(map(i)), &
     !!!       map (particle_set_get_real_parents (pset_in, i)))
     !!!  call particle_set_children (pset_out%prt(map(i)), &
     !!!       map (particle_set_get_real_children (pset_in, i)))
     !!! !!! workaround:
     call pset_out%prt(map(i))%set_parents &
          (pset_in%get_real_parents (i, kb))
     call pset_out%prt(map(i))%set_parents &
```

```
            (map (pset_out%prt(map(i))%parent))
        call pset_out%prt(map(i))%set_children &
            (pset_in%get_real_children (i, kb))
        call pset_out%prt(map(i))%set_children &
            (map (pset_out%prt(map(i))%child))
    end do
  contains
    subroutine copy_particles (stat)
      integer, intent(in) :: stat
      integer :: i
      do i = 1, pset_in%n_tot
         if (status(i) == stat) then
            j = j + 1
            map(i) = j
            call particle_init_particle (pset_out%prt(j), pset_in%prt(i))
         end if
      end do
    end subroutine copy_particles
  end subroutine particle_set_reduce
```

Remove the beam particles and beam remnants from the particle set if the keep beams flag is false. If keep beams is not given, the beam particles and the beam remnants are removed. The correlated state matrix, if any, is also ignored.

⟨*Particles: particle set: TBP*⟩+≡

```
  procedure :: filter_particles => particle_set_filter_particles
```

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_set_filter_particles &
      (pset_in, pset_out, keep_beams, real_parents, keep_virtuals)
    class(particle_set_t), intent(in) :: pset_in
    type(particle_set_t), intent(out) :: pset_out
    logical, intent(in), optional :: keep_beams, real_parents, keep_virtuals
    integer, dimension(:), allocatable :: status, map
    logical, dimension(:), allocatable :: filter
    integer :: i, j
    logical :: kb, rp, kv
    kb = .false.;  if (present (keep_beams))  kb = keep_beams
    rp = .false.; if (present (real_parents)) rp = real_parents
    kv = .true.; if (present (keep_virtuals)) kv = keep_virtuals
    call msg_debug (D_PARTICLES, "filter_particles")
    if (debug2_active (D_PARTICLES)) then
       print *, 'keep_beams =    ', kb
       print *, 'real_parents =    ', rp
       print *, 'keep_virtuals =    ', kv
       print *, '>>> pset_in : '
       call pset_in%write(compressed=.true.)
    end if
    call count_and_allocate()
    map = 0
    j = 0
    filter = .false.
    if (.not. kb) filter = status == PRT_BEAM .or. status == PRT_BEAM_REMNANT
    if (.not. kv) filter = filter .or. status == PRT_VIRTUAL
    call copy_particles ()
```

```
    do i = 1, pset_in%n_tot
       if (map(i) == 0)  cycle
       if (rp) then
          call pset_out%prt(map(i))%set_parents &
               (map (pset_in%get_real_parents (i, kb)))
          call pset_out%prt(map(i))%set_children &
               (map (pset_in%get_real_children (i, kb)))
       else
          call pset_out%prt(map(i))%set_parents &
               (map (pset_in%prt(i)%get_parents ()))
          call pset_out%prt(map(i))%set_children &
               (map (pset_in%prt(i)%get_children ()))
       end if
    end do
    if (debug2_active (D_PARTICLES)) then
       print *, '>>> pset_out : '
       call pset_out%write(compressed=.true.)
    end if
  contains
    ⟨filter particles: procedures⟩
  end subroutine particle_set_filter_particles
```

⟨*filter particles: procedures*⟩≡

```
    subroutine copy_particles ()
      integer :: i
      do i = 1, pset_in%n_tot
         if (.not. filter(i)) then
            j = j + 1
            map(i) = j
            call particle_init_particle (pset_out%prt(j), pset_in%prt(i))
         end if
      end do
    end subroutine copy_particles
```

⟨*filter particles: procedures*⟩+≡

```
  subroutine count_and_allocate
    allocate (status (pset_in%n_tot))
    !!! !!! !!! Workaround for ifort standard-semantics bug
    do i = 1, pset_in%n_tot
       status(i) = particle_get_status (pset_in%prt(i))
    end do
    !!! status = particle_get_status (pset_in%prt)
    if (kb)  pset_out%n_beam  = count (status == PRT_BEAM)
    pset_out%n_in  = count (status == PRT_INCOMING)
    if (kb .and. kv) then
       pset_out%n_vir = count (status == PRT_VIRTUAL) + &
            count (status == PRT_RESONANT) + &
            count (status == PRT_BEAM_REMNANT)
    else if (kb .and. .not. kv) then
       pset_out%n_vir = count (status == PRT_RESONANT) + &
            count (status == PRT_BEAM_REMNANT)
    else if (.not. kb .and. kv) then
       pset_out%n_vir = count (status == PRT_VIRTUAL) + &
```

```
        count (status == PRT_RESONANT)
  else
     pset_out%n_vir = count (status == PRT_RESONANT)
  end if
  pset_out%n_out = count (status == PRT_OUTGOING)
  pset_out%n_tot = &
       pset_out%n_beam + pset_out%n_in + pset_out%n_vir + pset_out%n_out
  allocate (pset_out%prt (pset_out%n_tot))
  allocate (map (pset_in%n_tot))
  allocate (filter (pset_in%n_tot))
end subroutine count_and_allocate
```

Transform a particle set into HEPEVT-compatible form. In this form, for each particle, the parents and the children are contiguous in the particle array. Usually, this requires to clone some particles.

We do not know in advance how many particles the canonical form will have. To be on the safe side, allocate four times the original size.

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: to_hepevt_form => particle_set_to_hepevt_form
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_to_hepevt_form (pset_in, pset_out)
  class(particle_set_t), intent(in) :: pset_in
  type(particle_set_t), intent(out) :: pset_out
  type :: particle_entry_t
     integer :: src = 0
     integer :: status = 0
     integer :: orig = 0
     integer :: copy = 0
  end type particle_entry_t
  type(particle_entry_t), dimension(:), allocatable :: prt
  integer, dimension(:), allocatable :: map1, map2
  integer, dimension(:), allocatable :: parent, child
  integer :: n_tot, n_parents, n_children, i, j, c, n

  n_tot = pset_in%n_tot
  allocate (prt (4 * n_tot))
  allocate (map1(4 * n_tot))
  allocate (map2(4 * n_tot))
  map1 = 0
  map2 = 0
  allocate (child (n_tot))
  allocate (parent (n_tot))
  n = 0
  do i = 1, n_tot
     if (pset_in%prt(i)%get_n_parents () == 0) then
        call append (i)
     end if
  end do
  do i = 1, n_tot
     n_children = pset_in%prt(i)%get_n_children ()
     if (n_children > 0) then
        child(1:n_children) = pset_in%prt(i)%get_children ()
        c = child(1)
```

```
      if (map1(c) == 0) then
         n_parents = pset_in%prt(c)%get_n_parents ()
         if (n_parents > 1) then
            parent(1:n_parents) = pset_in%prt(c)%get_parents ()
            if (i == parent(1) .and. &
                any( [(map1(i)+j-1, j=1,n_parents)] /= &
                       map1(parent(1:n_parents)))) then
               do j = 1, n_parents
                  call append (parent(j))
               end do
            end if
         else if (map1(i) == 0) then
            call append (i)
         end if
         do j = 1, n_children
            call append (child(j))
         end do
      end if
   else if (map1(i) == 0) then
      call append (i)
   end if
end do
do i = n, 1, -1
   if (prt(i)%status /= PRT_OUTGOING) then
      do j = 1, i-1
         if (prt(j)%status == PRT_OUTGOING) then
            call append(prt(j)%src)
         end if
      end do
      exit
   end if
end do
pset_out%n_beam = count (prt(1:n)%status == PRT_BEAM)
pset_out%n_in   = count (prt(1:n)%status == PRT_INCOMING)
pset_out%n_vir  = count (prt(1:n)%status == PRT_RESONANT)
pset_out%n_out  = count (prt(1:n)%status == PRT_OUTGOING)
pset_out%n_tot = n
allocate (pset_out%prt (n))
do i = 1, n
   call particle_init_particle (pset_out%prt(i), pset_in%prt(prt(i)%src))
   call pset_out%prt(i)%reset_status (prt(i)%status)
   if (prt(i)%orig == 0) then
      call pset_out%prt(i)%set_parents &
           (map2 (pset_in%prt(prt(i)%src)%get_parents ()))
   else
      call pset_out%prt(i)%set_parents ([ prt(i)%orig ])
   end if
   if (prt(i)%copy == 0) then
      call pset_out%prt(i)%set_children &
           (map1 (pset_in%prt(prt(i)%src)%get_children ()))
   else
      call pset_out%prt(i)%set_children ([ prt(i)%copy ])
   end if
end do
```

```
contains
  subroutine append (i)
    integer, intent(in) :: i
    n = n + 1
    if (n > size (prt)) &
         call msg_bug ("Particle set transform to HEPEVT: insufficient space")
    prt(n)%src = i
    prt(n)%status = pset_in%prt(i)%get_status ()
    if (map1(i) == 0) then
       map1(i) = n
    else
       prt(map2(i))%status = PRT_VIRTUAL
       prt(map2(i))%copy = n
       prt(n)%orig = map2(i)
    end if
    map2(i) = n
  end subroutine append
end subroutine particle_set_to_hepevt_form
```

This procedure aims at reconstructing the momenta of an interaction, given a particle set. Since the particle orderings

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: fill_interaction => particle_set_fill_interaction
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_fill_interaction &
     (pset, int, n_in, recover_beams, check_match, state_flv)
  class(particle_set_t), intent(in) :: pset
  type(interaction_t), intent(inout) :: int
  integer, intent(in) :: n_in
  logical, intent(in), optional :: recover_beams, check_match
  type(state_flv_content_t), intent(in), optional :: state_flv
  integer, dimension(:), allocatable :: map, pdg
  integer, dimension(:), allocatable :: i_in, i_out, p_in, p_out
  logical, dimension(:), allocatable :: i_set
  integer :: n_out, i, p
  logical :: r_beams, check
  r_beams = .false.;  if (present (recover_beams))  r_beams = recover_beams
  check = .true.;  if (present (check_match))  check = check_match
  if (check) then
     call find_hard_process_in_int  (i_in, i_out)
     call find_hard_process_in_pset (p_in, p_out)
     n_out = size (i_out)
     if (size (i_in) /= n_in)  call err_int_n_in
     if (size (p_in) /= n_in)  call err_pset_n_in
     if (size (p_out) /= n_out)  call err_pset_n_out
     call extract_hard_process_from_pset (pdg)
     call determine_map_for_hard_process (map, state_flv)
     if (.not. r_beams) then
        select case (n_in)
        case (1)
           call recover_parents (p_in(1), map)
        case (2)
           do i = 1, 2
```

```
            call recover_parents (p_in(i), map)
         end do
         do p = 1, 2
            call recover_radiation (p, map)
         end do
      end select
   end if
else
   allocate (map (int%get_n_tot ()))
   map = [(i, i = 1, size (map))]
   r_beams = .false.
end if
allocate (i_set (int%get_n_tot ()), source = .false.)
do p = 1, size (map)
   if (map(p) /= 0) then
      i_set(map(p)) = .true.
      call int%set_momentum &
           (pset%prt(p)%get_momentum (), map(p))
   end if
end do
if (r_beams) then
   do i = 1, n_in
      call reconstruct_beam_and_radiation (i, i_set)
   end do
end if
if (any (.not. i_set))  call err_map
contains
  subroutine find_hard_process_in_pset (p_in, p_out)
    integer, dimension(:), allocatable, intent(out) :: p_in, p_out
    integer, dimension(:), allocatable :: p_status, p_idx
    integer :: n_out_p
    integer :: i
    allocate (p_status (pset%n_tot), p_idx (pset%n_tot))
    !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
    !!! p_status = pset%prt%get_status ()
    do i = 1, pset%n_tot
       p_status(i) = pset%prt(i)%get_status ()
    end do
    p_idx = [(i, i = 1, pset%n_tot)]
    allocate (p_in (n_in))
    p_in = pack (p_idx, p_status == PRT_INCOMING)
    if (size (p_in) == 0)  call err_pset_hard
    i = p_in(1)
    n_out_p = particle_get_n_children (pset%prt(i))
    allocate (p_out (n_out_p))
    p_out = particle_get_children (pset%prt(i))
  end subroutine find_hard_process_in_pset
  subroutine find_hard_process_in_int (i_in, i_out)
    integer, dimension(:), allocatable, intent(out) :: i_in, i_out
    integer :: n_in_i
    integer :: i
    i = int%get_n_tot ()
    n_in_i = interaction_get_n_parents (int, i)
    if (n_in_i /= n_in)  call err_int_n_in
```

```
  allocate (i_in (n_in))
  i_in = interaction_get_parents (int, i)
  i = i_in(1)
  n_out = interaction_get_n_children (int, i)
  allocate (i_out (n_out))
  i_out = interaction_get_children (int, i)
end subroutine find_hard_process_in_int
subroutine extract_hard_process_from_pset (pdg)
  integer, dimension(:), allocatable, intent(out) :: pdg
  integer, dimension(:), allocatable :: pdg_p
  logical, dimension(:), allocatable :: mask_p
  integer :: i
  allocate (pdg_p (pset%n_tot))
  !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
  !!! pdg_p = pset%prt%get_pdg ()
  do i = 1, pset%n_tot
     pdg_p(i) = pset%prt(i)%get_pdg ()
  end do
  allocate (mask_p (pset%n_tot), source = .false.)
  mask_p (p_in) = .true.
  mask_p (p_out) = .true.
  allocate (pdg (n_in + n_out))
  pdg = pack (pdg_p, mask_p)
end subroutine extract_hard_process_from_pset
subroutine determine_map_for_hard_process (map, state_flv)
  integer, dimension(:), allocatable, intent(out) :: map
  type(state_flv_content_t), intent(in), optional :: state_flv
  integer, dimension(:), allocatable :: pdg_i, map_i
  integer :: n_tot
  logical, dimension(:), allocatable :: mask_i, mask_p
  logical :: success
  n_tot = int%get_n_tot ()
  if (present (state_flv)) then
     allocate (mask_i (n_tot), source = .false.)
     mask_i (i_in) = .true.
     mask_i (i_out) = .true.
     allocate (pdg_i (n_tot), map_i (n_tot))
     pdg_i = unpack (pdg, mask_i, 0)
     call state_flv%match (pdg_i, success, map_i)
     allocate (mask_p (pset%n_tot), source = .false.)
     mask_p (p_in) = .true.
     mask_p (p_out) = .true.
     allocate (map (size (mask_p)), &
          source = unpack (pack (map_i, mask_i), mask_p, 0))
     if (.not. success)  call err_mismatch
  else
     allocate (map (n_tot), source = 0)
     map(p_in) = i_in
     map(p_out) = i_out
  end if
end subroutine determine_map_for_hard_process
recursive subroutine recover_parents (p, map)
  integer, intent(in) :: p
  integer, dimension(:), intent(inout) :: map
```

```
  integer :: i, n, n_p, q, k
  integer, dimension(:), allocatable :: i_parents, p_parents
  integer, dimension(1) :: pp
  i = map(p)
  n = interaction_get_n_parents (int, i)
  q = p
  n_p = particle_get_n_parents (pset%prt(q))
  do while (n_p == 1)
     pp = particle_get_parents (pset%prt(q))
     if (pset%prt(pp(1))%get_n_children () > 1)  exit
     q = pp(1)
     n_p = pset%prt(q)%get_n_parents ()
  end do
  if (n_p /= n)  call err_map
  allocate (i_parents (n), p_parents (n))
  i_parents = interaction_get_parents (int, i)
  p_parents = pset%prt(q)%get_parents ()
  do k = 1, n
     q = p_parents(k)
     if (map(q) == 0) then
        map(q) = i_parents(k)
        call recover_parents (q, map)
     end if
  end do
end subroutine recover_parents
recursive subroutine recover_radiation (p, map)
  integer, intent(in) :: p
  integer, dimension(:), intent(inout) :: map
  integer :: i, n, n_p, q, k
  integer, dimension(:), allocatable :: i_children, p_children
  if (particle_get_status (pset%prt(p)) == PRT_INCOMING)  return
  i = map(p)
  n = interaction_get_n_children (int, i)
  n_p = pset%prt(p)%get_n_children ()
  if (n_p /= n)  call err_map
  allocate (i_children (n), p_children (n))
  i_children = interaction_get_children (int, i)
  p_children = pset%prt(p)%get_children ()
  do k = 1, n
     q = p_children(k)
     if (map(q) == 0) then
        i = i_children(k)
        if (interaction_get_n_children (int, i) == 0) then
           map(q) = i
        else
           select case (n)
           case (2)
              select case (k)
              case (1);  map(q) = i_children(2)
              case (2);  map(q) = i_children(1)
              end select
           case (4)
              select case (k)
              case (1);  map(q) = i_children(3)
```

```
            case (2);  map(q) = i_children(4)
            case (3);  map(q) = i_children(1)
            case (4);  map(q) = i_children(2)
            end select
         case default
            call err_radiation
         end select
      end if
   else
      call recover_radiation (q, map)
   end if
  end do
end subroutine recover_radiation
subroutine reconstruct_beam_and_radiation (k, i_set)
  integer, intent(in) :: k
  logical, dimension(:), intent(inout) :: i_set
  integer :: k_src, k_in, k_rad
  type(interaction_t), pointer :: int_src
  integer, dimension(2) :: i_child
  call int%find_source (k, int_src, k_src)
  call int%set_momentum (int_src%get_momentum (k_src), k)
  i_set(k) = .true.
  if (n_in == 2) then
     i_child = interaction_get_children (int, k)
     if (interaction_get_n_children (int, i_child(1)) > 0) then
        k_in = i_child(1);  k_rad = i_child(2)
     else
        k_in = i_child(2);  k_rad = i_child(1)
     end if
     if (.not. i_set(k_in))  call err_beams
     call int%set_momentum &
          (int%get_momentum (k) - int%get_momentum (k_in), k_rad)
     i_set(k_rad) = .true.
  end if
end subroutine reconstruct_beam_and_radiation
subroutine err_pset_hard
  call msg_fatal ("Reading particle set: no particles marked as incoming")
end subroutine err_pset_hard
subroutine err_int_n_in
  integer :: n
  if (allocated (i_in)) then
     n = size (i_in)
  else
     n = 0
  end if
  write (msg_buffer, "(A,I0,A,I0)") &
       "Filling hard process from particle set: expect ", n_in, &
       " incoming particle(s), found ", n
  call msg_bug
end subroutine err_int_n_in
subroutine err_pset_n_in
  write (msg_buffer, "(A,I0,A,I0)") &
       "Reading hard-process particle set: should contain ", n_in, &
       " incoming particle(s), found ", size (p_in)
```

```
      call msg_fatal
    end subroutine err_pset_n_in
    subroutine err_pset_n_out
      write (msg_buffer, "(A,I0,A,I0)") &
           "Reading hard-process particle set: should contain ", n_out, &
           " outgoing particle(s), found ", size (p_out)
      call msg_fatal
    end subroutine err_pset_n_out
    subroutine err_mismatch
      call pset%write ()
      call state_flv%write ()
      call msg_fatal ("Reading particle set: Flavor combination " &
           // "does not match requested process")
    end subroutine err_mismatch
    subroutine err_map
      call pset%write ()
      call int%basic_write ()
      call msg_fatal ("Reading hard-process particle set: " &
           // "Incomplete mapping from particle set to interaction")
    end subroutine err_map
    subroutine err_beams
      call pset%write ()
      call int%basic_write ()
      call msg_fatal ("Reading particle set: Beam structure " &
           // "does not match requested process")
    end subroutine err_beams
    subroutine err_radiation
      call int%basic_write ()
      call msg_bug ("Reading particle set: Interaction " &
           // "contains inconsistent radiation pattern.")
    end subroutine err_radiation
  end subroutine particle_set_fill_interaction
```

This procedure reconstructs an array of vertex indices from the parent-child information in the particle entries, according to the HepMC scheme. For each particle, we determine which vertex it comes from and which vertex it goes to. We return the two arrays and the maximum vertex index.

For each particle in the list, we first check its parents. If for any parent the vertex where it goes to is already known, this vertex index is assigned as the current 'from' vertex. Otherwise, a new index is created, assigned as the current 'from' vertex, and as the 'to' vertex for all parents.

Then, the analogous procedure is done for the children.

Furthermore, we assign to each vertex the vertex position from the parent(s). We check that these vertex positions coincide, and if not return a null vector.

⟨*Particles: particle set: TBP*⟩+≡

```
  procedure :: assign_vertices => particle_set_assign_vertices
```

⟨*Particles: procedures*⟩+≡

```
  subroutine particle_set_assign_vertices &
       (particle_set, v_from, v_to, n_vertices)
    class(particle_set_t), intent(in) :: particle_set
    integer, dimension(:), intent(out) :: v_from, v_to
    integer, intent(out) :: n_vertices
```

```
  integer, dimension(:), allocatable :: parent, child
  integer :: n_parents, n_children, vf, vt
  integer :: i, j, v
  v_from = 0
  v_to = 0
  vf = 0
  vt = 0
  do i = 1, particle_set%n_tot
     n_parents = particle_set%prt(i)%get_n_parents ()
     if (n_parents /= 0) then
        allocate (parent (n_parents))
        parent = particle_set%prt(i)%get_parents ()
        SCAN_PARENTS: do j = 1, size (parent)
           v = v_to(parent(j))
           if (v /= 0) then
              v_from(i) = v;  exit SCAN_PARENTS
           end if
        end do SCAN_PARENTS
        if (v_from(i) == 0) then
           vf = vf + 1;  v_from(i) = vf
           v_to(parent) = vf
        end if
        deallocate (parent)
     end if
     n_children = particle_set%prt(i)%get_n_children ()
     if (n_children /= 0) then
        allocate (child (n_children))
        child = particle_set%prt(i)%get_children ()
        SCAN_CHILDREN: do j = 1, size (child)
           v = v_from(child(j))
           if (v /= 0) then
              v_to(i) = v;  exit SCAN_CHILDREN
           end if
        end do SCAN_CHILDREN
        if (v_to(i) == 0) then
           vt = vt + 1;  v_to(i) = vt
           v_from(child) = vt
        end if
        deallocate (child)
     end if
  end do
  n_vertices = max (vf, vt)
end subroutine particle_set_assign_vertices
```

14.4.4 Expression interface

This converts a `particle_set` object as defined here to a more concise `subevt` object that can be used as the event root of an expression. In particular, the latter lacks virtual particles, spin correlations and parent-child relations.

We keep beam particles, incoming partons, and outgoing partons. Furthermore, we keep radiated particles (a.k.a. beam remnants) if they have no children in the current particle set, and mark them as outgoing particles.

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: to_subevt => particle_set_to_subevt
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_to_subevt (particle_set, subevt)
  class(particle_set_t), intent(in) :: particle_set
  type(subevt_t), intent(out) :: subevt
  integer :: n_tot, n_beam, n_in, n_out, n_rad
  integer :: i, k, n_active
  integer, dimension(2) :: hel
  logical :: keep
  n_tot  = particle_set_get_n_tot      (particle_set)
  n_beam = particle_set_get_n_beam     (particle_set)
  n_in   = particle_set_get_n_in       (particle_set)
  n_out  = particle_set_get_n_out      (particle_set)
  n_rad  = particle_set_get_n_remnants (particle_set)
  call subevt_init (subevt, n_beam + n_rad + n_in + n_out)
  k = 0
  do i = 1, n_tot
     associate (prt => particle_set%prt(i))
       keep = .false.
       select case (particle_get_status (prt))
       case (PRT_BEAM)
          k = k + 1
          call subevt_set_beam (subevt, k, &
               particle_get_pdg (prt), &
               particle_get_momentum (prt), &
               particle_get_p2 (prt))
          keep = .true.
       case (PRT_INCOMING)
          k = k + 1
          call subevt_set_incoming (subevt, k, &
               particle_get_pdg (prt), &
               particle_get_momentum (prt), &
               particle_get_p2 (prt))
          keep = .true.
       case (PRT_OUTGOING)
          k = k + 1
          call subevt_set_outgoing (subevt, k, &
               particle_get_pdg (prt), &
               particle_get_momentum (prt), &
               particle_get_p2 (prt))
          keep = .true.
       case (PRT_BEAM_REMNANT)
          if (particle_get_n_children (prt) == 0) then
             k = k + 1
             call subevt_set_outgoing (subevt, k, &
                  particle_get_pdg (prt), &
                  particle_get_momentum (prt), &
                  particle_get_p2 (prt))
             keep = .true.
          end if
       end select
       if (keep) then
          if (prt%polarization == PRT_DEFINITE_HELICITY) then
```

```
            if (prt%hel%is_diagonal ()) then
               hel = prt%hel%to_pair ()
               call subevt_polarize (subevt, k, hel(1))
            end if
         end if
      end if
   end associate
   n_active = k
end do
call subevt_reset (subevt, n_active)
end subroutine particle_set_to_subevt
```

This replaces the `particle\_set\%prt array` with a given array of particles

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: replace => particle_set_replace
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_replace (particle_set, newprt)
  class(particle_set_t), intent(inout) :: particle_set
  type(particle_t), intent(in), dimension(:), allocatable :: newprt
  if (allocated (particle_set%prt))  deallocate (particle_set%prt)
  allocate (particle_set%prt(size (newprt)))
  particle_set%prt = newprt
  particle_set%n_tot = size (newprt)
  particle_set%n_beam = count (particle_get_status (newprt) == PRT_BEAM)
  particle_set%n_in = count (particle_get_status (newprt) == PRT_INCOMING)
  particle_set%n_out = count (particle_get_status (newprt) == PRT_OUTGOING)
  particle_set%n_vir = particle_set%n_tot &
       - particle_set%n_beam - particle_set%n_in - particle_set%n_out
end subroutine particle_set_replace
```

This routines orders the outgoing particles into clusters of colorless particles and such of particles ordered corresponding to the indices of the color lines. All outgoing particles in the ordered set appear as child of the corresponding outgoing particle in the unordered set, including colored beam remnants. We always start continue via the anti-color line, such that color flows within each Lund string system is always continued from the anticolor of one particle to the identical color index of another particle.

⟨*Particles: particle set: TBP*⟩+≡

```
procedure :: order_color_lines => particle_set_order_color_lines
```

⟨*Particles: procedures*⟩+≡

```
subroutine particle_set_order_color_lines (pset_out, pset_in)
  class(particle_set_t), intent(inout) :: pset_out
  type(particle_set_t), intent(in) :: pset_in
  integer :: i, n, n_col_rem, remnant1, remnant2
  n_col_rem = 0
  do i = 1, pset_in%n_tot
     if (pset_in%prt(i)%get_status () == PRT_BEAM_REMNANT .and. &
          any (pset_in%prt(i)%get_color () /= 0)) then
        if (n == 0) then
           remnant1 = i
        else
```

```
            remnant2 = i
         end if
         n_col_rem = n_col_rem + 1
      end if
   end do
   pset_out%n_beam = pset_in%n_beam
   pset_out%n_in   = pset_in%n_in
   pset_out%n_vir  = pset_in%n_vir + pset_in%n_out + n_col_rem
   pset_out%n_out  = pset_in%n_out
   pset_out%n_tot  = pset_in%n_tot + pset_in%n_out + n_col_rem
   pset_out%correlated_state = pset_in%correlated_state
   pset_out%factorization_mode = pset_in%factorization_mode
   allocate (pset_out%prt (pset_out%n_tot))
   do i = 1, pset_in%n_tot
      call pset_out%prt(i)%init (pset_in%prt(i))
      call pset_out%prt(i)%set_children (pset_in%prt(i)%child)
      call pset_out%prt(i)%set_parents (pset_in%prt(i)%parent)
   end do
   n = pset_in%n_tot
   do i = 1, pset_in%n_tot
      if (pset_out%prt(i)%get_status () == PRT_OUTGOING .and. &
          all (pset_out%prt(i)%get_color () == 0) .and. &
          .not. pset_out%prt(i)%has_children ()) then
         n = n + 1
         call pset_out%prt(n)%init (pset_out%prt(i))
         call pset_out%prt(i)%reset_status (PRT_VIRTUAL)
         call pset_out%prt(i)%add_child (n)
         call pset_out%prt(i)%set_parents ([i])
      end if
   end do
   if (n_col_rem > 0) then
      do i = 1, n_col_rem
      end do
   end if
end subroutine particle_set_order_color_lines
```

Eliminate numerical noise

⟨*Particles: public*⟩+≡

```
public :: pacify
```

⟨*Particles: interfaces*⟩+≡

```
interface pacify
   module procedure pacify_particle
   module procedure pacify_particle_set
end interface pacify
```

⟨*Particles: procedures*⟩+≡

```
subroutine pacify_particle (prt)
  class(particle_t), intent(inout) :: prt
  real(default) :: e
  e = epsilon (1._default) * energy (prt%p)
  call pacify (prt%p, 10 * e)
  call pacify (prt%p2, 1e4 * e)
end subroutine pacify_particle
```

```
  subroutine pacify_particle_set (pset)
    class(particle_set_t), intent(inout) :: pset
    integer :: i
    do i = 1, pset%n_tot
       call pacify (pset%prt(i))
    end do
  end subroutine pacify_particle_set
```

14.4.5 Unit tests

Test module, followed by the corresponding implementation module.

⟨particles_ut.f90⟩≡

```
⟨File header⟩

module particles_ut
  use unit_tests
  use particles_uti

⟨Standard module head⟩

⟨Particles: public test⟩

contains

⟨Particles: test driver⟩

end module particles_ut
```

⟨particles_uti.f90⟩≡

```
⟨File header⟩

module particles_uti

⟨Use kinds⟩
  use io_units
  use numeric_utils
  use constants, only: one, tiny_07
  use lorentz
  use flavors
  use colors
  use helicities
  use quantum_numbers
  use state_matrices
  use interactions
  use evaluators
  use model_data
  use subevents

  use particles

⟨Standard module head⟩
```

⟨*Particles: test declarations*⟩

```
contains
```

⟨*Particles: tests*⟩

⟨*Particles: test auxiliary*⟩

```
end module particles_uti
```

API: driver for the unit tests below.

⟨*Particles: public test*⟩≡

```
public :: particles_test
```

⟨*Particles: test driver*⟩≡

```
subroutine particles_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```

⟨*Particles: execute tests*⟩

```
end subroutine particles_test
```

Check the basic setup of the `particle_set_t` type: Set up a chain of production and decay and factorize the result into particles. The process is $d\bar{d} \to Z \to q\bar{q}$.

⟨*Particles: execute tests*⟩≡

```
call test (particles_1, "particles_1", &
     "check particle_set routines", &
     u, results)
```

⟨*Particles: test declarations*⟩≡

```
public :: particles_1
```

⟨*Particles: tests*⟩≡

```
subroutine particles_1 (u)
  use os_interface
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t), dimension(3) :: flv
  type(color_t), dimension(3) :: col
  type(helicity_t), dimension(3) :: hel
  type(quantum_numbers_t), dimension(3) :: qn
  type(vector4_t), dimension(3) :: p
  type(interaction_t), target :: int1, int2
  type(quantum_numbers_mask_t) :: qn_mask_conn
  type(evaluator_t), target :: eval
  type(interaction_t) :: int
  type(particle_set_t) :: particle_set1, particle_set2
  type(particle_set_t) :: particle_set3, particle_set4
  type(subevt_t) :: subevt
  logical :: ok
  integer :: unit, iostat

  write (u, "(A)")  "* Test output: Particles"
  write (u, "(A)")  "*   Purpose: test particle_set routines"
  write (u, "(A)")
```

```
write (u, "(A)")  "* Reading model file"

call model%init_sm_test ()

write (u, "(A)")
write (u, "(A)")  "* Initializing production process"

call int1%basic_init (2, 0, 1, set_relations=.true.)
call flv%init ([1, -1, 23], model)
call col%init_col_acl ([0, 0, 0], [0, 0, 0])
call hel(3)%init (1, 1)
call qn%init (flv, col, hel)
call int1%add_state (qn, value=(0.25_default, 0._default))
call hel(3)%init (1,-1)
call qn%init (flv, col, hel)
call int1%add_state (qn, value=(0._default, 0.25_default))
call hel(3)%init (-1, 1)
call qn%init (flv, col, hel)
call int1%add_state (qn, value=(0._default,-0.25_default))
call hel(3)%init (-1,-1)
call qn%init (flv, col, hel)
call int1%add_state (qn, value=(0.25_default, 0._default))
call hel(3)%init (0, 0)
call qn%init (flv, col, hel)
call int1%add_state (qn, value=(0.5_default, 0._default))
call int1%freeze ()
p(1) = vector4_moving (45._default, 45._default, 3)
p(2) = vector4_moving (45._default,-45._default, 3)
p(3) = p(1) + p(2)
call int1%set_momenta (p)

write (u, "(A)")
write (u, "(A)")  "* Setup decay process"

call int2%basic_init (1, 0, 2, set_relations=.true.)
call flv%init ([23, 1, -1], model)
call col%init_col_acl ([0, 501, 0], [0, 0, 501])
call hel%init ([1, 1, 1], [1, 1, 1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(1._default, 0._default))
call hel%init ([1, 1, 1], [-1,-1,-1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(0._default, 0.1_default))
call hel%init ([-1,-1,-1], [1, 1, 1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(0._default,-0.1_default))
call hel%init ([-1,-1,-1], [-1,-1,-1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(1._default, 0._default))
call hel%init ([0, 1,-1], [0, 1,-1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(4._default, 0._default))
call hel%init ([0,-1, 1], [0, 1,-1])
call qn%init (flv, col, hel)
```

```
call int2%add_state (qn, value=(2._default, 0._default))
call hel%init ([0, 1,-1], [0,-1, 1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(2._default, 0._default))
call hel%init ([0,-1, 1], [0,-1, 1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(4._default, 0._default))
call flv%init ([23, 2, -2], model)
call hel%init ([0, 1,-1], [0, 1,-1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(0.5_default, 0._default))
call hel%init ([0,-1, 1], [0,-1, 1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(0.5_default, 0._default))
call int2%freeze ()
p(2) = vector4_moving (45._default, 45._default, 2)
p(3) = vector4_moving (45._default,-45._default, 2)
call int2%set_momenta (p)
call int2%set_source_link (1, int1, 3)
call int1%basic_write (u)
call int2%basic_write (u)

write (u, "(A)")
write (u, "(A)")  "* Concatenate production and decay"

call eval%init_product (int1, int2, qn_mask_conn, &
     connections_are_resonant=.true.)
call eval%receive_momenta ()
call eval%evaluate ()
call eval%write (u)

write (u, "(A)")
write (u, "(A)")  "* Factorize as subevent (complete, polarized)"
write (u, "(A)")

int = eval%interaction_t
call particle_set1%init &
     (ok, int, int, FM_FACTOR_HELICITY, &
      [0.2_default, 0.2_default], .false., .true.)
call particle_set1%write (u)

write (u, "(A)")
write (u, "(A)")  "* Factorize as subevent (in/out only, selected helicity)"
write (u, "(A)")

int = eval%interaction_t
call particle_set2%init &
     (ok, int, int, FM_SELECT_HELICITY, &
      [0.9_default, 0.9_default], .false., .false.)
call particle_set2%write (u)
call particle_set2%final ()

write (u, "(A)")
write (u, "(A)")  "* Factorize as subevent (complete, selected helicity)"
```

```
write (u, "(A)")

int = eval%interaction_t
call particle_set2%init &
     (ok, int, int, FM_SELECT_HELICITY, &
      [0.7_default, 0.7_default], .false., .true.)
call particle_set2%write (u)

write (u, "(A)")
write (u, "(A)")  &
     "* Factorize (complete, polarized, correlated); write and read again"
write (u, "(A)")

int = eval%interaction_t
call particle_set3%init &
     (ok, int, int, FM_FACTOR_HELICITY, &
      [0.7_default, 0.7_default], .true., .true.)
call particle_set3%write (u)

unit = free_unit ()
open (unit, action="readwrite", form="unformatted", status="scratch")
call particle_set3%write_raw (unit)
rewind (unit)
call particle_set4%read_raw (unit, iostat=iostat)
call particle_set4%set_model (model)
close (unit)

write (u, "(A)")
write (u, "(A)")  "* Result from reading"
write (u, "(A)")

call particle_set4%write (u)

write (u, "(A)")
write (u, "(A)")  "* Transform to a subevt object"
write (u, "(A)")

call particle_set4%to_subevt (subevt)
call subevt_write (subevt, u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call particle_set1%final ()
call particle_set2%final ()
call particle_set3%final ()
call particle_set4%final ()
call eval%final ()
call int1%final ()
call int2%final ()

call model%final ()

write (u, "(A)")
```

```
    write (u, "(A)")  "* Test output end: particles_1"

  end subroutine particles_1
```

Reconstruct a hard interaction from a particle set.

⟨*Particles: execute tests*⟩+≡

```
  call test (particles_2, "particles_2", &
       "reconstruct hard interaction", &
       u, results)
```

⟨*Particles: test declarations*⟩+≡

```
  public :: particles_2
```

⟨*Particles: tests*⟩+≡

```
  subroutine particles_2 (u)

    integer, intent(in) :: u
    type(interaction_t) :: int
    type(state_flv_content_t) :: state_flv
    type(particle_set_t) :: pset
    type(flavor_t), dimension(:), allocatable :: flv
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    integer :: i, j

    write (u, "(A)")  "* Test output: Particles"
    write (u, "(A)")  "*   Purpose: reconstruct simple interaction"
    write (u, "(A)")

    write (u, "(A)")  "* Set up a 2 -> 3 interaction"
    write (u, "(A)")  "    + incoming partons marked as virtual"
    write (u, "(A)")  "    + no quantum numbers"
    write (u, "(A)")

    call reset_interaction_counter ()
    call int%basic_init (0, 2, 3)
    do i = 1, 2
       do j = 3, 5
          call int%relate (i, j)
       end do
    end do

    allocate (qn (5))
    call int%add_state (qn)
    call int%freeze ()

    call int%basic_write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Manually set up a flavor-content record"
    write (u, "(A)")

    call state_flv%init (1, &
         mask = [.false., .false., .true., .true., .true.])
    call state_flv%set_entry (1, &
         pdg = [11, 12, 3, 4, 5], &
```

```
      map = [1, 2, 3, 4, 5])

  call state_flv%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Manually create a matching particle set"
  write (u, "(A)")

  pset%n_beam = 0
  pset%n_in   = 2
  pset%n_vir  = 0
  pset%n_out  = 3
  pset%n_tot  = 5
  allocate (pset%prt (pset%n_tot))
  do i = 1, 2
     call pset%prt(i)%reset_status (PRT_INCOMING)
     call pset%prt(i)%set_children ([3,4,5])
  end do
  do i = 3, 5
     call pset%prt(i)%reset_status (PRT_OUTGOING)
     call pset%prt(i)%set_parents ([1,2])
  end do
  call pset%prt(1)%set_momentum (vector4_at_rest (1._default))
  call pset%prt(2)%set_momentum (vector4_at_rest (2._default))
  call pset%prt(3)%set_momentum (vector4_at_rest (5._default))
  call pset%prt(4)%set_momentum (vector4_at_rest (4._default))
  call pset%prt(5)%set_momentum (vector4_at_rest (3._default))

  allocate (flv (5))
  call flv%init ([11,12,5,4,3])
  do i = 1, 5
     call pset%prt(i)%set_flavor (flv(i))
  end do

  call pset%write (u)

  write (u, "(A)")
  write (u, "(A)")  "*   Fill interaction from particle set"
  write (u, "(A)")

  call pset%fill_interaction (int, 2, state_flv=state_flv)
  call int%basic_write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call int%final ()
  call pset%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: particles_2"

end subroutine particles_2
```

Reconstruct an interaction with beam structure, e.g., a hadronic interaction, from a particle set.

⟨*Particles: execute tests*⟩+≡

```
call test (particles_3, "particles_3", &
     "reconstruct interaction with beam structure", &
     u, results)
```

⟨*Particles: test declarations*⟩+≡

```
public :: particles_3
```

⟨*Particles: tests*⟩+≡

```
subroutine particles_3 (u)

  integer, intent(in) :: u
  type(interaction_t) :: int
  type(state_flv_content_t) :: state_flv
  type(particle_set_t) :: pset
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  integer :: i, j

  write (u, "(A)")  "* Test output: Particles"
  write (u, "(A)")  "*   Purpose: reconstruct simple interaction"
  write (u, "(A)")

  write (u, "(A)")  "* Set up a 2 -> 2 -> 3 interaction with radiation"
  write (u, "(A)")  "    + no quantum numbers"
  write (u, "(A)")

  call reset_interaction_counter ()
  call int%basic_init (0, 6, 3)
  call int%relate (1, 3)
  call int%relate (1, 4)
  call int%relate (2, 5)
  call int%relate (2, 6)
  do i = 4, 6, 2
     do j = 7, 9
        call int%relate (i, j)
     end do
  end do

  allocate (qn (9))
  call int%add_state (qn)
  call int%freeze ()

  call int%basic_write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Manually set up a flavor-content record"
  write (u, "(A)")

  call state_flv%init (1, &
       mask = [.false., .false., .false., .false., .false., .false., &
       .true., .true., .true.])
  call state_flv%set_entry (1, &
       pdg = [2011, 2012, 91, 11, 92, 12, 3, 4, 5], &
```

```
        map = [1, 2, 3, 4, 5, 6, 7, 8, 9])

    call state_flv%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Manually create a matching particle set"
    write (u, "(A)")

    call create_test_particle_set_1 (pset)

    call pset%write (u)

    write (u, "(A)")
    write (u, "(A)")  "*   Fill interaction from particle set"
    write (u, "(A)")

    call pset%fill_interaction (int, 2, state_flv=state_flv)
    call int%basic_write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call int%final ()
    call pset%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: particles_3"

  end subroutine particles_3
```

⟨*Particles: test auxiliary*⟩≡

```
  subroutine create_test_particle_set_1 (pset)
    type(particle_set_t), intent(out) :: pset
    type(flavor_t), dimension(:), allocatable :: flv
    integer :: i
    pset%n_beam = 2
    pset%n_in   = 2
    pset%n_vir  = 2
    pset%n_out  = 3
    pset%n_tot  = 9

    allocate (pset%prt (pset%n_tot))
    call pset%prt(1)%reset_status (PRT_BEAM)
    call pset%prt(2)%reset_status (PRT_BEAM)
    call pset%prt(3)%reset_status (PRT_INCOMING)
    call pset%prt(4)%reset_status (PRT_INCOMING)
    call pset%prt(5)%reset_status (PRT_BEAM_REMNANT)
    call pset%prt(6)%reset_status (PRT_BEAM_REMNANT)
    call pset%prt(7)%reset_status (PRT_OUTGOING)
    call pset%prt(8)%reset_status (PRT_OUTGOING)
    call pset%prt(9)%reset_status (PRT_OUTGOING)

    call pset%prt(1)%set_children ([3,5])
    call pset%prt(2)%set_children ([4,6])
```

```
    call pset%prt(3)%set_children ([7,8,9])
    call pset%prt(4)%set_children ([7,8,9])

    call pset%prt(3)%set_parents ([1])
    call pset%prt(4)%set_parents ([2])
    call pset%prt(5)%set_parents ([1])
    call pset%prt(6)%set_parents ([2])
    call pset%prt(7)%set_parents ([3,4])
    call pset%prt(8)%set_parents ([3,4])
    call pset%prt(9)%set_parents ([3,4])

    call pset%prt(1)%set_momentum (vector4_at_rest (1._default))
    call pset%prt(2)%set_momentum (vector4_at_rest (2._default))
    call pset%prt(3)%set_momentum (vector4_at_rest (4._default))
    call pset%prt(4)%set_momentum (vector4_at_rest (6._default))
    call pset%prt(5)%set_momentum (vector4_at_rest (3._default))
    call pset%prt(6)%set_momentum (vector4_at_rest (5._default))
    call pset%prt(7)%set_momentum (vector4_at_rest (7._default))
    call pset%prt(8)%set_momentum (vector4_at_rest (8._default))
    call pset%prt(9)%set_momentum (vector4_at_rest (9._default))

    allocate (flv (9))
    call flv%init ([2011, 2012, 11, 12, 91, 92, 3, 4, 5])
    do i = 1, 9
       call pset%prt(i)%set_flavor (flv(i))
    end do
  end subroutine create_test_particle_set_1
```

Reconstruct an interaction with beam structure, e.g., a hadronic interaction, from a particle set that is missing the beam information.

⟨*Particles: execute tests*⟩+≡

```
  call test (particles_4, "particles_4", &
       "reconstruct interaction with missing beams", &
       u, results)
```

⟨*Particles: test declarations*⟩+≡

```
  public :: particles_4
```

⟨*Particles: tests*⟩+≡

```
  subroutine particles_4 (u)

    integer, intent(in) :: u
    type(interaction_t) :: int
    type(interaction_t), target :: int_beams
    type(state_flv_content_t) :: state_flv
    type(particle_set_t) :: pset
    type(flavor_t), dimension(:), allocatable :: flv
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    integer :: i, j

    write (u, "(A)")  "* Test output: Particles"
    write (u, "(A)")  "*   Purpose: reconstruct beams"
    write (u, "(A)")

    call reset_interaction_counter ()
```

```
write (u, "(A)")  "* Set up an interaction that contains beams only"
write (u, "(A)")

call int_beams%basic_init (0, 0, 2)
call int_beams%set_momentum (vector4_at_rest (1._default), 1)
call int_beams%set_momentum (vector4_at_rest (2._default), 2)
allocate (qn (2))
call int_beams%add_state (qn)
call int_beams%freeze ()

call int_beams%basic_write (u)

write (u, "(A)")
write (u, "(A)")  "* Set up a 2 -> 2 -> 3 interaction with radiation"
write (u, "(A)")  "    + no quantum numbers"
write (u, "(A)")

call int%basic_init (0, 6, 3)
call int%relate (1, 3)
call int%relate (1, 4)
call int%relate (2, 5)
call int%relate (2, 6)
do i = 4, 6, 2
   do j = 7, 9
      call int%relate (i, j)
   end do
end do
do i = 1, 2
   call int%set_source_link (i, int_beams, i)
end do

deallocate (qn)
allocate (qn (9))
call int%add_state (qn)
call int%freeze ()

call int%basic_write (u)

write (u, "(A)")
write (u, "(A)")  "* Manually set up a flavor-content record"
write (u, "(A)")

call state_flv%init (1, &
     mask = [.false., .false., .false., .false., .false., .false., &
     .true., .true., .true.])
call state_flv%set_entry (1, &
     pdg = [2011, 2012, 91, 11, 92, 12, 3, 4, 5], &
     map = [1, 2, 3, 4, 5, 6, 7, 8, 9])

call state_flv%write (u)

write (u, "(A)")
write (u, "(A)")  "* Manually create a matching particle set"
```

```
  write (u, "(A)")

  pset%n_beam = 0
  pset%n_in   = 2
  pset%n_vir  = 0
  pset%n_out  = 3
  pset%n_tot  = 5

  allocate (pset%prt (pset%n_tot))
  call pset%prt(1)%reset_status (PRT_INCOMING)
  call pset%prt(2)%reset_status (PRT_INCOMING)
  call pset%prt(3)%reset_status (PRT_OUTGOING)
  call pset%prt(4)%reset_status (PRT_OUTGOING)
  call pset%prt(5)%reset_status (PRT_OUTGOING)

  call pset%prt(1)%set_children ([3,4,5])
  call pset%prt(2)%set_children ([3,4,5])

  call pset%prt(3)%set_parents ([1,2])
  call pset%prt(4)%set_parents ([1,2])
  call pset%prt(5)%set_parents ([1,2])

  call pset%prt(1)%set_momentum (vector4_at_rest (6._default))
  call pset%prt(2)%set_momentum (vector4_at_rest (6._default))
  call pset%prt(3)%set_momentum (vector4_at_rest (3._default))
  call pset%prt(4)%set_momentum (vector4_at_rest (4._default))
  call pset%prt(5)%set_momentum (vector4_at_rest (5._default))

  allocate (flv (5))
  call flv%init ([11, 12, 3, 4, 5])
  do i = 1, 5
     call pset%prt(i)%set_flavor (flv(i))
  end do

  call pset%write (u)

  write (u, "(A)")
  write (u, "(A)")  "*   Fill interaction from particle set"
  write (u, "(A)")

  call pset%fill_interaction (int, 2, state_flv=state_flv, &
       recover_beams = .true.)
  call int%basic_write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call int%final ()
  call pset%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: particles_4"

end subroutine particles_4
```

Reconstruct an interaction with beam structure and cloned particles (radiated particles repeated in the event record, to maintain some canonical ordering).

⟨*Particles: execute tests*⟩+≡

```
call test (particles_5, "particles_5", &
     "reconstruct interaction with beams and duplicate entries", &
     u, results)
```

⟨*Particles: test declarations*⟩+≡

```
public :: particles_5
```

⟨*Particles: tests*⟩+≡

```
subroutine particles_5 (u)

  integer, intent(in) :: u
  type(interaction_t) :: int
  type(state_flv_content_t) :: state_flv
  type(particle_set_t) :: pset
  type(flavor_t), dimension(:), allocatable :: flv
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  integer :: i, j

  write (u, "(A)")  "* Test output: Particles"
  write (u, "(A)")  "*   Purpose: reconstruct event with duplicate entries"
  write (u, "(A)")

  write (u, "(A)")  "* Set up a 2 -> 2 -> 3 interaction with radiation"
  write (u, "(A)")  "    + no quantum numbers"
  write (u, "(A)")

  call reset_interaction_counter ()
  call int%basic_init (0, 6, 3)
  call int%relate (1, 3)
  call int%relate (1, 4)
  call int%relate (2, 5)
  call int%relate (2, 6)
  do i = 4, 6, 2
     do j = 7, 9
        call int%relate (i, j)
     end do
  end do

  allocate (qn (9))
  call int%add_state (qn)
  call int%freeze ()

  call int%basic_write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Manually set up a flavor-content record"
  write (u, "(A)")

  call state_flv%init (1, &
       mask = [.false., .false., .false., .false., .false., .false., &
       .true., .true., .true.])
```

```
call state_flv%set_entry (1, &
     pdg = [2011, 2012, 91, 11, 92, 12, 3, 4, 5], &
     map = [1, 2, 3, 4, 5, 6, 7, 8, 9])

call state_flv%write (u)

write (u, "(A)")
write (u, "(A)")  "* Manually create a matching particle set"
write (u, "(A)")

pset%n_beam = 2
pset%n_in   = 2
pset%n_vir  = 4
pset%n_out  = 5
pset%n_tot  = 13

allocate (pset%prt (pset%n_tot))
call pset%prt(1)%reset_status (PRT_BEAM)
call pset%prt(2)%reset_status (PRT_BEAM)
call pset%prt(3)%reset_status (PRT_VIRTUAL)
call pset%prt(4)%reset_status (PRT_VIRTUAL)
call pset%prt(5)%reset_status (PRT_VIRTUAL)
call pset%prt(6)%reset_status (PRT_VIRTUAL)
call pset%prt(7)%reset_status (PRT_INCOMING)
call pset%prt(8)%reset_status (PRT_INCOMING)
call pset%prt( 9)%reset_status (PRT_OUTGOING)
call pset%prt(10)%reset_status (PRT_OUTGOING)
call pset%prt(11)%reset_status (PRT_OUTGOING)
call pset%prt(12)%reset_status (PRT_OUTGOING)
call pset%prt(13)%reset_status (PRT_OUTGOING)

call pset%prt(1)%set_children ([3,4])
call pset%prt(2)%set_children ([5,6])
call pset%prt(3)%set_children ([ 7])
call pset%prt(4)%set_children ([ 9])
call pset%prt(5)%set_children ([ 8])
call pset%prt(6)%set_children ([10])
call pset%prt(7)%set_children ([11,12,13])
call pset%prt(8)%set_children ([11,12,13])

call pset%prt(3)%set_parents ([1])
call pset%prt(4)%set_parents ([1])
call pset%prt(5)%set_parents ([2])
call pset%prt(6)%set_parents ([2])
call pset%prt( 7)%set_parents ([3])
call pset%prt( 8)%set_parents ([5])
call pset%prt( 9)%set_parents ([4])
call pset%prt(10)%set_parents ([6])
call pset%prt(11)%set_parents ([7,8])
call pset%prt(12)%set_parents ([7,8])
call pset%prt(13)%set_parents ([7,8])

call pset%prt(1)%set_momentum (vector4_at_rest (1._default))
call pset%prt(2)%set_momentum (vector4_at_rest (2._default))
```

```
    call pset%prt(3)%set_momentum (vector4_at_rest (4._default))
    call pset%prt(4)%set_momentum (vector4_at_rest (3._default))
    call pset%prt(5)%set_momentum (vector4_at_rest (6._default))
    call pset%prt(6)%set_momentum (vector4_at_rest (5._default))
    call pset%prt(7)%set_momentum (vector4_at_rest (4._default))
    call pset%prt(8)%set_momentum (vector4_at_rest (6._default))
    call pset%prt( 9)%set_momentum (vector4_at_rest (3._default))
    call pset%prt(10)%set_momentum (vector4_at_rest (5._default))
    call pset%prt(11)%set_momentum (vector4_at_rest (7._default))
    call pset%prt(12)%set_momentum (vector4_at_rest (8._default))
    call pset%prt(13)%set_momentum (vector4_at_rest (9._default))

    allocate (flv (13))
    call flv%init ([2011, 2012, 11, 91, 12, 92, 11, 12, 91, 92, 3, 4, 5])
    do i = 1, 13
       call pset%prt(i)%set_flavor (flv(i))
    end do

    call pset%write (u)

    write (u, "(A)")
    write (u, "(A)")  "*   Fill interaction from particle set"
    write (u, "(A)")

    call pset%fill_interaction (int, 2, state_flv=state_flv)
    call int%basic_write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call int%final ()
    call pset%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: particles_5"

  end subroutine particles_5
```

Reconstruct an interaction with pair spectrum, e.g., beamstrahlung from a particle set.

⟨*Particles: execute tests*⟩+≡

```
  call test (particles_6, "particles_6", &
       "reconstruct interaction with pair spectrum", &
       u, results)
```

⟨*Particles: test declarations*⟩+≡

```
  public :: particles_6
```

⟨*Particles: tests*⟩+≡

```
  subroutine particles_6 (u)

    integer, intent(in) :: u
    type(interaction_t) :: int
    type(state_flv_content_t) :: state_flv
    type(particle_set_t) :: pset
```

```
type(flavor_t), dimension(:), allocatable :: flv
type(quantum_numbers_t), dimension(:), allocatable :: qn
integer :: i, j

write (u, "(A)")  "* Test output: Particles"
write (u, "(A)")  "*   Purpose: reconstruct interaction with pair spectrum"
write (u, "(A)")

write (u, "(A)")  "* Set up a 2 -> 2 -> 3 interaction with radiation"
write (u, "(A)")  "    + no quantum numbers"
write (u, "(A)")

call reset_interaction_counter ()
call int%basic_init (0, 6, 3)
do i = 1, 2
   do j = 3, 6
      call int%relate (i, j)
   end do
end do
do i = 5, 6
   do j = 7, 9
      call int%relate (i, j)
   end do
end do

allocate (qn (9))
call int%add_state (qn)
call int%freeze ()

call int%basic_write (u)

write (u, "(A)")
write (u, "(A)")  "* Manually set up a flavor-content record"
write (u, "(A)")

call state_flv%init (1, &
     mask = [.false., .false., .false., .false., .false., .false., &
     .true., .true., .true.])
call state_flv%set_entry (1, &
     pdg = [1011, 1012, 21, 22, 11, 12, 3, 4, 5], &
     map = [1, 2, 3, 4, 5, 6, 7, 8, 9])

call state_flv%write (u)

write (u, "(A)")
write (u, "(A)")  "* Manually create a matching particle set"
write (u, "(A)")

pset%n_beam = 2
pset%n_in   = 2
pset%n_vir  = 2
pset%n_out  = 3
pset%n_tot  = 9
```

```
allocate (pset%prt (pset%n_tot))
call pset%prt(1)%reset_status (PRT_BEAM)
call pset%prt(2)%reset_status (PRT_BEAM)
call pset%prt(3)%reset_status (PRT_INCOMING)
call pset%prt(4)%reset_status (PRT_INCOMING)
call pset%prt(5)%reset_status (PRT_OUTGOING)
call pset%prt(6)%reset_status (PRT_OUTGOING)
call pset%prt(7)%reset_status (PRT_OUTGOING)
call pset%prt(8)%reset_status (PRT_OUTGOING)
call pset%prt(9)%reset_status (PRT_OUTGOING)

call pset%prt(1)%set_children ([3,4,5,6])
call pset%prt(2)%set_children ([3,4,5,6])
call pset%prt(3)%set_children ([7,8,9])
call pset%prt(4)%set_children ([7,8,9])

call pset%prt(3)%set_parents ([1,2])
call pset%prt(4)%set_parents ([1,2])
call pset%prt(5)%set_parents ([1,2])
call pset%prt(6)%set_parents ([1,2])
call pset%prt(7)%set_parents ([3,4])
call pset%prt(8)%set_parents ([3,4])
call pset%prt(9)%set_parents ([3,4])

call pset%prt(1)%set_momentum (vector4_at_rest (1._default))
call pset%prt(2)%set_momentum (vector4_at_rest (2._default))
call pset%prt(3)%set_momentum (vector4_at_rest (5._default))
call pset%prt(4)%set_momentum (vector4_at_rest (6._default))
call pset%prt(5)%set_momentum (vector4_at_rest (3._default))
call pset%prt(6)%set_momentum (vector4_at_rest (4._default))
call pset%prt(7)%set_momentum (vector4_at_rest (7._default))
call pset%prt(8)%set_momentum (vector4_at_rest (8._default))
call pset%prt(9)%set_momentum (vector4_at_rest (9._default))

allocate (flv (9))
call flv%init ([1011, 1012, 11, 12, 21, 22, 3, 4, 5])
do i = 1, 9
   call pset%prt(i)%set_flavor (flv(i))
end do

call pset%write (u)

write (u, "(A)")
write (u, "(A)")  "*   Fill interaction from particle set"
write (u, "(A)")

call pset%fill_interaction (int, 2, state_flv=state_flv)
call int%basic_write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call int%final ()
call pset%final ()
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: particles_6"

  end subroutine particles_6
```

Reconstruct a hard decay interaction from a shuffled particle set.

⟨*Particles: execute tests*⟩+≡

```
  call test (particles_7, "particles_7", &
       "reconstruct decay interaction with reordering", &
       u, results)
```

⟨*Particles: test declarations*⟩+≡

```
  public :: particles_7
```

⟨*Particles: tests*⟩+≡

```
  subroutine particles_7 (u)

    integer, intent(in) :: u
    type(interaction_t) :: int
    type(state_flv_content_t) :: state_flv
    type(particle_set_t) :: pset
    type(flavor_t), dimension(:), allocatable :: flv
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    integer :: i, j

    write (u, "(A)")  "* Test output: Particles"
    write (u, "(A)")  "*   Purpose: reconstruct decay interaction with reordering"
    write (u, "(A)")

    write (u, "(A)")  "* Set up a 1 -> 3 interaction"
    write (u, "(A)")  "    + no quantum numbers"
    write (u, "(A)")

    call reset_interaction_counter ()
    call int%basic_init (0, 1, 3)
    do j = 2, 4
       call int%relate (1, j)
    end do

    allocate (qn (4))
    call int%add_state (qn)
    call int%freeze ()

    call int%basic_write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Manually set up a flavor-content record"
    write (u, "(A)")  "*   assumed interaction: 6 12 5 -11"
    write (u, "(A)")

    call state_flv%init (1, &
         mask = [.false., .true., .true., .true.])
    call state_flv%set_entry (1, &
         pdg = [6, 5, -11, 12], &
```

```
        map = [1, 4, 2, 3])

    call state_flv%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Manually create a matching particle set"
    write (u, "(A)")

    pset%n_beam = 0
    pset%n_in   = 1
    pset%n_vir  = 0
    pset%n_out  = 3
    pset%n_tot  = 4
    allocate (pset%prt (pset%n_tot))
    do i = 1, 1
       call pset%prt(i)%reset_status (PRT_INCOMING)
       call pset%prt(i)%set_children ([2,3,4])
    end do
    do i = 2, 4
       call pset%prt(i)%reset_status (PRT_OUTGOING)
       call pset%prt(i)%set_parents ([1])
    end do
    call pset%prt(1)%set_momentum (vector4_at_rest (1._default))
    call pset%prt(2)%set_momentum (vector4_at_rest (3._default))
    call pset%prt(3)%set_momentum (vector4_at_rest (2._default))
    call pset%prt(4)%set_momentum (vector4_at_rest (4._default))

    allocate (flv (4))
    call flv%init ([6,5,12,-11])
    do i = 1, 4
       call pset%prt(i)%set_flavor (flv(i))
    end do

    call pset%write (u)

    write (u, "(A)")
    write (u, "(A)")  "*   Fill interaction from particle set"
    write (u, "(A)")

    call pset%fill_interaction (int, 1, state_flv=state_flv)
    call int%basic_write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call int%final ()
    call pset%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: particles_7"

  end subroutine particles_7
```

⟨*Particles: execute tests*⟩+≡

```
    call test (particles_8, "particles_8", &
              "Test functions on particle sets", u, results)
```

⟨*Particles: test declarations*⟩+≡

```
    public :: particles_8
```

⟨*Particles: tests*⟩+≡

```
    subroutine particles_8 (u)
      integer, intent(in) :: u
      type(particle_set_t) :: particle_set
      type(particle_t), dimension(:), allocatable :: particles
      integer, allocatable, dimension(:) :: children, parents
      integer :: n_particles, i
      write (u, "(A)")  "* Test output: particles_8"
      write (u, "(A)")  "*   Purpose: Test functions on particle sets"
      write (u, "(A)")

      call create_test_particle_set_1 (particle_set)
      call particle_set%write (u)
      call assert_equal (u, particle_set%n_tot, 9)
      call assert_equal (u, particle_set%n_beam, 2)
      allocate (children (particle_set%prt(3)%get_n_children ()))
      children = particle_set%prt(3)%get_children()
      call assert_equal (u, particle_set%prt(children(1))%get_pdg (), 3)
      call assert_equal (u, size (particle_set%prt(1)%get_children ()), 2)
      call assert_equal (u, size (particle_set%prt(2)%get_children ()), 2)

      call particle_set%without_hadronic_remnants &
           (particles, n_particles, 3)
      call particle_set%replace (particles)
      write (u, "(A)")
      call particle_set%write (u)

      call assert_equal (u, n_particles, 7)
      call assert_equal (u, size(particles), 10)
      call assert_equal (u, particle_set%n_tot, 10)
      call assert_equal (u, particle_set%n_beam, 2)
      do i = 3, 4
         if (allocated (children))  deallocate (children)
         allocate (children (particle_set%prt(i)%get_n_children ()))
         children = particle_set%prt(i)%get_children()
         call assert_equal (u, particle_set%prt(children(1))%get_pdg (), 3)
         call assert_equal (u, particle_set%prt(children(2))%get_pdg (), 4)
         call assert_equal (u, particle_set%prt(children(3))%get_pdg (), 5)
      end do
      do i = 5, 7
         if (allocated (parents))  deallocate (parents)
         allocate (parents (particle_set%prt(i)%get_n_parents ()))
         parents = particle_set%prt(i)%get_parents()
         call assert_equal (u, particle_set%prt(parents(1))%get_pdg (), 11)
         call assert_equal (u, particle_set%prt(parents(2))%get_pdg (), 12)
      end do
      call assert_equal (u, size (particle_set%prt(1)%get_children ()), &
           1, "get children of 1")
      call assert_equal (u, size (particle_set%prt(2)%get_children ()), &
```

```
        1, "get children of 2")

    call assert_equal (u, particle_set%find_particle &
        (particle_set%prt(1)%get_pdg (), particle_set%prt(1)%p), &
        1, "find 1st particle")
    call assert_equal (u, particle_set%find_particle &
        (particle_set%prt(2)%get_pdg (), particle_set%prt(2)%p * &
         (one + tiny_07), rel_smallness=1.0E-6_default), &
        2, "find 2nd particle fuzzy")

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: particles_8"
  end subroutine particles_8
```

Order color lines into Lund string systems, without colored beam remnants first.

⟨*Particles: execute tests*⟩+≡

```
  call test (particles_9, "particles_9", &
      "order into Lund strings, uncolored beam remnants", &
      u, results)
```

⟨*Particles: test declarations*⟩+≡

```
  public :: particles_9
```

⟨*Particles: tests*⟩+≡

```
  subroutine particles_9 (u)
    integer, intent(in) :: u
    write (u, "(A)")  "* Test output: particles_9"
    write (u, "(A)")  "*   Purpose: Order into Lund strings, "
    write (u, "(A)")  "*              uncolored beam remnants"
    write (u, "(A)")
  end subroutine particles_9
```

Chapter 15

Beams

These modules implement beam configuration and beam structure, the latter in abstract terms.

beam_structures The `beam_structure_t` type is a messenger type that communicates the user settings to the `WHIZARD` core.

beams Beam configuration.

sf_aux Tools for handling structure functions and splitting

sf_mappings Mapping functions, useful for structure function implementation

sf_base The abstract structure-function interaction and structure-function chain types.

These are the implementation modules, the concrete counterparts of `sf_base`:

sf_isr ISR structure function (photon radiation inclusive and resummed in collinear and IR regions).

sf_epa Effective Photon Approximation.

sf_ewa Effective W (and Z) approximation.

sf_escan Energy spectrum that emulates a uniform energy scan.

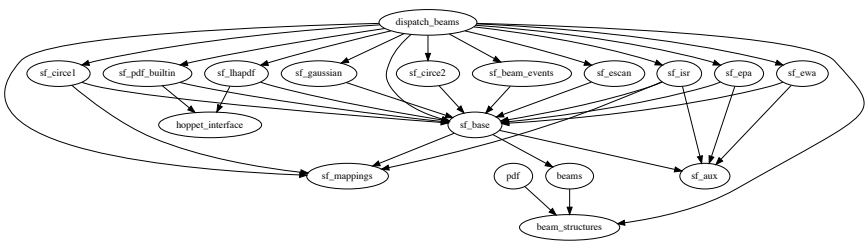

Figure 15.1: Module dependencies in `src/beams`.

sf_gaussian Gaussian beam spread

sf_beam_events Beam-event generator that reads its input from an external file.

sf_circe1 CIRCE1 beam spectra for electrons and photons.

sf_circe2 CIRCE2 beam spectra for electrons and photons.

hoppet_interface Support for b-quark matching, addon to PDF modules.

sf_pdf_builtin Direct support for selected hadron PDFs.

sf_lhapdf LHAPDF library support.

15.1 Beam structure

This module stores the beam structure definition as it is declared in the SINDARIN script. The structure definition is not analyzed, just recorded for later use.

We do not capture any numerical parameters, just names of particles and structure functions.

⟨beam_structures.f90⟩≡

```
⟨File header⟩

module beam_structures

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_defs, only: FMT_19
  use diagnostics
  use lorentz
  use polarizations

⟨Standard module head⟩

⟨Beam structures: public⟩

⟨Beam structures: types⟩

⟨Beam structures: interfaces⟩

contains

⟨Beam structures: procedures⟩

end module beam_structures
```

15.1.1 Beam structure elements

An entry in a beam-structure record consists of a string that denotes a type of structure function.

⟨*Beam structures: types*⟩≡

```
type :: beam_structure_entry_t
   logical :: is_valid = .false.
   type(string_t) :: name
 contains
 ⟨Beam structures: beam structure entry: TBP⟩
end type beam_structure_entry_t
```

Output.

⟨*Beam structures: beam structure entry: TBP*⟩≡

```
procedure :: to_string => beam_structure_entry_to_string
```

⟨*Beam structures: procedures*⟩≡

```
function beam_structure_entry_to_string (object) result (string)
  class(beam_structure_entry_t), intent(in) :: object
```

```
    type(string_t) :: string
    if (object%is_valid) then
       string = object%name
    else
       string = "none"
    end if
  end function beam_structure_entry_to_string
```

A record in the beam-structure sequence denotes either a structure-function entry, a pair of such entries, or a pair spectrum.

⟨*Beam structures: types*⟩+≡

```
  type :: beam_structure_record_t
     type(beam_structure_entry_t), dimension(:), allocatable :: entry
  end type beam_structure_record_t
```

15.1.2 Beam structure type

The beam-structure object contains the beam particle(s) as simple strings. The sequence of records indicates the structure functions by name. No numerical parameters are stored.

⟨*Beam structures: public*⟩≡

```
  public :: beam_structure_t
```

⟨*Beam structures: types*⟩+≡

```
  type :: beam_structure_t
     private
     integer :: n_beam = 0
     type(string_t), dimension(:), allocatable :: prt
     type(beam_structure_record_t), dimension(:), allocatable :: record
     type(smatrix_t), dimension(:), allocatable :: smatrix
     real(default), dimension(:), allocatable :: pol_f
     real(default), dimension(:), allocatable :: p
     real(default), dimension(:), allocatable :: theta
     real(default), dimension(:), allocatable :: phi
   contains
   ⟨Beam structures: beam structure: TBP⟩
  end type beam_structure_t
```

The finalizer deletes all contents explicitly, so we can continue with an empty beam record. (It is not needed for deallocation.) We have distinct finalizers for the independent parts of the beam structure.

⟨*Beam structures: beam structure: TBP*⟩≡

```
  procedure :: final_sf => beam_structure_final_sf
```

⟨*Beam structures: procedures*⟩+≡

```
  subroutine beam_structure_final_sf (object)
    class(beam_structure_t), intent(inout) :: object
    if (allocated (object%prt))  deallocate (object%prt)
    if (allocated (object%record))  deallocate (object%record)
    object%n_beam = 0
  end subroutine beam_structure_final_sf
```

Output. The actual information fits in a single line, therefore we can provide a `to_string` method. The `show` method also lists the current values of relevant global variables.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: write => beam_structure_write
procedure :: to_string => beam_structure_to_string
```

⟨*Beam structures: procedures*⟩+≡

```
subroutine beam_structure_write (object, unit)
  class(beam_structure_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1x,A,A)")  "Beam structure: ", char (object%to_string ())
  if (allocated (object%smatrix)) then
     do i = 1, size (object%smatrix)
        write (u, "(3x,A,I0,A)") "polarization (beam ", i, "):"
        call object%smatrix(i)%write (u, indent=2)
     end do
  end if
  if (allocated (object%pol_f)) then
     write (u, "(3x,A,F10.7,:,',',F10.7)")  "polarization degree =", &
          object%pol_f
  end if
  if (allocated (object%p)) then
     write (u, "(3x,A," // FMT_19 // ",:,',',"  // FMT_19 // &
          ")")  "momentum =", object%p
  end if
  if (allocated (object%theta)) then
     write (u, "(3x,A," // FMT_19 // ",:,',',"  // FMT_19 // &
          ")")  "angle th =", object%theta
  end if
  if (allocated (object%phi)) then
     write (u, "(3x,A," // FMT_19 // ",:,',',"  // FMT_19 // &
          ")")  "angle ph =", object%phi
  end if
end subroutine beam_structure_write

function beam_structure_to_string (object, sf_only) result (string)
  class(beam_structure_t), intent(in) :: object
  logical, intent(in), optional :: sf_only
  type(string_t) :: string
  integer :: i, j
  logical :: with_beams
  with_beams = .true.;  if (present (sf_only))  with_beams = .not. sf_only
  select case (object%n_beam)
  case (1)
     if (with_beams) then
        string = object%prt(1)
     else
        string = ""
     end if
  case (2)
     if (with_beams) then
```

```
         string = object%prt(1) // ", " // object%prt(2)
      else
         string = ""
      end if
      if (allocated (object%record)) then
         if (size (object%record) > 0) then
            if (with_beams)  string = string // " => "
            do i = 1, size (object%record)
               if (i > 1)  string = string // " => "
               do j = 1, size (object%record(i)%entry)
                  if (j > 1)  string = string // ", "
                  string = string // object%record(i)%entry(j)%to_string ()
               end do
            end do
         end if
      end if
   case default
      string = "[any particles]"
   end select
end function beam_structure_to_string
```

Initializer: dimension the beam structure record. Each array element denotes the number of entries for a record within the beam-structure sequence. The number of entries is either one or two, while the number of records is unlimited.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: init_sf => beam_structure_init_sf
```

⟨*Beam structures: procedures*⟩+≡

```
subroutine beam_structure_init_sf (beam_structure, prt, dim_array)
  class(beam_structure_t), intent(inout) :: beam_structure
  type(string_t), dimension(:), intent(in) :: prt
  integer, dimension(:), intent(in), optional :: dim_array
  integer :: i
  call beam_structure%final_sf ()
  beam_structure%n_beam = size (prt)
  allocate (beam_structure%prt (size (prt)))
  beam_structure%prt = prt
  if (present (dim_array)) then
     allocate (beam_structure%record (size (dim_array)))
     do i = 1, size (dim_array)
        allocate (beam_structure%record(i)%entry (dim_array(i)))
     end do
  else
     allocate (beam_structure%record (0))
  end if
end subroutine beam_structure_init_sf
```

Set an entry, specified by record number and entry number.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: set_sf => beam_structure_set_sf
```

⟨*Beam structures: procedures*⟩+≡

```
subroutine beam_structure_set_sf (beam_structure, i, j, name)
  class(beam_structure_t), intent(inout) :: beam_structure
```

```
    integer, intent(in) :: i, j
    type(string_t), intent(in) :: name
    associate (entry => beam_structure%record(i)%entry(j))
      entry%name = name
      entry%is_valid = .true.
    end associate
  end subroutine beam_structure_set_sf
```

Expand the beam-structure object. (i) For a pair spectrum, keep the entry. (ii) For a single-particle structure function written as a single entry, replace this by a record with two entries. (ii) For a record with two nontrivial entries, separate this into two records with one trivial entry each.

To achieve this, we need a function that tells us whether an entry is a spectrum or a structure function. It returns 0 for a trivial entry, 1 for a single-particle structure function, and 2 for a two-particle spectrum.

⟨*Beam structures: interfaces*⟩≡

```
  abstract interface
     function strfun_mode_fun (name) result (n)
       import
       type(string_t), intent(in) :: name
       integer :: n
     end function strfun_mode_fun
  end interface
```

Algorithm: (1) Mark entries as invalid where necessary. (2) Count the number of entries that we will need. (3) Expand and copy entries to a new record array. (4) Replace the old array by the new one.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
  procedure :: expand => beam_structure_expand
```

⟨*Beam structures: procedures*⟩+≡

```
  subroutine beam_structure_expand (beam_structure, strfun_mode)
    class(beam_structure_t), intent(inout) :: beam_structure
    procedure(strfun_mode_fun) :: strfun_mode
    type(beam_structure_record_t), dimension(:), allocatable :: new
    integer :: n_record, i, j
    if (.not. allocated (beam_structure%record))  return
    do i = 1, size (beam_structure%record)
       associate (entry => beam_structure%record(i)%entry)
         do j = 1, size (entry)
            select case (strfun_mode (entry(j)%name))
            case (0);  entry(j)%is_valid = .false.
            end select
         end do
       end associate
    end do
    n_record = 0
    do i = 1, size (beam_structure%record)
       associate (entry => beam_structure%record(i)%entry)
         select case (size (entry))
         case (1)
            if (entry(1)%is_valid) then
               select case (strfun_mode (entry(1)%name))
```

```
            case (1);  n_record = n_record + 2
            case (2);  n_record = n_record + 1
            end select
         end if
      case (2)
         do j = 1, 2
            if (entry(j)%is_valid) then
               select case (strfun_mode (entry(j)%name))
               case (1);  n_record = n_record + 1
               case (2)
                  call beam_structure%write ()
                  call msg_fatal ("Pair spectrum used as &
                       &single-particle structure function")
               end select
            end if
         end do
      end select
   end associate
end do
allocate (new (n_record))
n_record = 0
do i = 1, size (beam_structure%record)
   associate (entry => beam_structure%record(i)%entry)
      select case (size (entry))
      case (1)
         if (entry(1)%is_valid) then
            select case (strfun_mode (entry(1)%name))
            case (1)
               n_record = n_record + 1
               allocate (new(n_record)%entry (2))
               new(n_record)%entry(1) = entry(1)
               n_record = n_record + 1
               allocate (new(n_record)%entry (2))
               new(n_record)%entry(2) = entry(1)
            case (2)
               n_record = n_record + 1
               allocate (new(n_record)%entry (1))
               new(n_record)%entry(1) = entry(1)
            end select
         end if
      case (2)
         do j = 1, 2
            if (entry(j)%is_valid) then
               n_record = n_record + 1
               allocate (new(n_record)%entry (2))
               new(n_record)%entry(j) = entry(j)
            end if
         end do
      end select
   end associate
end do
call move_alloc (from = new, to = beam_structure%record)
end subroutine beam_structure_expand
```

15.1.3 Polarization

To record polarization, we provide an allocatable array of `smatrix` objects, sparse matrices. The polarization structure is independent of the structure-function setup, they are combined only when an actual beam object is constructed.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
procedure :: final_pol => beam_structure_final_pol
procedure :: init_pol => beam_structure_init_pol
```

⟨*Beam structures: procedures*⟩+≡
```
subroutine beam_structure_final_pol (beam_structure)
  class(beam_structure_t), intent(inout) :: beam_structure
  if (allocated (beam_structure%smatrix))  deallocate (beam_structure%smatrix)
  if (allocated (beam_structure%pol_f))  deallocate (beam_structure%pol_f)
end subroutine beam_structure_final_pol

subroutine beam_structure_init_pol (beam_structure, n)
  class(beam_structure_t), intent(inout) :: beam_structure
  integer, intent(in) :: n
  if (allocated (beam_structure%smatrix))  deallocate (beam_structure%smatrix)
  allocate (beam_structure%smatrix (n))
  if (.not. allocated (beam_structure%pol_f)) &
       allocate (beam_structure%pol_f (n), source = 1._default)
end subroutine beam_structure_init_pol
```

Check if polarized beams are used.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
procedure :: has_polarized_beams => beam_structure_has_polarized_beams
```

⟨*Beam structures: procedures*⟩+≡
```
elemental function beam_structure_has_polarized_beams (beam_structure) result (pol)
  logical :: pol
  class(beam_structure_t), intent(in) :: beam_structure
  if (allocated (beam_structure%pol_f)) then
     pol = any (beam_structure%pol_f /= 0)
  else
     pol = .false.
  end if
end function beam_structure_has_polarized_beams
```

Directly copy the spin density matrices.

⟨*Beam structures: beam structure: TBP*⟩+≡
```
procedure :: set_smatrix => beam_structure_set_smatrix
```

⟨*Beam structures: procedures*⟩+≡
```
subroutine beam_structure_set_smatrix (beam_structure, i, smatrix)
  class(beam_structure_t), intent(inout) :: beam_structure
  integer, intent(in) :: i
  type(smatrix_t), intent(in) :: smatrix
  beam_structure%smatrix(i) = smatrix
end subroutine beam_structure_set_smatrix
```

Initialize one of the spin density matrices manually.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: init_smatrix => beam_structure_init_smatrix
```

⟨*Beam structures: procedures*⟩+≡

```
subroutine beam_structure_init_smatrix (beam_structure, i, n_entry)
  class(beam_structure_t), intent(inout) :: beam_structure
  integer, intent(in) :: i
  integer, intent(in) :: n_entry
  call beam_structure%smatrix(i)%init (2, n_entry)
end subroutine beam_structure_init_smatrix
```

Set a polarization entry.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: set_sentry => beam_structure_set_sentry
```

⟨*Beam structures: procedures*⟩+≡

```
subroutine beam_structure_set_sentry &
     (beam_structure, i, i_entry, index, value)
  class(beam_structure_t), intent(inout) :: beam_structure
  integer, intent(in) :: i
  integer, intent(in) :: i_entry
  integer, dimension(:), intent(in) :: index
  complex(default), intent(in) :: value
  call beam_structure%smatrix(i)%set_entry (i_entry, index, value)
end subroutine beam_structure_set_sentry
```

Set the array of polarization fractions.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: set_pol_f => beam_structure_set_pol_f
```

⟨*Beam structures: procedures*⟩+≡

```
subroutine beam_structure_set_pol_f (beam_structure, f)
  class(beam_structure_t), intent(inout) :: beam_structure
  real(default), dimension(:), intent(in) :: f
  if (allocated (beam_structure%pol_f))  deallocate (beam_structure%pol_f)
  allocate (beam_structure%pol_f (size (f)), source = f)
end subroutine beam_structure_set_pol_f
```

15.1.4 Beam momenta

By default, beam momenta are deduced from the `sqrts` value or from the mass of the decaying particle, assuming a c.m. setup. Here we set them explicitly.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: final_mom => beam_structure_final_mom
```

⟨*Beam structures: procedures*⟩+≡

```
subroutine beam_structure_final_mom (beam_structure)
  class(beam_structure_t), intent(inout) :: beam_structure
  if (allocated (beam_structure%p))  deallocate (beam_structure%p)
  if (allocated (beam_structure%theta))  deallocate (beam_structure%theta)
  if (allocated (beam_structure%phi))  deallocate (beam_structure%phi)
end subroutine beam_structure_final_mom
```

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: set_momentum => beam_structure_set_momentum
procedure :: set_theta => beam_structure_set_theta
procedure :: set_phi => beam_structure_set_phi
```

⟨*Beam structures: procedures*⟩+≡

```
subroutine beam_structure_set_momentum (beam_structure, p)
  class(beam_structure_t), intent(inout) :: beam_structure
  real(default), dimension(:), intent(in) :: p
  if (allocated (beam_structure%p))  deallocate (beam_structure%p)
  allocate (beam_structure%p (size (p)), source = p)
end subroutine beam_structure_set_momentum

subroutine beam_structure_set_theta (beam_structure, theta)
  class(beam_structure_t), intent(inout) :: beam_structure
  real(default), dimension(:), intent(in) :: theta
  if (allocated (beam_structure%theta))  deallocate (beam_structure%theta)
  allocate (beam_structure%theta (size (theta)), source = theta)
end subroutine beam_structure_set_theta

subroutine beam_structure_set_phi (beam_structure, phi)
  class(beam_structure_t), intent(inout) :: beam_structure
  real(default), dimension(:), intent(in) :: phi
  if (allocated (beam_structure%phi))  deallocate (beam_structure%phi)
  allocate (beam_structure%phi (size (phi)), source = phi)
end subroutine beam_structure_set_phi
```

15.1.5 Get contents

Look at the incoming particles. We may also have the case that beam particles are not specified, but polarization.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: is_set => beam_structure_is_set
procedure :: get_n_beam => beam_structure_get_n_beam
procedure :: get_prt => beam_structure_get_prt
```

⟨*Beam structures: procedures*⟩+≡

```
function beam_structure_is_set (beam_structure) result (flag)
  class(beam_structure_t), intent(in) :: beam_structure
  logical :: flag
  flag = beam_structure%n_beam > 0 .or. beam_structure%asymmetric ()
end function beam_structure_is_set

function beam_structure_get_n_beam (beam_structure) result (n)
  class(beam_structure_t), intent(in) :: beam_structure
  integer :: n
  n = beam_structure%n_beam
end function beam_structure_get_n_beam

function beam_structure_get_prt (beam_structure) result (prt)
  class(beam_structure_t), intent(in) :: beam_structure
  type(string_t), dimension(:), allocatable :: prt
  allocate (prt (size (beam_structure%prt)))
```

```
    prt = beam_structure%prt
  end function beam_structure_get_prt
```

Return the number of records.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
  procedure :: get_n_record => beam_structure_get_n_record
```

⟨*Beam structures: procedures*⟩+≡

```
  function beam_structure_get_n_record (beam_structure) result (n)
    class(beam_structure_t), intent(in) :: beam_structure
    integer :: n
    if (allocated (beam_structure%record)) then
       n = size (beam_structure%record)
    else
       n = 0
    end if
  end function beam_structure_get_n_record
```

Return an array consisting of the beam indices affected by the valid entries within a record. After expansion, there should be exactly one valid entry per record.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
  procedure :: get_i_entry => beam_structure_get_i_entry
```

⟨*Beam structures: procedures*⟩+≡

```
  function beam_structure_get_i_entry (beam_structure, i) result (i_entry)
    class(beam_structure_t), intent(in) :: beam_structure
    integer, intent(in) :: i
    integer, dimension(:), allocatable :: i_entry
    associate (record => beam_structure%record(i))
      select case (size (record%entry))
      case (1)
         if (record%entry(1)%is_valid) then
            allocate (i_entry (2), source = [1, 2])
         else
            allocate (i_entry (0))
         end if
      case (2)
         if (all (record%entry%is_valid)) then
            allocate (i_entry (2), source = [1, 2])
         else if (record%entry(1)%is_valid) then
            allocate (i_entry (1), source = [1])
         else if (record%entry(2)%is_valid) then
            allocate (i_entry (1), source = [2])
         else
            allocate (i_entry (0))
         end if
      end select
    end associate
  end function beam_structure_get_i_entry
```

Return the name of the first valid entry within a record. After expansion, there should be exactly one valid entry per record.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: get_name => beam_structure_get_name
```

⟨*Beam structures: procedures*⟩+≡

```
function beam_structure_get_name (beam_structure, i) result (name)
  class(beam_structure_t), intent(in) :: beam_structure
  integer, intent(in) :: i
  type(string_t) :: name
  associate (record => beam_structure%record(i))
    if (record%entry(1)%is_valid) then
       name = record%entry(1)%name
    else if (size (record%entry) == 2) then
       name = record%entry(2)%name
    end if
  end associate
end function beam_structure_get_name
```

Return true if the beam structure contains a particular structure function identifier (such as `lhapdf`, `isr`, etc.)

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: contains => beam_structure_contains
```

⟨*Beam structures: procedures*⟩+≡

```
function beam_structure_contains (beam_structure, name) result (flag)
  class(beam_structure_t), intent(in) :: beam_structure
  character(*), intent(in) :: name
  logical :: flag
  integer :: i, j
  flag = .false.
  if (allocated (beam_structure%record)) then
     do i = 1, size (beam_structure%record)
        do j = 1, size (beam_structure%record(i)%entry)
           flag = beam_structure%record(i)%entry(j)%name == name
           if (flag)  return
        end do
     end do
  end if
end function beam_structure_contains
```

Return polarization data.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: polarized => beam_structure_polarized
procedure :: get_smatrix => beam_structure_get_smatrix
procedure :: get_pol_f => beam_structure_get_pol_f
procedure :: asymmetric => beam_structure_asymmetric
```

⟨*Beam structures: procedures*⟩+≡

```
function beam_structure_polarized (beam_structure) result (flag)
  class(beam_structure_t), intent(in) :: beam_structure
  logical :: flag
  flag = allocated (beam_structure%smatrix)
end function beam_structure_polarized

function beam_structure_get_smatrix (beam_structure) result (smatrix)
  class(beam_structure_t), intent(in) :: beam_structure
```

```
  type(smatrix_t), dimension(:), allocatable :: smatrix
  allocate (smatrix (size (beam_structure%smatrix)), &
       source = beam_structure%smatrix)
end function beam_structure_get_smatrix

function beam_structure_get_pol_f (beam_structure) result (pol_f)
  class(beam_structure_t), intent(in) :: beam_structure
  real(default), dimension(:), allocatable :: pol_f
  allocate (pol_f (size (beam_structure%pol_f)), &
       source = beam_structure%pol_f)
end function beam_structure_get_pol_f

function beam_structure_asymmetric (beam_structure) result (flag)
  class(beam_structure_t), intent(in) :: beam_structure
  logical :: flag
  flag = allocated (beam_structure%p) &
       .or. allocated (beam_structure%theta) &
       .or. allocated (beam_structure%phi)
end function beam_structure_asymmetric
```

Return the beam momenta (the space part, i.e., three-momenta). This is meaningful only if momenta and, optionally, angles have been set.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: get_momenta => beam_structure_get_momenta
```

⟨*Beam structures: procedures*⟩+≡

```
function beam_structure_get_momenta (beam_structure) result (p)
  class(beam_structure_t), intent(in) :: beam_structure
  type(vector3_t), dimension(:), allocatable :: p
  real(default), dimension(:), allocatable :: theta, phi
  integer :: n, i
  if (allocated (beam_structure%p)) then
     n = size (beam_structure%p)
     if (allocated (beam_structure%theta)) then
        if (size (beam_structure%theta) == n) then
           allocate (theta (n), source = beam_structure%theta)
        else
           call msg_fatal ("Beam structure: mismatch in momentum vs. &
                &angle theta specification")
        end if
     else
        allocate (theta (n), source = 0._default)
     end if
     if (allocated (beam_structure%phi)) then
        if (size (beam_structure%phi) == n) then
           allocate (phi (n), source = beam_structure%phi)
        else
           call msg_fatal ("Beam structure: mismatch in momentum vs. &
                &angle phi specification")
        end if
     else
        allocate (phi (n), source = 0._default)
     end if
     allocate (p (n))
```

```
      do i = 1, n
         p(i) = beam_structure%p(i) * vector3_moving ([ &
              sin (theta(i)) * cos (phi(i)), &
              sin (theta(i)) * sin (phi(i)), &
              cos (theta(i))])
      end do
      if (n == 2)  p(2) = - p(2)
   else
      call msg_fatal ("Beam structure: angle theta/phi specified but &
           &momentum/a p undefined")
   end if
end function beam_structure_get_momenta
```

Check for a complete beam structure. The `applies` flag tells if the beam structure should actually be used for a process with the given `n_in` number of incoming particles.

It set if the beam structure matches the process as either decay or scattering. It is unset if beam structure references a scattering setup but the process is a decay. It is also unset if the beam structure itself is empty.

If the beam structure cannot be used, terminate with fatal error.

⟨*Beam structures: beam structure: TBP*⟩+≡

```
procedure :: check_against_n_in => beam_structure_check_against_n_in
```

⟨*Beam structures: procedures*⟩+≡

```
subroutine beam_structure_check_against_n_in (beam_structure, n_in, applies)
   class(beam_structure_t), intent(in) :: beam_structure
   integer, intent(in) :: n_in
   logical, intent(out) :: applies
   if (beam_structure%is_set ()) then
      if (n_in == beam_structure%get_n_beam ()) then
         applies = .true.
      else if (beam_structure%get_n_beam () == 0) then
         call msg_fatal &
              ("Asymmetric beams: missing beam particle specification")
         applies = .false.
      else
         call msg_fatal &
              ("Mismatch of process and beam setup (scattering/decay)")
         applies = .false.
      end if
   else
      applies = .false.
   end if
end subroutine beam_structure_check_against_n_in
```

15.1.6 Unit Tests

Test module, followed by the corresponding implementation module.

⟨`beam_structures_ut.f90`⟩≡

⟨*File header*⟩

```
module beam_structures_ut
  use unit_tests
  use beam_structures_uti

⟨Standard module head⟩

⟨Beam structures: public test⟩

contains

⟨Beam structures: test driver⟩

end module beam_structures_ut
```

⟨beam_structures_uti.f90⟩≡

```
⟨File header⟩

module beam_structures_uti

⟨Use kinds⟩
⟨Use strings⟩

  use beam_structures

⟨Standard module head⟩

⟨Beam structures: test declarations⟩

contains

⟨Beam structures: tests⟩

⟨Beam structures: test auxiliary⟩

end module beam_structures_uti
```

API: driver for the unit tests below.

⟨*Beam structures: public test*⟩≡

```
  public :: beam_structures_test
```

⟨*Beam structures: test driver*⟩≡

```
  subroutine beam_structures_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨Beam structures: execute tests⟩
end subroutine beam_structures_test
```

Empty structure

⟨*Beam structures: execute tests*⟩≡

```
  call test (beam_structures_1, "beam_structures_1", &
       "empty beam structure record", &
       u, results)
```

⟨*Beam structures: test declarations*⟩≡

```
public :: beam_structures_1
```

⟨*Beam structures: tests*⟩≡

```
subroutine beam_structures_1 (u)
  integer, intent(in) :: u
  type(beam_structure_t) :: beam_structure

  write (u, "(A)")  "* Test output: beam_structures_1"
  write (u, "(A)")  "*   Purpose: display empty beam structure record"
  write (u, "(A)")

  call beam_structure%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: beam_structures_1"

end subroutine beam_structures_1
```

Nontrivial configurations

⟨*Beam structures: execute tests*⟩+≡

```
call test (beam_structures_2, "beam_structures_2", &
     "beam structure records", &
     u, results)
```

⟨*Beam structures: test declarations*⟩+≡

```
public :: beam_structures_2
```

⟨*Beam structures: tests*⟩+≡

```
subroutine beam_structures_2 (u)
  integer, intent(in) :: u
  type(beam_structure_t) :: beam_structure
  integer, dimension(0) :: empty_array
  type(string_t) :: s

  write (u, "(A)")  "* Test output: beam_structures_2"
  write (u, "(A)")  "*   Purpose: setup beam structure records"
  write (u, "(A)")

  s = "s"

  call beam_structure%init_sf ([s], empty_array)
  call beam_structure%write (u)

  write (u, "(A)")

  call beam_structure%init_sf ([s, s], [1])
  call beam_structure%set_sf (1, 1, var_str ("a"))
  call beam_structure%write (u)

  write (u, "(A)")

  call beam_structure%init_sf ([s, s], [2])
  call beam_structure%set_sf (1, 1, var_str ("a"))
```

```
    call beam_structure%set_sf (1, 2, var_str ("b"))
    call beam_structure%write (u)

    write (u, "(A)")

    call beam_structure%init_sf ([s, s], [2, 1])
    call beam_structure%set_sf (1, 1, var_str ("a"))
    call beam_structure%set_sf (1, 2, var_str ("b"))
    call beam_structure%set_sf (2, 1, var_str ("c"))
    call beam_structure%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: beam_structures_2"

  end subroutine beam_structures_2
```

Expansion

Provide a function that tells, for the dummy structure function names used here, whether they are considered a two-particle spectrum or a single-particle structure function:

⟨*Beam structures: test auxiliary*⟩≡

```
  function test_strfun_mode (name) result (n)
    type(string_t), intent(in) :: name
    integer :: n
    select case (char (name))
    case ("a");  n = 2
    case ("b");  n = 1
    case default;  n = 0
    end select
  end function test_strfun_mode
```

⟨*Beam structures: execute tests*⟩+≡

```
  call test (beam_structures_3, "beam_structures_3", &
       "beam structure expansion", &
       u, results)
```

⟨*Beam structures: test declarations*⟩+≡

```
  public :: beam_structures_3
```

⟨*Beam structures: tests*⟩+≡

```
  subroutine beam_structures_3 (u)
    integer, intent(in) :: u
    type(beam_structure_t) :: beam_structure
    type(string_t) :: s

    write (u, "(A)")  "* Test output: beam_structures_3"
    write (u, "(A)")  "*   Purpose: expand beam structure records"
    write (u, "(A)")

    s = "s"

    write (u, "(A)")  "* Pair spectrum (keep as-is)"
```

```
write (u, "(A)")

call beam_structure%init_sf ([s, s], [1])
call beam_structure%set_sf (1, 1, var_str ("a"))
call beam_structure%write (u)

write (u, "(A)")

call beam_structure%expand (test_strfun_mode)
call beam_structure%write (u)

write (u, "(A)")
write (u, "(A)")  "* Structure function pair (expand)"
write (u, "(A)")

call beam_structure%init_sf ([s, s], [2])
call beam_structure%set_sf (1, 1, var_str ("b"))
call beam_structure%set_sf (1, 2, var_str ("b"))
call beam_structure%write (u)

write (u, "(A)")

call beam_structure%expand (test_strfun_mode)
call beam_structure%write (u)

write (u, "(A)")
write (u, "(A)")  "* Structure function (separate and expand)"
write (u, "(A)")

call beam_structure%init_sf ([s, s], [1])
call beam_structure%set_sf (1, 1, var_str ("b"))
call beam_structure%write (u)

write (u, "(A)")

call beam_structure%expand (test_strfun_mode)
call beam_structure%write (u)

write (u, "(A)")
write (u, "(A)")  "* Combination"
write (u, "(A)")

call beam_structure%init_sf ([s, s], [1, 1])
call beam_structure%set_sf (1, 1, var_str ("a"))
call beam_structure%set_sf (2, 1, var_str ("b"))
call beam_structure%write (u)

write (u, "(A)")

call beam_structure%expand (test_strfun_mode)
call beam_structure%write (u)

write (u, "(A)")
write (u, "(A)")  "* Test output end: beam_structures_3"
```

```
end subroutine beam_structures_3
```

Public methods

Check the methods that can be called to get the beam-structure contents.

⟨*Beam structures: execute tests*⟩+≡

```
call test (beam_structures_4, "beam_structures_4", &
     "beam structure contents", &
     u, results)
```

⟨*Beam structures: test declarations*⟩+≡

```
public :: beam_structures_4
```

⟨*Beam structures: tests*⟩+≡

```
subroutine beam_structures_4 (u)
  integer, intent(in) :: u
  type(beam_structure_t) :: beam_structure
  type(string_t) :: s
  type(string_t), dimension(2) :: prt
  integer :: i

  write (u, "(A)")  "* Test output: beam_structures_4"
  write (u, "(A)")  "*   Purpose: check the API"
  write (u, "(A)")

  s = "s"

  write (u, "(A)")  "* Structure-function combination"
  write (u, "(A)")

  call beam_structure%init_sf ([s, s], [1, 2, 2])
  call beam_structure%set_sf (1, 1, var_str ("a"))
  call beam_structure%set_sf (2, 1, var_str ("b"))
  call beam_structure%set_sf (3, 2, var_str ("c"))
  call beam_structure%write (u)

  write (u, *)
  write (u, "(1x,A,I0)")  "n_beam = ", beam_structure%get_n_beam ()
  prt = beam_structure%get_prt ()
  write (u, "(1x,A,2(1x,A))")  "prt =", char (prt(1)), char (prt(2))

  write (u, *)
  write (u, "(1x,A,I0)")  "n_record = ", beam_structure%get_n_record ()

  do i = 1, 3
     write (u, "(A)")
     write (u, "(1x,A,I0,A,A)")  "name(", i, ") = ", &
          char (beam_structure%get_name (i))
     write (u, "(1x,A,I0,A,2(1x,I0))")  "i_entry(", i, ") =", &
          beam_structure%get_i_entry (i)
  end do

  write (u, "(A)")
```

```
    write (u, "(A)")  "* Test output end: beam_structures_4"

  end subroutine beam_structures_4
```

Polarization

The polarization properties are independent from the structure-function setup.

⟨*Beam structures: execute tests*⟩+≡

```
  call test (beam_structures_5, "beam_structures_5", &
       "polarization", &
       u, results)
```

⟨*Beam structures: test declarations*⟩+≡

```
  public :: beam_structures_5
```

⟨*Beam structures: tests*⟩+≡

```
  subroutine beam_structures_5 (u)
    integer, intent(in) :: u
    type(beam_structure_t) :: beam_structure
    integer, dimension(0) :: empty_array
    type(string_t) :: s

    write (u, "(A)")  "* Test output: beam_structures_5"
    write (u, "(A)")  "*   Purpose: setup polarization in beam structure records"
    write (u, "(A)")

    s = "s"

    call beam_structure%init_sf ([s], empty_array)
    call beam_structure%init_pol (1)
    call beam_structure%init_smatrix (1, 1)
    call beam_structure%set_sentry (1, 1, [0,0], (1._default, 0._default))
    call beam_structure%set_pol_f ([0.5_default])
    call beam_structure%write (u)


    write (u, "(A)")
    call beam_structure%final_sf ()
    call beam_structure%final_pol ()

    call beam_structure%init_sf ([s, s], [1])
    call beam_structure%set_sf (1, 1, var_str ("a"))
    call beam_structure%init_pol (2)
    call beam_structure%init_smatrix (1, 2)
    call beam_structure%set_sentry (1, 1, [-1,1], (0.5_default,-0.5_default))
    call beam_structure%set_sentry (1, 2, [ 1,1], (1._default, 0._default))
    call beam_structure%init_smatrix (2, 0)
    call beam_structure%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: beam_structures_5"

  end subroutine beam_structures_5
```

Momenta

The momenta are independent from the structure-function setup.

⟨*Beam structures: execute tests*⟩+≡
```
    call test (beam_structures_6, "beam_structures_6", &
         "momenta", &
         u, results)
```

⟨*Beam structures: test declarations*⟩+≡
```
    public :: beam_structures_6
```

⟨*Beam structures: tests*⟩+≡
```
    subroutine beam_structures_6 (u)
      integer, intent(in) :: u
      type(beam_structure_t) :: beam_structure
      integer, dimension(0) :: empty_array
      type(string_t) :: s

      write (u, "(A)")  "* Test output: beam_structures_6"
      write (u, "(A)")  "*   Purpose: setup momenta in beam structure records"
      write (u, "(A)")

      s = "s"

      call beam_structure%init_sf ([s], empty_array)
      call beam_structure%set_momentum ([500._default])
      call beam_structure%write (u)


      write (u, "(A)")
      call beam_structure%final_sf ()
      call beam_structure%final_mom ()

      call beam_structure%init_sf ([s, s], [1])
      call beam_structure%set_momentum ([500._default, 700._default])
      call beam_structure%set_theta ([0._default, 0.1_default])
      call beam_structure%set_phi ([0._default, 1.51_default])
      call beam_structure%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: beam_structures_6"

    end subroutine beam_structures_6
```

15.2 Beams for collisions and decays

⟨beams.f90⟩≡
```
  ⟨File header⟩

  module beams

  ⟨Use kinds⟩
  ⟨Use strings⟩
```

```
  use io_units
  use format_defs, only: FMT_19
  use numeric_utils
  use diagnostics
  use md5
  use lorentz
  use model_data
  use flavors
  use quantum_numbers
  use state_matrices
  use interactions
  use polarizations
  use beam_structures
```

⟨*Standard module head*⟩

⟨*Beams: public*⟩

⟨*Beams: types*⟩

⟨*Beams: interfaces*⟩

```
contains
```

⟨*Beams: procedures*⟩

```
end module beams
```

15.2.1 Beam data

The beam data type contains beam data for one or two beams, depending on whether we are dealing with beam collisions or particle decay. In addition, it holds the c.m. energy `sqrts`, the Lorentz transformation `L` that transforms the c.m. system into the lab system, and the pair of c.m. momenta.

⟨*Beams: public*⟩≡

```
  public :: beam_data_t
```

⟨*Beams: types*⟩≡

```
  type :: beam_data_t
     logical :: initialized = .false.
     integer :: n = 0
     type(flavor_t), dimension(:), allocatable :: flv
     real(default), dimension(:), allocatable :: mass
     type(pmatrix_t), dimension(:), allocatable :: pmatrix
     logical :: lab_is_cm_frame = .true.
     type(vector4_t), dimension(:), allocatable :: p_cm
     type(vector4_t), dimension(:), allocatable :: p
     type(lorentz_transformation_t), allocatable  :: L_cm_to_lab
     real(default) :: sqrts = 0
     character(32) :: md5sum = ""
   contains
   ⟨Beams: beam data: TBP⟩
  end type beam_data_t
```

Generic initializer. This is called by the specific initializers below. Initialize either for decay or for collision.

⟨*Beams: procedures*⟩≡

```
subroutine beam_data_init (beam_data, n)
  type(beam_data_t), intent(out) :: beam_data
  integer, intent(in) :: n
  beam_data%n = n
  allocate (beam_data%flv (n))
  allocate (beam_data%mass (n))
  allocate (beam_data%pmatrix (n))
  allocate (beam_data%p_cm (n))
  allocate (beam_data%p (n))
  beam_data%initialized = .true.
end subroutine beam_data_init
```

Finalizer: needed for the polarization components of the beams.

⟨*Beams: beam data: TBP*⟩≡

```
procedure :: final => beam_data_final
```

⟨*Beams: procedures*⟩+≡

```
subroutine beam_data_final (beam_data)
  class(beam_data_t), intent(inout) :: beam_data
  beam_data%initialized = .false.
end subroutine beam_data_final
```

The verbose (default) version is for debugging. The short version is for screen output in the UI.

⟨*Beams: beam data: TBP*⟩+≡

```
procedure :: write => beam_data_write
```

⟨*Beams: procedures*⟩+≡

```
subroutine beam_data_write (beam_data, unit, verbose, write_md5sum)
  class(beam_data_t), intent(in) :: beam_data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose, write_md5sum
  integer :: prt_name_len
  logical :: verb, write_md5
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  verb = .false.;  if (present (verbose))  verb = verbose
  write_md5 = verb;  if (present (write_md5sum)) write_md5 = write_md5sum
  if (.not. beam_data%initialized) then
     write (u, "(1x,A)") "Beam data: [undefined]"
     return
  end if
  prt_name_len = maxval (len (beam_data%flv%get_name ()))
  select case (beam_data%n)
  case (1)
     write (u, "(1x,A)") "Beam data (decay):"
     if (verb) then
        call write_prt (1)
        call beam_data%pmatrix(1)%write (u)
        write (u, *) "R.f. momentum:"
```

```
         call vector4_write (beam_data%p_cm(1), u)
         write (u, *) "Lab momentum:"
         call vector4_write (beam_data%p(1), u)
      else
         call write_prt (1)
      end if
   case (2)
      write (u, "(1x,A)") "Beam data (collision):"
      if (verb) then
         call write_prt (1)
         call beam_data%pmatrix(1)%write (u)
         call write_prt (2)
         call beam_data%pmatrix(2)%write (u)
         call write_sqrts
         write (u, *) "C.m. momenta:"
         call vector4_write (beam_data%p_cm(1), u)
         call vector4_write (beam_data%p_cm(2), u)
         write (u, *) "Lab momenta:"
         call vector4_write (beam_data%p(1), u)
         call vector4_write (beam_data%p(2), u)
      else
         call write_prt (1)
         call write_prt (2)
         call write_sqrts
      end if
   end select
   if (allocated (beam_data%L_cm_to_lab)) then
      if (verb) then
         call lorentz_transformation_write (beam_data%L_cm_to_lab, u)
      else
         write (u, "(1x,A)")  "Beam structure: lab and c.m. frame differ"
      end if
   end if
   if (write_md5) then
      write (u, *) "MD5 sum: ", beam_data%md5sum
   end if
contains
   subroutine write_sqrts
     character(80) :: sqrts_str
     write (sqrts_str, "(" // FMT_19 // ")")  beam_data%sqrts
     write (u, "(3x,A)")  "sqrts = " // trim (adjustl (sqrts_str)) // " GeV"
   end subroutine write_sqrts
   subroutine write_prt (i)
     integer, intent(in) :: i
     character(80) :: name_str, mass_str
     write (name_str, "(A)")  char (beam_data%flv(i)%get_name ())
     write (mass_str, "(ES13.7)")  beam_data%mass(i)
     write (u, "(3x,A)", advance="no") &
          name_str(:prt_name_len) // "  (mass = " &
          // trim (adjustl (mass_str)) // " GeV)"
     if (beam_data%pmatrix(i)%is_polarized ()) then
        write (u, "(2x,A)")  "polarized"
     else
        write (u, *)
```

```
      end if
    end subroutine write_prt
  end subroutine beam_data_write
```

Return initialization status:

⟨*Beams: beam data: TBP*⟩+≡

```
  procedure :: are_valid => beam_data_are_valid
```

⟨*Beams: procedures*⟩+≡

```
  function beam_data_are_valid (beam_data) result (flag)
    class(beam_data_t), intent(in) :: beam_data
    logical :: flag
    flag = beam_data%initialized
  end function beam_data_are_valid
```

Check whether beam data agree with the current values of relevant parameters.

⟨*Beams: beam data: TBP*⟩+≡

```
  procedure :: check_scattering => beam_data_check_scattering
```

⟨*Beams: procedures*⟩+≡

```
  subroutine beam_data_check_scattering (beam_data, sqrts)
    class(beam_data_t), intent(in) :: beam_data
    real(default), intent(in), optional :: sqrts
    if (beam_data_are_valid (beam_data)) then
       if (present (sqrts)) then
          if (.not. nearly_equal (sqrts, beam_data%sqrts)) then
             call msg_error ("Current setting of sqrts is inconsistent " &
                  // "with beam setup (ignored).")
          end if
       end if
    else
       call msg_bug ("Beam setup: invalid beam data")
    end if
  end subroutine beam_data_check_scattering
```

Return the number of beams (1 for decays, 2 for collisions).

⟨*Beams: beam data: TBP*⟩+≡

```
  procedure :: get_n_in => beam_data_get_n_in
```

⟨*Beams: procedures*⟩+≡

```
  function beam_data_get_n_in (beam_data) result (n_in)
    class(beam_data_t), intent(in) :: beam_data
    integer :: n_in
    n_in = beam_data%n
  end function beam_data_get_n_in
```

Return the beam flavor

⟨*Beams: beam data: TBP*⟩+≡

```
  procedure :: get_flavor => beam_data_get_flavor
```

⟨*Beams: procedures*⟩+≡

```
function beam_data_get_flavor (beam_data) result (flv)
  class(beam_data_t), intent(in) :: beam_data
  type(flavor_t), dimension(:), allocatable :: flv
  allocate (flv (beam_data%n))
  flv = beam_data%flv
end function beam_data_get_flavor
```

Return the beam energies

⟨*Beams: beam data: TBP*⟩+≡

```
procedure :: get_energy => beam_data_get_energy
```

⟨*Beams: procedures*⟩+≡

```
function beam_data_get_energy (beam_data) result (e)
  class(beam_data_t), intent(in) :: beam_data
  real(default), dimension(:), allocatable :: e
  !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
  integer :: i
  allocate (e (beam_data%n))
  if (beam_data%initialized) then
     do i = 1, beam_data%n
        e(i) = energy (beam_data%p(i))
     end do
  else
     e = 0
  end if
end function beam_data_get_energy
```

Return the c.m. energy.

⟨*Beams: beam data: TBP*⟩+≡

```
procedure :: get_sqrts => beam_data_get_sqrts
```

⟨*Beams: procedures*⟩+≡

```
function beam_data_get_sqrts (beam_data) result (sqrts)
  class(beam_data_t), intent(in) :: beam_data
  real(default) :: sqrts
  sqrts = beam_data%sqrts
end function beam_data_get_sqrts
```

Return true if the lab and c.m. frame are specified as identical.

⟨*Beams: beam data: TBP*⟩+≡

```
procedure :: cm_frame => beam_data_cm_frame
```

⟨*Beams: procedures*⟩+≡

```
function beam_data_cm_frame (beam_data) result (flag)
  class(beam_data_t), intent(in) :: beam_data
  logical :: flag
  flag = beam_data%lab_is_cm_frame
end function beam_data_cm_frame
```

Return the polarization in case it is just two degrees

⟨*Beams: beam data: TBP*⟩+≡

```
procedure :: get_polarization => beam_data_get_polarization
```

⟨*Beams: procedures*⟩+≡

```
function beam_data_get_polarization (beam_data) result (pol)
  class(beam_data_t), intent(in) :: beam_data
  real(default), dimension(2) :: pol
  if (beam_data%n /= 2) &
       call msg_fatal ("Beam data: can only treat scattering processes.")
  pol = beam_data%pmatrix%get_simple_pol ()
end function beam_data_get_polarization
```

⟨*Beams: beam data: TBP*⟩+≡

```
procedure :: get_helicity_state_matrix => beam_data_get_helicity_state_matrix
```

⟨*Beams: procedures*⟩+≡

```
function beam_data_get_helicity_state_matrix (beam_data) result (state_hel)
  type(state_matrix_t) :: state_hel
  class(beam_data_t), intent(in) :: beam_data
  type(polarization_t), dimension(:), allocatable :: pol
  integer :: i
  allocate (pol (beam_data%n))
  do i = 1, beam_data%n
     call pol(i)%init_pmatrix (beam_data%pmatrix(i))
  end do
  call combine_polarization_states (pol, state_hel)
end function beam_data_get_helicity_state_matrix
```

⟨*Beams: beam data: TBP*⟩+≡

```
procedure :: is_initialized => beam_data_is_initialized
```

⟨*Beams: procedures*⟩+≡

```
function beam_data_is_initialized (beam_data) result (initialized)
  logical :: initialized
  class(beam_data_t), intent(in) :: beam_data
  initialized = any (beam_data%pmatrix%exists ())
end function beam_data_is_initialized
```

Return a MD5 checksum for beam data. If no checksum is present (because beams have not been initialized), compute the checksum of the sqrts value.

⟨*Beams: beam data: TBP*⟩+≡

```
procedure :: get_md5sum => beam_data_get_md5sum
```

⟨*Beams: procedures*⟩+≡

```
function beam_data_get_md5sum (beam_data, sqrts) result (md5sum_beams)
  class(beam_data_t), intent(in) :: beam_data
  real(default), intent(in) :: sqrts
  character(32) :: md5sum_beams
  character(80) :: buffer
  if (beam_data%md5sum /= "") then
     md5sum_beams = beam_data%md5sum
  else
     write (buffer, *)  sqrts
     md5sum_beams = md5sum (buffer)
  end if
end function beam_data_get_md5sum
```

15.2.2 Initializers: beam structure

Initialize the beam data object from a beam structure object, given energy and model.

⟨*Beams: beam data: TBP*⟩+≡

```
procedure :: init_structure => beam_data_init_structure
```

⟨*Beams: procedures*⟩+≡

```
subroutine beam_data_init_structure &
     (beam_data, structure, sqrts, model, decay_rest_frame)
  class(beam_data_t), intent(out) :: beam_data
  type(beam_structure_t), intent(in) :: structure
  integer :: n_beam
  real(default), intent(in) :: sqrts
  class(model_data_t), intent(in), target :: model
  logical, intent(in), optional :: decay_rest_frame
  type(flavor_t), dimension(:), allocatable :: flv
  n_beam = structure%get_n_beam ()
  allocate (flv (n_beam))
  call flv%init (structure%get_prt (), model)
  if (structure%asymmetric ()) then
     if (structure%polarized ()) then
        call beam_data%init_momenta (structure%get_momenta (), flv, &
             structure%get_smatrix (), structure%get_pol_f ())
     else
        call beam_data%init_momenta (structure%get_momenta (), flv)
     end if
  else
     select case (n_beam)
     case (1)
        if (structure%polarized ()) then
           call beam_data%init_decay (flv, &
                structure%get_smatrix (), structure%get_pol_f (), &
                rest_frame = decay_rest_frame)
        else
           call beam_data%init_decay (flv, &
                rest_frame = decay_rest_frame)
        end if
     case (2)
        if (structure%polarized ()) then
           call beam_data%init_sqrts (sqrts, flv, &
                structure%get_smatrix (), structure%get_pol_f ())
        else
           call beam_data%init_sqrts (sqrts, flv)
        end if
     case default
        call msg_bug ("Beam data: invalid beam structure object")
     end select
  end if
end subroutine beam_data_init_structure
```

15.2.3 Initializers: collisions

This is the simplest one: just the two flavors, c.m. energy, polarization. Color is inferred from flavor. Beam momenta and c.m. momenta coincide.

⟨*Beams: beam data: TBP*⟩+≡

```
procedure :: init_sqrts => beam_data_init_sqrts
```

⟨*Beams: procedures*⟩+≡

```
subroutine beam_data_init_sqrts (beam_data, sqrts, flv, smatrix, pol_f)
  class(beam_data_t), intent(out) :: beam_data
  real(default), intent(in) :: sqrts
  type(flavor_t), dimension(:), intent(in) :: flv
  type(smatrix_t), dimension(:), intent(in), optional :: smatrix
  real(default), dimension(:), intent(in), optional :: pol_f
  real(default), dimension(size(flv)) :: E, p
  call beam_data_init (beam_data, size (flv))
  beam_data%sqrts = sqrts
  beam_data%lab_is_cm_frame = .true.
  select case (beam_data%n)
  case (1)
     E = sqrts;  p = 0
     beam_data%p_cm = vector4_moving (E, p, 3)
     beam_data%p = beam_data%p_cm
  case (2)
     beam_data%p_cm = colliding_momenta (sqrts, flv%get_mass ())
     beam_data%p = colliding_momenta (sqrts, flv%get_mass ())
  end select
  call beam_data_finish_initialization (beam_data, flv, smatrix, pol_f)
end subroutine beam_data_init_sqrts
```

This version sets beam momenta directly, assuming that they are asymmetric, i.e., lab frame and c.m. frame do not coincide. Polarization info is deferred to a common initializer.

The Lorentz transformation that we compute here is not actually used in the calculation; instead, it will be recomputed for each event in the subroutine `phs_set_incoming_momenta`. We compute it here for the nominal beam setup nevertheless, so we can print it and, in particular, include it in the MD5 sum.

⟨*Beams: beam data: TBP*⟩+≡

```
procedure :: init_momenta => beam_data_init_momenta
```

⟨*Beams: procedures*⟩+≡

```
subroutine beam_data_init_momenta (beam_data, p3, flv, smatrix, pol_f)
  class(beam_data_t), intent(out) :: beam_data
  type(vector3_t), dimension(:), intent(in) :: p3
  type(flavor_t), dimension(:), intent(in) :: flv
  type(smatrix_t), dimension(:), intent(in), optional :: smatrix
  real(default), dimension(:), intent(in), optional :: pol_f
  type(vector4_t) :: p0
  type(vector4_t), dimension(:), allocatable :: p, p_cm_rot
  real(default), dimension(size(p3)) :: e
  real(default), dimension(size(flv)) :: m
  type(lorentz_transformation_t) :: L_boost, L_rot
  !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
  integer :: i
```

```
  call beam_data_init (beam_data, size (flv))
  m = flv%get_mass ()
  e = sqrt (p3 ** 2 + m ** 2)
  allocate (p (beam_data%n))
  !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
  !!! p = vector4_moving (e, p3)
  do i = 1, beam_data%n
     p(i) = vector4_moving (e(i), p3(i))
  end do
  p0 = sum (p)
  beam_data%p = p
  beam_data%lab_is_cm_frame = .false.
  beam_data%sqrts = p0 ** 1
  L_boost = boost (p0, beam_data%sqrts)
  allocate (p_cm_rot (beam_data%n))
  p_cm_rot = inverse (L_boost) * p
  allocate (beam_data%L_cm_to_lab)
  select case (beam_data%n)
  case (1)
     beam_data%L_cm_to_lab = L_boost
     beam_data%p_cm = vector4_at_rest (beam_data%sqrts)
  case (2)
     L_rot = rotation_to_2nd (3, space_part (p_cm_rot(1)))
     beam_data%L_cm_to_lab = L_boost * L_rot
     beam_data%p_cm = &
          colliding_momenta (beam_data%sqrts, flv%get_mass ())
  end select
  call beam_data_finish_initialization (beam_data, flv, smatrix, pol_f)
end subroutine beam_data_init_momenta
```

Final steps: If requested, rotate the beams in the lab frame, and set the beam-data components.

⟨*Beams: procedures*⟩+≡

```
subroutine beam_data_finish_initialization (beam_data, flv, smatrix, pol_f)
  type(beam_data_t), intent(inout) :: beam_data
  type(flavor_t), dimension(:), intent(in) :: flv
  type(smatrix_t), dimension(:), intent(in), optional :: smatrix
  real(default), dimension(:), intent(in), optional :: pol_f
  integer :: i
  do i = 1, beam_data%n
     beam_data%flv(i) = flv(i)
     beam_data%mass(i) = flv(i)%get_mass ()
     if (present (smatrix)) then
        if (size (smatrix) /= beam_data%n) &
             call msg_fatal ("Beam data: &
             &polarization density array has wrong dimension")
        beam_data%pmatrix(i) = smatrix(i)
        if (present (pol_f)) then
           if (size (pol_f) /= size (smatrix)) &
                call msg_fatal ("Beam data: &
                &polarization fraction array has wrong dimension")
           call beam_data%pmatrix(i)%normalize (flv(i), pol_f(i))
        else
```

```
            call beam_data%pmatrix(i)%normalize (flv(i), 1._default)
         end if
      else
         call beam_data%pmatrix(i)%init (2, 0)
         call beam_data%pmatrix(i)%normalize (flv(i), 0._default)
      end if
   end do
   call beam_data%compute_md5sum ()
end subroutine beam_data_finish_initialization
```

The MD5 sum is stored within the beam-data record, so it can be checked for integrity in subsequent runs.

⟨*Beams: beam data: TBP*⟩+≡

```
procedure :: compute_md5sum => beam_data_compute_md5sum
```

⟨*Beams: procedures*⟩+≡

```
subroutine beam_data_compute_md5sum (beam_data)
  class(beam_data_t), intent(inout) :: beam_data
  integer :: unit
  unit = free_unit ()
  open (unit = unit, status = "scratch", action = "readwrite")
  call beam_data%write (unit, write_md5sum = .false., &
     verbose = .true.)
  rewind (unit)
  beam_data%md5sum = md5sum (unit)
  close (unit)
end subroutine beam_data_compute_md5sum
```

15.2.4 Initializers: decays

This is the simplest one: decay in rest frame. We need just flavor and polarization. Color is inferred from flavor. Beam momentum and c.m. momentum coincide.

⟨*Beams: beam data: TBP*⟩+≡

```
procedure :: init_decay => beam_data_init_decay
```

⟨*Beams: procedures*⟩+≡

```
subroutine beam_data_init_decay (beam_data, flv, smatrix, pol_f, rest_frame)
  class(beam_data_t), intent(out) :: beam_data
  type(flavor_t), dimension(1), intent(in) :: flv
  type(smatrix_t), dimension(1), intent(in), optional :: smatrix
  real(default), dimension(:), intent(in), optional :: pol_f
  logical, intent(in), optional :: rest_frame
  real(default), dimension(1) :: m
  m = flv%get_mass ()
  if (present (smatrix)) then
     call beam_data%init_sqrts (m(1), flv, smatrix, pol_f)
  else
     call beam_data%init_sqrts (m(1), flv, smatrix, pol_f)
  end if
  if (present (rest_frame))  beam_data%lab_is_cm_frame = rest_frame
end subroutine beam_data_init_decay
```

15.2.5 Sanity check

After the beams have been set, the initial-particle masses may have been modified. This can be checked here.

⟨*Beams: beam data: TBP*⟩+≡
```
procedure :: masses_are_consistent => beam_data_masses_are_consistent
```

⟨*Beams: procedures*⟩+≡
```
function beam_data_masses_are_consistent (beam_data) result (flag)
  logical :: flag
  class(beam_data_t), intent(in) :: beam_data
  flag = all (nearly_equal (beam_data%mass, beam_data%flv%get_mass ()))
end function beam_data_masses_are_consistent
```

15.2.6 The beams type

Beam objects are interaction objects that contain the actual beam data including polarization and density matrix. For collisions, the beam object actually contains two beams.

⟨*Beams: public*⟩+≡
```
public :: beam_t
```

⟨*Beams: types*⟩+≡
```
type :: beam_t
   private
   type(interaction_t) :: int
end type beam_t
```

The constructor contains code that converts beam data into the (entangled) particle-pair quantum state. First, we set the number of particles and polarization mask. (The polarization mask is handed over to all later interactions, so if helicity is diagonal or absent, this fact is used when constructing the hard-interaction events.) Then, we construct the entangled state that combines helicity, flavor and color of the two particles (where flavor and color are unique, while several helicity states are possible). Then, we transfer this state together with the associated values from the spin density matrix into the `interaction_t` object.

Calling the `add_state` method of the interaction object, we keep the entries of the helicity density matrix without adding them up. This ensures that for unpolarized states, we do not normalize but end up with an $1/N$ entry, where N is the initial-state multiplicity.

⟨*Beams: public*⟩+≡
```
public :: beam_init
```

⟨*Beams: procedures*⟩+≡
```
subroutine beam_init (beam, beam_data)
  type(beam_t), intent(out) :: beam
  type(beam_data_t), intent(in), target :: beam_data
  logical, dimension(beam_data%n) :: polarized, diagonal
  type(quantum_numbers_mask_t), dimension(beam_data%n) :: mask, mask_d
  type(state_matrix_t), target :: state_hel, state_fc, state_tmp
  type(state_iterator_t) :: it_hel, it_tmp
```

```
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    complex(default) :: value
    real(default), parameter :: tolerance = 100 * epsilon (1._default)
    polarized = beam_data%pmatrix%is_polarized ()
    diagonal = beam_data%pmatrix%is_diagonal ()
    mask = quantum_numbers_mask (.false., .false., &
         mask_h = .not. polarized, &
         mask_hd = diagonal)
    mask_d = quantum_numbers_mask (.false., .false., .false., &
         mask_hd = polarized .and. diagonal)
    call beam%int%basic_init &
         (0, 0, beam_data%n, mask=mask, store_values=.true.)
    state_hel = beam_data%get_helicity_state_matrix ()
    allocate (qn (beam_data%n))
    call qn%init (beam_data%flv, color_from_flavor (beam_data%flv, 1))
    call state_fc%init ()
    call state_fc%add_state (qn)
    call merge_state_matrices (state_hel, state_fc, state_tmp)
    call it_hel%init (state_hel)
    call it_tmp%init (state_tmp)
    do while (it_hel%is_valid ())
       qn = it_tmp%get_quantum_numbers ()
       value = it_hel%get_matrix_element ()
       if (any (qn%are_redundant (mask_d))) then
          ! skip off-diagonal elements for diagonal polarization
       else if (abs (value) <= tolerance) then
          ! skip zero entries
       else
          call beam%int%add_state (qn, value = value)
       end if
       call it_hel%advance ()
       call it_tmp%advance ()
    end do
    call beam%int%freeze ()
    call beam%int%set_momenta (beam_data%p, outgoing = .true.)
    call state_hel%final ()
    call state_fc%final ()
    call state_tmp%final ()
  end subroutine beam_init
```

Finalizer:

⟨*Beams: public*⟩+≡

```
  public :: beam_final
```

⟨*Beams: procedures*⟩+≡

```
  subroutine beam_final (beam)
    type(beam_t), intent(inout) :: beam
    call beam%int%final ()
  end subroutine beam_final
```

I/O:

⟨*Beams: public*⟩+≡

```
  public :: beam_write
```

⟨*Beams: procedures*⟩+≡

```
subroutine beam_write (beam, unit, verbose, show_momentum_sum, show_mass, col_verbose)
  type(beam_t), intent(in) :: beam
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose, show_momentum_sum, show_mass
  logical, intent(in), optional :: col_verbose
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  select case (beam%int%get_n_out ())
  case (1);  write (u, *) "Decaying particle:"
  case (2);  write (u, *) "Colliding beams:"
  end select
  call beam%int%basic_write &
       (unit, verbose = verbose, show_momentum_sum = &
        show_momentum_sum, show_mass = show_mass, &
        col_verbose = col_verbose)
end subroutine beam_write
```

Defined assignment: deep copy

⟨*Beams: public*⟩+≡

```
public :: assignment(=)
```

⟨*Beams: interfaces*⟩≡

```
interface assignment(=)
   module procedure beam_assign
end interface
```

⟨*Beams: procedures*⟩+≡

```
subroutine beam_assign (beam_out, beam_in)
  type(beam_t), intent(out) :: beam_out
  type(beam_t), intent(in) :: beam_in
  beam_out%int = beam_in%int
end subroutine beam_assign
```

15.2.7 Inherited procedures

⟨*Beams: public*⟩+≡

```
public :: interaction_set_source_link
```

⟨*Beams: interfaces*⟩+≡

```
interface interaction_set_source_link
   module procedure interaction_set_source_link_beam
end interface
```

⟨*Beams: procedures*⟩+≡

```
subroutine interaction_set_source_link_beam (int, i, beam1, i1)
  type(interaction_t), intent(inout) :: int
  type(beam_t), intent(in), target :: beam1
  integer, intent(in) :: i, i1
  call int%set_source_link (i, beam1%int, i1)
end subroutine interaction_set_source_link_beam
```

15.2.8 Accessing contents

Return the interaction component – as a pointer, to avoid any copying.

⟨*Beams: public*⟩+≡
```
public :: beam_get_int_ptr
```

⟨*Beams: procedures*⟩+≡
```
function beam_get_int_ptr (beam) result (int)
  type(interaction_t), pointer :: int
  type(beam_t), intent(in), target :: beam
  int => beam%int
end function beam_get_int_ptr
```

Set beam momenta directly. (Used for cascade decays.)

⟨*Beams: public*⟩+≡
```
public :: beam_set_momenta
```

⟨*Beams: procedures*⟩+≡
```
subroutine beam_set_momenta (beam, p)
  type(beam_t), intent(inout) :: beam
  type(vector4_t), dimension(:), intent(in) :: p
  call beam%int%set_momenta (p)
end subroutine beam_set_momenta
```

15.2.9 Unit tests

Test module, followed by the corresponding implementation module.

⟨beams_ut.f90⟩≡
```
⟨File header⟩

module beams_ut
  use unit_tests
  use beams_uti

⟨Standard module head⟩

⟨Beams: public test⟩

contains

⟨Beams: test driver⟩

end module beams_ut
```

⟨beams_uti.f90⟩≡
```
⟨File header⟩

module beams_uti

⟨Use kinds⟩
  use lorentz
  use flavors
  use interactions, only: reset_interaction_counter
```

```
  use polarizations, only: smatrix_t
  use model_data
  use beam_structures

  use beams
```

⟨*Standard module head*⟩

⟨*Beams: test declarations*⟩

```
contains
```

⟨*Beams: tests*⟩

```
end module beams_uti
```

API: driver for the unit tests below.

⟨*Beams: public test*⟩≡

```
  public :: beams_test
```

⟨*Beams: test driver*⟩≡

```
  subroutine beams_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```
⟨*Beams: execute tests*⟩
```
  end subroutine beams_test
```

Test the basic beam setup.

⟨*Beams: execute tests*⟩≡

```
  call test (beam_1, "beam_1", &
       "check basic beam setup", &
       u, results)
```

⟨*Beams: test declarations*⟩≡

```
  public :: beam_1
```

⟨*Beams: tests*⟩≡

```
  subroutine beam_1 (u)
    integer, intent(in) :: u
    type(beam_data_t), target :: beam_data
    type(beam_t) :: beam
    real(default) :: sqrts
    type(flavor_t), dimension(2) :: flv
    type(smatrix_t), dimension(2) :: smatrix
    real(default), dimension(2) :: pol_f
    type(model_data_t), target :: model

    write (u, "(A)")  "* Test output: beam_1"
    write (u, "(A)")  "*   Purpose: test basic beam setup"
    write (u, "(A)")

    write (u, "(A)")  "* Reading model file"
    write (u, "(A)")

    call reset_interaction_counter ()
```

```
call model%init_sm_test ()

write (u, "(A)")  "* Unpolarized scattering, massless fermions"
write (u, "(A)")

call reset_interaction_counter ()
sqrts = 500
call flv%init ([1,-1], model)

call beam_data%init_sqrts (sqrts, flv)
call beam_data%write (u)
write (u, "(A)")
call beam_init (beam, beam_data)
call beam_write (beam, u)
call beam_final (beam)
call beam_data%final ()

write (u, "(A)")
write (u, "(A)")  "* Unpolarized scattering, massless bosons"
write (u, "(A)")

call reset_interaction_counter ()
sqrts = 500
call flv%init ([22,22], model)

call beam_data%init_sqrts (sqrts, flv)
call beam_data%write (u)
write (u, "(A)")
call beam_init (beam, beam_data)
call beam_write (beam, u)
call beam_final (beam)
call beam_data%final ()

write (u, "(A)")
write (u, "(A)")  "* Unpolarized scattering, massive bosons"
write (u, "(A)")

call reset_interaction_counter ()
sqrts = 500
call flv%init ([24,-24], model)

call beam_data%init_sqrts (sqrts, flv)
call beam_data%write (u)
write (u, "(A)")
call beam_init (beam, beam_data)
call beam_write (beam, u)
call beam_final (beam)
call beam_data%final ()

write (u, "(A)")
write (u, "(A)")  "* Polarized scattering, massless fermions"
write (u, "(A)")

call reset_interaction_counter ()
```

```
sqrts = 500
call flv%init ([1,-1], model)

call smatrix(1)%init (2, 1)
call smatrix(1)%set_entry (1, [1,1], (1._default, 0._default))
pol_f(1) = 0.5_default

call smatrix(2)%init (2, 3)
call smatrix(2)%set_entry (1, [1,1], (1._default, 0._default))
call smatrix(2)%set_entry (2, [-1,-1], (1._default, 0._default))
call smatrix(2)%set_entry (3, [-1,1], (1._default, 0._default))
pol_f(2) = 1._default

call beam_data%init_sqrts (sqrts, flv, smatrix, pol_f)
call beam_data%write (u)
write (u, "(A)")
call beam_init (beam, beam_data)
call beam_write (beam, u)
call beam_final (beam)
call beam_data%final ()

write (u, "(A)")
write (u, "(A)")  "* Semi-polarized scattering, massless bosons"
write (u, "(A)")

call reset_interaction_counter ()
sqrts = 500
call flv%init ([22,22], model)

call smatrix(1)%init (2, 0)
pol_f(1) = 0._default

call smatrix(2)%init (2, 1)
call smatrix(2)%set_entry (1, [1,1], (1._default, 0._default))
pol_f(2) = 1._default

call beam_data%init_sqrts (sqrts, flv, smatrix, pol_f)
call beam_data%write (u)
write (u, "(A)")
call beam_init (beam, beam_data)
call beam_write (beam, u)
call beam_final (beam)
call beam_data%final ()

write (u, "(A)")
write (u, "(A)")  "* Semi-polarized scattering, massive bosons"
write (u, "(A)")

call reset_interaction_counter ()
sqrts = 500
call flv%init ([24,-24], model)

call smatrix(1)%init (2, 0)
pol_f(1) = 0._default
```

```
  call smatrix(2)%init (2, 1)
  call smatrix(2)%set_entry (1, [0,0], (1._default, 0._default))
  pol_f(2) = 1._default

  call beam_data%init_sqrts (sqrts, flv, smatrix, pol_f)
  call beam_data%write (u)
  write (u, "(A)")
  call beam_init (beam, beam_data)
  call beam_write (beam, u)
  call beam_final (beam)
  call beam_data%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Unpolarized decay, massive boson"
  write (u, "(A)")

  call reset_interaction_counter ()
  call flv(1)%init (23, model)

  call beam_data%init_decay (flv(1:1))
  call beam_data%write (u)
  write (u, "(A)")
  call beam_init (beam, beam_data)
  call beam_write (beam, u)

  write (u, "(A)")
  write (u, "(A)")  "* Polarized decay, massive boson"
  write (u, "(A)")

  call reset_interaction_counter ()
  call flv(1)%init (23, model)
  call smatrix(1)%init (2, 1)
  call smatrix(1)%set_entry (1, [0,0], (1._default, 0._default))
  pol_f(1) = 0.4_default

  call beam_data%init_decay (flv(1:1), smatrix(1:1), pol_f(1:1))
  call beam_data%write (u)
  write (u, "(A)")
  call beam_init (beam, beam_data)
  call beam_write (beam, u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call beam_final (beam)
  call beam_data%final ()

  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: beam_1"

end subroutine beam_1
```

Test advanced beam setup.

⟨*Beams: execute tests*⟩+≡

```
    call test (beam_2, "beam_2", &
         "beam initialization", &
         u, results)
```

⟨*Beams: test declarations*⟩+≡

```
    public :: beam_2
```

⟨*Beams: tests*⟩+≡

```
    subroutine beam_2 (u)
      integer, intent(in) :: u
      type(beam_data_t), target :: beam_data
      type(beam_t) :: beam
      real(default) :: sqrts
      type(flavor_t), dimension(2) :: flv
      integer, dimension(0) :: no_records
      type(beam_structure_t) :: beam_structure
      type(model_data_t), target :: model

      write (u, "(A)")  "* Test output: beam_2"
      write (u, "(A)")  "*   Purpose: transfer beam polarization using &
           &beam structure"
      write (u, "(A)")

      write (u, "(A)")  "* Reading model file"
      write (u, "(A)")

      call model%init_sm_test ()

      write (u, "(A)")  "* Unpolarized scattering, massless fermions"
      write (u, "(A)")

      call reset_interaction_counter ()
      sqrts = 500
      call flv%init ([1,-1], model)
      call beam_structure%init_sf (flv%get_name (), no_records)
      call beam_structure%final_pol ()

      call beam_structure%write (u)
      write (u, *)

      call beam_data%init_structure (beam_structure, sqrts, model)
      call beam_data%write (u)

      write (u, "(A)")
      call beam_init (beam, beam_data)
      call beam_write (beam, u)
      call beam_final (beam)
      call beam_data%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Unpolarized scattering, massless bosons"
      write (u, "(A)")
```

```
call reset_interaction_counter ()
sqrts = 500
call flv%init ([22,22], model)

call beam_structure%init_sf (flv%get_name (), no_records)
call beam_structure%final_pol ()

call beam_structure%write (u)
write (u, *)

call beam_data%init_structure (beam_structure, sqrts, model)
call beam_data%write (u)

write (u, "(A)")
call beam_init (beam, beam_data)
call beam_write (beam, u)
call beam_final (beam)
call beam_data%final ()

write (u, "(A)")
write (u, "(A)")  "* Unpolarized scattering, massive bosons"
write (u, "(A)")

call reset_interaction_counter ()
sqrts = 500
call flv%init ([24,-24], model)

call beam_structure%init_sf (flv%get_name (), no_records)
call beam_structure%final_pol ()

call beam_structure%write (u)
write (u, *)

call beam_data%init_structure (beam_structure, sqrts, model)
call beam_data%write (u)

write (u, "(A)")
call beam_init (beam, beam_data)
call beam_write (beam, u)
call beam_final (beam)
call beam_data%final ()

write (u, "(A)")
write (u, "(A)")  "* Polarized scattering, massless fermions"
write (u, "(A)")

call reset_interaction_counter ()
sqrts = 500
call flv%init ([1,-1], model)
call beam_structure%init_sf (flv%get_name (), no_records)
call beam_structure%init_pol (2)

call beam_structure%init_smatrix (1, 1)
```

```
call beam_structure%set_sentry (1, 1, [1,1], (1._default, 0._default))

call beam_structure%init_smatrix (2, 3)
call beam_structure%set_sentry (2, 1, [1,1], (1._default, 0._default))
call beam_structure%set_sentry (2, 2, [-1,-1], (1._default, 0._default))
call beam_structure%set_sentry (2, 3, [-1,1], (1._default, 0._default))

call beam_structure%set_pol_f ([0.5_default, 1._default])
call beam_structure%write (u)
write (u, *)

call beam_data%init_structure (beam_structure, sqrts, model)
call beam_data%write (u)
write (u, *)

call beam_init (beam, beam_data)
call beam_write (beam, u)

call beam_final (beam)
call beam_data%final ()
call beam_structure%final_pol ()
call beam_structure%final_sf ()

write (u, "(A)")
write (u, "(A)")  "* Semi-polarized scattering, massless bosons"
write (u, "(A)")

call reset_interaction_counter ()
sqrts = 500
call flv%init ([22,22], model)

call beam_structure%init_sf (flv%get_name (), no_records)
call beam_structure%init_pol (2)

call beam_structure%init_smatrix (1, 0)

call beam_structure%init_smatrix (2, 1)
call beam_structure%set_sentry (2, 1, [1,1], (1._default, 0._default))

call beam_structure%set_pol_f ([0._default, 1._default])
call beam_structure%write (u)
write (u, *)

call beam_data%init_structure (beam_structure, sqrts, model)
call beam_data%write (u)

write (u, "(A)")
call beam_init (beam, beam_data)
call beam_write (beam, u)
call beam_final (beam)
call beam_data%final ()

write (u, "(A)")
write (u, "(A)")  "* Semi-polarized scattering, massive bosons"
```

```
write (u, "(A)")

call reset_interaction_counter ()
sqrts = 500
call flv%init ([24,-24], model)

call beam_structure%init_sf (flv%get_name (), no_records)
call beam_structure%init_pol (2)

call beam_structure%init_smatrix (1, 0)

call beam_structure%init_smatrix (2, 1)
call beam_structure%set_sentry (2, 1, [0,0], (1._default, 0._default))
call beam_structure%write (u)

write (u, "(A)")
call beam_data%init_structure (beam_structure, sqrts, model)
call beam_data%write (u)

write (u, "(A)")
call beam_init (beam, beam_data)
call beam_write (beam, u)
call beam_final (beam)
call beam_data%final ()

write (u, "(A)")
write (u, "(A)")  "* Unpolarized decay, massive boson"
write (u, "(A)")

call reset_interaction_counter ()
call flv(1)%init (23, model)

call beam_structure%init_sf ([flv(1)%get_name ()], no_records)
call beam_structure%final_pol ()
call beam_structure%write (u)

write (u, "(A)")
call beam_data%init_structure (beam_structure, sqrts, model)
call beam_data%write (u)

write (u, "(A)")
call beam_init (beam, beam_data)
call beam_write (beam, u)

write (u, "(A)")
write (u, "(A)")  "* Polarized decay, massive boson"
write (u, "(A)")

call reset_interaction_counter ()
call flv(1)%init (23, model)
call beam_structure%init_sf ([flv(1)%get_name ()], no_records)

call beam_structure%init_pol (1)
```

```
    call beam_structure%init_smatrix (1, 1)
    call beam_structure%set_sentry (1, 1, [0,0], (1._default, 0._default))
    call beam_structure%set_pol_f ([0.4_default])
    call beam_structure%write (u)
    write (u, *)

    call beam_data%init_structure (beam_structure, sqrts, model)
    call beam_data%write (u)
    write (u, "(A)")
    call beam_init (beam, beam_data)
    call beam_write (beam, u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call beam_final (beam)
    call beam_data%final ()

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: beam_2"

  end subroutine beam_2
```

Test advanced beam setup, completely arbitrary momenta.

⟨*Beams: execute tests*⟩+≡

```
  call test (beam_3, "beam_3", &
       "generic beam momenta", &
       u, results)
```

⟨*Beams: test declarations*⟩+≡

```
  public :: beam_3
```

⟨*Beams: tests*⟩+≡

```
  subroutine beam_3 (u)
    integer, intent(in) :: u
    type(beam_data_t), target :: beam_data
    type(beam_t) :: beam
    type(flavor_t), dimension(2) :: flv
    integer, dimension(0) :: no_records
    type(model_data_t), target :: model
    type(beam_structure_t) :: beam_structure
    type(vector3_t), dimension(2) :: p3
    type(vector4_t), dimension(2) :: p

    write (u, "(A)")  "* Test output: beam_3"
    write (u, "(A)")  "*   Purpose: set up beams with generic momenta"
    write (u, "(A)")

    write (u, "(A)")  "* Reading model file"
    write (u, "(A)")

    call reset_interaction_counter ()
```

```
call model%init_sm_test ()

write (u, "(A)")  "* 1: Scattering process"
write (u, "(A)")

call flv%init ([2212,2212], model)

p3(1) = vector3_moving ([5._default, 0._default, 10._default])
p3(2) = -vector3_moving ([1._default, 1._default, -10._default])

call beam_structure%init_sf (flv%get_name (), no_records)
call beam_structure%set_momentum (p3 ** 1)
call beam_structure%set_theta (polar_angle (p3))
call beam_structure%set_phi (azimuthal_angle (p3))
call beam_structure%write (u)
write (u, *)

call beam_data%init_structure (beam_structure, 0._default, model)
call pacify (beam_data%l_cm_to_lab, 1e-20_default)
call beam_data%compute_md5sum ()
call beam_data%write (u, verbose = .true.)
write (u, *)

write (u, "(1x,A)")  "Beam momenta reconstructed from LT:"
p = beam_data%L_cm_to_lab * beam_data%p_cm
call pacify (p, 1e-12_default)
call vector4_write (p(1), u)
call vector4_write (p(2), u)
write (u, "(A)")

call beam_init (beam, beam_data)
call beam_write (beam, u)

call beam_final (beam)
call beam_data%final ()
call beam_structure%final_sf ()
call beam_structure%final_mom ()

write (u, "(A)")
write (u, "(A)")  "* 2: Decay"
write (u, "(A)")

call flv(1)%init (23, model)
p3(1) = vector3_moving ([10._default, 5._default, 50._default])

call beam_structure%init_sf ([flv(1)%get_name ()], no_records)
call beam_structure%set_momentum ([p3(1) ** 1])
call beam_structure%set_theta ([polar_angle (p3(1))])
call beam_structure%set_phi ([azimuthal_angle (p3(1))])
call beam_structure%write (u)
write (u, *)

call beam_data%init_structure (beam_structure, 0._default, model)
call beam_data%write (u, verbose = .true.)
```

```
    write (u, "(A)")

    write (u, "(1x,A)")  "Beam momentum reconstructed from LT:"
    p(1) = beam_data%L_cm_to_lab * beam_data%p_cm(1)
    call pacify (p(1), 1e-12_default)
    call vector4_write (p(1), u)
    write (u, "(A)")

    call beam_init (beam, beam_data)
    call beam_write (beam, u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call beam_final (beam)
    call beam_data%final ()
    call beam_structure%final_sf ()
    call beam_structure%final_mom ()

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: beam_3"

  end subroutine beam_3
```

15.3 Tools

This module contains auxiliary procedures that can be accessed by the structure function code.

⟨sf_aux.f90⟩≡

```
  ⟨File header⟩

  module sf_aux

  ⟨Use kinds⟩
    use io_units
    use constants, only: twopi
    use numeric_utils

    use lorentz

  ⟨Standard module head⟩

  ⟨SF aux: public⟩

  ⟨SF aux: parameters⟩

  ⟨SF aux: types⟩

  contains
```

⟨*SF aux: procedures*⟩

```
end module sf_aux
```

15.3.1 Momentum splitting

Let us consider first an incoming parton with momentum k and invariant mass squared $s = k^2$ that splits into two partons with momenta q, p and invariant masses $t = q^2$ and $u = p^2$. (This is an abuse of the Mandelstam notation. t is actually the momentum transfer, assuming that p is radiated and q initiates the hard process.) The energy is split among the partons such that if $E = k^0$, we have $q^0 = xE$ and $p^0 = \bar{x}E$, where $\bar{x} \equiv 1 - x$.

We define the angle θ as the polar angle of p w.r.t. the momentum axis of the incoming momentum k. Ignoring azimuthal angle, we can write the four-momenta in the basis (E, p_T, p_L) as

$$k = \begin{pmatrix} E \\ 0 \\ p \end{pmatrix}, \qquad p = \begin{pmatrix} \bar{x}E \\ \bar{x}\bar{p}\sin\theta \\ \bar{x}\bar{p}\cos\theta \end{pmatrix}, \qquad q = \begin{pmatrix} xE \\ -\bar{x}\bar{p}\sin\theta \\ p - \bar{x}\bar{p}\cos\theta \end{pmatrix}, \tag{15.1}$$

where the first two mass-shell conditions are

$$p^2 = E^2 - s, \qquad \bar{p}^2 = E^2 - \frac{u}{\bar{x}^2}. \tag{15.2}$$

The second condition implies that, for positive u, $\bar{x}^2 > u/E^2$, or equivalently

$$x < 1 - \sqrt{u}/E. \tag{15.3}$$

We are interested in the third mass-shell conditions: s and u are fixed, so we need t as a function of $\cos\theta$:

$$t = -2\bar{x}\left(E^2 - p\bar{p}\cos\theta\right) + s + u. \tag{15.4}$$

Solving for $\cos\theta$, we get

$$\cos\theta = \frac{2\bar{x}E^2 + t - s - u}{2\bar{x}p\bar{p}}. \tag{15.5}$$

We can compute $\sin\theta$ numerically as $\sin^2\theta = 1 - \cos^2\theta$, but it is important to reexpress this in view of numerical stability. To this end, we first determine the bounds for t. The cosine must be between -1 and 1, so the bounds are

$$t_0 = -2\bar{x}\left(E^2 + p\bar{p}\right) + s + u, \tag{15.6}$$

$$t_1 = -2\bar{x}\left(E^2 - p\bar{p}\right) + s + u. \tag{15.7}$$

Computing $\sin^2\theta$ from $\cos\theta$ above, we observe that the numerator is a quadratic polynomial in t which has the zeros t_0 and t_1, while the common denominator is given by $(2\bar{x}p\bar{p})^2$. Hence, we can write

$$\sin^2\theta = -\frac{(t - t_0)(t - t_1)}{(2\bar{x}p\bar{p})^2} \qquad \text{and} \qquad \cos\theta = \frac{(t - t_0) + (t - t_1)}{4\bar{x}p\bar{p}}, \tag{15.8}$$

which is free of large cancellations near $t = t_0$ or $t = t_1$.

If all is massless, i.e., $s = u = 0$, this simplifies to

$$t_0 = -4\bar{x}E^2, \qquad t_1 = 0, \tag{15.9}$$

$$\sin^2\theta = -\frac{t}{\bar{x}E^2}\left(1 + \frac{t}{4\bar{x}E^2}\right), \qquad \cos\theta = 1 + \frac{t}{2\bar{x}E^2}. \tag{15.10}$$

Here is the implementation. First, we define a container for the kinematical integration limits and some further data.

Note: contents are public only for easy access in unit test.

⟨*SF aux: public*⟩≡
```
public :: splitting_data_t
```

⟨*SF aux: types*⟩≡
```
type :: splitting_data_t
!     private
      logical :: collinear = .false.
      real(default) :: x0 = 0
      real(default) :: x1
      real(default) :: t0
      real(default) :: t1
      real(default) :: phi0 = 0
      real(default) :: phi1 = twopi
      real(default) :: E, p, s, u, m2
      real(default) :: x, xb, pb
      real(default) :: t = 0
      real(default) :: phi = 0
    contains
    ⟨SF aux: splitting data: TBP⟩
end type splitting_data_t
```

I/O for debugging:

⟨*SF aux: splitting data: TBP*⟩≡
```
procedure :: write => splitting_data_write
```

⟨*SF aux: procedures*⟩≡
```
subroutine splitting_data_write (d, unit)
  class(splitting_data_t), intent(in) :: d
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(A)") "Splitting data:"
  write (u, "(2x,A,L1)")  "collinear = ", d%collinear
1 format (2x,A,1x,ES15.8)
  write (u, 1) "x0   =", d%x0
  write (u, 1) "x    =", d%x
  write (u, 1) "xb   =", d%xb
  write (u, 1) "x1   =", d%x1
  write (u, 1) "t0   =", d%t0
  write (u, 1) "t    =", d%t
  write (u, 1) "t1   =", d%t1
  write (u, 1) "phi0 =", d%phi0
  write (u, 1) "phi  =", d%phi
  write (u, 1) "phi1 =", d%phi1
```

```
    write (u, 1) "E    =", d%E
    write (u, 1) "p    =", d%p
    write (u, 1) "pb   =", d%pb
    write (u, 1) "s    =", d%s
    write (u, 1) "u    =", d%u
    write (u, 1) "m2   =", d%m2
  end subroutine splitting_data_write
```

15.3.2 Constant data

This is the initializer for the data. The input consists of the incoming momentum, its invariant mass squared, and the invariant mass squared of the radiated particle. $m2$ is the *physical* mass squared of the outgoing particle. The t bounds depend on the chosen x value and cannot be determined yet.

⟨*SF aux: splitting data: TBP*⟩+≡

```
  procedure :: init => splitting_data_init
```

⟨*SF aux: procedures*⟩+≡

```
  subroutine splitting_data_init (d, k, mk2, mr2, mo2, collinear)
    class(splitting_data_t), intent(out) :: d
    type(vector4_t), intent(in) :: k
    real(default), intent(in) :: mk2, mr2, mo2
    logical, intent(in), optional :: collinear
    if (present (collinear))  d%collinear = collinear
    d%E = energy (k)
    d%x1 = 1 - sqrt (max (mr2, 0._default)) / d%E
    d%p = sqrt (d%E**2 - mk2)
    d%s = mk2
    d%u = mr2
    d%m2 = mo2
  end subroutine splitting_data_init
```

Retrieve the x bounds, if needed for x sampling. Generating an x value is done by the caller, since this is the part that depends on the nature of the structure function.

⟨*SF aux: splitting data: TBP*⟩+≡

```
  procedure :: get_x_bounds => splitting_get_x_bounds
```

⟨*SF aux: procedures*⟩+≡

```
  function splitting_get_x_bounds (d) result (x)
    class(splitting_data_t), intent(in) :: d
    real(default), dimension(2) :: x
    x = [ d%x0, d%x1 ]
  end function splitting_get_x_bounds
```

Now set the momentum fraction and compute t_0 and t_1.

[The calculation of t_1 is subject to numerical problems. The exact formula is ($s = m_i^2$, $u = m_r^2$)

$$t_1 = -2\bar{x}E^2 + m_i^2 + m_r^2 + 2\bar{x}\sqrt{E^2 - m_i^2}\,\sqrt{E^2 - m_r^2/\bar{x}^2}. \tag{15.11}$$

The structure-function paradigm is useful only if $E \gg m_i, m_r$. In a Taylor expansion for large E, the leading term cancels. The expansion of the square roots (to subleading order) yields

$$t_1 = x m_i^2 - \frac{x}{\bar{x}} m_r^2. \tag{15.12}$$

There are two cases of interest: $m_i = m_o$ and $m_r = 0$,

$$t_1 = x m_o^2 \tag{15.13}$$

and $m_i = m_r$ and $m_o = 0$,

$$t_1 = -\frac{x^2}{\bar{x}} m_i^2. \tag{15.14}$$

In both cases, $t_1 \leq m_o^2$.]

That said, it turns out that taking the t_1 evaluation at face value leads to less problems than the approximation. We express the angles in terms of $t - t_0$ and $t - t_1$. Numerical noise in t_1 can then be tolerated.

⟨*SF aux: splitting data: TBP*⟩+≡

```
procedure :: set_t_bounds => splitting_set_t_bounds
```

⟨*SF aux: procedures*⟩+≡

```
elemental subroutine splitting_set_t_bounds (d, x, xb)
  class(splitting_data_t), intent(inout) :: d
  real(default), intent(in), optional :: x, xb
  real(default) :: tp, tm
  if (present (x))  d%x = x
  if (present (xb)) d%xb = xb
  if (vanishes (d%u)) then
     d%pb = d%E
  else
     if (.not. vanishes (d%xb)) then
        d%pb = sqrt (max (d%E**2 - d%u / d%xb**2, 0._default))
     else
        d%pb = 0
     end if
  end if
  tp = -2 * d%xb * d%E**2 + d%s + d%u
  tm = -2 * d%xb * d%p * d%pb
  d%t0 = tp + tm
  d%t1 = tp - tm
  d%t = d%t1
end subroutine splitting_set_t_bounds
```

15.3.3 Sampling recoil

Compute a value for the momentum transfer t, using a random number r. We assume a logarithmic distribution for $t - m^2$, corresponding to the propagator $1/(t - m^2)$ with the physical mass m for the outgoing particle. Optionally, we can narrow the kinematical bounds.

If all three masses in the splitting vanish, the upper limit for t is zero. In that case, the t value is set to zero and the splitting will be collinear.

⟨*SF aux: splitting data: TBP*⟩+≡

```
procedure :: sample_t => splitting_sample_t
```

⟨*SF aux: procedures*⟩+≡

```
subroutine splitting_sample_t (d, r, t0, t1)
  class(splitting_data_t), intent(inout) :: d
  real(default), intent(in) :: r
  real(default), intent(in), optional :: t0, t1
  real(default) :: tt0, tt1, tt0m, tt1m
  if (d%collinear) then
     d%t = d%t1
  else
     tt0 = d%t0;  if (present (t0))  tt0 = max (t0, tt0)
     tt1 = d%t1;  if (present (t1))  tt1 = min (t1, tt1)
     tt0m = tt0 - d%m2
     tt1m = tt1 - d%m2
     if (tt0m < 0 .and. tt1m < 0 .and. abs(tt0m) > &
          epsilon(tt0m) .and. abs(tt1m) > epsilon(tt0m)) then
        d%t = d%m2 + tt0m * exp (r * log (tt1m / tt0m))
     else
        d%t = tt1
     end if
  end if
end subroutine splitting_sample_t
```

The inverse operation: Given t, we recover the value of r that would have produced this value.

⟨*SF aux: splitting data: TBP*⟩+≡

```
procedure :: inverse_t => splitting_inverse_t
```

⟨*SF aux: procedures*⟩+≡

```
subroutine splitting_inverse_t (d, r, t0, t1)
  class(splitting_data_t), intent(in) :: d
  real(default), intent(out) :: r
  real(default), intent(in), optional :: t0, t1
  real(default) :: tt0, tt1, tt0m, tt1m
  if (d%collinear) then
     r = 0
  else
     tt0 = d%t0;  if (present (t0))  tt0 = max (t0, tt0)
     tt1 = d%t1;  if (present (t1))  tt1 = min (t1, tt1)
     tt0m = tt0 - d%m2
     tt1m = tt1 - d%m2
     if (tt0m < 0 .and. tt1m < 0) then
        r = log ((d%t - d%m2) / tt0m) / log (tt1m / tt0m)
     else
        r = 0
     end if
  end if
end subroutine splitting_inverse_t
```

This is trivial, but provided for convenience:

⟨*SF aux: splitting data: TBP*⟩+≡

```
procedure :: sample_phi => splitting_sample_phi
```

⟨*SF aux: procedures*⟩+≡

```
subroutine splitting_sample_phi (d, r)
```

```
    class(splitting_data_t), intent(inout) :: d
    real(default), intent(in) :: r
    if (d%collinear) then
       d%phi = 0
    else
       d%phi = (1-r) * d%phi0 + r * d%phi1
    end if
  end subroutine splitting_sample_phi
```

Inverse:

⟨*SF aux: splitting data: TBP*⟩+≡

```
  procedure :: inverse_phi => splitting_inverse_phi
```

⟨*SF aux: procedures*⟩+≡

```
  subroutine splitting_inverse_phi (d, r)
    class(splitting_data_t), intent(in) :: d
    real(default), intent(out) :: r
    if (d%collinear) then
       r = 0
    else
       r = (d%phi - d%phi0) / (d%phi1 - d%phi0)
    end if
  end subroutine splitting_inverse_phi
```

15.3.4 Splitting

In this function, we actually perform the splitting. The incoming momentum k is split into (if no recoil) $q_1 = (1-x)k$ and $q_2 = xk$.

Apart from the splitting data, we need the incoming momentum k, the momentum transfer t, and the azimuthal angle ϕ. The momentum fraction x is already known here.

Alternatively, we can split without recoil. The azimuthal angle is irrelevant, and the momentum transfer is always equal to the upper limit t_1, so the polar angle is zero. Obviously, if there are nonzero masses it is not possible to keep both energy-momentum conservation and at the same time all particles on shell. We choose for dropping the on-shell condition here.

⟨*SF aux: splitting data: TBP*⟩+≡

```
  procedure :: split_momentum => splitting_split_momentum
```

⟨*SF aux: procedures*⟩+≡

```
  function splitting_split_momentum (d, k) result (q)
    class(splitting_data_t), intent(in) :: d
    type(vector4_t), dimension(2) :: q
    type(vector4_t), intent(in) :: k
    real(default) :: st2, ct2, st, ct, cp, sp
    type(lorentz_transformation_t) :: rot
    real(default) :: tt0, tt1, den
    type(vector3_t) :: kk, q1, q2
    if (d%collinear) then
       if (vanishes (d%s) .and. vanishes(d%u)) then
          q(1) = d%xb * k
          q(2) = d%x * k
```

```
      else
         kk = space_part (k)
         q1 = d%xb * (d%pb / d%p) * kk
         q2 = kk - q1
         q(1) = vector4_moving (d%xb * d%E, q1)
         q(2) = vector4_moving (d%x * d%E, q2)
      end if
   else
      den = 2 * d%xb * d%p * d%pb
      tt0 = max (d%t - d%t0, 0._default)
      tt1 = min (d%t - d%t1, 0._default)
      if (den**2 <= epsilon(den)) then
         st2 = 0
      else
         st2 = - (tt0 * tt1) / den ** 2
      end if
      if (st2 > 1) then
         st2 = 1
      end if
      ct2 = 1 - st2
      st = sqrt (max (st2, 0._default))
      ct = sqrt (max (ct2, 0._default))
      if ((d%t - d%t0 + d%t - d%t1) < 0) then
         ct = - ct
      end if
      sp = sin (d%phi)
      cp = cos (d%phi)
      rot = rotation_to_2nd (3, space_part (k))
      q1 = vector3_moving (d%xb * d%pb * [st * cp, st * sp, ct])
      q2 = vector3_moving (d%p, 3) - q1
      q(1) = rot * vector4_moving (d%xb * d%E, q1)
      q(2) = rot * vector4_moving (d%x * d%E, q2)
   end if
end function splitting_split_momentum
```

Momenta generated by splitting will in general be off-shell. They are on-shell only if they are collinear and massless. This subroutine puts them on shell by brute force, violating either momentum or energy conservation. The direction of three-momentum is always retained.

If the energy is below mass shell, we return a zero momentum.

⟨*SF aux: parameters*⟩≡

```
integer, parameter, public :: KEEP_ENERGY = 0, KEEP_MOMENTUM = 1
```

⟨*SF aux: public*⟩+≡

```
public :: on_shell
```

⟨*SF aux: procedures*⟩+≡

```
elemental subroutine on_shell (p, m2, keep)
  type(vector4_t), intent(inout) :: p
  real(default), intent(in) :: m2
  integer, intent(in) :: keep
  real(default) :: E, E2, pn
  select case (keep)
  case (KEEP_ENERGY)
```

```
      E = energy (p)
      E2 = E ** 2
      if (E2 >= m2) then
         pn = sqrt (E2 - m2)
         p = vector4_moving (E, pn * direction (space_part (p)))
      else
         p = vector4_null
      end if
   case (KEEP_MOMENTUM)
      E = sqrt (space_part (p) ** 2 + m2)
      p = vector4_moving (E, space_part (p))
   end select
end subroutine on_shell
```

15.3.5 Recovering the splitting

This is the inverse problem. We have on-shell momenta and want to deduce the splitting parameters x, t, and ϕ.

⟨*SF aux: splitting data: TBP*⟩+≡

```
procedure :: recover => splitting_recover
```

⟨*SF aux: procedures*⟩+≡

```
subroutine splitting_recover (d, k, q, keep)
  class(splitting_data_t), intent(inout) :: d
  type(vector4_t), intent(in) :: k, q
  integer, intent(in) :: keep
  type(lorentz_transformation_t) :: rot
  type(vector4_t) :: q0, k0
  real(default) :: p1, p2, p3, pt2, pp2, pl
  real(default) :: aux, den, norm
  real(default) :: st2, ct2, ct
  rot = inverse (rotation_to_2nd (3, space_part (k)))
  q0 = rot * q
  p1 = vector4_get_component (q0, 1)
  p2 = vector4_get_component (q0, 2)
  p3 = vector4_get_component (q0, 3)
  pt2 = p1 ** 2 + p2 ** 2
  pp2 = p1 ** 2 + p2 ** 2 + p3 ** 2
  pl = abs (p3)
  k0 = vector4_moving (d%E, d%p, 3)
  select case (keep)
  case (KEEP_ENERGY)
     d%x = energy (q0) / d%E
     d%xb = 1 - d%x
     call d%set_t_bounds ()
     if (.not. d%collinear) then
        aux = (d%xb * d%pb) ** 2 * pp2 - d%p ** 2 * pt2
        den = d%p ** 2 - (d%xb * d%pb) ** 2
        if (aux >= 0 .and. den > 0) then
           norm = (d%p * pl + sqrt (aux)) / den
        else
           norm = 1
        end if
```

```
      end if
    case (KEEP_MOMENTUM)
      d%xb = sqrt (space_part (k0 - q0) ** 2 + d%u) / d%E
      d%x = 1 - d%xb
      call d%set_t_bounds ()
      norm = 1
    end select
    if (d%collinear) then
      d%t = d%t1
      d%phi = 0
    else
      if ((d%xb * d%pb * norm)**2 < epsilon(d%xb)) then
        st2 = 1
      else
        st2 = pt2 / (d%xb * d%pb * norm ) ** 2
      end if
      if (st2 > 1) then
        st2 = 1
      end if
      ct2 = 1 - st2
      ct = sqrt (max (ct2, 0._default))
      if (.not. vanishes (1 + ct)) then
        d%t = d%t1 - 2 * d%xb * d%p * d%pb * st2 / (1 + ct)
      else
        d%t = d%t0
      end if
      if (.not. vanishes (p1) .or. .not. vanishes (p2)) then
        d%phi = atan2 (-p2, -p1)
      else
        d%phi = 0
      end if
    end if
  end subroutine splitting_recover
```

15.3.6 Extract data

⟨*SF aux: splitting data: TBP*⟩+≡

```
  procedure :: get_x => splitting_get_x
```

⟨*SF aux: procedures*⟩+≡

```
  function splitting_get_x (sd) result (x)
    class(splitting_data_t), intent(in) :: sd
    real(default) :: x
    x = sd%x
  end function splitting_get_x
```

15.3.7 Unit tests

Test module, followed by the corresponding implementation module.

⟨`sf_aux_ut.f90`⟩≡

⟨*File header*⟩

```
module sf_aux_ut
  use unit_tests
  use sf_aux_uti

⟨Standard module head⟩

⟨SF aux: public test⟩

contains

⟨SF aux: test driver⟩

end module sf_aux_ut
```

⟨sf_aux_uti.f90⟩≡

```
⟨File header⟩

module sf_aux_uti

⟨Use kinds⟩
  use lorentz

  use sf_aux

⟨Standard module head⟩

⟨SF aux: test declarations⟩

contains

⟨SF aux: tests⟩

end module sf_aux_uti
```

API: driver for the unit tests below.

⟨*SF aux: public test*⟩≡

```
  public :: sf_aux_test
```

⟨*SF aux: test driver*⟩≡

```
  subroutine sf_aux_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨SF aux: execute tests⟩
  end subroutine sf_aux_test
```

Momentum splitting: massless radiation

Compute momentum splitting for generic kinematics. It turns out that for $x = 0.5$, where $t - m^2$ is the geometric mean between its upper and lower bounds (this can be directly seen from the logarithmic distribution in the function `sample_t` for $r \equiv x = 1 - x = 0.5$), we arrive at an exact number $t = -0.15$ for the given input values.

⟨*SF aux: execute tests*⟩≡

```
call test (sf_aux_1, "sf_aux_1", &
     "massless radiation", &
     u, results)
```

⟨*SF aux: test declarations*⟩≡

```
public :: sf_aux_1
```

⟨*SF aux: tests*⟩≡

```
subroutine sf_aux_1 (u)
  integer, intent(in) :: u
  type(splitting_data_t) :: sd
  type(vector4_t) :: k
  type(vector4_t), dimension(2) :: q, q0
  real(default) :: E, mk, mp, mq
  real(default) :: x, r1, r2, r1o, r2o
  real(default) :: k2, q0_2, q1_2, q2_2

  write (u, "(A)")  "* Test output: sf_aux_1"
  write (u, "(A)")  "*   Purpose: compute momentum splitting"
  write (u, "(A)")  "             (massless radiated particle)"
  write (u, "(A)")

  E = 1
  mk = 0.3_default
  mp = 0
  mq = mk

  k = vector4_moving (E, sqrt (E**2 - mk**2), 3)
  k2 = k ** 2;  call pacify (k2, 1e-10_default)

  x = 0.6_default
  r1 = 0.5_default
  r2 = 0.125_default

  write (u, "(A)")  "* (1) Non-collinear setup"
  write (u, "(A)")

  call sd%init (k, mk**2, mp**2, mq**2)
  call sd%set_t_bounds (x, 1 - x)
  call sd%sample_t (r1)
  call sd%sample_phi (r2)

  call sd%write (u)

  q = sd%split_momentum (k)
  q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
  q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

  write (u, "(A)")
  write (u, "(A)")  "Incoming momentum k ="
  call vector4_write (k, u)
  write (u, "(A)")
  write (u, "(A)")  "Outgoing momentum sum p + q ="
  call vector4_write (sum (q), u)
  write (u, "(A)")
```

```
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

write (u, "(A)")  "Compare: u"
write (u, "(2(1x,F11.8))")  sd%u, q1_2

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

write (u, "(A)")  "Compare: 1-x"
write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q0_2 = q0(2) ** 2;  call pacify (q0_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q0_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)")  "Compare: x"
```

```
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t


call sd%inverse_t (r1o)

write (u, "(A)")  "Compare: r1"
write (u, "(2(1x,F11.8))")  r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)")  "Compare: r2"
write (u, "(2(1x,F11.8))")  r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q0_2 = q0(2) ** 2;  call pacify (q0_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q0_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t
```

```
call sd%inverse_t (r1o)

write (u, "(A)")  "Compare: r1"
write (u, "(2(1x,F11.8))")  r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)")  "Compare: r2"
write (u, "(2(1x,F11.8))")  r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* (2) Collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2, collinear = .true.)
call sd%set_t_bounds (x, 1 - x)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

write (u, "(A)")  "Compare: u"
write (u, "(2(1x,F11.8))")  sd%u, q1_2

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

write (u, "(A)")  "Compare: 1-x"
write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)
```

```
write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q0_2 = q0(2) ** 2;  call pacify (q0_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q0_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
```

```
    call vector4_write (q0(1), u)
    write (u, "(A)")
    write (u, "(A)")  "Outgoing momentum q ="
    call vector4_write (q0(2), u)
    write (u, "(A)")

    write (u, "(A)")  "Compare: mo^2"
    q0_2 = q0(2) ** 2;  call pacify (q0_2, 1e-10_default)
    write (u, "(2(1x,F11.8))")  sd%m2, q0_2
    write (u, "(A)")

    write (u, "(A)")  "* Recover parameters from outgoing momentum"
    write (u, "(A)")

    call sd%init (k, mk**2, mp**2, mq**2)
    call sd%recover (k, q0(2), KEEP_MOMENTUM)

    write (u, "(A)")  "Compare: x"
    write (u, "(2(1x,F11.8))")  x, sd%x
    write (u, "(A)")  "Compare: t"
    write (u, "(2(1x,F11.8))")  q2_2, sd%t

    write (u, "(A)")
    call sd%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_aux_1"

  end subroutine sf_aux_1
```

Momentum splitting: massless parton

Compute momentum splitting for generic kinematics. It turns out that for $x = 0.5$, where $t - m^2$ is the geometric mean between its upper and lower bounds, we arrive at an exact number $t = -0.36$ for the given input values.

⟨*SF aux: execute tests*⟩+≡

```
  call test (sf_aux_2, "sf_aux_2", &
       "massless parton", &
       u, results)
```

⟨*SF aux: test declarations*⟩+≡

```
  public :: sf_aux_2
```

⟨*SF aux: tests*⟩+≡

```
  subroutine sf_aux_2 (u)
    integer, intent(in) :: u
    type(splitting_data_t) :: sd
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q, q0
    real(default) :: E, mk, mp, mq
    real(default) :: x, r1, r2, r1o, r2o
    real(default) :: k2, q02_2, q1_2, q2_2
```

```
write (u, "(A)")  "* Test output: sf_aux_2"
write (u, "(A)")  "*   Purpose: compute momentum splitting"
write (u, "(A)")  "             (massless outgoing particle)"
write (u, "(A)")

E = 1
mk = 0.3_default
mp = mk
mq = 0

k = vector4_moving (E, sqrt (E**2 - mk**2), 3)
k2 = k ** 2;  call pacify (k2, 1e-10_default)

x = 0.6_default
r1 = 0.5_default
r2 = 0.125_default

write (u, "(A)")  "* (1) Non-collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%sample_t (r1)
call sd%sample_phi (r2)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

write (u, "(A)")  "Compare: u"
write (u, "(2(1x,F11.8))")  sd%u, q1_2
```

```
write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

write (u, "(A)")  "Compare: 1-x"
write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t


call sd%inverse_t (r1o)

write (u, "(A)")  "Compare: r1"
write (u, "(2(1x,F11.8))")  r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)")  "Compare: r2"
write (u, "(2(1x,F11.8))")  r2, r2o
```

```
write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t

call sd%inverse_t (r1o)

write (u, "(A)")  "Compare: r1"
write (u, "(2(1x,F11.8))")  r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)")  "Compare: r2"
write (u, "(2(1x,F11.8))")  r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
```

```
write (u, "(A)")  "* (2) Collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2, collinear = .true.)
call sd%set_t_bounds (x, 1 - x)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

write (u, "(A)")  "Compare: u"
write (u, "(2(1x,F11.8))")  sd%u, q1_2

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

write (u, "(A)")  "Compare: 1-x"
write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
```

```
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")
```

```
    call sd%init (k, mk**2, mp**2, mq**2)
    call sd%set_t_bounds (x, 1 - x)
    call sd%recover (k, q0(2), KEEP_MOMENTUM)

    write (u, "(A)")  "Compare: x"
    write (u, "(2(1x,F11.8))")  x, sd%x
    write (u, "(A)")  "Compare: t"
    write (u, "(2(1x,F11.8))")  q2_2, sd%t

    write (u, "(A)")
    call sd%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_aux_2"

  end subroutine sf_aux_2
```

Momentum splitting: all massless

Compute momentum splitting for massless kinematics. In the non-collinear case, we need a lower cutoff for $|t|$, otherwise a logarithmic distribution is not possible.

⟨*SF aux: execute tests*⟩+≡

```
  call test (sf_aux_3, "sf_aux_3", &
       "massless parton", &
       u, results)
```

⟨*SF aux: test declarations*⟩+≡

```
  public :: sf_aux_3
```

⟨*SF aux: tests*⟩+≡

```
  subroutine sf_aux_3 (u)
    integer, intent(in) :: u
    type(splitting_data_t) :: sd
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q, q0
    real(default) :: E, mk, mp, mq, qmin, qmax
    real(default) :: x, r1, r2, r1o, r2o
    real(default) :: k2, q02_2, q1_2, q2_2

    write (u, "(A)")  "* Test output: sf_aux_3"
    write (u, "(A)")  "*   Purpose: compute momentum splitting"
    write (u, "(A)")  "             (all massless, q cuts)"
    write (u, "(A)")

    E = 1
    mk = 0
    mp = 0
    mq = 0
    qmin = 1e-2_default
    qmax = 1e0_default
```

```
k = vector4_moving (E, sqrt (E**2 - mk**2), 3)
k2 = k ** 2;  call pacify (k2, 1e-10_default)

x = 0.6_default
r1 = 0.5_default
r2 = 0.125_default

write (u, "(A)")  "* (1) Non-collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%sample_t (r1, t1 = - qmin ** 2, t0 = - qmax **2)
call sd%sample_phi (r2)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

write (u, "(A)")  "Compare: u"
write (u, "(2(1x,F11.8))")  sd%u, q1_2

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

write (u, "(A)")  "Compare: 1-x"
write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep energy)"

q0 = q
```

```
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t


call sd%inverse_t (r1o, t1 = - qmin ** 2, t0 = - qmax **2)

write (u, "(A)")  "Compare: r1"
write (u, "(2(1x,F11.8))")  r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)")  "Compare: r2"
write (u, "(2(1x,F11.8))")  r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
```

```
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t

call sd%inverse_t (r1o, t1 = - qmin ** 2, t0 = - qmax **2)

write (u, "(A)")  "Compare: r1"
write (u, "(2(1x,F11.8))")  r1, r1o

call sd%inverse_phi (r2o)

write (u, "(A)")  "Compare: r2"
write (u, "(2(1x,F11.8))")  r2, r2o

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* (2) Collinear setup"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2, collinear = .true.)
call sd%set_t_bounds (x, 1 - x)

call sd%write (u)

q = sd%split_momentum (k)
q1_2 = q(1) ** 2;  call pacify (q1_2, 1e-10_default)
```

```
q2_2 = q(2) ** 2;  call pacify (q2_2, 1e-10_default)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: s"
write (u, "(2(1x,F11.8))")  sd%s, k2

write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  sd%t, q2_2

write (u, "(A)")  "Compare: u"
write (u, "(2(1x,F11.8))")  sd%u, q1_2

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  sd%x, energy (q(2)) / energy (k)

write (u, "(A)")  "Compare: 1-x"
write (u, "(2(1x,F11.8))")  sd%xb, energy (q(1)) / energy (k)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep energy)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_ENERGY)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
```

```
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_ENERGY)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t

write (u, "(A)")
call sd%write (u)

write (u, "(A)")
write (u, "(A)")  "* Project on-shell (keep momentum)"

q0 = q
call on_shell (q0, [mp**2, mq**2], KEEP_MOMENTUM)

write (u, "(A)")
write (u, "(A)")  "Incoming momentum k ="
call vector4_write (k, u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum sum p + q ="
call vector4_write (sum (q0), u)
write (u, "(A)")
write (u, "(A)")  "Radiated momentum p ="
call vector4_write (q0(1), u)
write (u, "(A)")
write (u, "(A)")  "Outgoing momentum q ="
call vector4_write (q0(2), u)
write (u, "(A)")

write (u, "(A)")  "Compare: mo^2"
q02_2 = q0(2) ** 2;  call pacify (q02_2, 1e-10_default)
write (u, "(2(1x,F11.8))")  sd%m2, q02_2
write (u, "(A)")

write (u, "(A)")  "* Recover parameters from outgoing momentum"
write (u, "(A)")

call sd%init (k, mk**2, mp**2, mq**2)
call sd%set_t_bounds (x, 1 - x)
call sd%recover (k, q0(2), KEEP_MOMENTUM)

write (u, "(A)")  "Compare: x"
write (u, "(2(1x,F11.8))")  x, sd%x
write (u, "(A)")  "Compare: t"
write (u, "(2(1x,F11.8))")  q2_2, sd%t
```

```
    write (u, "(A)")
    call sd%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_aux_3"

  end subroutine sf_aux_3
```

15.4 Mappings for structure functions

In this module, we provide a wrapper for useful mappings of the unit (hyper-)square that we can apply to a set of structure functions.

In some cases it is useful, or even mandatory, to map the MC input parameters nontrivially onto a set of structure functions for the two beams. In all cases considered here, instead of $x_1, x_2, \ldots$ as parameters for the beams, we generate one parameter that is equal, or related to, the product $x_1 x_2 \cdots$ (so it directly corresponds to $\sqrt{s}$). The other parameters describe the distribution of energy (loss) between beams and radiations.

⟨sf_mappings.f90⟩≡

```
⟨File header⟩

module sf_mappings

⟨Use kinds⟩
  use kinds, only: double
  use io_units
  use constants, only: pi, zero, one
  use numeric_utils
  use diagnostics

⟨Standard module head⟩

⟨SF mappings: public⟩

⟨SF mappings: parameters⟩

⟨SF mappings: types⟩

⟨SF mappings: interfaces⟩

contains

⟨SF mappings: procedures⟩

end module sf_mappings
```

15.4.1 Base type

First, we define an abstract base type for the mapping. In all cases we need to store the indices of the parameters on which the mapping applies. Additional

parameters can be stored in the extensions of this type.

⟨*SF mappings: public*⟩≡
```
public :: sf_mapping_t
```

⟨*SF mappings: types*⟩≡
```
type, abstract :: sf_mapping_t
   integer, dimension(:), allocatable :: i
 contains
 ⟨SF mappings: sf mapping: TBP⟩
end type sf_mapping_t
```

The output routine is deferred:

⟨*SF mappings: sf mapping: TBP*⟩≡
```
procedure (sf_mapping_write), deferred :: write
```

⟨*SF mappings: interfaces*⟩≡
```
abstract interface
   subroutine sf_mapping_write (object, unit)
     import
     class(sf_mapping_t), intent(in) :: object
     integer, intent(in), optional :: unit
   end subroutine sf_mapping_write
end interface
```

Initializer for the base type. The array of parameter indices is allocated but initialized to zero.

⟨*SF mappings: sf mapping: TBP*⟩+≡
```
procedure :: base_init => sf_mapping_base_init
```

⟨*SF mappings: procedures*⟩≡
```
subroutine sf_mapping_base_init (mapping, n_par)
  class(sf_mapping_t), intent(out) :: mapping
  integer, intent(in) :: n_par
  allocate (mapping%i (n_par))
  mapping%i = 0
end subroutine sf_mapping_base_init
```

Set an index value.

⟨*SF mappings: sf mapping: TBP*⟩+≡
```
procedure :: set_index => sf_mapping_set_index
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_mapping_set_index (mapping, j, i)
  class(sf_mapping_t), intent(inout) :: mapping
  integer, intent(in) :: j, i
  mapping%i(j) = i
end subroutine sf_mapping_set_index
```

Retrieve an index value.

⟨*SF mappings: sf mapping: TBP*⟩+≡
```
procedure :: get_index => sf_mapping_get_index
```

⟨*SF mappings: procedures*⟩+≡

```
function sf_mapping_get_index (mapping, j) result (i)
  class(sf_mapping_t), intent(inout) :: mapping
  integer, intent(in) :: j
  integer :: i
  i = mapping%i(j)
end function sf_mapping_get_index
```

Return the dimensionality, i.e., the number of parameters.

⟨*SF mappings: sf mapping: TBP*⟩+≡

```
procedure :: get_n_dim => sf_mapping_get_n_dim
```

⟨*SF mappings: procedures*⟩+≡

```
function sf_mapping_get_n_dim (mapping) result (n)
  class(sf_mapping_t), intent(in) :: mapping
  integer :: n
  n = size (mapping%i)
end function sf_mapping_get_n_dim
```

Computation: the values `p` are the input parameters, the values `r` are the output parameters. The values `rb` are defined as $\bar{r} = 1 - r$, but provided explicitly. They allow us to avoid numerical problems near $r = 1$.

The extra parameter `x_free` indicates that the total energy has already been renormalized by this factor. We have to take such a factor into account in a resonance or on-shell mapping.

The Jacobian is `f`. We modify only the two parameters indicated by the indices `i`.

⟨*SF mappings: sf mapping: TBP*⟩+≡

```
procedure (sf_mapping_compute), deferred :: compute
```

⟨*SF mappings: interfaces*⟩+≡

```
abstract interface
   subroutine sf_mapping_compute (mapping, r, rb, f, p, pb, x_free)
     import
     class(sf_mapping_t), intent(inout) :: mapping
     real(default), dimension(:), intent(out) :: r, rb
     real(default), intent(out) :: f
     real(default), dimension(:), intent(in) :: p, pb
     real(default), intent(inout), optional :: x_free
   end subroutine sf_mapping_compute
end interface
```

The inverse mapping. Use `r` and/or `rb` to reconstruct `p` and also compute `f`.

⟨*SF mappings: sf mapping: TBP*⟩+≡

```
procedure (sf_mapping_inverse), deferred :: inverse
```

⟨*SF mappings: interfaces*⟩+≡

```
abstract interface
   subroutine sf_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
     import
     class(sf_mapping_t), intent(inout) :: mapping
     real(default), dimension(:), intent(in) :: r, rb
     real(default), intent(out) :: f
```

```
      real(default), dimension(:), intent(out) :: p, pb
      real(default), intent(inout), optional :: x_free
   end subroutine sf_mapping_inverse
end interface
```

15.4.2 Methods for self-tests

This is a shorthand for: inject parameters, compute the mapping, display results, compute the inverse, display again. We provide an output format for the parameters and, optionally, a different output format for the Jacobians.

⟨*SF mappings: sf mapping: TBP*⟩+≡

```
procedure :: check => sf_mapping_check
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_mapping_check (mapping, u, p_in, pb_in, fmt_p, fmt_f)
  class(sf_mapping_t), intent(inout) :: mapping
  integer, intent(in) :: u
  real(default), dimension(:), intent(in) :: p_in, pb_in
  character(*), intent(in) :: fmt_p
  character(*), intent(in), optional :: fmt_f
  real(default), dimension(size(p_in)) :: p, pb, r, rb
  real(default) :: f, tolerance
  tolerance = 1.5E-17
  p = p_in
  pb= pb_in
  call mapping%compute (r, rb, f, p, pb)
  call pacify (p, tolerance)
  call pacify (pb, tolerance)
  call pacify (r, tolerance)
  call pacify (rb, tolerance)
  write (u, "(3x,A,9(1x," // fmt_p // "))")  "p =", p
  write (u, "(3x,A,9(1x," // fmt_p // "))")  "pb=", pb
  write (u, "(3x,A,9(1x," // fmt_p // "))")  "r =", r
  write (u, "(3x,A,9(1x," // fmt_p // "))")  "rb=", rb
  if (present (fmt_f)) then
     write (u, "(3x,A,9(1x," // fmt_f // "))")  "f =", f
  else
     write (u, "(3x,A,9(1x," // fmt_p // "))")  "f =", f
  end if
  write (u, *)
  call mapping%inverse (r, rb, f, p, pb)
  call pacify (p, tolerance)
  call pacify (pb, tolerance)
  call pacify (r, tolerance)
  call pacify (rb, tolerance)
  write (u, "(3x,A,9(1x," // fmt_p // "))")  "p =", p
  write (u, "(3x,A,9(1x," // fmt_p // "))")  "pb=", pb
  write (u, "(3x,A,9(1x," // fmt_p // "))")  "r =", r
  write (u, "(3x,A,9(1x," // fmt_p // "))")  "rb=", rb
  if (present (fmt_f)) then
     write (u, "(3x,A,9(1x," // fmt_f // "))")  "f =", f
  else
     write (u, "(3x,A,9(1x," // fmt_p // "))")  "f =", f
```

```
    end if
    write (u, *)
    write (u, "(3x,A,9(1x," // fmt_p // "))")  "*r=", product (r)
  end subroutine sf_mapping_check
```

This is a consistency check for the self-tests: the integral over the unit square should be unity. We estimate this by a simple binning and adding up the values; this should be sufficient for a self-test.

The argument is the requested number of sampling points. We take the square root for binning in both dimensions, so the precise number might be different.

⟨*SF mappings: sf mapping: TBP*⟩+≡

```
  procedure :: integral => sf_mapping_integral
```

⟨*SF mappings: procedures*⟩+≡

```
  function sf_mapping_integral (mapping, n_calls) result (integral)
    class(sf_mapping_t), intent(inout) :: mapping
    integer, intent(in) :: n_calls
    real(default) :: integral
    integer :: n_dim, n_bin, k
    real(default), dimension(:), allocatable :: p, pb, r, rb
    integer, dimension(:), allocatable :: ii
    real(default) :: dx, f, s

    n_dim = mapping%get_n_dim ()
    allocate (p (n_dim))
    allocate (pb(n_dim))
    allocate (r (n_dim))
    allocate (rb(n_dim))
    allocate (ii(n_dim))
    n_bin = nint (real (n_calls, default) ** (1._default / n_dim))
    dx = 1._default / n_bin
    s = 0
    ii = 1

    SAMPLE: do
       do k = 1, n_dim
          p(k)  = ii(k) * dx - dx/2
          pb(k) = (n_bin - ii(k)) * dx + dx/2
       end do
       call mapping%compute (r, rb, f, p, pb)
       s = s + f
       INCR: do k = 1, n_dim
          ii(k) = ii(k) + 1
          if (ii(k) <= n_bin) then
             exit INCR
          else if (k < n_dim) then
             ii(k) = 1
          else
             exit SAMPLE
          end if
       end do INCR
    end do SAMPLE
```

```
    integral = s / real (n_bin, default) ** n_dim

  end function sf_mapping_integral
```

15.4.3 Implementation: standard mapping

This maps the unit square (r_1, r_2) such that p_1 is the product $r_1 r_2$, while p_2 is related to the ratio.

⟨*SF mappings: public*⟩+≡

```
  public :: sf_s_mapping_t
```

⟨*SF mappings: types*⟩+≡

```
  type, extends (sf_mapping_t) :: sf_s_mapping_t
     logical :: power_set = .false.
     real(default) :: power = 1
   contains
   ⟨SF mappings: sf standard mapping: TBP⟩
  end type sf_s_mapping_t
```

Output.

⟨*SF mappings: sf standard mapping: TBP*⟩≡

```
  procedure :: write => sf_s_mapping_write
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_s_mapping_write (object, unit)
    class(sf_s_mapping_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(1x,A)", advance="no")  "map"
    if (any (object%i /= 0)) then
       write (u, "('(',I0,',',I0,')')", advance="no")  object%i
    end if
    write (u, "(A,F7.5,A)")  ": standard (", object%power, ")"
  end subroutine sf_s_mapping_write
```

Initialize: index pair and power parameter.

⟨*SF mappings: sf standard mapping: TBP*⟩+≡

```
  procedure :: init => sf_s_mapping_init
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_s_mapping_init (mapping, power)
    class(sf_s_mapping_t), intent(out) :: mapping
    real(default), intent(in), optional :: power
    call mapping%base_init (2)
    if (present (power)) then
       mapping%power_set = .true.
       mapping%power = power
    end if
  end subroutine sf_s_mapping_init
```

Apply mapping.

⟨*SF mappings: sf standard mapping: TBP*⟩+≡

```
procedure :: compute => sf_s_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_s_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_s_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: r2
  integer :: j
  if (mapping%power_set) then
     call map_unit_square (r2, f, p(mapping%i), mapping%power)
  else
     call map_unit_square (r2, f, p(mapping%i))
  end if
  r = p
  rb= pb
  do j = 1, 2
     r (mapping%i(j)) = r2(j)
     rb(mapping%i(j)) = 1 - r2(j)
  end do
end subroutine sf_s_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf standard mapping: TBP*⟩+≡

```
procedure :: inverse => sf_s_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_s_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
  class(sf_s_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(in) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: p2
  integer :: j
  if (mapping%power_set) then
     call map_unit_square_inverse (r(mapping%i), f, p2, mapping%power)
  else
     call map_unit_square_inverse (r(mapping%i), f, p2)
  end if
  p = r
  pb= rb
  do j = 1, 2
     p (mapping%i(j)) = p2(j)
     pb(mapping%i(j)) = 1 - p2(j)
  end do
end subroutine sf_s_mapping_inverse
```

15.4.4 Implementation: resonance pair mapping

This maps the unit square (r_1, r_2) such that p_1 is the product $r_1 r_2$, while p_2 is related to the ratio, then it maps p_1 to itself according to a Breit-Wigner shape, i.e., a flat prior distribution in p_1 results in a Breit-Wigner distribution. Mass and width of the BW are rescaled by the energy, thus dimensionless fractions.

⟨*SF mappings: public*⟩+≡
```
public :: sf_res_mapping_t
```

⟨*SF mappings: types*⟩+≡
```
type, extends (sf_mapping_t) :: sf_res_mapping_t
   real(default) :: m = 0
   real(default) :: w = 0
 contains
 ⟨SF mappings: sf resonance mapping: TBP⟩
end type sf_res_mapping_t
```

Output.

⟨*SF mappings: sf resonance mapping: TBP*⟩≡
```
procedure :: write => sf_res_mapping_write
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_res_mapping_write (object, unit)
  class(sf_res_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)", advance="no")  "map"
  if (any (object%i /= 0)) then
     write (u, "('(',I0,',',I0,')')", advance="no")  object%i
  end if
  write (u, "(A,F7.5,', ',F7.5,A)")  ": resonance (", object%m, object%w, ")"
end subroutine sf_res_mapping_write
```

Initialize: index pair and dimensionless mass and width parameters.

⟨*SF mappings: sf resonance mapping: TBP*⟩+≡
```
procedure :: init => sf_res_mapping_init
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_res_mapping_init (mapping, m, w)
  class(sf_res_mapping_t), intent(out) :: mapping
  real(default), intent(in) :: m, w
  call mapping%base_init (2)
  mapping%m = m
  mapping%w = w
end subroutine sf_res_mapping_init
```

Apply mapping.

⟨*SF mappings: sf resonance mapping: TBP*⟩+≡
```
procedure :: compute => sf_res_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_res_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_res_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: r2, p2
  real(default) :: fbw, f2, p1m
  integer :: j
  p2 = p(mapping%i)
  call map_breit_wigner &
       (p1m, fbw, p2(1), mapping%m, mapping%w, x_free)
  call map_unit_square (r2, f2, [p1m, p2(2)])
  f = fbw * f2
  r = p
  rb= pb
  do j = 1, 2
     r (mapping%i(j)) = r2(j)
     rb(mapping%i(j)) = 1 - r2(j)
  end do
end subroutine sf_res_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf resonance mapping: TBP*⟩+≡

```
procedure :: inverse => sf_res_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_res_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
  class(sf_res_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(in) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: p2
  real(default) :: fbw, f2, p1m
  call map_unit_square_inverse (r(mapping%i), f2, p2)
  call map_breit_wigner_inverse &
       (p2(1), fbw, p1m, mapping%m, mapping%w, x_free)
  p = r
  pb= rb
  p (mapping%i(1)) = p1m
  pb(mapping%i(1)) = 1 - p1m
  p (mapping%i(2)) = p2(2)
  pb(mapping%i(2)) = 1 - p2(2)
  f = fbw * f2
end subroutine sf_res_mapping_inverse
```

15.4.5 Implementation: resonance single mapping

While simpler, this is needed for structure-function setups only in exceptional cases.

This maps the unit interval (r_1) to itself according to a Breit-Wigner shape, i.e., a flat prior distribution in r_1 results in a Breit-Wigner distribution. Mass and width of the BW are rescaled by the energy, thus dimensionless fractions.

⟨*SF mappings: public*⟩+≡
```
public :: sf_res_mapping_single_t
```

⟨*SF mappings: types*⟩+≡
```
type, extends (sf_mapping_t) :: sf_res_mapping_single_t
   real(default) :: m = 0
   real(default) :: w = 0
 contains
 ⟨SF mappings: sf resonance single mapping: TBP⟩
end type sf_res_mapping_single_t
```

Output.

⟨*SF mappings: sf resonance single mapping: TBP*⟩≡
```
procedure :: write => sf_res_mapping_single_write
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_res_mapping_single_write (object, unit)
  class(sf_res_mapping_single_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)", advance="no")  "map"
  if (any (object%i /= 0)) then
     write (u, "(’(’,I0,’)’)", advance="no")  object%i
  end if
  write (u, "(A,F7.5,’, ’,F7.5,A)")  ": resonance (", object%m, object%w, ")"
end subroutine sf_res_mapping_single_write
```

Initialize: single index (!) and dimensionless mass and width parameters.

⟨*SF mappings: sf resonance single mapping: TBP*⟩+≡
```
procedure :: init => sf_res_mapping_single_init
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_res_mapping_single_init (mapping, m, w)
  class(sf_res_mapping_single_t), intent(out) :: mapping
  real(default), intent(in) :: m, w
  call mapping%base_init (1)
  mapping%m = m
  mapping%w = w
end subroutine sf_res_mapping_single_init
```

Apply mapping.

⟨*SF mappings: sf resonance single mapping: TBP*⟩+≡
```
procedure :: compute => sf_res_mapping_single_compute
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_res_mapping_single_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_res_mapping_single_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
```

```
    real(default), intent(inout), optional :: x_free
    real(default), dimension(1) :: r2, p2
    real(default) :: fbw
    integer :: j
    p2 = p(mapping%i)
    call map_breit_wigner &
         (r2(1), fbw, p2(1), mapping%m, mapping%w, x_free)
    f = fbw
    r = p
    rb= pb
    r (mapping%i(1)) = r2(1)
    rb(mapping%i(1)) = 1 - r2(1)
  end subroutine sf_res_mapping_single_compute
```

Apply inverse.

⟨*SF mappings: sf resonance single mapping: TBP*⟩+≡

```
  procedure :: inverse => sf_res_mapping_single_inverse
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_res_mapping_single_inverse (mapping, r, rb, f, p, pb, x_free)
    class(sf_res_mapping_single_t), intent(inout) :: mapping
    real(default), dimension(:), intent(in) :: r, rb
    real(default), intent(out) :: f
    real(default), dimension(:), intent(out) :: p, pb
    real(default), intent(inout), optional :: x_free
    real(default), dimension(1) :: p2
    real(default) :: fbw
    call map_breit_wigner_inverse &
         (r(mapping%i(1)), fbw, p2(1), mapping%m, mapping%w, x_free)
    p = r
    pb= rb
    p (mapping%i(1)) = p2(1)
    pb(mapping%i(1)) = 1 - p2(1)
    f = fbw
  end subroutine sf_res_mapping_single_inverse
```

15.4.6 Implementation: on-shell mapping

This is a degenerate version of the unit-square mapping where the product $r_1 r_2$ is constant. This product is given by the rescaled squared mass. We introduce an artificial first parameter p_1 to keep the counting, but nothing depends on it. The second parameter is the same p_2 as for the standard unit-square mapping for $\alpha = 1$, it parameterizes the ratio of r_1 and r_2.

⟨*SF mappings: public*⟩+≡

```
  public :: sf_os_mapping_t
```

⟨*SF mappings: types*⟩+≡

```
  type, extends (sf_mapping_t) :: sf_os_mapping_t
     real(default) :: m = 0
     real(default) :: lm2 = 0
   contains
```

⟨*SF mappings: sf on-shell mapping: TBP*⟩

```
end type sf_os_mapping_t
```

Output.

⟨*SF mappings: sf on-shell mapping: TBP*⟩≡

```
procedure :: write => sf_os_mapping_write
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_os_mapping_write (object, unit)
  class(sf_os_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)", advance="no")  "map"
  if (any (object%i /= 0)) then
     write (u, "(’(’,I0,’,’,I0,’)’)", advance="no")  object%i
  end if
  write (u, "(A,F7.5,A)")  ": on-shell (", object%m, ")"
end subroutine sf_os_mapping_write
```

Initialize: index pair and dimensionless mass parameter.

⟨*SF mappings: sf on-shell mapping: TBP*⟩+≡

```
procedure :: init => sf_os_mapping_init
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_os_mapping_init (mapping, m)
  class(sf_os_mapping_t), intent(out) :: mapping
  real(default), intent(in) :: m
  call mapping%base_init (2)
  mapping%m = m
  mapping%lm2 = abs (2 * log (mapping%m))
end subroutine sf_os_mapping_init
```

Apply mapping. The `x_free` parameter rescales the total energy, which must be accounted for in the enclosed mapping.

⟨*SF mappings: sf on-shell mapping: TBP*⟩+≡

```
procedure :: compute => sf_os_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_os_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_os_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: r2, p2
  integer :: j
  p2 = p(mapping%i)
  call map_on_shell (r2, f, p2, mapping%lm2, x_free)
  r = p
  rb= pb
  do j = 1, 2
     r (mapping%i(j)) = r2(j)
     rb(mapping%i(j)) = 1 - r2(j)
```

```
    end do
  end subroutine sf_os_mapping_compute
```

Apply inverse. The irrelevant parameter p_1 is always set zero.

⟨*SF mappings: sf on-shell mapping: TBP*⟩+≡

```
  procedure :: inverse => sf_os_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_os_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
    class(sf_os_mapping_t), intent(inout) :: mapping
    real(default), dimension(:), intent(in) :: r, rb
    real(default), intent(out) :: f
    real(default), dimension(:), intent(out) :: p, pb
    real(default), intent(inout), optional :: x_free
    real(default), dimension(2) :: p2, r2
    r2 = r(mapping%i)
    call map_on_shell_inverse (r2, f, p2, mapping%lm2, x_free)
    p = r
    pb= rb
    p (mapping%i(1)) = p2(1)
    pb(mapping%i(1)) = 1 - p2(1)
    p (mapping%i(2)) = p2(2)
    pb(mapping%i(2)) = 1 - p2(2)
  end subroutine sf_os_mapping_inverse
```

15.4.7 Implementation: on-shell single mapping

This is a degenerate version of the unit-interval mapping where the result r is constant. The value is given by the rescaled squared mass. The input parameter p_1 is actually ignored, nothing depends on it.

⟨*SF mappings: public*⟩+≡

```
  public :: sf_os_mapping_single_t
```

⟨*SF mappings: types*⟩+≡

```
  type, extends (sf_mapping_t) :: sf_os_mapping_single_t
     real(default) :: m = 0
     real(default) :: lm2 = 0
   contains
   ⟨SF mappings: sf on-shell mapping single: TBP⟩
  end type sf_os_mapping_single_t
```

Output.

⟨*SF mappings: sf on-shell mapping single: TBP*⟩≡

```
  procedure :: write => sf_os_mapping_single_write
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_os_mapping_single_write (object, unit)
    class(sf_os_mapping_single_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(1x,A)", advance="no")  "map"
```

```
    if (any (object%i /= 0)) then
       write (u, "('(',I0,')')", advance="no")  object%i
    end if
    write (u, "(A,F7.5,A)")  ": on-shell (", object%m, ")"
  end subroutine sf_os_mapping_single_write
```

Initialize: index pair and dimensionless mass parameter.

⟨*SF mappings: sf on-shell mapping single: TBP*⟩+≡

```
  procedure :: init => sf_os_mapping_single_init
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_os_mapping_single_init (mapping, m)
    class(sf_os_mapping_single_t), intent(out) :: mapping
    real(default), intent(in) :: m
    call mapping%base_init (1)
    mapping%m = m
    mapping%lm2 = abs (2 * log (mapping%m))
  end subroutine sf_os_mapping_single_init
```

Apply mapping. The `x_free` parameter rescales the total energy, which must be accounted for in the enclosed mapping.

⟨*SF mappings: sf on-shell mapping single: TBP*⟩+≡

```
  procedure :: compute => sf_os_mapping_single_compute
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_os_mapping_single_compute (mapping, r, rb, f, p, pb, x_free)
    class(sf_os_mapping_single_t), intent(inout) :: mapping
    real(default), dimension(:), intent(out) :: r, rb
    real(default), intent(out) :: f
    real(default), dimension(:), intent(in) :: p, pb
    real(default), intent(inout), optional :: x_free
    real(default), dimension(1) :: r2, p2
    integer :: j
    p2 = p(mapping%i)
    call map_on_shell_single (r2, f, p2, mapping%lm2, x_free)
    r = p
    rb= pb
    r (mapping%i(1)) = r2(1)
    rb(mapping%i(1)) = 1 - r2(1)
  end subroutine sf_os_mapping_single_compute
```

Apply inverse. The irrelevant parameter p_1 is always set zero.

⟨*SF mappings: sf on-shell mapping single: TBP*⟩+≡

```
  procedure :: inverse => sf_os_mapping_single_inverse
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_os_mapping_single_inverse (mapping, r, rb, f, p, pb, x_free)
    class(sf_os_mapping_single_t), intent(inout) :: mapping
    real(default), dimension(:), intent(in) :: r, rb
    real(default), intent(out) :: f
    real(default), dimension(:), intent(out) :: p, pb
    real(default), intent(inout), optional :: x_free
    real(default), dimension(1) :: p2, r2
```

```
    r2 = r(mapping%i)
    call map_on_shell_single_inverse (r2, f, p2, mapping%lm2, x_free)
    p = r
    pb= rb
    p (mapping%i(1)) = p2(1)
    pb(mapping%i(1)) = 1 - p2(1)
  end subroutine sf_os_mapping_single_inverse
```

15.4.8 Implementation: endpoint mapping

This maps the unit square (r_1, r_2) such that p_1 is the product $r_1 r_2$, while p_2 is related to the ratio. Furthermore, we enhance the region at $r_1 = 1$ and $r_2 = 1$, which translates into $p_1 = 1$ and $p_2 = 0, 1$. The enhancement is such that any power-like singularity is caught. This is useful for beamstrahlung spectra.

In addition, we allow for a delta-function singularity in r_1 and/or r_2. The singularity is smeared to an interval of width ϵ. If nonzero, we distinguish the kinematical momentum fractions r_i from effective values x_i, which should go into the structure-function evaluation. A bin of width ϵ in r is mapped to $x = 1$ exactly, while the interval $(0, 1-\epsilon)$ is mapped to $(0, 1)$ in x. The Jacobian reflects this distinction, and the logical `in_peak` allows for an unambiguous distinction.

The delta-peak fraction is used only for the integration self-test.

⟨*SF mappings: public*⟩+≡
```
  public :: sf_ep_mapping_t
```

⟨*SF mappings: types*⟩+≡
```
  type, extends (sf_mapping_t) :: sf_ep_mapping_t
     real(default) :: a = 1
   contains
   ⟨SF mappings: sf endpoint mapping: TBP⟩
  end type sf_ep_mapping_t
```

Output.

⟨*SF mappings: sf endpoint mapping: TBP*⟩≡
```
  procedure :: write => sf_ep_mapping_write
```

⟨*SF mappings: procedures*⟩+≡
```
  subroutine sf_ep_mapping_write (object, unit)
    class(sf_ep_mapping_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(1x,A)", advance="no")  "map"
    if (any (object%i /= 0)) then
       write (u, "('(',I0,',',I0,')')", advance="no")  object%i
    end if
    write (u, "(A,ES12.5,A)")  ": endpoint (a =", object%a, ")"
  end subroutine sf_ep_mapping_write
```

Initialize: no extra parameters.

⟨*SF mappings: sf endpoint mapping: TBP*⟩+≡
```
  procedure :: init => sf_ep_mapping_init
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ep_mapping_init (mapping, a)
  class(sf_ep_mapping_t), intent(out) :: mapping
  real(default), intent(in), optional :: a
  call mapping%base_init (2)
  if (present (a))  mapping%a = a
end subroutine sf_ep_mapping_init
```

Apply mapping.

⟨*SF mappings: sf endpoint mapping: TBP*⟩+≡

```
procedure :: compute => sf_ep_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ep_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_ep_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: px, r2
  real(default) :: f1, f2
  integer :: j
  call map_endpoint_1 (px(1), f1, p(mapping%i(1)), mapping%a)
  call map_endpoint_01 (px(2), f2, p(mapping%i(2)), mapping%a)
  call map_unit_square (r2, f, px)
  f = f * f1 * f2
  r = p
  rb= pb
  do j = 1, 2
     r (mapping%i(j)) = r2(j)
     rb(mapping%i(j)) = 1 - r2(j)
  end do
end subroutine sf_ep_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf endpoint mapping: TBP*⟩+≡

```
procedure :: inverse => sf_ep_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ep_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
  class(sf_ep_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(in) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: r2, px, p2
  real(default) :: f1, f2
  integer :: j
  do j = 1, 2
     r2(j) = r(mapping%i(j))
  end do
  call map_unit_square_inverse (r2, f, px)
  call map_endpoint_inverse_1 (px(1), f1, p2(1), mapping%a)
  call map_endpoint_inverse_01 (px(2), f2, p2(2), mapping%a)
```

```
    f = f * f1 * f2
    p = r
    pb= rb
    do j = 1, 2
       p (mapping%i(j)) = p2(j)
       pb(mapping%i(j)) = 1 - p2(j)
    end do
  end subroutine sf_ep_mapping_inverse
```

15.4.9 Implementation: endpoint mapping with resonance

Like the endpoint mapping for p_2, but replace the endpoint mapping by a Breit-Wigner mapping for p_1. This covers resonance production in the presence of beamstrahlung.

If the flag `resonance` is unset, we skip the resonance mapping, so the parameter p_1 remains equal to $r_1 r_2$, as in the standard s-channel mapping.

⟨*SF mappings: public*⟩+≡

```
  public :: sf_epr_mapping_t
```

⟨*SF mappings: types*⟩+≡

```
  type, extends (sf_mapping_t) :: sf_epr_mapping_t
     real(default) :: a = 1
     real(default) :: m = 0
     real(default) :: w = 0
     logical :: resonance = .true.
   contains
   ⟨SF mappings: sf endpoint/res mapping: TBP⟩
  end type sf_epr_mapping_t
```

Output.

⟨*SF mappings: sf endpoint/res mapping: TBP*⟩≡

```
  procedure :: write => sf_epr_mapping_write
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_epr_mapping_write (object, unit)
    class(sf_epr_mapping_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(1x,A)", advance="no")  "map"
    if (any (object%i /= 0)) then
       write (u, "(’(’,I0,’,’,I0,’)’)", advance="no")  object%i
    end if
    if (object%resonance) then
       write (u, "(A,F7.5,A,F7.5,’, ’,F7.5,A)")  ": ep/res (a = ", object%a, &
            " | ", object%m, object%w, ")"
    else
       write (u, "(A,F7.5,A)")  ": ep/nores (a = ", object%a, ")"
    end if
  end subroutine sf_epr_mapping_write
```

Initialize: if mass and width are not given, we initialize a non-resonant version of the mapping.

⟨*SF mappings: sf endpoint/res mapping: TBP*⟩+≡

```
procedure :: init => sf_epr_mapping_init
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_epr_mapping_init (mapping, a, m, w)
  class(sf_epr_mapping_t), intent(out) :: mapping
  real(default), intent(in) :: a
  real(default), intent(in), optional :: m, w
  call mapping%base_init (2)
  mapping%a = a
  if (present (m) .and. present (w)) then
     mapping%m = m
     mapping%w = w
  else
     mapping%resonance = .false.
  end if
end subroutine sf_epr_mapping_init
```

Apply mapping.

⟨*SF mappings: sf endpoint/res mapping: TBP*⟩+≡

```
procedure :: compute => sf_epr_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_epr_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_epr_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: px, r2
  real(default) :: f1, f2
  integer :: j
  if (mapping%resonance) then
     call map_breit_wigner &
          (px(1), f1, p(mapping%i(1)), mapping%m, mapping%w, x_free)
  else
     px(1) = p(mapping%i(1))
     f1 = 1
  end if
  call map_endpoint_01 (px(2), f2, p(mapping%i(2)), mapping%a)
  call map_unit_square (r2, f, px)
  f = f * f1 * f2
  r = p
  rb= pb
  do j = 1, 2
     r (mapping%i(j)) = r2(j)
     rb(mapping%i(j)) = 1 - r2(j)
  end do
end subroutine sf_epr_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf endpoint/res mapping: TBP*⟩+≡

```
procedure :: inverse => sf_epr_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_epr_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
  class(sf_epr_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(in) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: px, p2
  real(default) :: f1, f2
  integer :: j
  call map_unit_square_inverse (r(mapping%i), f, px)
  if (mapping%resonance) then
     call map_breit_wigner_inverse &
          (px(1), f1, p2(1), mapping%m, mapping%w, x_free)
  else
     p2(1) = px(1)
     f1 = 1
  end if
  call map_endpoint_inverse_01 (px(2), f2, p2(2), mapping%a)
  f = f * f1 * f2
  p = r
  pb= rb
  do j = 1, 2
     p (mapping%i(j)) = p2(j)
     pb(mapping%i(j)) = 1 - p2(j)
  end do
end subroutine sf_epr_mapping_inverse
```

15.4.10 Implementation: endpoint mapping for on-shell particle

Analogous to the resonance mapping, but the p_1 input is ignored altogether. This covers on-shell particle production in the presence of beamstrahlung.

⟨*SF mappings: public*⟩+≡

```
public :: sf_epo_mapping_t
```

⟨*SF mappings: types*⟩+≡

```
type, extends (sf_mapping_t) :: sf_epo_mapping_t
   real(default) :: a = 1
   real(default) :: m = 0
   real(default) :: lm2 = 0
 contains
 ⟨SF mappings: sf endpoint/os mapping: TBP⟩
end type sf_epo_mapping_t
```

Output.

⟨*SF mappings: sf endpoint/os mapping: TBP*⟩≡

```
procedure :: write => sf_epo_mapping_write
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_epo_mapping_write (object, unit)
  class(sf_epo_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)", advance="no")  "map"
  if (any (object%i /= 0)) then
     write (u, "('(',I0,',',I0,')')", advance="no")  object%i
  end if
  write (u, "(A,F7.5,A,F7.5,A)")  ": ep/on-shell (a = ", object%a, &
       " | ", object%m, ")"
end subroutine sf_epo_mapping_write
```

Initialize: no extra parameters.

⟨*SF mappings: sf endpoint/os mapping: TBP*⟩+≡

```
procedure :: init => sf_epo_mapping_init
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_epo_mapping_init (mapping, a, m)
  class(sf_epo_mapping_t), intent(out) :: mapping
  real(default), intent(in) :: a, m
  call mapping%base_init (2)
  mapping%a = a
  mapping%m = m
  mapping%lm2 = abs (2 * log (mapping%m))
end subroutine sf_epo_mapping_init
```

Apply mapping.

⟨*SF mappings: sf endpoint/os mapping: TBP*⟩+≡

```
procedure :: compute => sf_epo_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_epo_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_epo_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: px, r2
  real(default) :: f2
  integer :: j
  px(1) = 0
  call map_endpoint_01 (px(2), f2, p(mapping%i(2)), mapping%a)
  call map_on_shell (r2, f, px, mapping%lm2)
  f = f * f2
  r = p
  rb= pb
  do j = 1, 2
     r (mapping%i(j)) = r2(j)
     rb(mapping%i(j)) = 1 - r2(j)
  end do
end subroutine sf_epo_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf endpoint/os mapping: TBP*⟩+≡

```
procedure :: inverse => sf_epo_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_epo_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
  class(sf_epo_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(in) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: px, p2
  real(default) :: f2
  integer :: j
  call map_on_shell_inverse (r(mapping%i), f, px, mapping%lm2)
  p2(1) = 0
  call map_endpoint_inverse_01 (px(2), f2, p2(2), mapping%a)
  f = f * f2
  p = r
  pb= rb
  do j = 1, 2
     p (mapping%i(j)) = p2(j)
     pb(mapping%i(j)) = 1 - p2(j)
  end do
end subroutine sf_epo_mapping_inverse
```

15.4.11 Implementation: ISR endpoint mapping

Similar to the endpoint mapping above: This maps the unit square (r_1, r_2) such that p_1 is the product $r_1 r_2$, while p_2 is related to the ratio. Furthermore, we enhance the region at $r_1 = 1$ and $r_2 = 1$, which translates into $p_1 = 1$ and $p_2 = 0, 1$.

The enhancement is such that ISR singularity $(1-x)^{-1+\epsilon}$ is flattened. This would be easy in one dimension, but becomes nontrivial in two dimensions.

⟨*SF mappings: public*⟩+≡

```
public :: sf_ip_mapping_t
```

⟨*SF mappings: types*⟩+≡

```
type, extends (sf_mapping_t) :: sf_ip_mapping_t
   real(default) :: eps = 0
 contains
 ⟨SF mappings: sf power mapping: TBP⟩
end type sf_ip_mapping_t
```

Output.

⟨*SF mappings: sf power mapping: TBP*⟩≡

```
procedure :: write => sf_ip_mapping_write
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ip_mapping_write (object, unit)
  class(sf_ip_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
```

```
    integer :: u
    u = given_output_unit (unit)
    write (u, "(1x,A)", advance="no")  "map"
    if (any (object%i /= 0)) then
       write (u, "(’(’,I0,’,’,I0,’)’)", advance="no")  object%i
    end if
    write (u, "(A,ES12.5,A)")  ": isr (eps =", object%eps, ")"
  end subroutine sf_ip_mapping_write
```

Initialize: no extra parameters.

⟨*SF mappings: sf power mapping: TBP*⟩+≡

```
  procedure :: init => sf_ip_mapping_init
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_ip_mapping_init (mapping, eps)
    class(sf_ip_mapping_t), intent(out) :: mapping
    real(default), intent(in), optional :: eps
    call mapping%base_init (2)
    if (present (eps))  mapping%eps = eps
    if (mapping%eps <= 0) &
         call msg_fatal ("ISR mapping: regulator epsilon must not be zero")
  end subroutine sf_ip_mapping_init
```

Apply mapping.

⟨*SF mappings: sf power mapping: TBP*⟩+≡

```
  procedure :: compute => sf_ip_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_ip_mapping_compute (mapping, r, rb, f, p, pb, x_free)
    class(sf_ip_mapping_t), intent(inout) :: mapping
    real(default), dimension(:), intent(out) :: r, rb
    real(default), intent(out) :: f
    real(default), dimension(:), intent(in) :: p, pb
    real(default), intent(inout), optional :: x_free
    real(default), dimension(2) :: px, pxb, r2, r2b
    real(default) :: f1, f2, xb, y, yb
    integer :: j
    call map_power_1 (xb, f1, pb(mapping%i(1)), 2 * mapping%eps)
    call map_power_01 (y, yb, f2, pb(mapping%i(2)), mapping%eps)
    px(1)  = 1 - xb
    pxb(1) = xb
    px(2)  = y
    pxb(2) = yb
    call map_unit_square_prec (r2, r2b, f, px, pxb)
    f = f * f1 * f2
    r = p
    rb= pb
    do j = 1, 2
       r (mapping%i(j)) = r2 (j)
       rb(mapping%i(j)) = r2b(j)
    end do
  end subroutine sf_ip_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf power mapping: TBP*⟩+≡

```
procedure :: inverse => sf_ip_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ip_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
  class(sf_ip_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(in) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: r2, r2b, px, pxb, p2, p2b
  real(default) :: f1, f2, xb, y, yb
  integer :: j
  do j = 1, 2
     r2 (j) = r (mapping%i(j))
     r2b(j) = rb(mapping%i(j))
  end do
  call map_unit_square_inverse_prec (r2, r2b, f, px, pxb)
  xb = pxb(1)
  if (px(1) > 0) then
     y  = px(2)
     yb = pxb(2)
  else
     y  = 0.5_default
     yb = 0.5_default
  end if
  call map_power_inverse_1 (xb, f1, p2b(1), 2 * mapping%eps)
  call map_power_inverse_01 (y, yb, f2, p2b(2), mapping%eps)
  p2 = 1 - p2b
  f = f * f1 * f2
  p = r
  pb= rb
  do j = 1, 2
     p (mapping%i(j)) = p2(j)
     pb(mapping%i(j)) = p2b(j)
  end do
end subroutine sf_ip_mapping_inverse
```

15.4.12 Implementation: ISR endpoint mapping, resonant

Similar to the endpoint mapping above: This maps the unit square (r_1, r_2) such that p_1 is the product $r_1 r_2$, while p_2 is related to the ratio. Furthermore, we enhance the region at $r_1 = 1$ and $r_2 = 1$, which translates into $p_1 = 1$ and $p_2 = 0, 1$.

The enhancement is such that ISR singularity $(1-x)^{-1+\epsilon}$ is flattened. This would be easy in one dimension, but becomes nontrivial in two dimensions.

The resonance can be turned off by the flag `resonance`.

⟨*SF mappings: public*⟩+≡

```
public :: sf_ipr_mapping_t
```

⟨*SF mappings: types*⟩+≡

```
type, extends (sf_mapping_t) :: sf_ipr_mapping_t
   real(default) :: eps = 0
   real(default) :: m = 0
   real(default) :: w = 0
   logical :: resonance = .true.
 contains
 ⟨SF mappings: sf power/res mapping: TBP⟩
end type sf_ipr_mapping_t
```

Output.

⟨*SF mappings: sf power/res mapping: TBP*⟩≡

```
procedure :: write => sf_ipr_mapping_write
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ipr_mapping_write (object, unit)
  class(sf_ipr_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)", advance="no")  "map"
  if (any (object%i /= 0)) then
     write (u, "('(',I0,',',I0,')')", advance="no")  object%i
  end if
  if (object%resonance) then
     write (u, "(A,F7.5,A,F7.5,', ',F7.5,A)")  ": isr/res (eps = ", &
          object%eps, " | ", object%m, object%w, ")"
  else
     write (u, "(A,F7.5,A)")  ": isr/res (eps = ", object%eps, ")"
  end if
end subroutine sf_ipr_mapping_write
```

Initialize:

⟨*SF mappings: sf power/res mapping: TBP*⟩+≡

```
procedure :: init => sf_ipr_mapping_init
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ipr_mapping_init (mapping, eps, m, w)
  class(sf_ipr_mapping_t), intent(out) :: mapping
  real(default), intent(in), optional :: eps, m, w
  call mapping%base_init (2)
  if (present (eps))  mapping%eps = eps
  if (mapping%eps <= 0) &
       call msg_fatal ("ISR mapping: regulator epsilon must not be zero")
  if (present (m) .and. present (w)) then
     mapping%m = m
     mapping%w = w
  else
     mapping%resonance = .false.
  end if
end subroutine sf_ipr_mapping_init
```

Apply mapping.

⟨*SF mappings: sf power/res mapping: TBP*⟩+≡

```
procedure :: compute => sf_ipr_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ipr_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_ipr_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: px, pxb, r2, r2b
  real(default) :: f1, f2, y, yb
  integer :: j
  if (mapping%resonance) then
     call map_breit_wigner &
          (px(1), f1, p(mapping%i(1)), mapping%m, mapping%w, x_free)
  else
     px(1) = p(mapping%i(1))
     f1 = 1
  end if
  call map_power_01 (y, yb, f2, pb(mapping%i(2)), mapping%eps)
  pxb(1) = 1 - px(1)
  px(2)  = y
  pxb(2) = yb
  call map_unit_square_prec (r2, r2b, f, px, pxb)
  f = f * f1 * f2
  r = p
  rb= pb
  do j = 1, 2
     r (mapping%i(j)) = r2 (j)
     rb(mapping%i(j)) = r2b(j)
  end do
end subroutine sf_ipr_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf power/res mapping: TBP*⟩+≡

```
procedure :: inverse => sf_ipr_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ipr_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
  class(sf_ipr_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(in) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: r2, r2b, px, pxb, p2, p2b
  real(default) :: f1, f2, y, yb
  integer :: j
  do j = 1, 2
     r2 (j) = r (mapping%i(j))
     r2b(j) = rb(mapping%i(j))
  end do
  call map_unit_square_inverse_prec (r2, r2b, f, px, pxb)
```

```
    if (px(1) > 0) then
       y  = px(2)
       yb = pxb(2)
    else
       y  = 0.5_default
       yb = 0.5_default
    end if
    if (mapping%resonance) then
       call map_breit_wigner_inverse &
            (px(1), f1, p2(1), mapping%m, mapping%w, x_free)
    else
       p2(1) = px(1)
       f1 = 1
    end if
    call map_power_inverse_01 (y, yb, f2, p2b(2), mapping%eps)
    p2b(1) = 1 - p2(1)
    p2 (2) = 1 - p2b(2)
    f = f * f1 * f2
    p = r
    pb= rb
    do j = 1, 2
       p (mapping%i(j)) = p2(j)
       pb(mapping%i(j)) = p2b(j)
    end do
  end subroutine sf_ipr_mapping_inverse
```

15.4.13 Implementation: ISR on-shell mapping

Similar to the endpoint mapping above: This maps the unit square (r_1, r_2) such that p_1 is ignored while the product $r_1 r_2$ is constant. p_2 is related to the ratio. Furthermore, we enhance the region at $r_1 = 1$ and $r_2 = 1$, which translates into $p_1 = 1$ and $p_2 = 0, 1$.

The enhancement is such that ISR singularity $(1-x)^{-1+\epsilon}$ is flattened. This would be easy in one dimension, but becomes nontrivial in two dimensions.

⟨*SF mappings: public*⟩+≡

```
  public :: sf_ipo_mapping_t
```

⟨*SF mappings: types*⟩+≡

```
  type, extends (sf_mapping_t) :: sf_ipo_mapping_t
     real(default) :: eps = 0
     real(default) :: m = 0
   contains
   ⟨SF mappings: sf power/os mapping: TBP⟩
  end type sf_ipo_mapping_t
```

Output.

⟨*SF mappings: sf power/os mapping: TBP*⟩≡

```
  procedure :: write => sf_ipo_mapping_write
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_ipo_mapping_write (object, unit)
    class(sf_ipo_mapping_t), intent(in) :: object
```

```
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(1x,A)", advance="no")  "map"
    if (any (object%i /= 0)) then
       write (u, "('(',I0,',',I0,')')", advance="no")  object%i
    end if
    write (u, "(A,F7.5,A,F7.5,A)")  ": isr/os (eps = ", object%eps, &
         " | ", object%m, ")"
  end subroutine sf_ipo_mapping_write
```

Initialize: no extra parameters.

⟨*SF mappings: sf power/os mapping: TBP*⟩+≡

```
  procedure :: init => sf_ipo_mapping_init
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_ipo_mapping_init (mapping, eps, m)
    class(sf_ipo_mapping_t), intent(out) :: mapping
    real(default), intent(in), optional :: eps, m
    call mapping%base_init (2)
    if (present (eps))  mapping%eps = eps
    if (mapping%eps <= 0) &
         call msg_fatal ("ISR mapping: regulator epsilon must not be zero")
    mapping%m = m
  end subroutine sf_ipo_mapping_init
```

Apply mapping.

⟨*SF mappings: sf power/os mapping: TBP*⟩+≡

```
  procedure :: compute => sf_ipo_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_ipo_mapping_compute (mapping, r, rb, f, p, pb, x_free)
    class(sf_ipo_mapping_t), intent(inout) :: mapping
    real(default), dimension(:), intent(out) :: r, rb
    real(default), intent(out) :: f
    real(default), dimension(:), intent(in) :: p, pb
    real(default), intent(inout), optional :: x_free
    real(default), dimension(2) :: px, pxb, r2, r2b
    real(default) :: f1, f2, y, yb
    integer :: j
    call map_power_01 (y, yb, f2, pb(mapping%i(2)), mapping%eps)
    px(1)  = mapping%m ** 2
    if (present (x_free))  px(1) = px(1) / x_free
    pxb(1) = 1 - px(1)
    px(2)  = y
    pxb(2) = yb
    call map_unit_square_prec (r2, r2b, f1, px, pxb)
    f = f1 * f2
    r = p
    rb= pb
    do j = 1, 2
       r (mapping%i(j)) = r2 (j)
       rb(mapping%i(j)) = r2b(j)
    end do
```

```
end subroutine sf_ipo_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf power/os mapping: TBP*⟩+≡

```
procedure :: inverse => sf_ipo_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ipo_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
  class(sf_ipo_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(in) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(2) :: r2, r2b, px, pxb, p2, p2b
  real(default) :: f1, f2, y, yb
  integer :: j
  do j = 1, 2
     r2 (j) = r (mapping%i(j))
     r2b(j) = rb(mapping%i(j))
  end do
  call map_unit_square_inverse_prec (r2, r2b, f1, px, pxb)
  y  = px(2)
  yb = pxb(2)
  call map_power_inverse_01 (y, yb, f2, p2b(2), mapping%eps)
  p2(1) = 0
  p2b(1)= 1
  p2(2) = 1 - p2b(2)
  f = f1 * f2
  p = r
  pb= rb
  do j = 1, 2
     p (mapping%i(j)) = p2(j)
     pb(mapping%i(j)) = p2b(j)
  end do
end subroutine sf_ipo_mapping_inverse
```

15.4.14 Implementation: Endpoint + ISR power mapping

This is a combination of endpoint (i.e., beamstrahlung) and ISR power mapping. The first two parameters apply to the beamstrahlung spectrum, the last two to the ISR function for the first and second beam, respectively.

⟨*SF mappings: public*⟩+≡

```
public :: sf_ei_mapping_t
```

⟨*SF mappings: types*⟩+≡

```
type, extends (sf_mapping_t) :: sf_ei_mapping_t
   type(sf_ep_mapping_t) :: ep
   type(sf_ip_mapping_t) :: ip
 contains
 ⟨SF mappings: sf ep-ip mapping: TBP⟩
end type sf_ei_mapping_t
```

Output.

⟨*SF mappings: sf ep-ip mapping: TBP*⟩≡

```
procedure :: write => sf_ei_mapping_write
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ei_mapping_write (object, unit)
  class(sf_ei_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)", advance="no")  "map"
  if (any (object%i /= 0)) then
     write (u, "(’(’,I0,3(’,’,I0),’)’)", advance="no")  object%i
  end if
  write (u, "(A,ES12.5,A,ES12.5,A)")  ": ep/isr (a =", object%ep%a, &
       ", eps =", object%ip%eps, ")"
end subroutine sf_ei_mapping_write
```

Initialize: no extra parameters.

⟨*SF mappings: sf ep-ip mapping: TBP*⟩+≡

```
procedure :: init => sf_ei_mapping_init
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ei_mapping_init (mapping, a, eps)
  class(sf_ei_mapping_t), intent(out) :: mapping
  real(default), intent(in), optional :: a, eps
  call mapping%base_init (4)
  call mapping%ep%init (a)
  call mapping%ip%init (eps)
end subroutine sf_ei_mapping_init
```

Set an index value. We should communicate the appropriate indices to the enclosed sub-mappings, therefore override the method.

⟨*SF mappings: sf ep-ip mapping: TBP*⟩+≡

```
procedure :: set_index => sf_ei_mapping_set_index
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ei_mapping_set_index (mapping, j, i)
  class(sf_ei_mapping_t), intent(inout) :: mapping
  integer, intent(in) :: j, i
  mapping%i(j) = i
  select case (j)
  case (1:2);  call mapping%ep%set_index (j, i)
  case (3:4);  call mapping%ip%set_index (j-2, i)
  end select
end subroutine sf_ei_mapping_set_index
```

Apply mapping. Now, the beamstrahlung and ISR mappings are independent of each other. The parameter subsets that are actually used should not overlap. The Jacobians are multiplied.

⟨*SF mappings: sf ep-ip mapping: TBP*⟩+≡

```
procedure :: compute => sf_ei_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ei_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_ei_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(size(p)) :: q, qb
  real(default) :: f1, f2
  call mapping%ep%compute (q, qb, f1, p, pb, x_free)
  call mapping%ip%compute (r, rb, f2, q, qb, x_free)
  f = f1 * f2
end subroutine sf_ei_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf ep-ip mapping: TBP*⟩+≡

```
procedure :: inverse => sf_ei_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_ei_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
  class(sf_ei_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(in) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(size(p)) :: q, qb
  real(default) :: f1, f2
  call mapping%ip%inverse (r, rb, f2, q, qb, x_free)
  call mapping%ep%inverse (q, qb, f1, p, pb, x_free)
  f = f1 * f2
end subroutine sf_ei_mapping_inverse
```

15.4.15 Implementation: Endpoint + ISR + resonance

This is a combination of endpoint (i.e., beamstrahlung) and ISR power mapping, adapted for an s-channel resonance. The first two internal parameters apply to the beamstrahlung spectrum, the last two to the ISR function for the first and second beam, respectively. The first and third parameters are the result of an overall resonance mapping, so on the outside, the first parameter is the total momentum fraction, the third one describes the distribution between beamstrahlung and ISR.

⟨*SF mappings: public*⟩+≡

```
public :: sf_eir_mapping_t
```

⟨*SF mappings: types*⟩+≡

```
type, extends (sf_mapping_t) :: sf_eir_mapping_t
   type(sf_res_mapping_t) :: res
   type(sf_epr_mapping_t) :: ep
   type(sf_ipr_mapping_t) :: ip
 contains
 ⟨SF mappings: sf ep-ip-res mapping: TBP⟩
end type sf_eir_mapping_t
```

Output.

⟨*SF mappings: sf ep-ip-res mapping: TBP*⟩≡

```
procedure :: write => sf_eir_mapping_write
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_eir_mapping_write (object, unit)
  class(sf_eir_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)", advance="no")  "map"
  if (any (object%i /= 0)) then
     write (u, "(’(’,I0,3(’,’,I0),’)’)", advance="no")  object%i
  end if
  write (u, "(A,F7.5,A,F7.5,A,F7.5,’, ’,F7.5,A)")  &
       ": ep/isr/res (a =", object%ep%a, &
       ", eps =", object%ip%eps, " | ", object%res%m, object%res%w, ")"
end subroutine sf_eir_mapping_write
```

Initialize: no extra parameters.

⟨*SF mappings: sf ep-ip-res mapping: TBP*⟩+≡

```
procedure :: init => sf_eir_mapping_init
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_eir_mapping_init (mapping, a, eps, m, w)
  class(sf_eir_mapping_t), intent(out) :: mapping
  real(default), intent(in) :: a, eps, m, w
  call mapping%base_init (4)
  call mapping%res%init (m, w)
  call mapping%ep%init (a)
  call mapping%ip%init (eps)
end subroutine sf_eir_mapping_init
```

Set an index value. We should communicate the appropriate indices to the enclosed sub-mappings, therefore override the method.

⟨*SF mappings: sf ep-ip-res mapping: TBP*⟩+≡

```
procedure :: set_index => sf_eir_mapping_set_index
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_eir_mapping_set_index (mapping, j, i)
  class(sf_eir_mapping_t), intent(inout) :: mapping
  integer, intent(in) :: j, i
  mapping%i(j) = i
  select case (j)
  case (1);  call mapping%res%set_index (1, i)
  case (3);  call mapping%res%set_index (2, i)
  end select
  select case (j)
  case (1:2);  call mapping%ep%set_index (j, i)
  case (3:4);  call mapping%ip%set_index (j-2, i)
  end select
end subroutine sf_eir_mapping_set_index
```

Apply mapping. Now, the beamstrahlung and ISR mappings are independent of each other. The parameter subsets that are actually used should not overlap. The Jacobians are multiplied.

⟨*SF mappings: sf ep-ip-res mapping: TBP*⟩+≡
```
procedure :: compute => sf_eir_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_eir_mapping_compute (mapping, r, rb, f, p, pb, x_free)
  class(sf_eir_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(size(p)) :: px, pxb, q, qb
  real(default) :: f0, f1, f2
  call mapping%res%compute (px, pxb, f0, p, pb, x_free)
  call mapping%ep%compute (q, qb, f1, px, pxb, x_free)
  call mapping%ip%compute (r, rb, f2, q, qb, x_free)
  f = f0 * f1 * f2
end subroutine sf_eir_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf ep-ip-res mapping: TBP*⟩+≡
```
procedure :: inverse => sf_eir_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_eir_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
  class(sf_eir_mapping_t), intent(inout) :: mapping
  real(default), dimension(:), intent(in) :: r, rb
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: p, pb
  real(default), intent(inout), optional :: x_free
  real(default), dimension(size(p)) :: px, pxb, q, qb
  real(default) :: f0, f1, f2
  call mapping%ip%inverse (r, rb, f2, q, qb, x_free)
  call mapping%ep%inverse (q, qb, f1, px, pxb, x_free)
  call mapping%res%inverse (px, pxb, f0, p, pb, x_free)
  f = f0 * f1 * f2
end subroutine sf_eir_mapping_inverse
```

15.4.16 Implementation: Endpoint + ISR power mapping, on-shell

This is a combination of endpoint (i.e., beamstrahlung) and ISR power mapping. The first two parameters apply to the beamstrahlung spectrum, the last two to the ISR function for the first and second beam, respectively. On top of that, we map the first and third parameter such that the product is constant. From the outside, the first parameter is irrelevant while the third parameter describes the distribution of energy (loss) among beamstrahlung and ISR.

⟨*SF mappings: public*⟩+≡
```
public :: sf_eio_mapping_t
```

⟨*SF mappings: types*⟩+≡

```
type, extends (sf_mapping_t) :: sf_eio_mapping_t
   type(sf_os_mapping_t) :: os
   type(sf_epr_mapping_t) :: ep
   type(sf_ipr_mapping_t) :: ip
 contains
 ⟨SF mappings: sf ep-ip-os mapping: TBP⟩
end type sf_eio_mapping_t
```

Output.

⟨*SF mappings: sf ep-ip-os mapping: TBP*⟩≡

```
procedure :: write => sf_eio_mapping_write
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_eio_mapping_write (object, unit)
  class(sf_eio_mapping_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)", advance="no")  "map"
  if (any (object%i /= 0)) then
     write (u, "(’(’,I0,3(’,’,I0),’)’)", advance="no")  object%i
  end if
  write (u, "(A,F7.5,A,F7.5,A,F7.5,A)")  ": ep/isr/os (a =", object%ep%a, &
       ", eps =", object%ip%eps, " | ", object%os%m, ")"
end subroutine sf_eio_mapping_write
```

Initialize: no extra parameters.

⟨*SF mappings: sf ep-ip-os mapping: TBP*⟩+≡

```
procedure :: init => sf_eio_mapping_init
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_eio_mapping_init (mapping, a, eps, m)
  class(sf_eio_mapping_t), intent(out) :: mapping
  real(default), intent(in), optional :: a, eps, m
  call mapping%base_init (4)
  call mapping%os%init (m)
  call mapping%ep%init (a)
  call mapping%ip%init (eps)
end subroutine sf_eio_mapping_init
```

Set an index value. We should communicate the appropriate indices to the enclosed sub-mappings, therefore override the method.

⟨*SF mappings: sf ep-ip-os mapping: TBP*⟩+≡

```
procedure :: set_index => sf_eio_mapping_set_index
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_eio_mapping_set_index (mapping, j, i)
  class(sf_eio_mapping_t), intent(inout) :: mapping
  integer, intent(in) :: j, i
  mapping%i(j) = i
  select case (j)
  case (1);  call mapping%os%set_index (1, i)
```

```
    case (3);  call mapping%os%set_index (2, i)
    end select
    select case (j)
    case (1:2);  call mapping%ep%set_index (j, i)
    case (3:4);  call mapping%ip%set_index (j-2, i)
    end select
  end subroutine sf_eio_mapping_set_index
```

Apply mapping. Now, the beamstrahlung and ISR mappings are independent of each other. The parameter subsets that are actually used should not overlap. The Jacobians are multiplied.

⟨*SF mappings: sf ep-ip-os mapping: TBP*⟩+≡

```
  procedure :: compute => sf_eio_mapping_compute
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_eio_mapping_compute (mapping, r, rb, f, p, pb, x_free)
    class(sf_eio_mapping_t), intent(inout) :: mapping
    real(default), dimension(:), intent(out) :: r, rb
    real(default), intent(out) :: f
    real(default), dimension(:), intent(in) :: p, pb
    real(default), intent(inout), optional :: x_free
    real(default), dimension(size(p)) :: px, pxb, q, qb
    real(default) :: f0, f1, f2
    call mapping%os%compute (px, pxb, f0, p, pb, x_free)
    call mapping%ep%compute (q, qb, f1, px, pxb, x_free)
    call mapping%ip%compute (r, rb, f2, q, qb, x_free)
    f = f0 * f1 * f2
  end subroutine sf_eio_mapping_compute
```

Apply inverse.

⟨*SF mappings: sf ep-ip-os mapping: TBP*⟩+≡

```
  procedure :: inverse => sf_eio_mapping_inverse
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_eio_mapping_inverse (mapping, r, rb, f, p, pb, x_free)
    class(sf_eio_mapping_t), intent(inout) :: mapping
    real(default), dimension(:), intent(in) :: r, rb
    real(default), intent(out) :: f
    real(default), dimension(:), intent(out) :: p, pb
    real(default), intent(inout), optional :: x_free
    real(default), dimension(size(p)) :: px, pxb, q, qb
    real(default) :: f0, f1, f2
    call mapping%ip%inverse (r, rb, f2, q, qb, x_free)
    call mapping%ep%inverse (q, qb, f1, px, pxb, x_free)
    call mapping%os%inverse (px, pxb, f0, p, pb, x_free)
    f = f0 * f1 * f2
  end subroutine sf_eio_mapping_inverse
```

15.4.17 Basic formulas

Standard mapping of the unit square

This mapping of the unit square is appropriate in particular for structure functions which are concentrated at the lower end. Instead of a rectangular grid, one set of grid lines corresponds to constant parton c.m. energy. The other set is chosen such that the jacobian is only mildly singular ($\ln x$ which is zero at $x = 1$), corresponding to an initial concentration of sampling points at the maximum energy. If `power` is greater than one (the default), points are also concentrated at the lower end.

The formula is (`power`$=\alpha$):

$$r_1 = (p_1^{p_2})^\alpha \tag{15.15}$$

$$r_2 = (p_1^{1-p_2})^\alpha \tag{15.16}$$

$$f = \alpha^2 p_1^{\alpha-1} |\log p_1| \tag{15.17}$$

and for the default case $\alpha = 1$:

$$r_1 = p_1^{p_2} \tag{15.18}$$

$$r_2 = p_1^{1-p_2} \tag{15.19}$$

$$f = |\log p_1| \tag{15.20}$$

⟨*SF mappings: procedures*⟩+≡

```
subroutine map_unit_square (r, factor, p, power)
  real(default), dimension(2), intent(out) :: r
  real(default), intent(out) :: factor
  real(default), dimension(2), intent(in) :: p
  real(default), intent(in), optional :: power
  real(default) :: xx, yy
  factor = 1
  xx = p(1)
  yy = p(2)
  if (present(power)) then
     if (p(1) > 0 .and. power > 1) then
        xx = p(1)**power
        factor = factor * power * xx / p(1)
     end if
  end if
  if (.not. vanishes (xx)) then
     r(1) = xx ** yy
     r(2) = xx / r(1)
     factor = factor * abs (log (xx))
  else
     r = 0
  end if
end subroutine map_unit_square
```

This is the inverse mapping.

⟨*SF mappings: procedures*⟩+≡

```
subroutine map_unit_square_inverse (r, factor, p, power)
  real(kind=default), dimension(2), intent(in) :: r
```

```
    real(kind=default), intent(out) :: factor
    real(kind=default), dimension(2), intent(out) :: p
    real(kind=default), intent(in), optional :: power
    real(kind=default) :: lg, xx, yy
    factor = 1
    xx = r(1) * r(2)
    if (.not. vanishes (xx)) then
       lg = log (xx)
       if (.not. vanishes (lg)) then
          yy = log (r(1)) / lg
       else
          yy = 0
       end if
       p(2) = yy
       factor = factor * abs (lg)
       if (present(power)) then
          p(1) = xx**(1._default/power)
          factor = factor * power * xx / p(1)
       else
          p(1) = xx
       end if
    else
       p = 0
    end if
  end subroutine map_unit_square_inverse
```

Precise mapping of the unit square

A more precise version (with unit power parameter). This version should be numerically stable near $x = 1$ and $y = 0, 1$. The formulas are again

$$r_1 = p_1^{p_2}, \qquad r_2 = p_1^{\bar{p}_2}, \qquad f = -\log p_1 \tag{15.21}$$

but we compute both r_i and $\bar{r}_i$ simultaneously and make direct use of both p_i and $\bar{p}_i$ as appropriate.

⟨*SF mappings: procedures*⟩+≡

```
  subroutine map_unit_square_prec (r, rb, factor, p, pb)
    real(default), dimension(2), intent(out) :: r
    real(default), dimension(2), intent(out) :: rb
    real(default), intent(out) :: factor
    real(default), dimension(2), intent(in) :: p
    real(default), dimension(2), intent(in) :: pb
    if (p(1) > 0.5_default) then
       call compute_prec_xy_1 (r(1), rb(1), p(1), pb(1), p (2))
       call compute_prec_xy_1 (r(2), rb(2), p(1), pb(1), pb(2))
       factor = - log_prec (p(1), pb(1))
    else if (.not. vanishes (p(1))) then
       call compute_prec_xy_0 (r(1), rb(1), p(1), pb(1), p (2))
       call compute_prec_xy_0 (r(2), rb(2), p(1), pb(1), pb(2))
       factor = - log_prec (p(1), pb(1))
    else
       r  = 0
       rb = 1
```

```
      factor = 0
    end if
  end subroutine map_unit_square_prec
```

This is the inverse mapping.

⟨*SF mappings: procedures*⟩+≡

```
  subroutine map_unit_square_inverse_prec (r, rb, factor, p, pb)
    real(default), dimension(2), intent(in) :: r
    real(default), dimension(2), intent(in) :: rb
    real(default), intent(out) :: factor
    real(default), dimension(2), intent(out) :: p
    real(default), dimension(2), intent(out) :: pb
    call inverse_prec_x (r, rb, p(1), pb(1))
    if (all (r > 0)) then
       if (rb(1) < rb(2)) then
          call inverse_prec_y (r, rb, p(2), pb(2))
       else
          call inverse_prec_y ([r(2),r(1)], [rb(2),rb(1)], pb(2), p(2))
       end if
       factor = - log_prec (p(1), pb(1))
    else
       p(1)  = 0
       pb(1) = 1
       p(2)  = 0.5_default
       pb(2) = 0.5_default
       factor = 0
    end if
  end subroutine map_unit_square_inverse_prec
```

This is an auxiliary function: evaluate the expression $\bar{z} = 1 - x^y$ in a numerically stable way. Instabilities occur for $y = 0$ and $x = 1$. The idea is to replace the bracket by the first terms of its Taylor expansion around $x = 1$ (read $\bar{x} \equiv 1 - x$)

$$1 - x^y = y\bar{x}\left(1 + \frac{1}{2}(1-y)\bar{x} + \frac{1}{6}(2-y)(1-y)\bar{x}^2\right) \tag{15.22}$$

whenever this is the better approximation. Actually, the relative numerical error of the exact formula is about $\eta/(y\bar{x})$ where η is given by `epsilon(KIND)` in Fortran. The relative error of the approximation is better than the last included term divided by $(y\bar{x})$.

The first subroutine computes z and $\bar{z}$ near $x = 1$ where $\log x$ should be expanded, the second one near $x = 0$ where $\log x$ can be kept.

⟨*SF mappings: procedures*⟩+≡

```
  subroutine compute_prec_xy_1 (z, zb, x, xb, y)
    real(default), intent(out) :: z, zb
    real(default), intent(in) :: x, xb, y
    real(default) :: a1, a2, a3
    a1 = y * xb
    a2 = a1 * (1 - y) * xb / 2
    a3 = a2 * (2 - y) * xb / 3
    if (abs (a3) < epsilon (a3)) then
       zb = a1 + a2 + a3
       z = 1 - zb
```

```
  else
     z = x ** y
     zb = 1 - z
  end if
end subroutine compute_prec_xy_1

subroutine compute_prec_xy_0 (z, zb, x, xb, y)
  real(default), intent(out) :: z, zb
  real(default), intent(in) :: x, xb, y
  real(default) :: a1, a2, a3, lx
  lx = -log (x)
  a1 = y * lx
  a2 = a1 * y * lx / 2
  a3 = a2 * y * lx / 3
  if (abs (a3) < epsilon (a3)) then
     zb = a1 + a2 + a3
     z = 1 - zb
  else
     z = x ** y
     zb = 1 - z
  end if
end subroutine compute_prec_xy_0
```

For the inverse calculation, we evaluate $x = r_1 r_2$ in a stable way. Since it is just a polynomial, the expansion near $x = 1$ is analytically exact, and we don't need to choose based on precision.

⟨*SF mappings: procedures*⟩+≡

```
subroutine inverse_prec_x (r, rb, x, xb)
  real(default), dimension(2), intent(in) :: r, rb
  real(default), intent(out) :: x, xb
  real(default) :: a0, a1
  a0 = rb(1) + rb(2)
  a1 = rb(1) * rb(2)
  if (a0 > 0.5_default) then
     xb = a0 - a1
     x = 1 - xb
  else
     x = r(1) * r(2)
     xb = 1 - x
  end if
end subroutine inverse_prec_x
```

The inverse calculation for the relative momentum fraction

$$y = \frac{1}{1 + \frac{\log r_2}{\log r_1}} \tag{15.23}$$

is slightly more complicated. We should take the precise form of the logarithm, so we are safe near $r_i = 1$. A series expansion is required if $r_1 \ll r_2$, since then y becomes small. (We assume $r_1 < r_2$ here; for the opposite case, the arguments can be exchanged.)

⟨*SF mappings: procedures*⟩+≡

```
subroutine inverse_prec_y (r, rb, y, yb)
```

```
    real(default), dimension(2), intent(in) :: r, rb
    real(default), intent(out) :: y, yb
    real(default) :: log1, log2, a1, a2, a3
    log1 = log_prec (r(1), rb(1))
    log2 = log_prec (r(2), rb(2))
    if (abs (log2**3) < epsilon (one)) then
       if (abs(log1) < epsilon (one)) then
          y = zero
       else
          y = one / (one + log2 / log1)
       end if
       if (abs(log2) < epsilon (one)) then
          yb = zero
       else
          yb = one / (one + log1 / log2)
       end if
       return
    end if
    a1 = - rb(1) / log2
    a2 = - rb(1) ** 2 * (one / log2**2 + one / (2 * log2))
    a3 = - rb(1) ** 3 * (one / log2**3 + one / log2**2 + one/(3 * log2))
    if (abs (a3) < epsilon (a3)) then
       y  = a1 + a2 + a3
       yb = one - y
    else
       y  = one / (one + log2 / log1)
       yb = one / (one + log1 / log2)
    end if
  end subroutine inverse_prec_y
```

We also need an evaluation of $\log x$ which is stable near $x = 1$.

⟨*SF mappings: public*⟩+≡

```
  public :: log_prec
```

⟨*SF mappings: procedures*⟩+≡

```
  function log_prec (x, xb) result (lx)
    real(default), intent(in) :: x, xb
    real(default) :: a1, a2, a3, lx
    a1 = xb
    a2 = a1 * xb / 2
    a3 = a2 * xb * 2 / 3
    if (abs (a3) < epsilon (a3)) then
       lx = - a1 - a2 - a3
    else
       lx = log (x)
    end if
  end function log_prec
```

Mapping for on-shell s-channel

The limiting case, if the product $r_1 r_2$ is fixed for on-shell production. The parameter p_1 is ignored. In the inverse mapping, it is returned zero.

The parameter x_free, if present, rescales the total energy. If it is less than one, the rescaled mass parameter m^2 should be increased accordingly.

Public for access in unit test.

⟨*SF mappings: public*⟩+≡
```
public :: map_on_shell
public :: map_on_shell_inverse
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine map_on_shell (r, factor, p, lm2, x_free)
  real(default), dimension(2), intent(out) :: r
  real(default), intent(out) :: factor
  real(default), dimension(2), intent(in) :: p
  real(default), intent(in) :: lm2
  real(default), intent(in), optional :: x_free
  real(default) :: lx
  lx = lm2;  if (present (x_free))  lx = lx + log (x_free)
  r(1) = exp (- p(2) * lx)
  r(2) = exp (- (1 - p(2)) * lx)
  factor = lx
end subroutine map_on_shell

subroutine map_on_shell_inverse (r, factor, p, lm2, x_free)
  real(default), dimension(2), intent(in) :: r
  real(default), intent(out) :: factor
  real(default), dimension(2), intent(out) :: p
  real(default), intent(in) :: lm2
  real(default), intent(in), optional :: x_free
  real(default) :: lx
  lx = lm2;  if (present (x_free))  lx = lx + log (x_free)
  p(1) = 0
  p(2) = abs (log (r(1))) / lx
  factor = lx
end subroutine map_on_shell_inverse
```

Mapping for on-shell s-channel, single parameter

This is a pseudo-mapping which applies if there is actually just one parameter p. The output parameter r is fixed for on-shell production. The lone parameter p_1 is ignored. In the inverse mapping, it is returned zero.

The parameter x_free, if present, rescales the total energy. If it is less than one, the rescaled mass parameter m^2 should be increased accordingly.

Public for access in unit test.

⟨*SF mappings: public*⟩+≡
```
public :: map_on_shell_single
public :: map_on_shell_single_inverse
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine map_on_shell_single (r, factor, p, lm2, x_free)
  real(default), dimension(1), intent(out) :: r
  real(default), intent(out) :: factor
  real(default), dimension(1), intent(in) :: p
  real(default), intent(in) :: lm2
  real(default), intent(in), optional :: x_free
```

```
  real(default) :: lx
  lx = lm2;  if (present (x_free))  lx = lx + log (x_free)
  r(1) = exp (- lx)
  factor = 1
end subroutine map_on_shell_single

subroutine map_on_shell_single_inverse (r, factor, p, lm2, x_free)
  real(default), dimension(1), intent(in) :: r
  real(default), intent(out) :: factor
  real(default), dimension(1), intent(out) :: p
  real(default), intent(in) :: lm2
  real(default), intent(in), optional :: x_free
  real(default) :: lx
  lx = lm2;  if (present (x_free))  lx = lx + log (x_free)
  p(1) = 0
  factor = 1
end subroutine map_on_shell_single_inverse
```

Mapping for a Breit-Wigner resonance

This is the standard Breit-Wigner mapping. We apply it to a single variable, independently of or in addition to a unit-square mapping. We assume here that the limits for the variable are 0 and 1, and that the mass m and width w are rescaled appropriately, so they are dimensionless and usually between 0 and 1.

If `x_free` is set, it rescales the total energy and thus mass and width, since these are defined with respect to the total energy.

⟨*SF mappings: procedures*⟩+≡

```
subroutine map_breit_wigner (r, factor, p, m, w, x_free)
  real(default), intent(out) :: r
  real(default), intent(out) :: factor
  real(default), intent(in) :: p
  real(default), intent(in) :: m
  real(default), intent(in) :: w
  real(default), intent(in), optional :: x_free
  real(default) :: m2, mw, a1, a2, a3, z, tmp
  m2 = m ** 2
  mw = m * w
  if (present (x_free)) then
     m2 = m2 / x_free
     mw = mw / x_free
  end if
  a1 = atan (- m2 / mw)
  a2 = atan ((1 - m2) / mw)
  a3 = (a2 - a1) * mw
  z = (1-p) * a1 + p * a2
  if (-pi/2 < z .and. z < pi/2) then
     tmp = tan (z)
     r = max (m2 + mw * tmp, 0._default)
     factor = a3 * (1 + tmp ** 2)
  else
     r = 0
     factor = 0
```

```
  end if
end subroutine map_breit_wigner

subroutine map_breit_wigner_inverse (r, factor, p, m, w, x_free)
  real(default), intent(in) :: r
  real(default), intent(out) :: factor
  real(default), intent(out) :: p
  real(default), intent(in) :: m
  real(default), intent(in) :: w
  real(default) :: m2, mw, a1, a2, a3, tmp
  real(default), intent(in), optional :: x_free
  m2 = m ** 2
  mw = m * w
  if (present (x_free)) then
     m2 = m2 / x_free
     mw = mw / x_free
  end if
  a1 = atan (- m2 / mw)
  a2 = atan ((1 - m2) / mw)
  a3 = (a2 - a1) * mw
  tmp = (r - m2) / mw
  p = (atan (tmp) - a1) / (a2 - a1)
  factor = a3 * (1 + tmp ** 2)
end subroutine map_breit_wigner_inverse
```

Mapping with endpoint enhancement

This is a mapping which is close to the unit mapping, except that at the endpoint(s), the output values are exponentially enhanced.

$$y = \tanh(a \tan(\frac{\pi}{2} x)) \tag{15.24}$$

We have two variants: one covers endpoints at 0 and 1 symmetrically, while the other one (which essentially maps one-half of the range), covers only the endpoint at 1.

⟨*SF mappings: procedures*⟩+≡

```
subroutine map_endpoint_1 (x3, factor, x1, a)
  real(default), intent(out) :: x3, factor
  real(default), intent(in) :: x1
  real(default), intent(in) :: a
  real(default) :: x2
  if (abs (x1) < 1) then
     x2 = tan (x1 * pi / 2)
     x3 = tanh (a * x2)
     factor = a * pi/2 * (1 + x2 ** 2) * (1 - x3 ** 2)
  else
     x3 = x1
     factor = 0
  end if
end subroutine map_endpoint_1

subroutine map_endpoint_inverse_1 (x3, factor, x1, a)
```

```
  real(default), intent(in) :: x3
  real(default), intent(out) :: x1, factor
  real(default), intent(in) :: a
  real(default) :: x2
  if (abs (x3) < 1) then
     x2 = atanh (x3) / a
     x1 = 2 / pi * atan (x2)
     factor = a * pi/2 * (1 + x2 ** 2) * (1 - x3 ** 2)
  else
     x1 = x3
     factor = 0
  end if
end subroutine map_endpoint_inverse_1

subroutine map_endpoint_01 (x4, factor, x0, a)
  real(default), intent(out) :: x4, factor
  real(default), intent(in) :: x0
  real(default), intent(in) :: a
  real(default) :: x1, x3
  x1 = 2 * x0 - 1
  call map_endpoint_1 (x3, factor, x1, a)
  x4 = (x3 + 1) / 2
end subroutine map_endpoint_01

subroutine map_endpoint_inverse_01 (x4, factor, x0, a)
  real(default), intent(in) :: x4
  real(default), intent(out) :: x0, factor
  real(default), intent(in) :: a
  real(default) :: x1, x3
  x3 = 2 * x4 - 1
  call map_endpoint_inverse_1 (x3, factor, x1, a)
  x0 = (x1 + 1) / 2
end subroutine map_endpoint_inverse_01
```

Mapping with endpoint enhancement (ISR)

This is another endpoint mapping. It is designed to flatten the ISR singularity which is of power type at $x = 1$, i.e., if

$$\sigma = \int_0^1 dx\, f(x)\, G(x) = \int_0^1 dx\, \epsilon(1-x)^{-1+\epsilon} G(x), \tag{15.25}$$

we replace this by

$$r = x^\epsilon \quad \Longrightarrow \quad \sigma = \int_0^1 dr\, G(1-(1-r)^{1/\epsilon}). \tag{15.26}$$

We expect that ϵ is small.

The actual mapping is $r \to x$ (so x emerges closer to 1). The Jacobian that we return is thus $1/f(x)$. We compute the mapping in terms of $\bar{x} \equiv 1 - x$, so we can achieve the required precision. Because some compilers show quite wild numeric fluctuations, we internally convert numeric types to explicit `double` precision.

⟨*SF mappings: public*⟩+≡

```
public :: map_power_1
public :: map_power_inverse_1
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine map_power_1 (xb, factor, rb, eps)
  real(default), intent(out) :: xb, factor
  real(default), intent(in) :: rb
  real(double) :: rb_db, factor_db, eps_db, xb_db
  real(default), intent(in) :: eps
  rb_db = real (rb, kind=double)
  eps_db = real (eps, kind=double)
  xb_db = rb_db ** (1 / eps_db)
  if (rb_db > 0) then
     factor_db = xb_db / rb_db / eps_db
     factor = real (factor_db, kind=default)
  else
     factor = 0
  end if
  xb = real (xb_db, kind=default)
end subroutine map_power_1

subroutine map_power_inverse_1 (xb, factor, rb, eps)
  real(default), intent(in) :: xb
  real(default), intent(out) :: rb, factor
  real(double) :: xb_db, factor_db, eps_db, rb_db
  real(default), intent(in) :: eps
  xb_db = real (xb, kind=double)
  eps_db = real (eps, kind=double)
  rb_db = xb_db ** eps_db
  if (xb_db > 0) then
     factor_db = xb_db / rb_db / eps_db
     factor = real (factor_db, kind=default)
  else
     factor = 0
  end if
  rb = real (rb_db, kind=default)
end subroutine map_power_inverse_1
```

Here we apply a power mapping to both endpoints. We divide the interval in two equal halves and apply the power mapping for the nearest endpoint, either 0 or 1.

⟨*SF mappings: procedures*⟩+≡

```
subroutine map_power_01 (y, yb, factor, r, eps)
  real(default), intent(out) :: y, yb, factor
  real(default), intent(in) :: r
  real(default), intent(in) :: eps
  real(default) :: u, ub, zp, zm
  u = 2 * r - 1
  if (u > 0) then
     ub = 2 * (1 - r)
     call map_power_1 (zm, factor, ub, eps)
     zp = 2 - zm
  else if (u < 0) then
     ub = 2 * r
```

```
      call map_power_1 (zp, factor, ub, eps)
      zm = 2 - zp
   else
      factor = 1 / eps
      zp = 1
      zm = 1
   end if
   y  = zp / 2
   yb = zm / 2
end subroutine map_power_01

subroutine map_power_inverse_01 (y, yb, factor, r, eps)
   real(default), intent(in) :: y, yb
   real(default), intent(out) :: r, factor
   real(default), intent(in) :: eps
   real(default) :: ub, zp, zm
   zp = 2 * y
   zm = 2 * yb
   if (zm < zp) then
      call map_power_inverse_1 (zm, factor, ub, eps)
      r = 1 - ub / 2
   else if (zp < zm) then
      call map_power_inverse_1 (zp, factor, ub, eps)
      r = ub / 2
   else
      factor = 1 / eps
      ub = 1
      r = ub / 2
   end if
end subroutine map_power_inverse_01
```

Structure-function channels

A structure-function chain parameterization (channel) may contain a mapping that applies to multiple structure functions. This is described by an extension of the `sf_mapping_t` type. In addition, it may contain mappings that apply to (other) individual structure functions. The details of these mappings are implementation-specific.

The `sf_channel_t` type combines this information. It contains an array of map codes, one for each structure-function entry. The code values are:

none MC input parameters r directly become energy fractions x

single default mapping for a single structure-function entry

multi/s map $r \to x$ such that one MC input parameter is $\hat{s}/s$

multi/resonance as before, adapted to s-channel resonance

multi/on-shell as before, adapted to an on-shell particle in the s channel

multi/endpoint like multi/s, but enhance the region near $r_i = 1$

multi/endpoint/res endpoint mapping with resonance

multi/endpoint/os endpoint mapping for on-shell

multi/power/os like multi/endpoint, regulating a power singularity

⟨*SF mappings: parameters*⟩≡
```
integer, parameter :: SFMAP_NONE = 0
integer, parameter :: SFMAP_SINGLE = 1
integer, parameter :: SFMAP_MULTI_S = 2
integer, parameter :: SFMAP_MULTI_RES = 3
integer, parameter :: SFMAP_MULTI_ONS = 4
integer, parameter :: SFMAP_MULTI_EP = 5
integer, parameter :: SFMAP_MULTI_EPR = 6
integer, parameter :: SFMAP_MULTI_EPO = 7
integer, parameter :: SFMAP_MULTI_IP = 8
integer, parameter :: SFMAP_MULTI_IPR = 9
integer, parameter :: SFMAP_MULTI_IPO = 10
integer, parameter :: SFMAP_MULTI_EI = 11
integer, parameter :: SFMAP_MULTI_SRS = 13
integer, parameter :: SFMAP_MULTI_SON = 14
```

Then, it contains an allocatable entry for the multi mapping. This entry holds the MC-parameter indices on which the mapping applies (there may be more than one MC parameter per structure-function entry) and any parameters associated with the mapping.

There can be only one multi-mapping per channel.

⟨*SF mappings: public*⟩+≡
```
public :: sf_channel_t
```

⟨*SF mappings: types*⟩+≡
```
type :: sf_channel_t
   integer, dimension(:), allocatable :: map_code
   class(sf_mapping_t), allocatable :: multi_mapping
 contains
 ⟨SF mappings: sf channel: TBP⟩
end type sf_channel_t
```

The output format prints a single character for each structure-function entry and, if applicable, an account of the mapping parameters.

⟨*SF mappings: sf channel: TBP*⟩≡
```
procedure :: write => sf_channel_write
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_channel_write (object, unit)
  class(sf_channel_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  if (allocated (object%map_code)) then
     do i = 1, size (object%map_code)
        select case (object%map_code (i))
        case (SFMAP_NONE)
           write (u, "(1x,A)", advance="no") "-"
        case (SFMAP_SINGLE)
           write (u, "(1x,A)", advance="no") "+"
```

```
            case (SFMAP_MULTI_S)
               write (u, "(1x,A)", advance="no") "s"
            case (SFMAP_MULTI_RES, SFMAP_MULTI_SRS)
               write (u, "(1x,A)", advance="no") "r"
            case (SFMAP_MULTI_ONS, SFMAP_MULTI_SON)
               write (u, "(1x,A)", advance="no") "o"
            case (SFMAP_MULTI_EP)
               write (u, "(1x,A)", advance="no") "e"
            case (SFMAP_MULTI_EPR)
               write (u, "(1x,A)", advance="no") "p"
            case (SFMAP_MULTI_EPO)
               write (u, "(1x,A)", advance="no") "q"
            case (SFMAP_MULTI_IP)
               write (u, "(1x,A)", advance="no") "i"
            case (SFMAP_MULTI_IPR)
               write (u, "(1x,A)", advance="no") "i"
            case (SFMAP_MULTI_IPO)
               write (u, "(1x,A)", advance="no") "i"
            case (SFMAP_MULTI_EI)
               write (u, "(1x,A)", advance="no") "i"
            case default
               write (u, "(1x,A)", advance="no") "?"
            end select
         end do
      else
         write (u, "(1x,A)", advance="no") "-"
      end if
      if (allocated (object%multi_mapping)) then
         write (u, "(1x,'/')", advance="no")
         call object%multi_mapping%write (u)
      else
         write (u, *)
      end if
   end subroutine sf_channel_write
```

Initializer for a single `sf_channel` object.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
   procedure :: init => sf_channel_init
```

⟨*SF mappings: procedures*⟩+≡

```
   subroutine sf_channel_init (channel, n_strfun)
     class(sf_channel_t), intent(out) :: channel
     integer, intent(in) :: n_strfun
     allocate (channel%map_code (n_strfun))
     channel%map_code = SFMAP_NONE
   end subroutine sf_channel_init
```

Assignment. This merely copies intrinsic assignment, but apparently the latter is bugged in gfortran 4.6.3, causing memory corruption.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
   generic :: assignment (=) => sf_channel_assign
   procedure :: sf_channel_assign
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_channel_assign (copy, original)
  class(sf_channel_t), intent(out) :: copy
  type(sf_channel_t), intent(in) :: original
  allocate (copy%map_code (size (original%map_code)))
  copy%map_code = original%map_code
  if (allocated (original%multi_mapping)) then
     allocate (copy%multi_mapping, source = original%multi_mapping)
  end if
end subroutine sf_channel_assign
```

This initializer allocates an array of channels with common number of structure-function entries, therefore it is not a type-bound procedure.

⟨*SF mappings: public*⟩+≡

```
public :: allocate_sf_channels
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine allocate_sf_channels (channel, n_channel, n_strfun)
  type(sf_channel_t), dimension(:), intent(out), allocatable :: channel
  integer, intent(in) :: n_channel
  integer, intent(in) :: n_strfun
  integer :: c
  allocate (channel (n_channel))
  do c = 1, n_channel
     call channel(c)%init (n_strfun)
  end do
end subroutine allocate_sf_channels
```

This marks a given subset of indices as single-mapping.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
procedure :: activate_mapping => sf_channel_activate_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_channel_activate_mapping (channel, i_sf)
  class(sf_channel_t), intent(inout) :: channel
  integer, dimension(:), intent(in) :: i_sf
  channel%map_code(i_sf) = SFMAP_SINGLE
end subroutine sf_channel_activate_mapping
```

This sets an s-channel multichannel mapping. The parameter indices are not yet set.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
procedure :: set_s_mapping => sf_channel_set_s_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
subroutine sf_channel_set_s_mapping (channel, i_sf, power)
  class(sf_channel_t), intent(inout) :: channel
  integer, dimension(:), intent(in) :: i_sf
  real(default), intent(in), optional :: power
  channel%map_code(i_sf) = SFMAP_MULTI_S
  allocate (sf_s_mapping_t :: channel%multi_mapping)
  select type (mapping => channel%multi_mapping)
  type is (sf_s_mapping_t)
```

```
      call mapping%init (power)
    end select
  end subroutine sf_channel_set_s_mapping
```

This sets an s-channel resonance multichannel mapping.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: set_res_mapping => sf_channel_set_res_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_channel_set_res_mapping (channel, i_sf, m, w, single)
    class(sf_channel_t), intent(inout) :: channel
    integer, dimension(:), intent(in) :: i_sf
    real(default), intent(in) :: m, w
    logical, intent(in) :: single
    if (single) then
       channel%map_code(i_sf) = SFMAP_MULTI_SRS
       allocate (sf_res_mapping_single_t :: channel%multi_mapping)
       select type (mapping => channel%multi_mapping)
       type is (sf_res_mapping_single_t)
          call mapping%init (m, w)
       end select
    else
       channel%map_code(i_sf) = SFMAP_MULTI_RES
       allocate (sf_res_mapping_t :: channel%multi_mapping)
       select type (mapping => channel%multi_mapping)
       type is (sf_res_mapping_t)
          call mapping%init (m, w)
       end select
    end if
  end subroutine sf_channel_set_res_mapping
```

This sets an s-channel on-shell multichannel mapping. The length of the `i_sf` array must be 2. (The first parameter actually becomes an irrelevant dummy.)

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: set_os_mapping => sf_channel_set_os_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_channel_set_os_mapping (channel, i_sf, m, single)
    class(sf_channel_t), intent(inout) :: channel
    integer, dimension(:), intent(in) :: i_sf
    real(default), intent(in) :: m
    logical, intent(in) :: single
    if (single) then
       channel%map_code(i_sf) = SFMAP_MULTI_SON
       allocate (sf_os_mapping_single_t :: channel%multi_mapping)
       select type (mapping => channel%multi_mapping)
       type is (sf_os_mapping_single_t)
          call mapping%init (m)
       end select
    else
       channel%map_code(i_sf) = SFMAP_MULTI_ONS
       allocate (sf_os_mapping_t :: channel%multi_mapping)
       select type (mapping => channel%multi_mapping)
       type is (sf_os_mapping_t)
```

```
        call mapping%init (m)
      end select
    end if
  end subroutine sf_channel_set_os_mapping
```

This sets an s-channel endpoint mapping. The parameter a is the slope parameter (default 1); increasing it moves the endpoint region (at $x = 1$ to lower values in the input parameter. region even more.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: set_ep_mapping => sf_channel_set_ep_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_channel_set_ep_mapping (channel, i_sf, a)
    class(sf_channel_t), intent(inout) :: channel
    integer, dimension(:), intent(in) :: i_sf
    real(default), intent(in), optional :: a
    channel%map_code(i_sf) = SFMAP_MULTI_EP
    allocate (sf_ep_mapping_t :: channel%multi_mapping)
    select type (mapping => channel%multi_mapping)
    type is (sf_ep_mapping_t)
       call mapping%init (a = a)
    end select
  end subroutine sf_channel_set_ep_mapping
```

This sets a resonant endpoint mapping.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: set_epr_mapping => sf_channel_set_epr_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_channel_set_epr_mapping (channel, i_sf, a, m, w)
    class(sf_channel_t), intent(inout) :: channel
    integer, dimension(:), intent(in) :: i_sf
    real(default), intent(in) :: a, m, w
    channel%map_code(i_sf) = SFMAP_MULTI_EPR
    allocate (sf_epr_mapping_t :: channel%multi_mapping)
    select type (mapping => channel%multi_mapping)
    type is (sf_epr_mapping_t)
       call mapping%init (a, m, w)
    end select
  end subroutine sf_channel_set_epr_mapping
```

This sets an on-shell endpoint mapping.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: set_epo_mapping => sf_channel_set_epo_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_channel_set_epo_mapping (channel, i_sf, a, m)
    class(sf_channel_t), intent(inout) :: channel
    integer, dimension(:), intent(in) :: i_sf
    real(default), intent(in) :: a, m
    channel%map_code(i_sf) = SFMAP_MULTI_EPO
    allocate (sf_epo_mapping_t :: channel%multi_mapping)
    select type (mapping => channel%multi_mapping)
```

```
    type is (sf_epo_mapping_t)
       call mapping%init (a, m)
    end select
  end subroutine sf_channel_set_epo_mapping
```

This sets an s-channel power mapping, regulating a singularity of type $(1-x)^{-1+\epsilon}$. The parameter ϵ depends on the structure function.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: set_ip_mapping => sf_channel_set_ip_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_channel_set_ip_mapping (channel, i_sf, eps)
    class(sf_channel_t), intent(inout) :: channel
    integer, dimension(:), intent(in) :: i_sf
    real(default), intent(in), optional :: eps
    channel%map_code(i_sf) = SFMAP_MULTI_IP
    allocate (sf_ip_mapping_t :: channel%multi_mapping)
    select type (mapping => channel%multi_mapping)
    type is (sf_ip_mapping_t)
       call mapping%init (eps)
    end select
  end subroutine sf_channel_set_ip_mapping
```

This sets an s-channel resonant power mapping, regulating a singularity of type $(1-x)^{-1+\epsilon}$ in the presence of an s-channel resonance. The parameter ϵ depends on the structure function.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: set_ipr_mapping => sf_channel_set_ipr_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_channel_set_ipr_mapping (channel, i_sf, eps, m, w)
    class(sf_channel_t), intent(inout) :: channel
    integer, dimension(:), intent(in) :: i_sf
    real(default), intent(in), optional :: eps, m, w
    channel%map_code(i_sf) = SFMAP_MULTI_IPR
    allocate (sf_ipr_mapping_t :: channel%multi_mapping)
    select type (mapping => channel%multi_mapping)
    type is (sf_ipr_mapping_t)
       call mapping%init (eps, m, w)
    end select
  end subroutine sf_channel_set_ipr_mapping
```

This sets an on-shell power mapping, regulating a singularity of type $(1-x)^{-1+\epsilon}$ for the production of a single on-shell particle.. The parameter ϵ depends on the structure function.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: set_ipo_mapping => sf_channel_set_ipo_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_channel_set_ipo_mapping (channel, i_sf, eps, m)
    class(sf_channel_t), intent(inout) :: channel
    integer, dimension(:), intent(in) :: i_sf
    real(default), intent(in), optional :: eps, m
```

```
    channel%map_code(i_sf) = SFMAP_MULTI_IPO
    allocate (sf_ipo_mapping_t :: channel%multi_mapping)
    select type (mapping => channel%multi_mapping)
    type is (sf_ipo_mapping_t)
       call mapping%init (eps, m)
    end select
  end subroutine sf_channel_set_ipo_mapping
```

This sets a combined endpoint/ISR mapping.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: set_ei_mapping => sf_channel_set_ei_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_channel_set_ei_mapping (channel, i_sf, a, eps)
    class(sf_channel_t), intent(inout) :: channel
    integer, dimension(:), intent(in) :: i_sf
    real(default), intent(in), optional :: a, eps
    channel%map_code(i_sf) = SFMAP_MULTI_EI
    allocate (sf_ei_mapping_t :: channel%multi_mapping)
    select type (mapping => channel%multi_mapping)
    type is (sf_ei_mapping_t)
       call mapping%init (a, eps)
    end select
  end subroutine sf_channel_set_ei_mapping
```

This sets a combined endpoint/ISR mapping with resonance.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: set_eir_mapping => sf_channel_set_eir_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_channel_set_eir_mapping (channel, i_sf, a, eps, m, w)
    class(sf_channel_t), intent(inout) :: channel
    integer, dimension(:), intent(in) :: i_sf
    real(default), intent(in), optional :: a, eps, m, w
    channel%map_code(i_sf) = SFMAP_MULTI_EI
    allocate (sf_eir_mapping_t :: channel%multi_mapping)
    select type (mapping => channel%multi_mapping)
    type is (sf_eir_mapping_t)
       call mapping%init (a, eps, m, w)
    end select
  end subroutine sf_channel_set_eir_mapping
```

This sets a combined endpoint/ISR mapping, on-shell.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: set_eio_mapping => sf_channel_set_eio_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
  subroutine sf_channel_set_eio_mapping (channel, i_sf, a, eps, m)
    class(sf_channel_t), intent(inout) :: channel
    integer, dimension(:), intent(in) :: i_sf
    real(default), intent(in), optional :: a, eps, m
    channel%map_code(i_sf) = SFMAP_MULTI_EI
    allocate (sf_eio_mapping_t :: channel%multi_mapping)
    select type (mapping => channel%multi_mapping)
```

```
    type is (sf_eio_mapping_t)
       call mapping%init (a, eps, m)
    end select
  end subroutine sf_channel_set_eio_mapping
```

Return true if the mapping code at position `i_sf` is `SFMAP_SINGLE`.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: is_single_mapping => sf_channel_is_single_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
  function sf_channel_is_single_mapping (channel, i_sf) result (flag)
    class(sf_channel_t), intent(in) :: channel
    integer, intent(in) :: i_sf
    logical :: flag
    flag = channel%map_code(i_sf) == SFMAP_SINGLE
  end function sf_channel_is_single_mapping
```

Return true if the mapping code at position `i_sf` is any of the `SFMAP_MULTI` mappings.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: is_multi_mapping => sf_channel_is_multi_mapping
```

⟨*SF mappings: procedures*⟩+≡

```
  function sf_channel_is_multi_mapping (channel, i_sf) result (flag)
    class(sf_channel_t), intent(in) :: channel
    integer, intent(in) :: i_sf
    logical :: flag
    select case (channel%map_code(i_sf))
    case (SFMAP_NONE, SFMAP_SINGLE)
       flag = .false.
    case default
       flag = .true.
    end select
  end function sf_channel_is_multi_mapping
```

Return the number of parameters that the multi-mapping requires. The mapping object must be allocated.

⟨*SF mappings: sf channel: TBP*⟩+≡

```
  procedure :: get_multi_mapping_n_par => sf_channel_get_multi_mapping_n_par
```

⟨*SF mappings: procedures*⟩+≡

```
  function sf_channel_get_multi_mapping_n_par (channel) result (n_par)
    class(sf_channel_t), intent(in) :: channel
    integer :: n_par
    if (allocated (channel%multi_mapping)) then
       n_par = channel%multi_mapping%get_n_dim ()
    else
       n_par = 0
    end if
  end function sf_channel_get_multi_mapping_n_par
```

Return true if there is any nontrivial mapping in any of the channels.

Note: we provide an explicit public function. gfortran 4.6.3 has problems with the alternative implementation as a type-bound procedure for an array base object.

⟨*SF mappings: public*⟩+≡
```
public :: any_sf_channel_has_mapping
```

⟨*SF mappings: procedures*⟩+≡
```
function any_sf_channel_has_mapping (channel) result (flag)
  type(sf_channel_t), dimension(:), intent(in) :: channel
  logical :: flag
  integer :: c
  flag = .false.
  do c = 1, size (channel)
     flag = flag .or. any (channel(c)%map_code /= SFMAP_NONE)
  end do
end function any_sf_channel_has_mapping
```

Set a parameter index for an active multi mapping. We assume that the index array is allocated properly.

⟨*SF mappings: sf channel: TBP*⟩+≡
```
procedure :: set_par_index => sf_channel_set_par_index
```

⟨*SF mappings: procedures*⟩+≡
```
subroutine sf_channel_set_par_index (channel, j, i_par)
  class(sf_channel_t), intent(inout) :: channel
  integer, intent(in) :: j
  integer, intent(in) :: i_par
  associate (mapping => channel%multi_mapping)
    if (j >= 1 .and. j <= mapping%get_n_dim ()) then
       if (mapping%get_index (j) == 0) then
          call channel%multi_mapping%set_index (j, i_par)
       else
          call msg_bug ("Structure-function setup: mapping index set twice")
       end if
    else
       call msg_bug ("Structure-function setup: mapping index out of range")
    end if
  end associate
end subroutine sf_channel_set_par_index
```

15.4.18 Unit tests

Test module, followed by the corresponding implementation module.

⟨`sf_mappings_ut.f90`⟩≡
```
⟨File header⟩

module sf_mappings_ut
  use unit_tests
  use sf_mappings_uti

⟨Standard module head⟩
```

```
⟨SF mappings: public test⟩

contains

⟨SF mappings: test driver⟩

end module sf_mappings_ut
```

⟨sf_mappings_uti.f90⟩≡

```
⟨File header⟩

module sf_mappings_uti

⟨Use kinds⟩
  use format_defs, only: FMT_11, FMT_12, FMT_13, FMT_14, FMT_15, FMT_16

  use sf_mappings

⟨Standard module head⟩

⟨SF mappings: test declarations⟩

contains

⟨SF mappings: tests⟩

end module sf_mappings_uti
```

API: driver for the unit tests below.

⟨*SF mappings: public test*⟩≡

```
public :: sf_mappings_test
```

⟨*SF mappings: test driver*⟩≡

```
subroutine sf_mappings_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨SF mappings: execute tests⟩
end subroutine sf_mappings_test
```

Check standard mapping

Probe the standard mapping of the unit square for different parameter values. Also calculates integrals. For a finite number of bins, they differ slightly from 1, but the result is well-defined because we are not using random points.

⟨*SF mappings: execute tests*⟩≡

```
call test (sf_mappings_1, "sf_mappings_1", &
     "standard pair mapping", &
     u, results)
```

⟨*SF mappings: test declarations*⟩≡

```
public :: sf_mappings_1
```

⟨*SF mappings: tests*⟩≡

```
subroutine sf_mappings_1 (u)
  integer, intent(in) :: u
  class(sf_mapping_t), allocatable :: mapping
  real(default), dimension(2) :: p

  write (u, "(A)")  "* Test output: sf_mappings_1"
  write (u, "(A)")  "*   Purpose: probe standard mapping"
  write (u, "(A)")

  allocate (sf_s_mapping_t :: mapping)
  select type (mapping)
  type is (sf_s_mapping_t)
     call mapping%init ()
     call mapping%set_index (1, 1)
     call mapping%set_index (2, 2)
  end select

  call mapping%write (u)

  write (u, *)
  write (u, "(A)")  "Probe at (0,0):"
  p = [0._default, 0._default]
  call mapping%check (u, p, 1-p, "F7.5")

  write (u, *)
  write (u, "(A)")  "Probe at (0.5,0.5):"
  p = [0.5_default, 0.5_default]
  call mapping%check (u, p, 1-p, "F7.5")

  write (u, *)
  write (u, "(A)")  "Probe at (0.1,0.5):"
  p = [0.1_default, 0.5_default]
  call mapping%check (u, p, 1-p, "F7.5")

  write (u, *)
  write (u, "(A)")  "Probe at (0.1,0.1):"
  p = [0.1_default, 0.1_default]
  call mapping%check (u, p, 1-p, "F7.5")

  write (u, *)
  write (u, "(A)")  "Compute integral:"
  write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

  deallocate (mapping)
  allocate (sf_s_mapping_t :: mapping)
  select type (mapping)
  type is (sf_s_mapping_t)
     call mapping%init (power=2._default)
     call mapping%set_index (1, 1)
     call mapping%set_index (2, 2)
  end select

  write (u, *)
```

```
    call mapping%write (u)

    write (u, *)
    write (u, "(A)")  "Probe at (0,0):"
    p = [0._default, 0._default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.5,0.5):"
    p = [0.5_default, 0.5_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.1,0.5):"
    p = [0.1_default, 0.5_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.1,0.1):"
    p = [0.1_default, 0.1_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Compute integral:"
    write (u, "(3x,A,1x,F7.5)")  "I =", mapping%integral (100000)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_mappings_1"

  end subroutine sf_mappings_1
```

Channel entries

Construct channel entries and print them.

⟨*SF mappings: execute tests*⟩+≡

```
  call test (sf_mappings_2, "sf_mappings_2", &
       "structure-function mapping channels", &
       u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
  public :: sf_mappings_2
```

⟨*SF mappings: tests*⟩+≡

```
  subroutine sf_mappings_2 (u)
    integer, intent(in) :: u
    type(sf_channel_t), dimension(:), allocatable :: channel
    integer :: c

    write (u, "(A)")  "* Test output: sf_mappings_2"
    write (u, "(A)")  "*   Purpose: construct and display &
         &mapping-channel objects"
    write (u, "(A)")

    call allocate_sf_channels (channel, n_channel = 8, n_strfun = 2)
```

```
    call channel(2)%activate_mapping ([1])
    call channel(3)%set_s_mapping ([1,2])
    call channel(4)%set_s_mapping ([1,2], power=2._default)
    call channel(5)%set_res_mapping ([1,2], m = 0.5_default, w = 0.1_default, single = .false.)
    call channel(6)%set_os_mapping ([1,2], m = 0.5_default, single = .false.)
    call channel(7)%set_res_mapping ([1], m = 0.5_default, w = 0.1_default, single = .true.)
    call channel(8)%set_os_mapping ([1], m = 0.5_default, single = .true.)

    call channel(3)%set_par_index (1, 1)
    call channel(3)%set_par_index (2, 4)

    call channel(4)%set_par_index (1, 1)
    call channel(4)%set_par_index (2, 4)

    call channel(5)%set_par_index (1, 1)
    call channel(5)%set_par_index (2, 3)

    call channel(6)%set_par_index (1, 1)
    call channel(6)%set_par_index (2, 2)

    call channel(7)%set_par_index (1, 1)

    call channel(8)%set_par_index (1, 1)

    do c = 1, size (channel)
       write (u, "(I0,':')", advance="no")  c
       call channel(c)%write (u)
    end do

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_mappings_2"

  end subroutine sf_mappings_2
```

Check resonance mapping

Probe the resonance mapping of the unit square for different parameter values. Also calculates integrals. For a finite number of bins, they differ slightly from 1, but the result is well-defined because we are not using random points.

The resonance mass is at 1/2 the energy, the width is 1/10.

⟨*SF mappings: execute tests*⟩+≡

```
  call test (sf_mappings_3, "sf_mappings_3", &
       "resonant pair mapping", &
       u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
  public :: sf_mappings_3
```

⟨*SF mappings: tests*⟩+≡

```
  subroutine sf_mappings_3 (u)
    integer, intent(in) :: u
    class(sf_mapping_t), allocatable :: mapping
    real(default), dimension(2) :: p
```

```
    write (u, "(A)")  "* Test output: sf_mappings_3"
    write (u, "(A)")  "*   Purpose: probe resonance pair mapping"
    write (u, "(A)")

    allocate (sf_res_mapping_t :: mapping)
    select type (mapping)
    type is (sf_res_mapping_t)
       call mapping%init (0.5_default, 0.1_default)
       call mapping%set_index (1, 1)
       call mapping%set_index (2, 2)
    end select

    call mapping%write (u)

    write (u, *)
    write (u, "(A)")  "Probe at (0,0):"
    p = [0._default, 0._default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.5,0.5):"
    p = [0.5_default, 0.5_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.1,0.5):"
    p = [0.1_default, 0.5_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.1,0.1):"
    p = [0.1_default, 0.1_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Compute integral:"
    write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

    deallocate (mapping)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_mappings_3"

  end subroutine sf_mappings_3
```

Check on-shell mapping

Probe the on-shell mapping of the unit square for different parameter values. Also calculates integrals. In this case, the Jacobian is constant and given by $|\log m^2|$, so this is also the value of the integral. The factor results from the variable change in the δ function $\delta(m^2 - x_1 x_2)$ which multiplies the cross section

for the case at hand.

For the test, the (rescaled) resonance mass is set at 1/2 the energy.

⟨*SF mappings: execute tests*⟩+≡
```
call test (sf_mappings_4, "sf_mappings_4", &
     "on-shell pair mapping", &
     u, results)
```

⟨*SF mappings: test declarations*⟩+≡
```
public :: sf_mappings_4
```

⟨*SF mappings: tests*⟩+≡
```
subroutine sf_mappings_4 (u)
  integer, intent(in) :: u
  class(sf_mapping_t), allocatable :: mapping
  real(default), dimension(2) :: p

  write (u, "(A)")  "* Test output: sf_mappings_4"
  write (u, "(A)")  "*   Purpose: probe on-shell pair mapping"
  write (u, "(A)")

  allocate (sf_os_mapping_t :: mapping)
  select type (mapping)
  type is (sf_os_mapping_t)
     call mapping%init (0.5_default)
     call mapping%set_index (1, 1)
     call mapping%set_index (2, 2)
  end select

  call mapping%write (u)

  write (u, *)
  write (u, "(A)")  "Probe at (0,0):"
  p = [0._default, 0._default]
  call mapping%check (u, p, 1-p, "F7.5")

  write (u, *)
  write (u, "(A)")  "Probe at (0.5,0.5):"
  p = [0.5_default, 0.5_default]
  call mapping%check (u, p, 1-p, "F7.5")

  write (u, *)
  write (u, "(A)")  "Probe at (0,0.1):"
  p = [0._default, 0.1_default]
  call mapping%check (u, p, 1-p, "F7.5")

  write (u, *)
  write (u, "(A)")  "Probe at (0,1.0):"
  p = [0._default, 1.0_default]
  call mapping%check (u, p, 1-p, "F7.5")

  write (u, *)
  write (u, "(A)")  "Compute integral:"
  write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

  deallocate (mapping)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_mappings_4"

  end subroutine sf_mappings_4
```

Check endpoint mapping

Probe the endpoint mapping of the unit square for different parameter values. Also calculates integrals. For a finite number of bins, they differ slightly from 1, but the result is well-defined because we are not using random points.

⟨*SF mappings: execute tests*⟩+≡

```
  call test (sf_mappings_5, "sf_mappings_5", &
       "endpoint pair mapping", &
       u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
  public :: sf_mappings_5
```

⟨*SF mappings: tests*⟩+≡

```
  subroutine sf_mappings_5 (u)
    integer, intent(in) :: u
    class(sf_mapping_t), allocatable :: mapping
    real(default), dimension(2) :: p

    write (u, "(A)")  "* Test output: sf_mappings_5"
    write (u, "(A)")  "*   Purpose: probe endpoint pair mapping"
    write (u, "(A)")

    allocate (sf_ep_mapping_t :: mapping)
    select type (mapping)
    type is (sf_ep_mapping_t)
       call mapping%init ()
       call mapping%set_index (1, 1)
       call mapping%set_index (2, 2)
    end select

    call mapping%write (u)

    write (u, *)
    write (u, "(A)")  "Probe at (0,0):"
    p = [0._default, 0._default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.5,0.5):"
    p = [0.5_default, 0.5_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.1,0.5):"
    p = [0.1_default, 0.5_default]
    call mapping%check (u, p, 1-p, "F7.5")
```

```
    write (u, *)
    write (u, "(A)")  "Probe at (0.7,0.2):"
    p = [0.7_default, 0.2_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Compute integral:"
    write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

    deallocate (mapping)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_mappings_5"

  end subroutine sf_mappings_5
```

Check endpoint resonant mapping

Probe the endpoint mapping with resonance. Also calculates integrals.

⟨*SF mappings: execute tests*⟩+≡

```
  call test (sf_mappings_6, "sf_mappings_6", &
       "endpoint resonant mapping", &
       u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
  public :: sf_mappings_6
```

⟨*SF mappings: tests*⟩+≡

```
  subroutine sf_mappings_6 (u)
    integer, intent(in) :: u
    class(sf_mapping_t), allocatable :: mapping
    real(default), dimension(2) :: p

    write (u, "(A)")  "* Test output: sf_mappings_6"
    write (u, "(A)")  "*   Purpose: probe endpoint resonant mapping"
    write (u, "(A)")

    allocate (sf_epr_mapping_t :: mapping)
    select type (mapping)
    type is (sf_epr_mapping_t)
       call mapping%init (a = 1._default, m = 0.5_default, w = 0.1_default)
       call mapping%set_index (1, 1)
       call mapping%set_index (2, 2)
    end select

    call mapping%write (u)

    write (u, *)
    write (u, "(A)")  "Probe at (0,0):"
    p = [0._default, 0._default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
```

```
write (u, "(A)")  "Probe at (0.5,0.5):"
p = [0.5_default, 0.5_default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Probe at (0.1,0.5):"
p = [0.1_default, 0.5_default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Probe at (0.7,0.2):"
p = [0.7_default, 0.2_default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Compute integral:"
write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

deallocate (mapping)

write (u, "(A)")
write (u, "(A)")  "* Same mapping without resonance:"
write (u, "(A)")

allocate (sf_epr_mapping_t :: mapping)
select type (mapping)
type is (sf_epr_mapping_t)
   call mapping%init (a = 1._default)
   call mapping%set_index (1, 1)
   call mapping%set_index (2, 2)
end select

call mapping%write (u)

write (u, *)
write (u, "(A)")  "Probe at (0,0):"
p = [0._default, 0._default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Probe at (0.5,0.5):"
p = [0.5_default, 0.5_default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Probe at (0.1,0.5):"
p = [0.1_default, 0.5_default]
call mapping%check (u, p, 1-p, "F7.5")

write (u, *)
write (u, "(A)")  "Probe at (0.7,0.2):"
p = [0.7_default, 0.2_default]
call mapping%check (u, p, 1-p, "F7.5")
```

```
    write (u, *)
    write (u, "(A)")  "Compute integral:"
    write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

    deallocate (mapping)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_mappings_6"

  end subroutine sf_mappings_6
```

Check endpoint on-shell mapping

Probe the endpoint mapping with an on-shell particle. Also calculates integrals.

⟨*SF mappings: execute tests*⟩+≡

```
  call test (sf_mappings_7, "sf_mappings_7", &
       "endpoint on-shell mapping", &
       u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
  public :: sf_mappings_7
```

⟨*SF mappings: tests*⟩+≡

```
  subroutine sf_mappings_7 (u)
    integer, intent(in) :: u
    class(sf_mapping_t), allocatable :: mapping
    real(default), dimension(2) :: p

    write (u, "(A)")  "* Test output: sf_mappings_7"
    write (u, "(A)")  "*   Purpose: probe endpoint on-shell mapping"
    write (u, "(A)")

    allocate (sf_epo_mapping_t :: mapping)
    select type (mapping)
    type is (sf_epo_mapping_t)
       call mapping%init (a = 1._default, m = 0.5_default)
       call mapping%set_index (1, 1)
       call mapping%set_index (2, 2)
    end select

    call mapping%write (u)

    write (u, *)
    write (u, "(A)")  "Probe at (0,0):"
    p = [0._default, 0._default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.5,0.5):"
    p = [0.5_default, 0.5_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.1,0.5):"
```

```
    p = [0.1_default, 0.5_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.7,0.2):"
    p = [0.7_default, 0.2_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Compute integral:"
    write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

    deallocate (mapping)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_mappings_7"

  end subroutine sf_mappings_7
```

Check power mapping

Probe the power mapping of the unit square for different parameter values. Also calculates integrals. For a finite number of bins, they differ slightly from 1, but the result is well-defined because we are not using random points.

⟨*SF mappings: execute tests*⟩+≡

```
  call test (sf_mappings_8, "sf_mappings_8", &
       "power pair mapping", &
       u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
  public :: sf_mappings_8
```

⟨*SF mappings: tests*⟩+≡

```
  subroutine sf_mappings_8 (u)
    integer, intent(in) :: u
    class(sf_mapping_t), allocatable :: mapping
    real(default), dimension(2) :: p, pb

    write (u, "(A)")  "* Test output: sf_mappings_8"
    write (u, "(A)")  "*   Purpose: probe power pair mapping"
    write (u, "(A)")

    allocate (sf_ip_mapping_t :: mapping)
    select type (mapping)
    type is (sf_ip_mapping_t)
       call mapping%init (eps = 0.1_default)
       call mapping%set_index (1, 1)
       call mapping%set_index (2, 2)
    end select

    call mapping%write (u)

    write (u, *)
```

```
  write (u, "(A)")  "Probe at (0,0.5):"
  p = [0._default, 0.5_default]
  pb= [1._default, 0.5_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.5,0.5):"
  p = [0.5_default, 0.5_default]
  pb= [0.5_default, 0.5_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.9,0.5):"
  p = [0.9_default, 0.5_default]
  pb= [0.1_default, 0.5_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.7,0.2):"
  p = [0.7_default, 0.2_default]
  pb= [0.3_default, 0.8_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.7,0.8):"
  p = [0.7_default, 0.8_default]
  pb= [0.3_default, 0.2_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.99,0.02):"
  p = [0.99_default, 0.02_default]
  pb= [0.01_default, 0.98_default]
  call mapping%check (u, p, pb, FMT_14, FMT_12)

  write (u, *)
  write (u, "(A)")  "Probe at (0.99,0.98):"
  p = [0.99_default, 0.98_default]
  pb= [0.01_default, 0.02_default]
  call mapping%check (u, p, pb, FMT_14, FMT_12)

  write (u, *)
  write (u, "(A)")  "Compute integral:"
  write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

  deallocate (mapping)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_mappings_8"

end subroutine sf_mappings_8
```

Check resonant power mapping

Probe the power mapping of the unit square, adapted for an s-channel resonance, for different parameter values. Also calculates integrals. For a finite number of bins, they differ slightly from 1, but the result is well-defined because we are not using random points.

⟨*SF mappings: execute tests*⟩+≡

```
call test (sf_mappings_9, "sf_mappings_9", &
     "power resonance mapping", &
     u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
public :: sf_mappings_9
```

⟨*SF mappings: tests*⟩+≡

```
subroutine sf_mappings_9 (u)
  integer, intent(in) :: u
  class(sf_mapping_t), allocatable :: mapping
  real(default), dimension(2) :: p, pb

  write (u, "(A)")  "* Test output: sf_mappings_9"
  write (u, "(A)")  "*   Purpose: probe power resonant pair mapping"
  write (u, "(A)")

  allocate (sf_ipr_mapping_t :: mapping)
  select type (mapping)
  type is (sf_ipr_mapping_t)
     call mapping%init (eps = 0.1_default, m = 0.5_default, w = 0.1_default)
     call mapping%set_index (1, 1)
     call mapping%set_index (2, 2)
  end select

  call mapping%write (u)

  write (u, *)
  write (u, "(A)")  "Probe at (0,0.5):"
  p = [0._default, 0.5_default]
  pb= [1._default, 0.5_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.5,0.5):"
  p = [0.5_default, 0.5_default]
  pb= [0.5_default, 0.5_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.9,0.5):"
  p = [0.9_default, 0.5_default]
  pb= [0.1_default, 0.5_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.7,0.2):"
  p = [0.7_default, 0.2_default]
```

```
pb= [0.3_default, 0.8_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
write (u, "(A)")  "Probe at (0.7,0.8):"
p = [0.7_default, 0.8_default]
pb= [0.3_default, 0.2_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
write (u, "(A)")  "Probe at (0.9999,0.02):"
p = [0.9999_default, 0.02_default]
pb= [0.0001_default, 0.98_default]
call mapping%check (u, p, pb, FMT_11, FMT_12)

write (u, *)
write (u, "(A)")  "Probe at (0.9999,0.98):"
p = [0.9999_default, 0.98_default]
pb= [0.0001_default, 0.02_default]
call mapping%check (u, p, pb, FMT_11, FMT_12)

write (u, *)
write (u, "(A)")  "Compute integral:"
write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

deallocate (mapping)

write (u, "(A)")
write (u, "(A)")  "* Same mapping without resonance:"
write (u, "(A)")

allocate (sf_ipr_mapping_t :: mapping)
select type (mapping)
type is (sf_ipr_mapping_t)
   call mapping%init (eps = 0.1_default)
   call mapping%set_index (1, 1)
   call mapping%set_index (2, 2)
end select

call mapping%write (u)

write (u, *)
write (u, "(A)")  "Probe at (0,0.5):"
p = [0._default, 0.5_default]
pb= [1._default, 0.5_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
write (u, "(A)")  "Probe at (0.5,0.5):"
p = [0.5_default, 0.5_default]
pb= [0.5_default, 0.5_default]
call mapping%check (u, p, pb, FMT_16)

write (u, *)
```

```
    write (u, "(A)")  "Probe at (0.9,0.5):"
    p = [0.9_default, 0.5_default]
    pb= [0.1_default, 0.5_default]
    call mapping%check (u, p, pb, FMT_16)

    write (u, *)
    write (u, "(A)")  "Probe at (0.7,0.2):"
    p = [0.7_default, 0.2_default]
    pb= [0.3_default, 0.8_default]
    call mapping%check (u, p, pb, FMT_16)

    write (u, *)
    write (u, "(A)")  "Probe at (0.7,0.8):"
    p = [0.7_default, 0.8_default]
    pb= [0.3_default, 0.2_default]
    call mapping%check (u, p, pb, FMT_16)

    write (u, *)
    write (u, "(A)")  "Compute integral:"
    write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

    deallocate (mapping)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_mappings_9"

  end subroutine sf_mappings_9
```

Check on-shell power mapping

Probe the power mapping of the unit square, adapted for single-particle production, for different parameter values. Also calculates integrals. For a finite number of bins, they differ slightly from 1, but the result is well-defined because we are not using random points.

⟨*SF mappings: execute tests*⟩+≡

```
  call test (sf_mappings_10, "sf_mappings_10", &
       "power on-shell mapping", &
       u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
  public :: sf_mappings_10
```

⟨*SF mappings: tests*⟩+≡

```
  subroutine sf_mappings_10 (u)
    integer, intent(in) :: u
    class(sf_mapping_t), allocatable :: mapping
    real(default), dimension(2) :: p, pb

    write (u, "(A)")  "* Test output: sf_mappings_10"
    write (u, "(A)")  "*   Purpose: probe power on-shell mapping"
    write (u, "(A)")

    allocate (sf_ipo_mapping_t :: mapping)
```

```
    select type (mapping)
    type is (sf_ipo_mapping_t)
       call mapping%init (eps = 0.1_default, m = 0.5_default)
       call mapping%set_index (1, 1)
       call mapping%set_index (2, 2)
    end select

    call mapping%write (u)

    write (u, *)
    write (u, "(A)")  "Probe at (0,0.5):"
    p = [0._default, 0.5_default]
    pb= [1._default, 0.5_default]
    call mapping%check (u, p, pb, FMT_16)

    write (u, *)
    write (u, "(A)")  "Probe at (0,0.02):"
    p = [0._default, 0.02_default]
    pb= [1._default, 0.98_default]
    call mapping%check (u, p, pb, FMT_15, FMT_12)

    write (u, *)
    write (u, "(A)")  "Probe at (0,0.98):"
    p = [0._default, 0.98_default]
    pb= [1._default, 0.02_default]
    call mapping%check (u, p, pb, FMT_15, FMT_12)

    write (u, *)
    write (u, "(A)")  "Compute integral:"
    write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

    deallocate (mapping)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_mappings_10"

  end subroutine sf_mappings_10
```

Check combined endpoint-power mapping

Probe the mapping for the beamstrahlung/ISR combination.

⟨*SF mappings: execute tests*⟩+≡

```
  call test (sf_mappings_11, "sf_mappings_11", &
       "endpoint/power combined mapping", &
       u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
  public :: sf_mappings_11
```

⟨*SF mappings: tests*⟩+≡

```
  subroutine sf_mappings_11 (u)
    integer, intent(in) :: u
    class(sf_mapping_t), allocatable :: mapping
    real(default), dimension(4) :: p, pb
```

```
  write (u, "(A)")  "* Test output: sf_mappings_11"
  write (u, "(A)")  "*   Purpose: probe power pair mapping"
  write (u, "(A)")

  allocate (sf_ei_mapping_t :: mapping)
  select type (mapping)
  type is (sf_ei_mapping_t)
     call mapping%init (eps = 0.1_default)
     call mapping%set_index (1, 1)
     call mapping%set_index (2, 2)
     call mapping%set_index (3, 3)
     call mapping%set_index (4, 4)
  end select

  call mapping%write (u)

  write (u, *)
  write (u, "(A)")  "Probe at (0.5, 0.5, 0.5, 0.5):"
  p = [0.5_default, 0.5_default, 0.5_default, 0.5_default]
  pb= [0.5_default, 0.5_default, 0.5_default, 0.5_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.7, 0.2, 0.4, 0.8):"
  p = [0.7_default, 0.2_default, 0.4_default, 0.8_default]
  pb= [0.3_default, 0.8_default, 0.6_default, 0.2_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.9, 0.06, 0.95, 0.1):"
  p = [0.9_default, 0.06_default, 0.95_default, 0.1_default]
  pb= [0.1_default, 0.94_default, 0.05_default, 0.9_default]
  call mapping%check (u, p, pb, FMT_13, FMT_12)

  write (u, *)
  write (u, "(A)")  "Compute integral:"
  write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

  deallocate (mapping)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_mappings_11"

end subroutine sf_mappings_11
```

Check resonant endpoint-power mapping

Probe the mapping for the beamstrahlung/ISR combination.

⟨*SF mappings: execute tests*⟩+≡

```
call test (sf_mappings_12, "sf_mappings_12", &
     "endpoint/power resonant combined mapping", &
     u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
public :: sf_mappings_12
```

⟨*SF mappings: tests*⟩+≡

```
subroutine sf_mappings_12 (u)
  integer, intent(in) :: u
  class(sf_mapping_t), allocatable :: mapping
  real(default), dimension(4) :: p, pb

  write (u, "(A)")  "* Test output: sf_mappings_12"
  write (u, "(A)")  "*   Purpose: probe resonant combined mapping"
  write (u, "(A)")

  allocate (sf_eir_mapping_t :: mapping)
  select type (mapping)
  type is (sf_eir_mapping_t)
     call mapping%init (a = 1._default, &
          eps = 0.1_default, m = 0.5_default, w = 0.1_default)
     call mapping%set_index (1, 1)
     call mapping%set_index (2, 2)
     call mapping%set_index (3, 3)
     call mapping%set_index (4, 4)
  end select

  call mapping%write (u)

  write (u, *)
  write (u, "(A)")  "Probe at (0.5, 0.5, 0.5, 0.5):"
  p = [0.5_default, 0.5_default, 0.5_default, 0.5_default]
  pb= [0.5_default, 0.5_default, 0.5_default, 0.5_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.7, 0.2, 0.4, 0.8):"
  p = [0.7_default, 0.2_default, 0.4_default, 0.8_default]
  pb= [0.3_default, 0.8_default, 0.6_default, 0.2_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.9, 0.06, 0.95, 0.1):"
  p = [0.9_default, 0.06_default, 0.95_default, 0.1_default]
  pb= [0.1_default, 0.94_default, 0.05_default, 0.9_default]
  call mapping%check (u, p, pb, FMT_15, FMT_12)

  write (u, *)
  write (u, "(A)")  "Compute integral:"
  write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

  deallocate (mapping)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_mappings_12"

end subroutine sf_mappings_12
```

Check on-shell endpoint-power mapping

Probe the mapping for the beamstrahlung/ISR combination.

⟨*SF mappings: execute tests*⟩+≡

```
call test (sf_mappings_13, "sf_mappings_13", &
     "endpoint/power on-shell combined mapping", &
     u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
public :: sf_mappings_13
```

⟨*SF mappings: tests*⟩+≡

```
subroutine sf_mappings_13 (u)
  integer, intent(in) :: u
  class(sf_mapping_t), allocatable :: mapping
  real(default), dimension(4) :: p, pb

  write (u, "(A)")  "* Test output: sf_mappings_13"
  write (u, "(A)")  "*   Purpose: probe on-shell combined mapping"
  write (u, "(A)")

  allocate (sf_eio_mapping_t :: mapping)
  select type (mapping)
  type is (sf_eio_mapping_t)
     call mapping%init (a = 1._default, eps = 0.1_default, m = 0.5_default)
     call mapping%set_index (1, 1)
     call mapping%set_index (2, 2)
     call mapping%set_index (3, 3)
     call mapping%set_index (4, 4)
  end select

  call mapping%write (u)

  write (u, *)
  write (u, "(A)")  "Probe at (0.5, 0.5, 0.5, 0.5):"
  p = [0.5_default, 0.5_default, 0.5_default, 0.5_default]
  pb= [0.5_default, 0.5_default, 0.5_default, 0.5_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.7, 0.2, 0.4, 0.8):"
  p = [0.7_default, 0.2_default, 0.4_default, 0.8_default]
  pb= [0.3_default, 0.8_default, 0.6_default, 0.2_default]
  call mapping%check (u, p, pb, FMT_16)

  write (u, *)
  write (u, "(A)")  "Probe at (0.9, 0.06, 0.95, 0.1):"
  p = [0.9_default, 0.06_default, 0.95_default, 0.1_default]
  pb= [0.1_default, 0.94_default, 0.05_default, 0.9_default]
  call mapping%check (u, p, pb, FMT_14, FMT_12)

  write (u, *)
  write (u, "(A)")  "Compute integral:"
  write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)
```

```
    deallocate (mapping)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_mappings_13"

  end subroutine sf_mappings_13
```

Check rescaling

Check the rescaling factor in on-shell basic mapping.

⟨*SF mappings: execute tests*⟩+≡

```
  call test (sf_mappings_14, "sf_mappings_14", &
       "rescaled on-shell mapping", &
       u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
  public :: sf_mappings_14
```

⟨*SF mappings: tests*⟩+≡

```
  subroutine sf_mappings_14 (u)
    integer, intent(in) :: u
    real(default), dimension(2) :: p2, r2
    real(default), dimension(1) :: p1, r1
    real(default) :: f, x_free, m2

    write (u, "(A)")  "* Test output: sf_mappings_14"
    write (u, "(A)")  "*   Purpose: probe rescaling in os mapping"
    write (u, "(A)")

    x_free = 0.9_default
    m2 = 0.5_default

    write (u, "(A)")  "* Two parameters"
    write (u, "(A)")

    p2 = [0.1_default, 0.2_default]

    call map_on_shell (r2, f, p2, -log (m2), x_free)

    write (u, "(A,9(1x," // FMT_14 // "))")  "p =", p2
    write (u, "(A,9(1x," // FMT_14 // "))")  "r =", r2
    write (u, "(A,9(1x," // FMT_14 // "))")  "f =", f
    write (u, "(A,9(1x," // FMT_14 // "))")  "*r=", x_free * product (r2)

    write (u, *)

    call map_on_shell_inverse (r2, f, p2, -log (m2), x_free)

    write (u, "(A,9(1x," // FMT_14 // "))")  "p =", p2
    write (u, "(A,9(1x," // FMT_14 // "))")  "r =", r2
    write (u, "(A,9(1x," // FMT_14 // "))")  "f =", f
    write (u, "(A,9(1x," // FMT_14 // "))")  "*r=", x_free * product (r2)

    write (u, "(A)")
```

```
    write (u, "(A)")  "* One parameter"
    write (u, "(A)")

    p1 = [0.1_default]

    call map_on_shell_single (r1, f, p1, -log (m2), x_free)

    write (u, "(A,9(1x," // FMT_14 // "))")  "p =", p1
    write (u, "(A,9(1x," // FMT_14 // "))")  "r =", r1
    write (u, "(A,9(1x," // FMT_14 // "))")  "f =", f
    write (u, "(A,9(1x," // FMT_14 // "))")  "*r=", x_free * product (r1)

    write (u, *)

    call map_on_shell_single_inverse (r1, f, p1, -log (m2), x_free)

    write (u, "(A,9(1x," // FMT_14 // "))")  "p =", p1
    write (u, "(A,9(1x," // FMT_14 // "))")  "r =", r1
    write (u, "(A,9(1x," // FMT_14 // "))")  "f =", f
    write (u, "(A,9(1x," // FMT_14 // "))")  "*r=", x_free * product (r1)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_mappings_14"

  end subroutine sf_mappings_14
```

Check single parameter resonance mapping

Probe the resonance mapping of the unit interval for different parameter values. Also calculates integrals.

The resonance mass is at 1/2 the energy, the width is 1/10.

⟨*SF mappings: execute tests*⟩+≡

```
  call test (sf_mappings_15, "sf_mappings_15", &
       "resonant single mapping", &
       u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
  public :: sf_mappings_15
```

⟨*SF mappings: tests*⟩+≡

```
  subroutine sf_mappings_15 (u)
    integer, intent(in) :: u
    class(sf_mapping_t), allocatable :: mapping
    real(default), dimension(1) :: p

    write (u, "(A)")  "* Test output: sf_mappings_15"
    write (u, "(A)")  "*   Purpose: probe resonance single mapping"
    write (u, "(A)")

    allocate (sf_res_mapping_single_t :: mapping)
    select type (mapping)
    type is (sf_res_mapping_single_t)
       call mapping%init (0.5_default, 0.1_default)
```

```
      call mapping%set_index (1, 1)
    end select

    call mapping%write (u)

    write (u, *)
    write (u, "(A)")  "Probe at (0):"
    p = [0._default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.5):"
    p = [0.5_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.1):"
    p = [0.1_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Compute integral:"
    write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

    deallocate (mapping)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_mappings_15"

  end subroutine sf_mappings_15
```

Check single parameter on-shell mapping

Probe the on-shell (pseudo) mapping of the unit interval for different parameter values. Also calculates integrals.

The resonance mass is at 1/2 the energy.

⟨*SF mappings: execute tests*⟩+≡

```
  call test (sf_mappings_16, "sf_mappings_16", &
       "on-shell single mapping", &
       u, results)
```

⟨*SF mappings: test declarations*⟩+≡

```
  public :: sf_mappings_16
```

⟨*SF mappings: tests*⟩+≡

```
  subroutine sf_mappings_16 (u)
    integer, intent(in) :: u
    class(sf_mapping_t), allocatable :: mapping
    real(default), dimension(1) :: p

    write (u, "(A)")  "* Test output: sf_mappings_16"
    write (u, "(A)")  "*   Purpose: probe on-shell single mapping"
    write (u, "(A)")
```

```
    allocate (sf_os_mapping_single_t :: mapping)
    select type (mapping)
    type is (sf_os_mapping_single_t)
       call mapping%init (0.5_default)
       call mapping%set_index (1, 1)
    end select

    call mapping%write (u)

    write (u, *)
    write (u, "(A)")  "Probe at (0):"
    p = [0._default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Probe at (0.5):"
    p = [0.5_default]
    call mapping%check (u, p, 1-p, "F7.5")

    write (u, *)
    write (u, "(A)")  "Compute integral:"
    write (u, "(3x,A,1x,F7.5)") "I =", mapping%integral (100000)

    deallocate (mapping)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_mappings_16"

  end subroutine sf_mappings_16
```

15.5 Structure function base

⟨sf_base.f90⟩≡

```
  ⟨File header⟩

  module sf_base

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use io_units
    use format_utils, only: write_separator
    use format_defs, only: FMT_17, FMT_19
    use diagnostics
    use lorentz
    use quantum_numbers
    use interactions
    use evaluators
    use pdg_arrays
    use beams
    use sf_aux
    use sf_mappings
```

⟨*Standard module head*⟩

⟨*SF base: public*⟩

⟨*SF base: parameters*⟩

⟨*SF base: types*⟩

⟨*SF base: interfaces*⟩

```
contains
```

⟨*SF base: procedures*⟩

```
end module sf_base
```

15.5.1 Abstract structure-function data type

This type should hold all configuration data for a specific type of structure function. The base object is empty; the implementations will fill it.

⟨*SF base: public*⟩≡

```
public :: sf_data_t
```

⟨*SF base: types*⟩≡

```
type, abstract :: sf_data_t
 contains
 ⟨SF base: sf data: TBP⟩
end type sf_data_t
```

Output.

⟨*SF base: sf data: TBP*⟩≡

```
procedure (sf_data_write), deferred :: write
```

⟨*SF base: interfaces*⟩≡

```
abstract interface
   subroutine sf_data_write (data, unit, verbose)
     import
     class(sf_data_t), intent(in) :: data
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: verbose
   end subroutine sf_data_write
end interface
```

Return true if this structure function is in generator mode. In that case, all parameters are free, otherwise bound. (We do not support mixed cases.) Default is: no generator.

⟨*SF base: sf data: TBP*⟩+≡

```
procedure :: is_generator => sf_data_is_generator
```

⟨*SF base: procedures*⟩≡
```
function sf_data_is_generator (data) result (flag)
  class(sf_data_t), intent(in) :: data
  logical :: flag
  flag = .false.
end function sf_data_is_generator
```

Return the number of input parameters that determine the structure function.

⟨*SF base: sf data: TBP*⟩+≡
```
procedure (sf_data_get_int), deferred :: get_n_par
```

⟨*SF base: interfaces*⟩+≡
```
abstract interface
   function sf_data_get_int (data) result (n)
     import
     class(sf_data_t), intent(in) :: data
     integer :: n
   end function sf_data_get_int
end interface
```

Return the outgoing particle PDG codes for the current setup. The codes can be an array of particles, for each beam.

⟨*SF base: sf data: TBP*⟩+≡
```
procedure (sf_data_get_pdg_out), deferred :: get_pdg_out
```

⟨*SF base: interfaces*⟩+≡
```
abstract interface
   subroutine sf_data_get_pdg_out (data, pdg_out)
     import
     class(sf_data_t), intent(in) :: data
     type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
   end subroutine sf_data_get_pdg_out
end interface
```

Allocate a matching structure function interaction object and properly initialize it.

⟨*SF base: sf data: TBP*⟩+≡
```
procedure (sf_data_allocate_sf_int), deferred :: allocate_sf_int
```

⟨*SF base: interfaces*⟩+≡
```
abstract interface
   subroutine sf_data_allocate_sf_int (data, sf_int)
     import
     class(sf_data_t), intent(in) :: data
     class(sf_int_t), intent(inout), allocatable :: sf_int
   end subroutine sf_data_allocate_sf_int
end interface
```

Return the PDF set index, if applicable. We implement a default method which returns zero. The PDF (builtin and LHA) implementations will override this.

⟨*SF base: sf data: TBP*⟩+≡
```
procedure :: get_pdf_set => sf_data_get_pdf_set
```

⟨*SF base: procedures*⟩+≡
```
function sf_data_get_pdf_set (data) result (pdf_set)
  class(sf_data_t), intent(in) :: data
  integer :: pdf_set
  pdf_set = 0
end function sf_data_get_pdf_set
```

Return the spectrum file, if applicable. We implement a default method which returns zero. CIRCE1, CIRCE2 and the beam spectrum will override this.

⟨*SF base: sf data: TBP*⟩+≡
```
procedure :: get_beam_file => sf_data_get_beam_file
```

⟨*SF base: procedures*⟩+≡
```
function sf_data_get_beam_file (data) result (file)
  class(sf_data_t), intent(in) :: data
  type(string_t) :: file
  file = ""
end function sf_data_get_beam_file
```

15.5.2 Structure-function chain configuration

This is the data type that the `process` module uses for setting up its structure-function chain. For each structure function described by the beam data, there is an entry. The `i` array indicates the beam(s) to which this structure function applies, and the `data` object contains the actual configuration data.

⟨*SF base: public*⟩+≡
```
public :: sf_config_t
```

⟨*SF base: types*⟩+≡
```
type :: sf_config_t
   integer, dimension(:), allocatable :: i
   class(sf_data_t), allocatable :: data
 contains
 ⟨SF base: sf config: TBP⟩
end type sf_config_t
```

Output:

⟨*SF base: sf config: TBP*⟩≡
```
procedure :: write => sf_config_write
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_config_write (object, unit)
  class(sf_config_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  if (allocated (object%i)) then
     write (u, "(1x,A,2(1x,I0))")  "Structure-function configuration: &
          &beam(s)", object%i
     if (allocated (object%data))  call object%data%write (u)
  else
     write (u, "(1x,A)")  "Structure-function configuration: [undefined]"
```

```
    end if
  end subroutine sf_config_write
```

Initialize.

⟨*SF base: sf config: TBP*⟩+≡

```
  procedure :: init => sf_config_init
```

⟨*SF base: procedures*⟩+≡

```
  subroutine sf_config_init (sf_config, i_beam, sf_data)
    class(sf_config_t), intent(out) :: sf_config
    integer, dimension(:), intent(in) :: i_beam
    class(sf_data_t), intent(in) :: sf_data
    allocate (sf_config%i (size (i_beam)), source = i_beam)
    allocate (sf_config%data, source = sf_data)
  end subroutine sf_config_init
```

Return the PDF set, if any.

⟨*SF base: sf config: TBP*⟩+≡

```
  procedure :: get_pdf_set => sf_config_get_pdf_set
```

⟨*SF base: procedures*⟩+≡

```
  function sf_config_get_pdf_set (sf_config) result (pdf_set)
    class(sf_config_t), intent(in) :: sf_config
    integer :: pdf_set
    pdf_set = sf_config%data%get_pdf_set ()
  end function sf_config_get_pdf_set
```

Return the beam spectrum file, if any.

⟨*SF base: sf config: TBP*⟩+≡

```
  procedure :: get_beam_file => sf_config_get_beam_file
```

⟨*SF base: procedures*⟩+≡

```
  function sf_config_get_beam_file (sf_config) result (file)
    class(sf_config_t), intent(in) :: sf_config
    type(string_t) :: file
    file = sf_config%data%get_beam_file ()
  end function sf_config_get_beam_file
```

15.5.3 Structure-function instance

The `sf_int_t` data type contains an `interaction_t` object (it is an extension of this type) and a pointer to the `sf_data_t` configuration data. This interaction, or copies of it, is used to implement structure-function kinematics and dynamics in the context of process evaluation.

The status code `status` tells whether the interaction is undefined, has defined kinematics (but matrix elements invalid), or is completely defined. There is also a status code for failure. The implementation is responsible for updating the status.

The entries `mi2`, `mr2`, and `mo2` hold the squared invariant masses of the incoming, radiated, and outgoing particle, respectively. They are supposed to be set upon initialization, but could also be varied event by event.

If the radiated or outgoing mass is nonzero, we may need to apply an on-shell projection. The projection mode is stored as `on_shell_mode`.

The array `beam_index` is the list of beams on which this structure function applies (1, 2, or both). The arrays `incoming`, `radiated`, and `outgoing` contain the indices of the respective particle sets within the interaction, for convenient lookup. The array `par_index` indicates the MC input parameters that this entry will use up in the structure-function chain. The first parameter (or the first two, for a spectrum) in this array determines the momentum fraction and is thus subject to global mappings.

In the abstract base type, we do not implement the data pointer. This allows us to restrict its type in the implementations.

⟨*SF base: public*⟩+≡
```
public :: sf_int_t
```

⟨*SF base: types*⟩+≡
```
type, abstract, extends (interaction_t) :: sf_int_t
   integer :: status = SF_UNDEFINED
   real(default), dimension(:), allocatable :: mi2
   real(default), dimension(:), allocatable :: mr2
   real(default), dimension(:), allocatable :: mo2
   integer :: on_shell_mode = KEEP_ENERGY
   logical :: qmin_defined = .false.
   logical :: qmax_defined = .false.
   real(default), dimension(:), allocatable :: qmin
   real(default), dimension(:), allocatable :: qmax
   integer, dimension(:), allocatable :: beam_index
   integer, dimension(:), allocatable :: incoming
   integer, dimension(:), allocatable :: radiated
   integer, dimension(:), allocatable :: outgoing
   integer, dimension(:), allocatable :: par_index
   integer, dimension(:), allocatable :: par_primary
 contains
 ⟨SF base: sf int: TBP⟩
end type sf_int_t
```

Status codes. The codes that refer to links, masks, and connections, apply to structure-function chains only.

The status codes are public.

⟨*SF base: parameters*⟩≡
```
integer, parameter, public :: SF_UNDEFINED = 0
integer, parameter, public :: SF_INITIAL = 1
integer, parameter, public :: SF_DONE_LINKS = 2
integer, parameter, public :: SF_FAILED_MASK = 3
integer, parameter, public :: SF_DONE_MASK = 4
integer, parameter, public :: SF_FAILED_CONNECTIONS = 5
integer, parameter, public :: SF_DONE_CONNECTIONS = 6
integer, parameter, public :: SF_SEED_KINEMATICS = 10
integer, parameter, public :: SF_FAILED_KINEMATICS = 11
integer, parameter, public :: SF_DONE_KINEMATICS = 12
integer, parameter, public :: SF_FAILED_EVALUATION = 13
integer, parameter, public :: SF_EVALUATED = 20
```

Write a string version of the status code:

⟨*SF base: procedures*⟩+≡

```
subroutine write_sf_status (status, u)
  integer, intent(in) :: status
  integer, intent(in) :: u
  select case (status)
  case (SF_UNDEFINED)
     write (u, "(1x,'[',A,']')")  "undefined"
  case (SF_INITIAL)
     write (u, "(1x,'[',A,']')")  "initialized"
  case (SF_DONE_LINKS)
     write (u, "(1x,'[',A,']')")  "links set"
  case (SF_FAILED_MASK)
     write (u, "(1x,'[',A,']')")  "mask mismatch"
  case (SF_DONE_MASK)
     write (u, "(1x,'[',A,']')")  "mask set"
  case (SF_FAILED_CONNECTIONS)
     write (u, "(1x,'[',A,']')")  "connections failed"
  case (SF_DONE_CONNECTIONS)
     write (u, "(1x,'[',A,']')")  "connections set"
  case (SF_SEED_KINEMATICS)
     write (u, "(1x,'[',A,']')")  "incoming momenta set"
  case (SF_FAILED_KINEMATICS)
     write (u, "(1x,'[',A,']')")  "kinematics failed"
  case (SF_DONE_KINEMATICS)
     write (u, "(1x,'[',A,']')")  "kinematics set"
  case (SF_FAILED_EVALUATION)
     write (u, "(1x,'[',A,']')")  "evaluation failed"
  case (SF_EVALUATED)
     write (u, "(1x,'[',A,']')")  "evaluated"
  end select
end subroutine write_sf_status
```

This is the basic output routine. Display status and interaction.

⟨*SF base: sf int: TBP*⟩≡

```
procedure :: base_write => sf_int_base_write
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_int_base_write (object, unit, testflag)
  class(sf_int_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)", advance="no")  "SF instance:"
  call write_sf_status (object%status, u)
  if (allocated (object%beam_index)) &
       write (u, "(3x,A,2(1x,I0))")  "beam      =", object%beam_index
  if (allocated (object%incoming)) &
       write (u, "(3x,A,2(1x,I0))")  "incoming  =", object%incoming
  if (allocated (object%radiated)) &
       write (u, "(3x,A,2(1x,I0))")  "radiated  =", object%radiated
  if (allocated (object%outgoing)) &
       write (u, "(3x,A,2(1x,I0))")  "outgoing  =", object%outgoing
```

```
    if (allocated (object%par_index)) &
         write (u, "(3x,A,2(1x,I0))")  "parameter =", object%par_index
    if (object%qmin_defined) &
         write (u, "(3x,A,1x," // FMT_19 // ")")  "q_min     =", object%qmin
    if (object%qmax_defined) &
         write (u, "(3x,A,1x," // FMT_19 // ")")  "q_max     =", object%qmax
    call object%interaction_t%basic_write (u, testflag = testflag)
  end subroutine sf_int_base_write
```

The type string identifies the structure function class, and possibly more details about the structure function.

⟨*SF base: sf int: TBP*⟩+≡

```
  procedure (sf_int_type_string), deferred :: type_string
```

⟨*SF base: interfaces*⟩+≡

```
  abstract interface
     function sf_int_type_string (object) result (string)
       import
       class(sf_int_t), intent(in) :: object
       type(string_t) :: string
     end function sf_int_type_string
  end interface
```

Output of the concrete object. We should not forget to call the output routine for the base type.

⟨*SF base: sf int: TBP*⟩+≡

```
  procedure (sf_int_write), deferred :: write
```

⟨*SF base: interfaces*⟩+≡

```
  abstract interface
     subroutine sf_int_write (object, unit, testflag)
       import
       class(sf_int_t), intent(in) :: object
       integer, intent(in), optional :: unit
       logical, intent(in), optional :: testflag
     end subroutine sf_int_write
  end interface
```

Basic initialization: set the invariant masses for the particles and initialize the interaction. The caller should then add states to the interaction and freeze it.

The dimension of the mask should be equal to the sum of the dimensions of the mass-squared arrays, which determine incoming, radiated, and outgoing particles, respectively.

Optionally, we can define minimum and maximum values for the momentum transfer to the outgoing particle(s). If all masses are zero, this is actually required for non-collinear splitting.

⟨*SF base: sf int: TBP*⟩+≡

```
  procedure :: base_init => sf_int_base_init
```

⟨*SF base: procedures*⟩+≡

```
  subroutine sf_int_base_init &
       (sf_int, mask, mi2, mr2, mo2, qmin, qmax, hel_lock)
    class(sf_int_t), intent(out) :: sf_int
```

```
    type (quantum_numbers_mask_t), dimension(:), intent(in) :: mask
    real(default), dimension(:), intent(in) :: mi2, mr2, mo2
    real(default), dimension(:), intent(in), optional :: qmin, qmax
    integer, dimension(:), intent(in), optional :: hel_lock
    allocate (sf_int%mi2 (size (mi2)))
    sf_int%mi2 = mi2
    allocate (sf_int%mr2 (size (mr2)))
    sf_int%mr2 = mr2
    allocate (sf_int%mo2 (size (mo2)))
    sf_int%mo2 = mo2
    if (present (qmin)) then
       sf_int%qmin_defined = .true.
       allocate (sf_int%qmin (size (qmin)))
       sf_int%qmin = qmin
    end if
    if (present (qmax)) then
       sf_int%qmax_defined = .true.
       allocate (sf_int%qmax (size (qmax)))
       sf_int%qmax = qmax
    end if
    call sf_int%interaction_t%basic_init &
         (size (mi2), 0, size (mr2) + size (mo2), &
         mask = mask, hel_lock = hel_lock, set_relations = .true.)
  end subroutine sf_int_base_init
```

Set the indices of the incoming, radiated, and outgoing particles, respectively.

⟨*SF base: sf int: TBP*⟩+≡

```
  procedure :: set_incoming => sf_int_set_incoming
  procedure :: set_radiated => sf_int_set_radiated
  procedure :: set_outgoing => sf_int_set_outgoing
```

⟨*SF base: procedures*⟩+≡

```
  subroutine sf_int_set_incoming (sf_int, incoming)
    class(sf_int_t), intent(inout) :: sf_int
    integer, dimension(:), intent(in) :: incoming
    allocate (sf_int%incoming (size (incoming)))
    sf_int%incoming = incoming
  end subroutine sf_int_set_incoming

  subroutine sf_int_set_radiated (sf_int, radiated)
    class(sf_int_t), intent(inout) :: sf_int
    integer, dimension(:), intent(in) :: radiated
    allocate (sf_int%radiated (size (radiated)))
    sf_int%radiated = radiated
  end subroutine sf_int_set_radiated

  subroutine sf_int_set_outgoing (sf_int, outgoing)
    class(sf_int_t), intent(inout) :: sf_int
    integer, dimension(:), intent(in) :: outgoing
    allocate (sf_int%outgoing (size (outgoing)))
    sf_int%outgoing = outgoing
  end subroutine sf_int_set_outgoing
```

Initialization. This proceeds via an abstract data object, which for the actual implementation should have the matching concrete type. Since all implementations have the same signature, we can prepare a deferred procedure. The data object will become the target of a corresponding pointer within the `sf_int_t` implementation.

This should call the previous procedure.

⟨*SF base: sf int: TBP*⟩+≡
```
procedure (sf_int_init), deferred :: init
```

⟨*SF base: interfaces*⟩+≡
```
abstract interface
   subroutine sf_int_init (sf_int, data)
     import
     class(sf_int_t), intent(out) :: sf_int
     class(sf_data_t), intent(in), target :: data
   end subroutine sf_int_init
end interface
```

Complete initialization. This routine contains initializations that can only be performed after the interaction object got its final shape, i.e., redundant helicities have been eliminated by matching with beams and process.

The default implementation does nothing.

The `target` attribute is formally required since some overriding implementations use a temporary pointer (iterator) to the state-matrix component. It doesn't appear to make a real difference, though.

⟨*SF base: sf int: TBP*⟩+≡
```
procedure :: setup_constants => sf_int_setup_constants
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_int_setup_constants (sf_int)
  class(sf_int_t), intent(inout), target :: sf_int
end subroutine sf_int_setup_constants
```

Set beam indices, i.e., the beam(s) on which this structure function applies.

⟨*SF base: sf int: TBP*⟩+≡
```
procedure :: set_beam_index => sf_int_set_beam_index
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_int_set_beam_index (sf_int, beam_index)
  class(sf_int_t), intent(inout) :: sf_int
  integer, dimension(:), intent(in) :: beam_index
  allocate (sf_int%beam_index (size (beam_index)))
  sf_int%beam_index = beam_index
end subroutine sf_int_set_beam_index
```

Set parameter indices, indicating which MC input parameters are to be used for evaluating this structure function.

⟨*SF base: sf int: TBP*⟩+≡
```
procedure :: set_par_index => sf_int_set_par_index
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_int_set_par_index (sf_int, par_index)
  class(sf_int_t), intent(inout) :: sf_int
  integer, dimension(:), intent(in) :: par_index
  allocate (sf_int%par_index (size (par_index)))
  sf_int%par_index = par_index
end subroutine sf_int_set_par_index
```

Initialize the structure-function kinematics, setting incoming momenta. We assume that array shapes match.

Three versions. The first version relies on the momenta being linked to another interaction. The second version sets the momenta explicitly. In the third version, we first compute momenta for the specified energies and store those.

⟨*SF base: sf int: TBP*⟩+≡

```
generic :: seed_kinematics => sf_int_receive_momenta
generic :: seed_kinematics => sf_int_seed_momenta
generic :: seed_kinematics => sf_int_seed_energies
procedure :: sf_int_receive_momenta
procedure :: sf_int_seed_momenta
procedure :: sf_int_seed_energies
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_int_receive_momenta (sf_int)
  class(sf_int_t), intent(inout) :: sf_int
  if (sf_int%status >= SF_INITIAL) then
     call sf_int%receive_momenta ()
     sf_int%status = SF_SEED_KINEMATICS
  end if
end subroutine sf_int_receive_momenta

subroutine sf_int_seed_momenta (sf_int, k)
  class(sf_int_t), intent(inout) :: sf_int
  type(vector4_t), dimension(:), intent(in) :: k
  if (sf_int%status >= SF_INITIAL) then
     call sf_int%set_momenta (k, outgoing=.false.)
     sf_int%status = SF_SEED_KINEMATICS
  end if
end subroutine sf_int_seed_momenta

subroutine sf_int_seed_energies (sf_int, E)
  class(sf_int_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: E
  type(vector4_t), dimension(:), allocatable :: k
  integer :: j
  if (sf_int%status >= SF_INITIAL) then
     allocate (k (size (E)))
     if (all (E**2 >= sf_int%mi2)) then
        do j = 1, size (E)
           k(j) = vector4_moving (E(j), &
                (3-2*j) * sqrt (E(j)**2 - sf_int%mi2(j)), 3)
        end do
        call sf_int%seed_kinematics (k)
     end if
```

```
    end if
  end subroutine sf_int_seed_energies
```

Tell if in generator mode. By default, this is false. To be overridden where appropriate; we may refer to the `is_generator` method of the `data` component in the concrete type.

⟨*SF base: sf int: TBP*⟩+≡

```
  procedure :: is_generator => sf_int_is_generator
```

⟨*SF base: procedures*⟩+≡

```
  function sf_int_is_generator (sf_int) result (flag)
    class(sf_int_t), intent(in) :: sf_int
    logical :: flag
    flag = .false.
  end function sf_int_is_generator
```

Generate free parameters `r`. Parameters are free if they do not correspond to integration parameters (i.e., are bound), but are generated by the structure function object itself. By default, all parameters are bound, and the output values of this procedure will be discarded. With free parameters, we have to override this procedure.

The value `x_free` is the renormalization factor of the total energy that corresponds to the free parameters. If there are no free parameters, the procedure will not change its value, which starts as unity. Otherwise, the fraction is typically decreased, but may also be increased in some cases.

⟨*SF base: sf int: TBP*⟩+≡

```
  procedure :: generate_free => sf_int_generate_free
```

⟨*SF base: procedures*⟩+≡

```
  subroutine sf_int_generate_free (sf_int, r, rb,  x_free)
    class(sf_int_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: r, rb
    real(default), intent(inout) :: x_free
    r = 0
    rb= 1
  end subroutine sf_int_generate_free
```

Complete the structure-function kinematics, derived from an input parameter (array) r between 0 and 1. The interaction momenta are calculated, and we return x (the momentum fraction), and f (the Jacobian factor for the map $r \to x$), if `map` is set.

If the `map` flag is unset, r and x values will coincide, and f will become unity. If it is set, the structure-function implementation chooses a convenient mapping from r to x with Jacobian f.

In the `inverse_kinematics` variant, we exchange the intent of `x` and `r`. The momenta are calculated only if the optional flag `set_momenta` is present and set.

⟨*SF base: sf int: TBP*⟩+≡

```
  procedure (sf_int_complete_kinematics), deferred :: complete_kinematics
  procedure (sf_int_inverse_kinematics), deferred :: inverse_kinematics
```

⟨*SF base: interfaces*⟩+≡

```
abstract interface
   subroutine sf_int_complete_kinematics (sf_int, x, f, r, rb, map)
     import
     class(sf_int_t), intent(inout) :: sf_int
     real(default), dimension(:), intent(out) :: x
     real(default), intent(out) :: f
     real(default), dimension(:), intent(in) :: r
     real(default), dimension(:), intent(in) :: rb
     logical, intent(in) :: map
   end subroutine sf_int_complete_kinematics
end interface

abstract interface
   subroutine sf_int_inverse_kinematics (sf_int, x, f, r, rb, map, &
        set_momenta)
     import
     class(sf_int_t), intent(inout) :: sf_int
     real(default), dimension(:), intent(in) :: x
     real(default), intent(out) :: f
     real(default), dimension(:), intent(out) :: r
     real(default), dimension(:), intent(out) :: rb
     logical, intent(in) :: map
     logical, intent(in), optional :: set_momenta
   end subroutine sf_int_inverse_kinematics
end interface
```

Single splitting: compute momenta, given x input parameters. We assume that the incoming momentum is set. The status code is set to `SF_FAILED_KINEMATICS` if the x array does not correspond to a valid momentum configuration. Otherwise, it is updated to `SF_DONE_KINEMATICS`.

We force the outgoing particle on-shell. The on-shell projection is determined by the `on_shell_mode`. The radiated particle should already be on shell.

⟨*SF base: sf int: TBP*⟩+≡

```
procedure :: split_momentum => sf_int_split_momentum
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_int_split_momentum (sf_int, x, xb1)
  class(sf_int_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(in) :: xb1
  type(vector4_t) :: k
  type(vector4_t), dimension(2) :: q
  type(splitting_data_t) :: sd
  real(default) :: E1, E2
  logical :: fail
  if (sf_int%status >= SF_SEED_KINEMATICS) then
     k = sf_int%get_momentum (1)
     call sd%init (k, &
          sf_int%mi2(1), sf_int%mr2(1), sf_int%mo2(1), &
          collinear = size (x) == 1)
     call sd%set_t_bounds (x(1), xb1)
     select case (size (x))
     case (1)
```

```
      case (3)
         if (sf_int%qmax_defined) then
            if (sf_int%qmin_defined) then
               call sd%sample_t (x(2), &
                    t0 = - sf_int%qmax(1) ** 2, t1 = - sf_int%qmin(1) ** 2)
            else
               call sd%sample_t (x(2), &
                    t0 = - sf_int%qmax(1) ** 2)
            end if
         else
            if (sf_int%qmin_defined) then
               call sd%sample_t (x(2), t1 = - sf_int%qmin(1) ** 2)
            else
               call sd%sample_t (x(2))
            end if
         end if
         call sd%sample_phi (x(3))
      case default
         call msg_bug ("Structure function: impossible number of parameters")
      end select
      q = sd%split_momentum (k)
      call on_shell (q, [sf_int%mr2, sf_int%mo2], &
           sf_int%on_shell_mode)
      call sf_int%set_momenta (q, outgoing=.true.)
      E1 = energy (q(1))
      E2 = energy (q(2))
      fail = E1 < 0 .or. E2 < 0 &
           .or. E1 ** 2 < sf_int%mr2(1) &
           .or. E2 ** 2 < sf_int%mo2(1)
      if (fail) then
         sf_int%status = SF_FAILED_KINEMATICS
      else
         sf_int%status = SF_DONE_KINEMATICS
      end if
   end if
end subroutine sf_int_split_momentum
```

Pair splitting: two incoming momenta, two radiated, two outgoing. This is simple because we insist on all momenta being collinear.

⟨*SF base: sf int: TBP*⟩+≡

```
procedure :: split_momenta => sf_int_split_momenta
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_int_split_momenta (sf_int, x, xb1)
  class(sf_int_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), dimension(:), intent(in) :: xb1
  type(vector4_t), dimension(2) :: k
  type(vector4_t), dimension(4) :: q
  real(default), dimension(4) :: E
  logical :: fail
  if (sf_int%status >= SF_SEED_KINEMATICS) then
     select case (size (x))
     case (2)
```

```
      case default
         call msg_bug ("Pair structure function: recoil requested &
              &but not implemented yet")
      end select
      k(1) = sf_int%get_momentum (1)
      k(2) = sf_int%get_momentum (2)
      q(1:2) = xb1 * k
      q(3:4) = x * k
      select case (size (sf_int%mr2))
      case (2)
         call on_shell (q, &
              [sf_int%mr2(1), sf_int%mr2(2), &
              sf_int%mo2(1), sf_int%mo2(2)], &
              sf_int%on_shell_mode)
         call sf_int%set_momenta (q, outgoing=.true.)
         E = energy (q)
         fail = any (E < 0) &
              .or. any (E(1:2) ** 2 < sf_int%mr2) &
              .or. any (E(3:4) ** 2 < sf_int%mo2)
      case default;  call msg_bug ("split momenta: incorrect use")
      end select
      if (fail) then
         sf_int%status = SF_FAILED_KINEMATICS
      else
         sf_int%status = SF_DONE_KINEMATICS
      end if
   end if
end subroutine sf_int_split_momenta
```

Pair spectrum: the reduced version of the previous splitting, without radiated momenta.

⟨*SF base: sf int: TBP*⟩+≡

```
procedure :: reduce_momenta => sf_int_reduce_momenta
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_int_reduce_momenta (sf_int, x)
  class(sf_int_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  type(vector4_t), dimension(2) :: k
  type(vector4_t), dimension(2) :: q
  real(default), dimension(2) :: E
  logical :: fail
  if (sf_int%status >= SF_SEED_KINEMATICS) then
     select case (size (x))
     case (2)
     case default
        call msg_bug ("Pair spectrum: recoil requested &
             &but not implemented yet")
     end select
     k(1) = sf_int%get_momentum (1)
     k(2) = sf_int%get_momentum (2)
     q = x * k
     call on_shell (q, &
          [sf_int%mo2(1), sf_int%mo2(2)], &
```

```
            sf_int%on_shell_mode)
         call sf_int%set_momenta (q, outgoing=.true.)
         E = energy (q)
         fail = any (E < 0) &
              .or. any (E ** 2 < sf_int%mo2)
         if (fail) then
            sf_int%status = SF_FAILED_KINEMATICS
         else
            sf_int%status = SF_DONE_KINEMATICS
         end if
      end if
   end subroutine sf_int_reduce_momenta
```

The inverse procedure: we compute the `x` array from the momentum configuration. In an overriding TBP, we may also set internal data that depend on this, for convenience.

NOTE: Here and above, the single-particle case is treated in detail, allowing for non-collinearity and non-vanishing masses and nontrivial momentum-transfer bounds. For the pair case, we currently implement only collinear splitting and assume massless particles. This should be improved.

NOTE: We should also improve this in the sense of extracting both x and $\bar{x}$, where for the latter we may consult the beam remnant. The current implementation is not sensitive to small $\bar{x}$ values.

⟨*SF base: sf int: TBP*⟩+≡

```
   procedure :: recover_x => sf_int_recover_x
   procedure :: base_recover_x => sf_int_recover_x
```

⟨*SF base: procedures*⟩+≡

```
   subroutine sf_int_recover_x (sf_int, x, x_free)
     class(sf_int_t), intent(inout) :: sf_int
     real(default), dimension(:), intent(out) :: x
     real(default), intent(inout), optional :: x_free
     type(vector4_t), dimension(:), allocatable :: k
     type(vector4_t), dimension(:), allocatable :: q
     type(splitting_data_t) :: sd
     if (sf_int%status >= SF_SEED_KINEMATICS) then
        allocate (k (sf_int%interaction_t%get_n_in ()))
        allocate (q (sf_int%interaction_t%get_n_out ()))
        k = sf_int%get_momenta (outgoing=.false.)
        q = sf_int%get_momenta (outgoing=.true.)
        select case (size (k))
        case (1)
           call sd%init (k(1), &
                sf_int%mi2(1), sf_int%mr2(1), sf_int%mo2(1), &
                collinear = size (x) == 1)
           call sd%recover (k(1), q(2), sf_int%on_shell_mode)
           x(1) = sd%get_x ()
           select case (size (x))
           case (1)
           case (3)
              if (sf_int%qmax_defined) then
                 if (sf_int%qmin_defined) then
                    call sd%inverse_t (x(2), &
```

```
                t0 = - sf_int%qmax(1) ** 2, t1 = - sf_int%qmin(1) ** 2)
            else
               call sd%inverse_t (x(2), &
                    t0 = - sf_int%qmax(1) ** 2)
            end if
         else
            if (sf_int%qmin_defined) then
               call sd%inverse_t (x(2), t1 = - sf_int%qmin(1) ** 2)
            else
               call sd%inverse_t (x(2))
            end if
         end if
         call sd%inverse_phi (x(3))
      case default
         call msg_bug ("Structure function: impossible number &
              &of parameters")
      end select
   case (2)
      select case (size (x))
      case (2)
      case default
         call msg_bug ("Pair structure function: recoil requested &
              &but not implemented yet")
      end select
      select case (sf_int%on_shell_mode)
      case (KEEP_ENERGY)
         select case (size (q))
         case (4)
            x = energy (q(3:4)) / energy (k)
         case (2)
            x = energy (q) / energy (k)
         end select
      case (KEEP_MOMENTUM)
         select case (size (q))
         case (4)
            x = longitudinal_part (q(3:4)) / longitudinal_part (k)
         case (2)
            x = longitudinal_part (q) / longitudinal_part (k)
         end select
      end select
   end select
  end if
end subroutine sf_int_recover_x
```

Apply the structure function, i.e., evaluate the interaction. For the calculation, we may use the stored momenta, any further information stored inside the `sf_int` implementation during kinematics setup, and the given energy scale. It may happen that for the given kinematics the value is not defined. This should be indicated by the status code.

⟨*SF base: sf int: TBP*⟩+≡

```
procedure (sf_int_apply), deferred :: apply
```

⟨*SF base: interfaces*⟩+≡

```
abstract interface
```

```
    subroutine sf_int_apply (sf_int, scale)
      import
      class(sf_int_t), intent(inout) :: sf_int
      real(default), intent(in) :: scale
    end subroutine sf_int_apply
end interface
```

15.5.4 Accessing the structure function

Return metadata. Once `interaction_t` is rewritten in OO, some of this will be inherited.

The number of outgoing is equal to the number of incoming particles. The radiated particles are the difference.

⟨*SF base: sf int: TBP*⟩+≡

```
procedure :: get_n_in => sf_int_get_n_in
procedure :: get_n_rad => sf_int_get_n_rad
procedure :: get_n_out => sf_int_get_n_out
```

⟨*SF base: procedures*⟩+≡

```
pure function sf_int_get_n_in (object) result (n_in)
  class(sf_int_t), intent(in) :: object
  integer :: n_in
  n_in = object%interaction_t%get_n_in ()
end function sf_int_get_n_in

pure function sf_int_get_n_rad (object) result (n_rad)
  class(sf_int_t), intent(in) :: object
  integer :: n_rad
  n_rad = object%interaction_t%get_n_out () &
       - object%interaction_t%get_n_in ()
end function sf_int_get_n_rad

pure function sf_int_get_n_out (object) result (n_out)
  class(sf_int_t), intent(in) :: object
  integer :: n_out
  n_out = object%interaction_t%get_n_in ()
end function sf_int_get_n_out
```

Number of matrix element entries in the interaction:

⟨*SF base: sf int: TBP*⟩+≡

```
procedure :: get_n_states => sf_int_get_n_states
```

⟨*SF base: procedures*⟩+≡

```
function sf_int_get_n_states (sf_int) result (n_states)
  class(sf_int_t), intent(in) :: sf_int
  integer :: n_states
  n_states = sf_int%get_n_matrix_elements ()
end function sf_int_get_n_states
```

Return a specific state as a quantum-number array.

⟨*SF base: sf int: TBP*⟩+≡

```
procedure :: get_state => sf_int_get_state
```

⟨*SF base: procedures*⟩+≡

```
function sf_int_get_state (sf_int, i) result (qn)
  class(sf_int_t), intent(in) :: sf_int
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  integer, intent(in) :: i
  allocate (qn (sf_int%get_n_tot ()))
  qn = sf_int%get_quantum_numbers (i)
end function sf_int_get_state
```

Return the matrix-element values for all states. We can assume that the matrix elements are real, so we take the real part.

⟨*SF base: sf int: TBP*⟩+≡

```
procedure :: get_values => sf_int_get_values
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_int_get_values (sf_int, value)
  class(sf_int_t), intent(in) :: sf_int
  real(default), dimension(:), intent(out) :: value
  integer :: i
  if (sf_int%status >= SF_EVALUATED) then
     do i = 1, size (value)
        value(i) = real (sf_int%get_matrix_element (i))
     end do
  else
     value = 0
  end if
end subroutine sf_int_get_values
```

15.5.5 Direct calculations

Compute a structure function value (array) directly, given an array of x values and a scale. If the energy is also given, we initialize the kinematics for that energy, otherwise take it from a previous run.

We assume that the `E` array has dimension `n_in`, and the `x` array has `n_par`.

⟨*SF base: sf int: TBP*⟩+≡

```
procedure :: compute_values => sf_int_compute_values
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_int_compute_values (sf_int, value, x, xb, scale, E)
  class(sf_int_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: value
  real(default), dimension(:), intent(in) :: x
  real(default), dimension(:), intent(in) :: xb
  real(default), intent(in) :: scale
  real(default), dimension(:), intent(in), optional :: E
  real(default), dimension(size (x)) :: xx
  real(default) :: f
  if (present (E))  call sf_int%seed_kinematics (E)
  if (sf_int%status >= SF_SEED_KINEMATICS) then
     call sf_int%complete_kinematics (xx, f, x, xb, map=.false.)
     call sf_int%apply (scale)
     call sf_int%get_values (value)
```

```
      value = value * f
   else
      value = 0
   end if
end subroutine sf_int_compute_values
```

Compute just a single value for one of the states, i.e., throw the others away.

⟨*SF base: sf int: TBP*⟩+≡

```
procedure :: compute_value => sf_int_compute_value
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_int_compute_value &
     (sf_int, i_state, value, x, xb, scale, E)
   class(sf_int_t), intent(inout) :: sf_int
   integer, intent(in) :: i_state
   real(default), intent(out) :: value
   real(default), dimension(:), intent(in) :: x
   real(default), dimension(:), intent(in) :: xb
   real(default), intent(in) :: scale
   real(default), dimension(:), intent(in), optional :: E
   real(default), dimension(:), allocatable :: value_array
   if (sf_int%status >= SF_INITIAL) then
      allocate (value_array (sf_int%get_n_states ()))
      call sf_int%compute_values (value_array, x, xb, scale, E)
      value = value_array(i_state)
   else
      value = 0
   end if
end subroutine sf_int_compute_value
```

15.5.6 Structure-function instance

This is a wrapper for `sf_int_t` objects, such that we can build an array with different structure-function types. The structure-function contains an array (a sequence) of `sf_int_t` objects.

The object, it holds the evaluator that connects the preceding part of the structure-function chain to the current interaction.

It also stores the input and output parameter values for the contained structure function. The `r` array has a second dimension, corresponding to the mapping channels in a multi-channel configuration. There is a Jacobian entry `f` for each channel. The corresponding logical array `mapping` tells whether we apply the mapping appropriate for the current structure function in this channel. The `x` parameter values (energy fractions) are common to all channels.

⟨*SF base: types*⟩+≡

```
type :: sf_instance_t
   class(sf_int_t), allocatable :: int
   type(evaluator_t) :: eval
   real(default), dimension(:,:), allocatable :: r
   real(default), dimension(:,:), allocatable :: rb
   real(default), dimension(:), allocatable :: f
   logical, dimension(:), allocatable :: m
```

```
      real(default), dimension(:), allocatable :: x
   end type sf_instance_t
```

15.5.7 Structure-function chain

A chain is an array of structure functions `sf`, initiated by a beam setup. We do not use this directly for evaluation, but create instances with copies of the contained interactions.

`n_par` is the total number of parameters that is necessary for completely determining the structure-function chain. `n_bound` is the number of MC input parameters that are requested from the integrator. The difference of `n_par` and `n_bound` is the number of free parameters, which are generated by a structure-function object in generator mode.

⟨*SF base: public*⟩+≡

```
   public :: sf_chain_t
```

⟨*SF base: types*⟩+≡

```
   type, extends (beam_t) :: sf_chain_t
      type(beam_data_t), pointer :: beam_data => null ()
      integer :: n_in = 0
      integer :: n_strfun = 0
      integer :: n_par = 0
      integer :: n_bound = 0
      type(sf_instance_t), dimension(:), allocatable :: sf
      logical :: trace_enable = .false.
      integer :: trace_unit = 0
    contains
    ⟨SF base: sf chain: TBP⟩
   end type sf_chain_t
```

Finalizer.

⟨*SF base: sf chain: TBP*⟩≡

```
   procedure :: final => sf_chain_final
```

⟨*SF base: procedures*⟩+≡

```
   subroutine sf_chain_final (object)
     class(sf_chain_t), intent(inout) :: object
     integer :: i
     call object%final_tracing ()
     if (allocated (object%sf)) then
        do i = 1, size (object%sf, 1)
           associate (sf => object%sf(i))
             if (allocated (sf%int)) then
                call sf%int%final ()
             end if
           end associate
        end do
     end if
     call beam_final (object%beam_t)
   end subroutine sf_chain_final
```

Output.

⟨*SF base: sf chain: TBP*⟩+≡

```
procedure :: write => sf_chain_write
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_write (object, unit)
  class(sf_chain_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Incoming particles / structure-function chain:"
  if (associated (object%beam_data)) then
     write (u, "(3x,A,I0)")  "n_in      = ", object%n_in
     write (u, "(3x,A,I0)")  "n_strfun  = ", object%n_strfun
     write (u, "(3x,A,I0)")  "n_par     = ", object%n_par
     if (object%n_par /= object%n_bound) then
        write (u, "(3x,A,I0)")  "n_bound   = ", object%n_bound
     end if
     call object%beam_data%write (u)
     call write_separator (u)
     call beam_write (object%beam_t, u)
     if (allocated (object%sf)) then
        do i = 1, object%n_strfun
           associate (sf => object%sf(i))
             call write_separator (u)
             if (allocated (sf%int)) then
                call sf%int%write (u)
             else
                write (u, "(1x,A)")  "SF instance: [undefined]"
             end if
           end associate
        end do
     end if
  else
     write (u, "(3x,A)")  "[undefined]"
  end if
end subroutine sf_chain_write
```

Initialize: setup beams. The `beam_data` target must remain valid for the lifetime of the chain, since we just establish a pointer. The structure-function configuration array is used to initialize the individual structure-function entries. The target attribute is needed because the `sf_int` entries establish pointers to the configuration data.

⟨*SF base: sf chain: TBP*⟩+≡

```
procedure :: init => sf_chain_init
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_init (sf_chain, beam_data, sf_config)
  class(sf_chain_t), intent(out) :: sf_chain
  type(beam_data_t), intent(in), target :: beam_data
  type(sf_config_t), dimension(:), intent(in), optional, target :: sf_config
  integer :: i
  sf_chain%beam_data => beam_data
  sf_chain%n_in = beam_data%get_n_in ()
```

```
    call beam_init (sf_chain%beam_t, beam_data)
    if (present (sf_config)) then
       sf_chain%n_strfun = size (sf_config)
       allocate (sf_chain%sf (sf_chain%n_strfun))
       do i = 1, sf_chain%n_strfun
          call sf_chain%set_strfun (i, sf_config(i)%i, sf_config(i)%data)
       end do
    end if
  end subroutine sf_chain_init
```

Receive the beam momenta from a source to which the beam interaction is linked.

⟨*SF base: sf chain: TBP*⟩+≡

```
  procedure :: receive_beam_momenta => sf_chain_receive_beam_momenta
```

⟨*SF base: procedures*⟩+≡

```
  subroutine sf_chain_receive_beam_momenta (sf_chain)
    class(sf_chain_t), intent(inout), target :: sf_chain
    type(interaction_t), pointer :: beam_int
    beam_int => sf_chain%get_beam_int_ptr ()
    call beam_int%receive_momenta ()
  end subroutine sf_chain_receive_beam_momenta
```

Explicitly set the beam momenta.

⟨*SF base: sf chain: TBP*⟩+≡

```
  procedure :: set_beam_momenta => sf_chain_set_beam_momenta
```

⟨*SF base: procedures*⟩+≡

```
  subroutine sf_chain_set_beam_momenta (sf_chain, p)
    class(sf_chain_t), intent(inout) :: sf_chain
    type(vector4_t), dimension(:), intent(in) :: p
    call beam_set_momenta (sf_chain%beam_t, p)
  end subroutine sf_chain_set_beam_momenta
```

Set a structure-function entry. We use the `data` input to allocate the `int` structure-function instance with appropriate type, then initialize the entry. The entry establishes a pointer to `data`.

The index `i` is the structure-function index in the chain.

⟨*SF base: sf chain: TBP*⟩+≡

```
  procedure :: set_strfun => sf_chain_set_strfun
```

⟨*SF base: procedures*⟩+≡

```
  subroutine sf_chain_set_strfun (sf_chain, i, beam_index, data)
    class(sf_chain_t), intent(inout) :: sf_chain
    integer, intent(in) :: i
    integer, dimension(:), intent(in) :: beam_index
    class(sf_data_t), intent(in), target :: data
    integer :: n_par, j
    n_par = data%get_n_par ()
    call data%allocate_sf_int (sf_chain%sf(i)%int)
    associate (sf_int => sf_chain%sf(i)%int)
      call sf_int%init (data)
      call sf_int%set_beam_index (beam_index)
```

```
      call sf_int%set_par_index &
           ([(j, j = sf_chain%n_par + 1, sf_chain%n_par + n_par)])
      sf_chain%n_par = sf_chain%n_par + n_par
      if (.not. data%is_generator ()) then
         sf_chain%n_bound = sf_chain%n_bound + n_par
      end if
    end associate
  end subroutine sf_chain_set_strfun
```

Return the number of structure-function parameters.

⟨*SF base: sf chain: TBP*⟩+≡

```
  procedure :: get_n_par => sf_chain_get_n_par
  procedure :: get_n_bound => sf_chain_get_n_bound
```

⟨*SF base: procedures*⟩+≡

```
  function sf_chain_get_n_par (sf_chain) result (n)
    class(sf_chain_t), intent(in) :: sf_chain
    integer :: n
    n = sf_chain%n_par
  end function sf_chain_get_n_par

  function sf_chain_get_n_bound (sf_chain) result (n)
    class(sf_chain_t), intent(in) :: sf_chain
    integer :: n
    n = sf_chain%n_bound
  end function sf_chain_get_n_bound
```

Return a pointer to the beam interaction.

⟨*SF base: sf chain: TBP*⟩+≡

```
  procedure :: get_beam_int_ptr => sf_chain_get_beam_int_ptr
```

⟨*SF base: procedures*⟩+≡

```
  function sf_chain_get_beam_int_ptr (sf_chain) result (int)
    class(sf_chain_t), intent(in), target :: sf_chain
    type(interaction_t), pointer :: int
    int => beam_get_int_ptr (sf_chain%beam_t)
  end function sf_chain_get_beam_int_ptr
```

Enable the trace feature: record structure function data (input parameters, x values, evaluation result) to an external file.

⟨*SF base: sf chain: TBP*⟩+≡

```
  procedure :: setup_tracing => sf_chain_setup_tracing
  procedure :: final_tracing => sf_chain_final_tracing
```

⟨*SF base: procedures*⟩+≡

```
  subroutine sf_chain_setup_tracing (sf_chain, file)
    class(sf_chain_t), intent(inout) :: sf_chain
    type(string_t), intent(in) :: file
    if (sf_chain%n_strfun > 0) then
       sf_chain%trace_enable = .true.
       sf_chain%trace_unit = free_unit ()
       open (sf_chain%trace_unit, file = char (file), action = "write", &
            status = "replace")
```

```
      call sf_chain%write_trace_header ()
    else
      call msg_error ("Beam structure: no structure functions, tracing &
           &disabled")
    end if
  end subroutine sf_chain_setup_tracing

  subroutine sf_chain_final_tracing (sf_chain)
    class(sf_chain_t), intent(inout) :: sf_chain
    if (sf_chain%trace_enable) then
      close (sf_chain%trace_unit)
      sf_chain%trace_enable = .false.
    end if
  end subroutine sf_chain_final_tracing
```

Write the header for the tracing file.

⟨*SF base: sf chain: TBP*⟩+≡

```
  procedure :: write_trace_header => sf_chain_write_trace_header
```

⟨*SF base: procedures*⟩+≡

```
  subroutine sf_chain_write_trace_header (sf_chain)
    class(sf_chain_t), intent(in) :: sf_chain
    integer :: u
    if (sf_chain%trace_enable) then
      u = sf_chain%trace_unit
      write (u, "('# ',A)")  "WHIZARD output: &
           &structure-function sampling data"
      write (u, "('# ',A,1x,I0)")  "Number of sf records:", sf_chain%n_strfun
      write (u, "('# ',A,1x,I0)")  "Number of parameters:", sf_chain%n_par
      write (u, "('# ',A)")  "Columns: channel, p(n_par), x(n_par), f, Jac * f"
    end if
  end subroutine sf_chain_write_trace_header
```

Write a record which collects the structure function data for the current data point. For the selected channel, we print first the input integration parameters, then the x values, then the structure-function value summed over all quantum numbers, then the structure function value times the mapping Jacobian.

⟨*SF base: sf chain: TBP*⟩+≡

```
  procedure :: trace => sf_chain_trace
```

⟨*SF base: procedures*⟩+≡

```
  subroutine sf_chain_trace (sf_chain, c_sel, p, x, f, sf_sum)
    class(sf_chain_t), intent(in) :: sf_chain
    integer, intent(in) :: c_sel
    real(default), dimension(:,:), intent(in) :: p
    real(default), dimension(:), intent(in) :: x
    real(default), dimension(:), intent(in) :: f
    real(default), intent(in) :: sf_sum
    real(default) :: sf_sum_pac, f_sf_sum_pac
    integer :: u, i
    if (sf_chain%trace_enable) then
      u = sf_chain%trace_unit
      write (u, "(1x,I0)", advance="no")  c_sel
      write (u, "(2x)", advance="no")
```

```
      do i = 1, sf_chain%n_par
         write (u, "(1x," // FMT_17 // ")", advance="no")  p(i,c_sel)
      end do
      write (u, "(2x)", advance="no")
      do i = 1, sf_chain%n_par
         write (u, "(1x," // FMT_17 // ")", advance="no")  x(i)
      end do
      write (u, "(2x)", advance="no")
      sf_sum_pac = sf_sum
      f_sf_sum_pac = f(c_sel) * sf_sum
      call pacify (sf_sum_pac, 1.E-28_default)
      call pacify (f_sf_sum_pac, 1.E-28_default)
      write (u, "(2(1x," // FMT_17 // "))")  sf_sum_pac, f_sf_sum_pac
    end if
  end subroutine sf_chain_trace
```

15.5.8 Chain instances

A structure-function chain instance contains copies of the interactions in the configuration chain, suitably linked to each other and connected by evaluators.

After initialization, `out_sf` should point, for each beam, to the last structure function that affects this beam. `out_sf_i` should indicate the index of the corresponding outgoing particle within that structure-function interaction.

Analogously, `out_eval` is the last evaluator in the structure-function chain, which contains the complete set of outgoing particles. `out_eval_i` should indicate the index of the outgoing particles, within that evaluator, which will initiate the collision.

When calculating actual kinematics, we fill the `p`, `r`, and `x` arrays and the `f` factor. The `p` array denotes the MC input parameters as they come from the random-number generator. The `r` array results from applying global mappings. The `x` array results from applying structure-function local mappings. The x values can be interpreted directly as momentum fractions (or angle fractions, where recoil is involved). The `f` factor is the Jacobian that results from applying all mappings.

The `mapping` entry may store a global mapping that is applied to a combination of x values and structure functions, as opposed to mappings that affect only a single structure function. It is applied before the latter mappings, in the transformation from the `p` array to the `r` array. For parameters affected by this mapping, we should ensure that they are not involved in a local mapping.

⟨*SF base: public*⟩+≡
```
public :: sf_chain_instance_t
```

⟨*SF base: types*⟩+≡
```
type, extends (beam_t) :: sf_chain_instance_t
   type(sf_chain_t), pointer :: config => null ()
   integer :: status = SF_UNDEFINED
   type(sf_instance_t), dimension(:), allocatable :: sf
   integer, dimension(:), allocatable :: out_sf
   integer, dimension(:), allocatable :: out_sf_i
   integer :: out_eval = 0
   integer, dimension(:), allocatable :: out_eval_i
```

```
    integer :: selected_channel = 0
    real(default), dimension(:,:), allocatable :: p, pb
    real(default), dimension(:,:), allocatable :: r, rb
    real(default), dimension(:), allocatable :: f
    real(default), dimension(:), allocatable :: x
    logical, dimension(:), allocatable :: bound
    real(default) :: x_free = 1
    type(sf_channel_t), dimension(:), allocatable :: channel
  contains
  ⟨SF base: sf chain instance: TBP⟩
end type sf_chain_instance_t
```

Finalizer.

⟨*SF base: sf chain instance: TBP*⟩≡

```
procedure :: final => sf_chain_instance_final
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_instance_final (object)
  class(sf_chain_instance_t), intent(inout) :: object
  integer :: i
  if (allocated (object%sf)) then
     do i = 1, size (object%sf, 1)
        associate (sf => object%sf(i))
          if (allocated (sf%int)) then
             call sf%eval%final ()
             call sf%int%final ()
          end if
        end associate
     end do
  end if
  call beam_final (object%beam_t)
end subroutine sf_chain_instance_final
```

Output.

Note: nagfor 5.3.1 appears to be slightly confused with the allocation status. We check both for allocation and nonzero size.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
procedure :: write => sf_chain_instance_write
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_instance_write (object, unit, col_verbose)
  class(sf_chain_instance_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: col_verbose
  integer :: u, i, c
  u = given_output_unit (unit)
  write (u, "(1x,A)", advance="no")  "Structure-function chain instance:"
  call write_sf_status (object%status, u)
  if (allocated (object%out_sf)) then
     write (u, "(3x,A)", advance="no")  "outgoing (interactions) ="
     do i = 1, size (object%out_sf)
        write (u, "(1x,I0,':',I0)", advance="no") &
             object%out_sf(i), object%out_sf_i(i)
     end do
```

```
   write (u, *)
end if
if (object%out_eval /= 0) then
   write (u, "(3x,A)", advance="no")  "outgoing (evaluators)   ="
   do i = 1, size (object%out_sf)
      write (u, "(1x,I0,':',I0)", advance="no") &
           object%out_eval, object%out_eval_i(i)
   end do
   write (u, *)
end if
if (allocated (object%sf)) then
   if (size (object%sf) /= 0) then
      write (u, "(1x,A)")  "Structure-function parameters:"
      do c = 1, size (object%f)
         write (u, "(1x,A,I0,A)", advance="no")  "Channel #", c, ":"
         if (c == object%selected_channel) then
            write (u, "(1x,A)")  "[selected]"
         else
            write (u, *)
         end if
         write (u, "(3x,A,9(1x,F9.7))")  "p =", object%p(:,c)
         write (u, "(3x,A,9(1x,F9.7))")  "r =", object%r(:,c)
         write (u, "(3x,A,9(1x,ES13.7))")  "f =", object%f(c)
         write (u, "(3x,A)", advance="no") "m ="
         call object%channel(c)%write (u)
      end do
      write (u, "(3x,A,9(1x,F9.7))")  "x =", object%x
      if (.not. all (object%bound)) then
         write (u, "(3x,A,9(1x,L1))")  "bound =", object%bound
      end if
   end if
end if
call write_separator (u)
call beam_write (object%beam_t, u, col_verbose = col_verbose)
if (allocated (object%sf)) then
   do i = 1, size (object%sf)
      associate (sf => object%sf(i))
        call write_separator (u)
        if (allocated (sf%int)) then
           if (allocated (sf%r)) then
              write (u, "(1x,A)")  "Structure-function parameters:"
              do c = 1, size (sf%f)
                 write (u, "(1x,A,I0,A)", advance="no")  "Channel #", c, ":"
                 if (c == object%selected_channel) then
                    write (u, "(1x,A)")  "[selected]"
                 else
                    write (u, *)
                 end if
                 write (u, "(3x,A,9(1x,F9.7))")  "r =", sf%r(:,c)
                 write (u, "(3x,A,9(1x,ES13.7))")  "f =", sf%f(c)
                 write (u, "(3x,A,9(1x,L1,7x))") "m =", sf%m(c)
              end do
              write (u, "(3x,A,9(1x,F9.7))")  "x =", sf%x
           end if
```

```
            call sf%int%write(u)
            if (.not. sf%eval%is_empty ()) then
                  call sf%eval%write (u, col_verbose = col_verbose)
            end if
          end if
        end associate
      end do
    end if
  end subroutine sf_chain_instance_write
```

Initialize. This creates a copy of the interactions in the configuration chain, assumed to be properly initialized. In the copy, we allocate the `p` etc. arrays.

The brute-force assignment of the `sf` component would be straightforward, but at least gfortran 4.6.3 would like a more fine-grained copy. In any case, the copy is deep as far as allocatables are concerned, but for the contained `interaction_t` objects the copy is shallow, as long as we do not bind defined assignment to the type. Therefore, we have to re-assign the `interaction_t` components explicitly, this time calling the proper defined assignment. Furthermore, we allocate the parameter arrays for each structure function.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
  procedure :: init => sf_chain_instance_init
```

⟨*SF base: procedures*⟩+≡

```
  subroutine sf_chain_instance_init (chain, config, n_channel)
    class(sf_chain_instance_t), intent(out), target :: chain
    type(sf_chain_t), intent(in), target :: config
    integer, intent(in) :: n_channel
    integer :: i, j
    integer :: n_par_tot, n_par, n_strfun
    chain%config => config
    n_strfun = config%n_strfun
    chain%beam_t = config%beam_t
    allocate (chain%out_sf (config%n_in), chain%out_sf_i (config%n_in))
    allocate (chain%out_eval_i (config%n_in))
    chain%out_sf = 0
    chain%out_sf_i = [(i, i = 1, config%n_in)]
    chain%out_eval_i = chain%out_sf_i
    n_par_tot = 0
    if (n_strfun /= 0) then
       allocate (chain%sf (n_strfun))
       do i = 1, n_strfun
          associate (sf => chain%sf(i))
            allocate (sf%int, source=config%sf(i)%int)
            sf%int%interaction_t = config%sf(i)%int%interaction_t
            n_par = size (sf%int%par_index)
            allocate (sf%r (n_par, n_channel));  sf%r = 0
            allocate (sf%rb(n_par, n_channel));  sf%rb= 0
            allocate (sf%f (n_channel));         sf%f = 0
            allocate (sf%m (n_channel));         sf%m = .false.
            allocate (sf%x (n_par));             sf%x = 0
            n_par_tot = n_par_tot + n_par
          end associate
       end do
```

```
      allocate (chain%p (n_par_tot, n_channel));  chain%p = 0
      allocate (chain%pb(n_par_tot, n_channel));  chain%pb= 0
      allocate (chain%r (n_par_tot, n_channel));  chain%r = 0
      allocate (chain%rb(n_par_tot, n_channel));  chain%rb= 0
      allocate (chain%f (n_channel));             chain%f = 0
      allocate (chain%x (n_par_tot));             chain%x = 0
      call allocate_sf_channels &
           (chain%channel, n_channel=n_channel, n_strfun=n_strfun)
   end if
   allocate (chain%bound (n_par_tot), source = .true.)
   do i = 1, n_strfun
      associate (sf => chain%sf(i))
        if (sf%int%is_generator ()) then
           do j = 1, size (sf%int%par_index)
              chain%bound(sf%int%par_index(j)) = .false.
           end do
        end if
      end associate
   end do
   chain%status = SF_INITIAL
end subroutine sf_chain_instance_init
```

Manually select a channel.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
procedure :: select_channel => sf_chain_instance_select_channel
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_instance_select_channel (chain, channel)
  class(sf_chain_instance_t), intent(inout) :: chain
  integer, intent(in), optional :: channel
  if (present (channel)) then
     chain%selected_channel = channel
  else
     chain%selected_channel = 0
  end if
end subroutine sf_chain_instance_select_channel
```

Copy a channel-mapping object to the structure-function chain instance. We assume that assignment is sufficient, i.e., any non-static components of the `channel` object are allocatable und thus recursively copied.

After the copy, we extract the single-entry mappings and activate them for the individual structure functions. If there is a multi-entry mapping, we obtain the corresponding MC parameter indices and set them in the copy of the channel object.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
procedure :: set_channel => sf_chain_instance_set_channel
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_instance_set_channel (chain, c, channel)
  class(sf_chain_instance_t), intent(inout) :: chain
  integer, intent(in) :: c
  type(sf_channel_t), intent(in) :: channel
  integer :: i, j, k
```

```
    if (chain%status >= SF_INITIAL) then
       chain%channel(c) = channel
       j = 0
       do i = 1, chain%config%n_strfun
          associate (sf => chain%sf(i))
            sf%m(c) = channel%is_single_mapping (i)
            if (channel%is_multi_mapping (i)) then
               do k = 1, size (sf%int%beam_index)
                  j = j + 1
                  call chain%channel(c)%set_par_index &
                       (j, sf%int%par_index(k))
               end do
            end if
          end associate
       end do
       if (j /= chain%channel(c)%get_multi_mapping_n_par ()) then
          print *, "index last filled    = ", j
          print *, "number of parameters = ", &
               chain%channel(c)%get_multi_mapping_n_par ()
          call msg_bug ("Structure-function setup: mapping index mismatch")
       end if
       chain%status = SF_INITIAL
    end if
  end subroutine sf_chain_instance_set_channel
```

Link the interactions in the chain. First, link the beam instance to its template in the configuration chain, which should have the appropriate momenta fixed.

Then, we follow the chain via the arrays `out_sf` and `out_sf_i`. The arrays are (up to) two-dimensional, the entries correspond to the beam particle(s). For each beam, the entry `out_sf` points to the last interaction that affected this beam, and `out_sf_i` is the out-particle index within that interaction. For the initial beam, `out_sf` is zero by definition.

For each entry in the chain, we scan the affected beams (one or two). We look for `out_sf` and link the out-particle there to the corresponding in-particle in the current interaction. Then, we update the entry in `out_sf` and `out_sf_i` to point to the current interaction.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
  procedure :: link_interactions => sf_chain_instance_link_interactions
```

⟨*SF base: procedures*⟩+≡

```
  subroutine sf_chain_instance_link_interactions (chain)
    class(sf_chain_instance_t), intent(inout), target :: chain
    type(interaction_t), pointer :: int
    integer :: i, j, b
    if (chain%status >= SF_INITIAL) then
       do b = 1, chain%config%n_in
          int => beam_get_int_ptr (chain%beam_t)
          call interaction_set_source_link (int, b, &
               chain%config%beam_t, b)
       end do
       if (allocated (chain%sf)) then
          do i = 1, size (chain%sf)
             associate (sf_int => chain%sf(i)%int)
```

```
            do j = 1, size (sf_int%beam_index)
               b = sf_int%beam_index(j)
               call link (sf_int%interaction_t, b, sf_int%incoming(j))
               chain%out_sf(b) = i
               chain%out_sf_i(b) = sf_int%outgoing(j)
            end do
         end associate
      end do
   end if
   chain%status = SF_DONE_LINKS
end if
contains
  subroutine link (int, b, in_index)
    type(interaction_t), intent(inout) :: int
    integer, intent(in) :: b, in_index
    integer :: i
    i = chain%out_sf(b)
    select case (i)
    case (0)
       call interaction_set_source_link (int, in_index, &
            chain%beam_t, chain%out_sf_i(b))
    case default
       call int%set_source_link (in_index, &
            chain%sf(i)%int, chain%out_sf_i(b))
    end select
  end subroutine link
end subroutine sf_chain_instance_link_interactions
```

Exchange the quantum-number masks between the interactions in the chain, so we can combine redundant entries and detect any obvious mismatch.

We proceed first in the forward direction and then backwards again.

After this is finished, we finalize initialization by calling the `setup_constants` method, which prepares constant data that depend on the matrix element structure.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
procedure :: exchange_mask => sf_chain_exchange_mask
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_exchange_mask (chain)
  class(sf_chain_instance_t), intent(inout), target :: chain
  type(interaction_t), pointer :: int
  type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
  integer :: i
  if (chain%status >= SF_DONE_LINKS) then
     if (allocated (chain%sf)) then
        int => beam_get_int_ptr (chain%beam_t)
        allocate (mask (int%get_n_out ()))
        mask = int%get_mask ()
        if (size (chain%sf) /= 0) then
           do i = 1, size (chain%sf) - 1
              call interaction_exchange_mask (chain%sf(i)%int%interaction_t)
           end do
           do i = size (chain%sf), 1, -1
              call interaction_exchange_mask (chain%sf(i)%int%interaction_t)
```

```
            end do
            if (any (mask .neqv. int%get_mask ())) then
               chain%status = SF_FAILED_MASK
               return
            end if
            do i = 1, size (chain%sf)
               call chain%sf(i)%int%setup_constants ()
            end do
         end if
      end if
      chain%status = SF_DONE_MASK
   end if
end subroutine sf_chain_exchange_mask
```

Initialize the evaluators that connect the interactions in the chain.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
procedure :: init_evaluators => sf_chain_instance_init_evaluators
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_instance_init_evaluators (chain)
  class(sf_chain_instance_t), intent(inout), target :: chain
  type(interaction_t), pointer :: int
  type(quantum_numbers_mask_t) :: mask
  integer :: i
  if (chain%status >= SF_DONE_MASK) then
     if (allocated (chain%sf)) then
        if (size (chain%sf) /= 0) then
           mask = quantum_numbers_mask (.false., .false., .true.)
           int => beam_get_int_ptr (chain%beam_t)
           do i = 1, size (chain%sf)
              associate (sf => chain%sf(i))
                call sf%eval%init_product &
                     (int, sf%int%interaction_t, mask)
                if (sf%eval%is_empty ()) then
                   chain%status = SF_FAILED_CONNECTIONS
                   return
                end if
                int => sf%eval%interaction_t
              end associate
           end do
           call find_outgoing_particles ()
        end if
     end if
     chain%status = SF_DONE_CONNECTIONS
  end if
contains
⟨SF base: init evaluators: find outgoing particles⟩
end subroutine sf_chain_instance_init_evaluators
```

This is an internal subroutine of the previous one: After evaluators are set, trace the outgoing particles to the last evaluator. We only need the first channel, all channels are equivalent for this purpose.

For each beam, the outgoing particle is located by out_sf (the structure-function object where it originates) and out_sf_i (the index within that object). This particle is referenced by the corresponding evaluator, which in turn is referenced by the next evaluator, until we are at the end of the chain. We can trace back references by interaction_find_link. Knowing that out_eval is the index of the last evaluator, we thus determine out_eval_i, the index of the outgoing particle within that evaluator.

⟨*SF base: init evaluators: find outgoing particles*⟩≡

```
subroutine find_outgoing_particles ()
  type(interaction_t), pointer :: int, int_next
  integer :: i, j, out_sf, out_i
  chain%out_eval = size (chain%sf)
  do j = 1, size (chain%out_eval_i)
     out_sf = chain%out_sf(j)
     out_i = chain%out_sf_i(j)
     if (out_sf == 0) then
        int => beam_get_int_ptr (chain%beam_t)
        out_sf = 1
     else
        int => chain%sf(out_sf)%int%interaction_t
     end if
     do i = out_sf, chain%out_eval
        int_next => chain%sf(i)%eval%interaction_t
        out_i = interaction_find_link (int_next, int, out_i)
        int => int_next
     end do
     chain%out_eval_i(j) = out_i
  end do
end subroutine find_outgoing_particles
```

Compute the kinematics in the chain instance. We can assume that the seed momenta are set in the configuration beams. Scanning the chain, we first transfer the incoming momenta. Then, the use up the MC input parameter array p to compute the radiated and outgoing momenta.

In the multi-channel case, c_sel is the channel which we use for computing the kinematics and the x values. In the other channels, we invert the kinematics in order to recover the corresponding rows in the r array, and the Jacobian f.

We first apply any global mapping to transform the input p into the array r. This is then given to the structure functions which compute the final array x and Jacobian factors f, which we multiply to obtain the overall Jacobian.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
procedure :: compute_kinematics => sf_chain_instance_compute_kinematics
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_instance_compute_kinematics (chain, c_sel, p_in)
  class(sf_chain_instance_t), intent(inout), target :: chain
  integer, intent(in) :: c_sel
  real(default), dimension(:), intent(in) :: p_in
  type(interaction_t), pointer :: int
  real(default) :: f_mapping
  logical, dimension(size (chain%bound)) :: bound
  integer :: i, j, c
  if (chain%status >= SF_DONE_CONNECTIONS) then
     call chain%select_channel (c_sel)
```

```
int => beam_get_int_ptr (chain%beam_t)
call int%receive_momenta ()
if (allocated (chain%sf)) then
   if (size (chain%sf) /= 0) then
      forall (i = 1:size (chain%sf))  chain%sf(i)%int%status = SF_INITIAL
      !!! Bug in nagfor 5.3.1(907), fixed in 5.3.1(982)
      ! chain%p (:,c_sel) = unpack (p_in, chain%bound, 0._default)
      !!! Workaround:
      bound = chain%bound
      chain%p (:,c_sel) = unpack (p_in, bound, 0._default)
      chain%pb(:,c_sel) = 1 - chain%p(:,c_sel)
      chain%f = 1
      chain%x_free = 1
      do i = 1, size (chain%sf)
         associate (sf => chain%sf(i))
           call sf%int%generate_free (sf%r(:,c_sel), sf%rb(:,c_sel), &
                chain%x_free)
           do j = 1, size (sf%x)
              if (.not. chain%bound(sf%int%par_index(j))) then
                 chain%p (sf%int%par_index(j),c_sel) = sf%r (j,c_sel)
                 chain%pb(sf%int%par_index(j),c_sel) = sf%rb(j,c_sel)
              end if
           end do
         end associate
      end do
      if (allocated (chain%channel(c_sel)%multi_mapping)) then
         call chain%channel(c_sel)%multi_mapping%compute &
              (chain%r(:,c_sel), chain%rb(:,c_sel), &
               f_mapping, &
               chain%p(:,c_sel), chain%pb(:,c_sel), &
               chain%x_free)
         chain%f(c_sel) = f_mapping
      else
         chain%r (:,c_sel) = chain%p (:,c_sel)
         chain%rb(:,c_sel) = chain%pb(:,c_sel)
         chain%f(c_sel) = 1
      end if
      do i = 1, size (chain%sf)
         associate (sf => chain%sf(i))
           call sf%int%seed_kinematics ()
           do j = 1, size (sf%x)
              sf%r (j,c_sel) = chain%r (sf%int%par_index(j),c_sel)
              sf%rb(j,c_sel) = chain%rb(sf%int%par_index(j),c_sel)
           end do
           call sf%int%complete_kinematics &
                (sf%x, sf%f(c_sel), sf%r(:,c_sel), sf%rb(:,c_sel), &
                 sf%m(c_sel))
           do j = 1, size (sf%x)
              chain%x(sf%int%par_index(j)) = sf%x(j)
           end do
           if (sf%int%status <= SF_FAILED_KINEMATICS) then
              chain%status = SF_FAILED_KINEMATICS
              return
           end if
```

```
            do c = 1, size (sf%f)
               if (c /= c_sel) then
                  call sf%int%inverse_kinematics &
                       (sf%x, sf%f(c), sf%r(:,c), sf%rb(:,c), sf%m(c))
                  do j = 1, size (sf%x)
                     chain%r (sf%int%par_index(j),c) = sf%r (j,c)
                     chain%rb(sf%int%par_index(j),c) = sf%rb(j,c)
                  end do
               end if
               chain%f(c) = chain%f(c) * sf%f(c)
            end do
            if (.not. sf%eval%is_empty ()) then
               call sf%eval%receive_momenta ()
            end if
         end associate
      end do
      do c = 1, size (chain%f)
         if (c /= c_sel) then
            if (allocated (chain%channel(c)%multi_mapping)) then
               call chain%channel(c)%multi_mapping%inverse &
                    (chain%r(:,c), chain%rb(:,c), &
                     f_mapping, &
                     chain%p(:,c), chain%pb(:,c), &
                     chain%x_free)
               chain%f(c) = chain%f(c) * f_mapping
            else
               chain%p (:,c) = chain%r (:,c)
               chain%pb(:,c) = chain%rb(:,c)
            end if
         end if
      end do
   end if
end if
chain%status = SF_DONE_KINEMATICS
end if
end subroutine sf_chain_instance_compute_kinematics
```

This is a variant of the previous procedure. We know the x parameters and reconstruct the momenta and the MC input parameters `p`. We do not need to select a channel.

Note: this is probably redundant, since the method we actually want starts from the momenta, recovers all x parameters, and then inverts mappings. See below `recover_kinematics`.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
procedure :: inverse_kinematics => sf_chain_instance_inverse_kinematics
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_instance_inverse_kinematics (chain, x)
  class(sf_chain_instance_t), intent(inout), target :: chain
  real(default), dimension(:), intent(in) :: x
  type(interaction_t), pointer :: int
  real(default) :: f_mapping
  integer :: i, j, c
  if (chain%status >= SF_DONE_CONNECTIONS) then
```

```
      call chain%select_channel ()
      int => beam_get_int_ptr (chain%beam_t)
      call int%receive_momenta ()
      if (allocated (chain%sf)) then
         chain%f = 1
         if (size (chain%sf) /= 0) then
            forall (i = 1:size (chain%sf))  chain%sf(i)%int%status = SF_INITIAL
            chain%x = x
            do i = 1, size (chain%sf)
               associate (sf => chain%sf(i))
                 call sf%int%seed_kinematics ()
                 do j = 1, size (sf%x)
                    sf%x(j) = chain%x(sf%int%par_index(j))
                 end do
                 do c = 1, size (sf%f)
                    call sf%int%inverse_kinematics &
                         (sf%x, sf%f(c), sf%r(:,c), sf%rb(:,c), sf%m(c), c==1)
                    chain%f(c) = chain%f(c) * sf%f(c)
                    do j = 1, size (sf%x)
                       chain%r (sf%int%par_index(j),c) = sf%r (j,c)
                       chain%rb(sf%int%par_index(j),c) = sf%rb(j,c)
                    end do
                 end do
                 if (.not. sf%eval%is_empty ()) then
                    call sf%eval%receive_momenta ()
                 end if
               end associate
            end do
            do c = 1, size (chain%f)
               if (allocated (chain%channel(c)%multi_mapping)) then
                  call chain%channel(c)%multi_mapping%inverse &
                       (chain%r(:,c), chain%rb(:,c), &
                       f_mapping, &
                       chain%p(:,c), chain%pb(:,c), &
                       chain%x_free)
                  chain%f(c) = chain%f(c) * f_mapping
               else
                  chain%p (:,c) = chain%r (:,c)
                  chain%pb(:,c) = chain%rb(:,c)
               end if
            end do
         end if
      end if
      chain%status = SF_DONE_KINEMATICS
   end if
end subroutine sf_chain_instance_inverse_kinematics
```

Recover the kinematics: assuming that the last evaluator has been filled with a valid set of momenta, we travel the momentum links backwards and fill the preceding evaluators and, as a side effect, interactions. We stop at the beam interaction.

After all momenta are set, apply the `inverse_kinematics` procedure above, suitably modified, to recover the x and p parameters and the Jacobian factors.

The c_sel (channel) argument is just used to mark a selected channel for the records, otherwise the recovery procedure is independent of this.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
procedure :: recover_kinematics => sf_chain_instance_recover_kinematics
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_instance_recover_kinematics (chain, c_sel)
  class(sf_chain_instance_t), intent(inout), target :: chain
  integer, intent(in) :: c_sel
  real(default) :: f_mapping
  integer :: i, j, c
  if (chain%status >= SF_DONE_CONNECTIONS) then
     call chain%select_channel (c_sel)
     if (allocated (chain%sf)) then
        do i = size (chain%sf), 1, -1
           associate (sf => chain%sf(i))
             if (.not. sf%eval%is_empty ()) then
                call interaction_send_momenta (sf%eval%interaction_t)
             end if
           end associate
        end do
        chain%f = 1
        if (size (chain%sf) /= 0) then
           forall (i = 1:size (chain%sf))  chain%sf(i)%int%status = SF_INITIAL
           chain%x_free = 1
           do i = 1, size (chain%sf)
              associate (sf => chain%sf(i))
                call sf%int%seed_kinematics ()
                call sf%int%recover_x (sf%x, chain%x_free)
                do j = 1, size (sf%x)
                   chain%x(sf%int%par_index(j)) = sf%x(j)
                end do
                do c = 1, size (sf%f)
                   call sf%int%inverse_kinematics &
                        (sf%x, sf%f(c), sf%r(:,c), sf%rb(:,c), sf%m(c), c==1)
                   chain%f(c) = chain%f(c) * sf%f(c)
                   do j = 1, size (sf%x)
                      chain%r (sf%int%par_index(j),c) = sf%r (j,c)
                      chain%rb(sf%int%par_index(j),c) = sf%rb(j,c)
                   end do
                end do
              end associate
           end do
           do c = 1, size (chain%f)
              if (allocated (chain%channel(c)%multi_mapping)) then
                 call chain%channel(c)%multi_mapping%inverse &
                      (chain%r(:,c), chain%rb(:,c), &
                      f_mapping, &
                      chain%p(:,c), chain%pb(:,c), &
                      chain%x_free)
                 chain%f(c) = chain%f(c) * f_mapping
              else
                 chain%p (:,c) = chain%r (:,c)
                 chain%pb(:,c) = chain%rb(:,c)
```

```
            end if
         end do
      end if
   end if
   chain%status = SF_DONE_KINEMATICS
end if
end subroutine sf_chain_instance_recover_kinematics
```

Return the initial beam momenta to their source, thus completing kinematics recovery. Obviously, this works as a side effect.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
procedure :: return_beam_momenta => sf_chain_instance_return_beam_momenta
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_instance_return_beam_momenta (chain)
  class(sf_chain_instance_t), intent(in), target :: chain
  type(interaction_t), pointer :: int
  if (chain%status >= SF_DONE_KINEMATICS) then
     int => beam_get_int_ptr (chain%beam_t)
     call interaction_send_momenta (int)
  end if
end subroutine sf_chain_instance_return_beam_momenta
```

Evaluate all interactions in the chain and the product evaluators. We provide a `scale` argument that is given to all structure functions in the chain.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
procedure :: evaluate => sf_chain_instance_evaluate
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_instance_evaluate (chain, scale)
  class(sf_chain_instance_t), intent(inout), target :: chain
  real(default), intent(in) :: scale
  type(interaction_t), pointer :: out_int
  real(default) :: sf_sum
  integer :: i
  if (chain%status >= SF_DONE_KINEMATICS) then
     if (allocated (chain%sf)) then
        if (size (chain%sf) /= 0) then
           do i = 1, size (chain%sf)
              associate (sf => chain%sf(i))
                call sf%int%apply (scale)
                if (sf%int%status <= SF_FAILED_EVALUATION) then
                   chain%status = SF_FAILED_EVALUATION
                   return
                end if
                if (.not. sf%eval%is_empty ()) then
                   call sf%eval%evaluate ()
                end if
              end associate
           end do
           out_int => chain%get_out_int_ptr ()
           sf_sum = real (out_int%sum ())
           call chain%config%trace &
                (chain%selected_channel, chain%p, chain%x, chain%f, sf_sum)
```

```
         end if
      end if
      chain%status = SF_EVALUATED
   end if
end subroutine sf_chain_instance_evaluate
```

15.5.9 Access to the chain instance

Transfer the outgoing momenta to the array p. We assume that array sizes match.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
procedure :: get_out_momenta => sf_chain_instance_get_out_momenta
```

⟨*SF base: procedures*⟩+≡

```
subroutine sf_chain_instance_get_out_momenta (chain, p)
  class(sf_chain_instance_t), intent(in), target :: chain
  type(vector4_t), dimension(:), intent(out) :: p
  type(interaction_t), pointer :: int
  integer :: i, j
  if (chain%status >= SF_DONE_KINEMATICS) then
     do j = 1, size (chain%out_sf)
        i = chain%out_sf(j)
        select case (i)
        case (0)
           int => beam_get_int_ptr (chain%beam_t)
        case default
           int => chain%sf(i)%int%interaction_t
        end select
        p(j) = int%get_momentum (chain%out_sf_i(j))
     end do
  end if
end subroutine sf_chain_instance_get_out_momenta
```

Return a pointer to the last evaluator in the chain (to the interaction).

⟨*SF base: sf chain instance: TBP*⟩+≡

```
procedure :: get_out_int_ptr => sf_chain_instance_get_out_int_ptr
```

⟨*SF base: procedures*⟩+≡

```
function sf_chain_instance_get_out_int_ptr (chain) result (int)
  class(sf_chain_instance_t), intent(in), target :: chain
  type(interaction_t), pointer :: int
  if (chain%out_eval == 0) then
     int => beam_get_int_ptr (chain%beam_t)
  else
     int => chain%sf(chain%out_eval)%eval%interaction_t
  end if
end function sf_chain_instance_get_out_int_ptr
```

Return the index of the j-th outgoing particle, within the last evaluator.

⟨*SF base: sf chain instance: TBP*⟩+≡

```
procedure :: get_out_i => sf_chain_instance_get_out_i
```

⟨*SF base: procedures*⟩+≡
```
function sf_chain_instance_get_out_i (chain, j) result (i)
  class(sf_chain_instance_t), intent(in) :: chain
  integer, intent(in) :: j
  integer :: i
  i = chain%out_eval_i(j)
end function sf_chain_instance_get_out_i
```

Return the mask for the outgoing particle(s), within the last evaluator.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
procedure :: get_out_mask => sf_chain_instance_get_out_mask
```

⟨*SF base: procedures*⟩+≡
```
function sf_chain_instance_get_out_mask (chain) result (mask)
  class(sf_chain_instance_t), intent(in), target :: chain
  type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
  type(interaction_t), pointer :: int
  allocate (mask (chain%config%n_in))
  int => chain%get_out_int_ptr ()
  mask = int%get_mask (chain%out_eval_i)
end function sf_chain_instance_get_out_mask
```

Return the array of MC input parameters that corresponds to channel `c`. This is the `p` array, the parameters before all mappings.

The `p` array may be deallocated. This should correspond to a zero-size `r` argument, so nothing to do then.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
procedure :: get_mcpar => sf_chain_instance_get_mcpar
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_chain_instance_get_mcpar (chain, c, r)
  class(sf_chain_instance_t), intent(in) :: chain
  integer, intent(in) :: c
  real(default), dimension(:), intent(out) :: r
  if (allocated (chain%p))  r = pack (chain%p(:,c), chain%bound)
end subroutine sf_chain_instance_get_mcpar
```

Return the Jacobian factor that corresponds to channel `c`.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
procedure :: get_f => sf_chain_instance_get_f
```

⟨*SF base: procedures*⟩+≡
```
function sf_chain_instance_get_f (chain, c) result (f)
  class(sf_chain_instance_t), intent(in) :: chain
  integer, intent(in) :: c
  real(default) :: f
  if (allocated (chain%f)) then
     f = chain%f(c)
  else
     f = 1
  end if
end function sf_chain_instance_get_f
```

Return the evaluation status.

⟨*SF base: sf chain instance: TBP*⟩+≡
```
procedure :: get_status => sf_chain_instance_get_status
```

⟨*SF base: procedures*⟩+≡
```
function sf_chain_instance_get_status (chain) result (status)
  class(sf_chain_instance_t), intent(in) :: chain
  integer :: status
  status = chain%status
end function sf_chain_instance_get_status
```

⟨*SF base: sf chain instance: TBP*⟩+≡
```
procedure :: get_matrix_elements => sf_chain_instance_get_matrix_elements
```

⟨*SF base: procedures*⟩+≡
```
subroutine sf_chain_instance_get_matrix_elements (chain, i, ff)
   class(sf_chain_instance_t), intent(in) :: chain
   integer, intent(in) :: i
   real(default), intent(out), dimension(:), allocatable :: ff

   associate (sf => chain%sf(i))
      ff = real (sf%int%get_matrix_element ())
   end associate
end subroutine sf_chain_instance_get_matrix_elements
```

15.5.10 Unit tests

Test module, followed by the corresponding implementation module.

⟨sf_base_ut.f90⟩≡
```
⟨File header⟩

module sf_base_ut
  use unit_tests
  use sf_base_uti

⟨Standard module head⟩

⟨SF base: public test auxiliary⟩

⟨SF base: public test⟩

contains

⟨SF base: test driver⟩

end module sf_base_ut
```

⟨sf_base_uti.f90⟩≡
```
⟨File header⟩

module sf_base_uti

⟨Use kinds⟩
```

```
⟨Use strings⟩
  use io_units
  use format_defs, only: FMT_19
  use format_utils, only: write_separator
  use diagnostics
  use lorentz
  use pdg_arrays
  use flavors
  use colors
  use helicities
  use quantum_numbers
  use state_matrices, only: FM_IGNORE_HELICITY
  use interactions
  use particles
  use model_data
  use beams
  use sf_aux
  use sf_mappings

  use sf_base

⟨Standard module head⟩

⟨SF base: test declarations⟩

⟨SF base: public test auxiliary⟩

⟨SF base: test types⟩

contains

⟨SF base: tests⟩

⟨SF base: test auxiliary⟩

end module sf_base_uti
```

API: driver for the unit tests below.

```
⟨SF base: public test⟩≡
    public :: sf_base_test
```

```
⟨SF base: test driver⟩≡
    subroutine sf_base_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨SF base: execute tests⟩
    end subroutine sf_base_test
```

15.5.11 Test implementation: structure function

This is a template for the actual structure-function implementation which will be defined in separate modules.

Configuration data

The test structure function uses the `Test` model. It describes a scalar within an arbitrary initial particle, which is given in the initialization. The radiated particle is also a scalar, the same one, but we set its mass artificially to zero.

⟨*SF base: public test auxiliary*⟩≡
```
public :: sf_test_data_t
```

⟨*SF base: test types*⟩≡
```
type, extends (sf_data_t) :: sf_test_data_t
   class(model_data_t), pointer :: model => null ()
   integer :: mode = 0
   type(flavor_t) :: flv_in
   type(flavor_t) :: flv_out
   type(flavor_t) :: flv_rad
   real(default) :: m = 0
   logical :: collinear = .true.
   real(default), dimension(:), allocatable :: qbounds
 contains
 ⟨SF base: sf test data: TBP⟩
end type sf_test_data_t
```

Output.

⟨*SF base: sf test data: TBP*⟩≡
```
procedure :: write => sf_test_data_write
```

⟨*SF base: test auxiliary*⟩≡
```
subroutine sf_test_data_write (data, unit, verbose)
  class(sf_test_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "SF test data:"
  write (u, "(3x,A,A)") "model     = ", char (data%model%get_name ())
  write (u, "(3x,A)", advance="no") "incoming  = "
  call data%flv_in%write (u);  write (u, *)
  write (u, "(3x,A)", advance="no") "outgoing  = "
  call data%flv_out%write (u);  write (u, *)
  write (u, "(3x,A)", advance="no") "radiated  = "
  call data%flv_rad%write (u);  write (u, *)
  write (u, "(3x,A," // FMT_19 // ")")  "mass      = ", data%m
  write (u, "(3x,A,L1)")  "collinear = ", data%collinear
  if (.not. data%collinear .and. allocated (data%qbounds)) then
     write (u, "(3x,A," // FMT_19 // ")")  "qmin      = ", data%qbounds(1)
     write (u, "(3x,A," // FMT_19 // ")")  "qmax      = ", data%qbounds(2)
  end if
end subroutine sf_test_data_write
```

Initialization.

⟨*SF base: sf test data: TBP*⟩+≡
```
procedure :: init => sf_test_data_init
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_data_init (data, model, pdg_in, collinear, qbounds, mode)
  class(sf_test_data_t), intent(out) :: data
  class(model_data_t), intent(in), target :: model
  type(pdg_array_t), intent(in) :: pdg_in
  logical, intent(in), optional :: collinear
  real(default), dimension(2), intent(in), optional :: qbounds
  integer, intent(in), optional :: mode
  data%model => model
  if (present (mode))  data%mode = mode
  if (pdg_array_get (pdg_in, 1) /= 25) then
     call msg_fatal ("Test spectrum function: input flavor must be 's'")
  end if
  call data%flv_in%init (25, model)
  data%m = data%flv_in%get_mass ()
  if (present (collinear))  data%collinear = collinear
  call data%flv_out%init (25, model)
  call data%flv_rad%init (25, model)
  if (present (qbounds)) then
     allocate (data%qbounds (2))
     data%qbounds = qbounds
  end if
end subroutine sf_test_data_init
```

Return the number of parameters: 1 if only consider collinear splitting, 3 otherwise.

⟨*SF base: sf test data: TBP*⟩+≡

```
procedure :: get_n_par => sf_test_data_get_n_par
```

⟨*SF base: test auxiliary*⟩+≡

```
function sf_test_data_get_n_par (data) result (n)
  class(sf_test_data_t), intent(in) :: data
  integer :: n
  if (data%collinear) then
     n = 1
  else
     n = 3
  end if
end function sf_test_data_get_n_par
```

Return the outgoing particle PDG code: 25

⟨*SF base: sf test data: TBP*⟩+≡

```
procedure :: get_pdg_out => sf_test_data_get_pdg_out
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_data_get_pdg_out (data, pdg_out)
  class(sf_test_data_t), intent(in) :: data
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
  pdg_out(1) = 25
end subroutine sf_test_data_get_pdg_out
```

Allocate the matching interaction.

⟨*SF base: sf test data: TBP*⟩+≡

```
procedure :: allocate_sf_int => sf_test_data_allocate_sf_int
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_data_allocate_sf_int (data, sf_int)
  class(sf_test_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  if (allocated (sf_int)) deallocate (sf_int)
  allocate (sf_test_t :: sf_int)
end subroutine sf_test_data_allocate_sf_int
```

Interaction

⟨*SF base: test types*⟩+≡

```
type, extends (sf_int_t) :: sf_test_t
   type(sf_test_data_t), pointer :: data => null ()
   real(default) :: x = 0
 contains
 ⟨SF base: sf test int: TBP⟩
end type sf_test_t
```

Type string: constant

⟨*SF base: sf test int: TBP*⟩≡

```
procedure :: type_string => sf_test_type_string
```

⟨*SF base: test auxiliary*⟩+≡

```
function sf_test_type_string (object) result (string)
  class(sf_test_t), intent(in) :: object
  type(string_t) :: string
  string = "Test"
end function sf_test_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF base: sf test int: TBP*⟩+≡

```
procedure :: write => sf_test_write
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_write (object, unit, testflag)
  class(sf_test_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     call object%base_write (u, testflag)
  else
     write (u, "(1x,A)")  "SF test data: [undefined]"
  end if
end subroutine sf_test_write
```

Initialize. We know that `data` will be of concrete type `sf_test_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass, but keep the radiated mass zero.

Optionally, we can provide minimum and maximum values for the momentum transfer.

⟨*SF base: sf test int: TBP*⟩+≡
```
procedure :: init => sf_test_init
```

⟨*SF base: test auxiliary*⟩+≡
```
subroutine sf_test_init (sf_int, data)
  class(sf_test_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  type(quantum_numbers_mask_t), dimension(3) :: mask
  type(helicity_t) :: hel0
  type(color_t) :: col0
  type(quantum_numbers_t), dimension(3) :: qn
  mask = quantum_numbers_mask (.false., .false., .false.)
  select type (data)
  type is (sf_test_data_t)
     if (allocated (data%qbounds)) then
        call sf_int%base_init (mask, &
             [data%m**2], [0._default], [data%m**2], &
             [data%qbounds(1)], [data%qbounds(2)])
     else
        call sf_int%base_init (mask, &
             [data%m**2], [0._default], [data%m**2])
     end if
     sf_int%data => data
     call hel0%init (0)
     call col0%init ()
     call qn(1)%init (data%flv_in,  col0, hel0)
     call qn(2)%init (data%flv_rad, col0, hel0)
     call qn(3)%init (data%flv_out, col0, hel0)
     call sf_int%add_state (qn)
     call sf_int%freeze ()
     call sf_int%set_incoming ([1])
     call sf_int%set_radiated ([2])
     call sf_int%set_outgoing ([3])
  end select
  sf_int%status = SF_INITIAL
end subroutine sf_test_init
```

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set $x = r^2$ and consequently $f(r) = 2r$.

⟨*SF base: sf test int: TBP*⟩+≡
```
procedure :: complete_kinematics => sf_test_complete_kinematics
```

⟨*SF base: test auxiliary*⟩+≡
```
subroutine sf_test_complete_kinematics (sf_int, x, f, r, rb, map)
  class(sf_test_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  real(default), dimension(:), intent(in) :: rb
  logical, intent(in) :: map
```

```
  real(default) :: xb1
  if (map) then
     x(1) = r(1)**2
     f = 2 * r(1)
  else
     x(1) = r(1)
     f = 1
  end if
  xb1 = 1 - x(1)
  if (size (x) == 3)  x(2:3) = r(2:3)
  call sf_int%split_momentum (x, xb1)
  sf_int%x = x(1)
  select case (sf_int%status)
  case (SF_FAILED_KINEMATICS);  f = 0
  end select
end subroutine sf_test_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF base: sf test int: TBP*⟩+≡

```
procedure :: inverse_kinematics => sf_test_inverse_kinematics
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
  class(sf_test_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  real(default), dimension(:), intent(out) :: rb
  logical, intent(in) :: map
  logical, intent(in), optional :: set_momenta
  real(default) :: xb1
  logical :: set_mom
  set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
  if (map) then
     r(1) = sqrt (x(1))
     f = 2 * r(1)
  else
     r(1) = x(1)
     f = 1
  end if
  xb1 = 1 - x(1)
  if (size (x) == 3)  r(2:3) = x(2:3)
  rb = 1 - r
  sf_int%x = x(1)
  if (set_mom) then
     call sf_int%split_momentum (x, xb1)
     select case (sf_int%status)
     case (SF_FAILED_KINEMATICS);  f = 0
     end select
  end if
end subroutine sf_test_inverse_kinematics
```

Apply the structure function. The matrix element becomes unity and the application always succeeds.

If the `mode` indicator is one, the matrix element is equal to the parameter x.

⟨*SF base: sf test int: TBP*⟩+≡
```
procedure :: apply => sf_test_apply
```

⟨*SF base: test auxiliary*⟩+≡
```
subroutine sf_test_apply (sf_int, scale)
  class(sf_test_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  select case (sf_int%data%mode)
  case (0)
     call sf_int%set_matrix_element &
          (cmplx (1._default, kind=default))
  case (1)
     call sf_int%set_matrix_element &
          (cmplx (sf_int%x, kind=default))
  end select
  sf_int%status = SF_EVALUATED
end subroutine sf_test_apply
```

15.5.12 Test implementation: pair spectrum

Another template, this time for a incoming particle pair, splitting into two radiated and two outgoing particles.

Configuration data

For simplicity, the spectrum contains two mirror images of the previous structure-function configuration: the incoming and all outgoing particles are test scalars.

We have two versions, one with radiated particles, one without.

⟨*SF base: test types*⟩+≡
```
type, extends (sf_data_t) :: sf_test_spectrum_data_t
   class(model_data_t), pointer :: model => null ()
   type(flavor_t) :: flv_in
   type(flavor_t) :: flv_out
   type(flavor_t) :: flv_rad
   logical :: with_radiation = .true.
   real(default) :: m = 0
 contains
 ⟨SF base: sf test spectrum data: TBP⟩
end type sf_test_spectrum_data_t
```

Output.

⟨*SF base: sf test spectrum data: TBP*⟩≡
```
procedure :: write => sf_test_spectrum_data_write
```

⟨*SF base: test auxiliary*⟩+≡
```
subroutine sf_test_spectrum_data_write (data, unit, verbose)
  class(sf_test_spectrum_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
```

```
    integer :: u
    u = given_output_unit (unit)
    write (u, "(1x,A)")  "SF test spectrum data:"
    write (u, "(3x,A,A)") "model     = ", char (data%model%get_name ())
    write (u, "(3x,A)", advance="no") "incoming  = "
    call data%flv_in%write (u);  write (u, *)
    write (u, "(3x,A)", advance="no") "outgoing  = "
    call data%flv_out%write (u);  write (u, *)
    write (u, "(3x,A)", advance="no") "radiated  = "
    call data%flv_rad%write (u);  write (u, *)
    write (u, "(3x,A," // FMT_19 // ")")  "mass      = ", data%m
  end subroutine sf_test_spectrum_data_write
```

Initialization.

⟨*SF base: sf test spectrum data: TBP*⟩+≡

```
  procedure :: init => sf_test_spectrum_data_init
```

⟨*SF base: test auxiliary*⟩+≡

```
  subroutine sf_test_spectrum_data_init (data, model, pdg_in, with_radiation)
    class(sf_test_spectrum_data_t), intent(out) :: data
    class(model_data_t), intent(in), target :: model
    type(pdg_array_t), intent(in) :: pdg_in
    logical, intent(in) :: with_radiation
    data%model => model
    data%with_radiation = with_radiation
    if (pdg_array_get (pdg_in, 1) /= 25) then
       call msg_fatal ("Test structure function: input flavor must be 's'")
    end if
    call data%flv_in%init (25, model)
    data%m = data%flv_in%get_mass ()
    call data%flv_out%init (25, model)
    if (with_radiation) then
       call data%flv_rad%init (25, model)
    end if
  end subroutine sf_test_spectrum_data_init
```

Return the number of parameters: 2, since we have only collinear splitting here.

⟨*SF base: sf test spectrum data: TBP*⟩+≡

```
  procedure :: get_n_par => sf_test_spectrum_data_get_n_par
```

⟨*SF base: test auxiliary*⟩+≡

```
  function sf_test_spectrum_data_get_n_par (data) result (n)
    class(sf_test_spectrum_data_t), intent(in) :: data
    integer :: n
    n = 2
  end function sf_test_spectrum_data_get_n_par
```

Return the outgoing particle PDG codes: 25

⟨*SF base: sf test spectrum data: TBP*⟩+≡

```
  procedure :: get_pdg_out => sf_test_spectrum_data_get_pdg_out
```

⟨*SF base: test auxiliary*⟩+≡

```
  subroutine sf_test_spectrum_data_get_pdg_out (data, pdg_out)
    class(sf_test_spectrum_data_t), intent(in) :: data
```

```
    type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
    pdg_out(1) = 25
    pdg_out(2) = 25
  end subroutine sf_test_spectrum_data_get_pdg_out
```

Allocate the matching interaction.

⟨*SF base: sf test spectrum data: TBP*⟩+≡

```
  procedure :: allocate_sf_int => &
       sf_test_spectrum_data_allocate_sf_int
```

⟨*SF base: test auxiliary*⟩+≡

```
  subroutine sf_test_spectrum_data_allocate_sf_int (data, sf_int)
    class(sf_test_spectrum_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (sf_test_spectrum_t :: sf_int)
  end subroutine sf_test_spectrum_data_allocate_sf_int
```

Interaction

⟨*SF base: test types*⟩+≡

```
  type, extends (sf_int_t) :: sf_test_spectrum_t
     type(sf_test_spectrum_data_t), pointer :: data => null ()
   contains
   ⟨SF base: sf test spectrum: TBP⟩
  end type sf_test_spectrum_t
```

⟨*SF base: sf test spectrum: TBP*⟩≡

```
  procedure :: type_string => sf_test_spectrum_type_string
```

⟨*SF base: test auxiliary*⟩+≡

```
  function sf_test_spectrum_type_string (object) result (string)
    class(sf_test_spectrum_t), intent(in) :: object
    type(string_t) :: string
    string = "Test Spectrum"
  end function sf_test_spectrum_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF base: sf test spectrum: TBP*⟩+≡

```
  procedure :: write => sf_test_spectrum_write
```

⟨*SF base: test auxiliary*⟩+≡

```
  subroutine sf_test_spectrum_write (object, unit, testflag)
    class(sf_test_spectrum_t), intent(in) :: object
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: testflag
    integer :: u
    u = given_output_unit (unit)
    if (associated (object%data)) then
       call object%data%write (u)
       call object%base_write (u, testflag)
    else
       write (u, "(1x,A)")  "SF test spectrum data: [undefined]"
```

```
    end if
  end subroutine sf_test_spectrum_write
```

Initialize. We know that `data` will be of concrete type `sf_test_spectrum_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass, but keep the radiated mass zero.

Optionally, we can provide minimum and maximum values for the momentum transfer.

⟨*SF base: sf test spectrum: TBP*⟩+≡

```
  procedure :: init => sf_test_spectrum_init
```

⟨*SF base: test auxiliary*⟩+≡

```
  subroutine sf_test_spectrum_init (sf_int, data)
    class(sf_test_spectrum_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    type(quantum_numbers_mask_t), dimension(6) :: mask
    type(helicity_t) :: hel0
    type(color_t) :: col0
    type(quantum_numbers_t), dimension(6) :: qn
    mask = quantum_numbers_mask (.false., .false., .false.)
    select type (data)
    type is (sf_test_spectrum_data_t)
       if (data%with_radiation) then
          call sf_int%base_init (mask(1:6), &
               [data%m**2, data%m**2], &
               [0._default, 0._default], &
               [data%m**2, data%m**2])
          sf_int%data => data
          call hel0%init (0)
          call col0%init ()
          call qn(1)%init (data%flv_in,  col0, hel0)
          call qn(2)%init (data%flv_in,  col0, hel0)
          call qn(3)%init (data%flv_rad, col0, hel0)
          call qn(4)%init (data%flv_rad, col0, hel0)
          call qn(5)%init (data%flv_out, col0, hel0)
          call qn(6)%init (data%flv_out, col0, hel0)
          call sf_int%add_state (qn(1:6))
          call sf_int%set_incoming ([1,2])
          call sf_int%set_radiated ([3,4])
          call sf_int%set_outgoing ([5,6])
       else
          call sf_int%base_init (mask(1:4), &
               [data%m**2, data%m**2], &
               [real(default) :: ], &
               [data%m**2, data%m**2])
          sf_int%data => data
          call hel0%init (0)
          call col0%init ()
          call qn(1)%init (data%flv_in,  col0, hel0)
          call qn(2)%init (data%flv_in,  col0, hel0)
          call qn(3)%init (data%flv_out, col0, hel0)
          call qn(4)%init (data%flv_out, col0, hel0)
          call sf_int%add_state (qn(1:4))
```

```
        call sf_int%set_incoming ([1,2])
        call sf_int%set_outgoing ([3,4])
     end if
     call sf_int%freeze ()
  end select
  sf_int%status = SF_INITIAL
end subroutine sf_test_spectrum_init
```

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set $x = r^2$ (as above) for both x parameters and consequently $f(r) = 4r_1r_2$.

⟨*SF base: sf test spectrum: TBP*⟩+≡

```
procedure :: complete_kinematics => sf_test_spectrum_complete_kinematics
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_spectrum_complete_kinematics (sf_int, x, f, r, rb, map)
  class(sf_test_spectrum_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  real(default), dimension(:), intent(in) :: rb
  logical, intent(in) :: map
  real(default), dimension(2) :: xb1
  if (map) then
     x = r**2
     f = 4 * r(1) * r(2)
  else
     x = r
     f = 1
  end if
  if (sf_int%data%with_radiation) then
     xb1 = 1 - x
     call sf_int%split_momenta (x, xb1)
  else
     call sf_int%reduce_momenta (x)
  end if
  select case (sf_int%status)
  case (SF_FAILED_KINEMATICS);  f = 0
  end select
end subroutine sf_test_spectrum_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF base: sf test spectrum: TBP*⟩+≡

```
procedure :: inverse_kinematics => sf_test_spectrum_inverse_kinematics
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_spectrum_inverse_kinematics &
     (sf_int, x, f, r, rb, map, set_momenta)
  class(sf_test_spectrum_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
```

```
    real(default), intent(out) :: f
    real(default), dimension(:), intent(out) :: r
    real(default), dimension(:), intent(out) :: rb
    logical, intent(in) :: map
    logical, intent(in), optional :: set_momenta
    real(default), dimension(2) :: xb1
    logical :: set_mom
    set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
    if (map) then
       r = sqrt (x)
       f = 4 * r(1) * r(2)
    else
       r = x
       f = 1
    end if
    rb = 1 - r
    if (set_mom)  then
       if (sf_int%data%with_radiation) then
          xb1 = 1 - x
          call sf_int%split_momenta (x, xb1)
       else
          call sf_int%reduce_momenta (x)
       end if
       select case (sf_int%status)
       case (SF_FAILED_KINEMATICS);  f = 0
       end select
    end if
  end subroutine sf_test_spectrum_inverse_kinematics
```

Apply the structure function. The matrix element becomes unity and the application always succeeds.

⟨*SF base: sf test spectrum: TBP*⟩+≡

```
  procedure :: apply => sf_test_spectrum_apply
```

⟨*SF base: test auxiliary*⟩+≡

```
  subroutine sf_test_spectrum_apply (sf_int, scale)
    class(sf_test_spectrum_t), intent(inout) :: sf_int
    real(default), intent(in) :: scale
    call sf_int%set_matrix_element &
         (cmplx (1._default, kind=default))
    sf_int%status = SF_EVALUATED
  end subroutine sf_test_spectrum_apply
```

15.5.13 Test implementation: generator spectrum

A generator for two beams, no radiation (for simplicity).

Configuration data

For simplicity, the spectrum contains two mirror images of the previous structure-function configuration: the incoming and all outgoing particles are test scalars.

We have two versions, one with radiated particles, one without.

⟨*SF base: test types*⟩+≡

```
type, extends (sf_data_t) :: sf_test_generator_data_t
   class(model_data_t), pointer :: model => null ()
   type(flavor_t) :: flv_in
   type(flavor_t) :: flv_out
   type(flavor_t) :: flv_rad
   real(default) :: m = 0
 contains
 ⟨SF base: sf test generator data: TBP⟩
end type sf_test_generator_data_t
```

Output.

⟨*SF base: sf test generator data: TBP*⟩≡

```
procedure :: write => sf_test_generator_data_write
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_generator_data_write (data, unit, verbose)
  class(sf_test_generator_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "SF test generator data:"
  write (u, "(3x,A,A)") "model     = ", char (data%model%get_name ())
  write (u, "(3x,A)", advance="no") "incoming  = "
  call data%flv_in%write (u);  write (u, *)
  write (u, "(3x,A)", advance="no") "outgoing  = "
  call data%flv_out%write (u);  write (u, *)
  write (u, "(3x,A," // FMT_19 // ")")  "mass      = ", data%m
end subroutine sf_test_generator_data_write
```

Initialization.

⟨*SF base: sf test generator data: TBP*⟩+≡

```
procedure :: init => sf_test_generator_data_init
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_generator_data_init (data, model, pdg_in)
  class(sf_test_generator_data_t), intent(out) :: data
  class(model_data_t), intent(in), target :: model
  type(pdg_array_t), intent(in) :: pdg_in
  data%model => model
  if (pdg_array_get (pdg_in, 1) /= 25) then
     call msg_fatal ("Test generator: input flavor must be 's'")
  end if
  call data%flv_in%init (25, model)
  data%m = data%flv_in%get_mass ()
  call data%flv_out%init (25, model)
end subroutine sf_test_generator_data_init
```

This structure function is a generator.

⟨*SF base: sf test generator data: TBP*⟩+≡

```
procedure :: is_generator => sf_test_generator_data_is_generator
```

⟨*SF base: test auxiliary*⟩+≡
```
function sf_test_generator_data_is_generator (data) result (flag)
  class(sf_test_generator_data_t), intent(in) :: data
  logical :: flag
  flag = .true.
end function sf_test_generator_data_is_generator
```

Return the number of parameters: 2, since we have only collinear splitting here.

⟨*SF base: sf test generator data: TBP*⟩+≡
```
procedure :: get_n_par => sf_test_generator_data_get_n_par
```

⟨*SF base: test auxiliary*⟩+≡
```
function sf_test_generator_data_get_n_par (data) result (n)
  class(sf_test_generator_data_t), intent(in) :: data
  integer :: n
  n = 2
end function sf_test_generator_data_get_n_par
```

Return the outgoing particle PDG codes: 25

⟨*SF base: sf test generator data: TBP*⟩+≡
```
procedure :: get_pdg_out => sf_test_generator_data_get_pdg_out
```

⟨*SF base: test auxiliary*⟩+≡
```
subroutine sf_test_generator_data_get_pdg_out (data, pdg_out)
  class(sf_test_generator_data_t), intent(in) :: data
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
  pdg_out(1) = 25
  pdg_out(2) = 25
end subroutine sf_test_generator_data_get_pdg_out
```

Allocate the matching interaction.

⟨*SF base: sf test generator data: TBP*⟩+≡
```
procedure :: allocate_sf_int => &
     sf_test_generator_data_allocate_sf_int
```

⟨*SF base: test auxiliary*⟩+≡
```
subroutine sf_test_generator_data_allocate_sf_int (data, sf_int)
  class(sf_test_generator_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  allocate (sf_test_generator_t :: sf_int)
end subroutine sf_test_generator_data_allocate_sf_int
```

Interaction

⟨*SF base: test types*⟩+≡
```
type, extends (sf_int_t) :: sf_test_generator_t
   type(sf_test_generator_data_t), pointer :: data => null ()
 contains
 ⟨SF base: sf test generator: TBP⟩
end type sf_test_generator_t
```

⟨*SF base: sf test generator: TBP*⟩≡

```
procedure :: type_string => sf_test_generator_type_string
```

⟨*SF base: test auxiliary*⟩+≡

```
function sf_test_generator_type_string (object) result (string)
  class(sf_test_generator_t), intent(in) :: object
  type(string_t) :: string
  string = "Test Generator"
end function sf_test_generator_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF base: sf test generator: TBP*⟩+≡

```
procedure :: write => sf_test_generator_write
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_generator_write (object, unit, testflag)
  class(sf_test_generator_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     call object%base_write (u, testflag)
  else
     write (u, "(1x,A)")  "SF test generator data: [undefined]"
  end if
end subroutine sf_test_generator_write
```

Initialize. We know that `data` will be of concrete type `sf_test_generator_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass. No radiation.

⟨*SF base: sf test generator: TBP*⟩+≡

```
procedure :: init => sf_test_generator_init
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_generator_init (sf_int, data)
  class(sf_test_generator_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  type(quantum_numbers_mask_t), dimension(4) :: mask
  type(helicity_t) :: hel0
  type(color_t) :: col0
  type(quantum_numbers_t), dimension(4) :: qn
  mask = quantum_numbers_mask (.false., .false., .false.)
  select type (data)
  type is (sf_test_generator_data_t)
     call sf_int%base_init (mask(1:4), &
          [data%m**2, data%m**2], &
          [real(default) :: ], &
          [data%m**2, data%m**2])
     sf_int%data => data
     call hel0%init (0)
     call col0%init ()
```

```
      call qn(1)%init (data%flv_in,  col0, hel0)
      call qn(2)%init (data%flv_in,  col0, hel0)
      call qn(3)%init (data%flv_out, col0, hel0)
      call qn(4)%init (data%flv_out, col0, hel0)
      call sf_int%add_state (qn(1:4))
      call sf_int%set_incoming ([1,2])
      call sf_int%set_outgoing ([3,4])
      call sf_int%freeze ()
    end select
    sf_int%status = SF_INITIAL
  end subroutine sf_test_generator_init
```

This structure function is a generator.

⟨*SF base: sf test generator: TBP*⟩+≡

```
  procedure :: is_generator => sf_test_generator_is_generator
```

⟨*SF base: test auxiliary*⟩+≡

```
  function sf_test_generator_is_generator (sf_int) result (flag)
    class(sf_test_generator_t), intent(in) :: sf_int
    logical :: flag
    flag = sf_int%data%is_generator ()
  end function sf_test_generator_is_generator
```

Generate free parameters. This mock generator always produces the nubmers 0.8 and 0.5.

⟨*SF base: sf test generator: TBP*⟩+≡

```
  procedure :: generate_free => sf_test_generator_generate_free
```

⟨*SF base: test auxiliary*⟩+≡

```
  subroutine sf_test_generator_generate_free (sf_int, r, rb,  x_free)
    class(sf_test_generator_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: r, rb
    real(default), intent(inout) :: x_free
    r = [0.8, 0.5]
    rb= 1 - r
    x_free = x_free * product (r)
  end subroutine sf_test_generator_generate_free
```

Recover momentum fractions. Since the x values are free, we also set the `x_free` parameter.

⟨*SF base: sf test generator: TBP*⟩+≡

```
  procedure :: recover_x => sf_test_generator_recover_x
```

⟨*SF base: test auxiliary*⟩+≡

```
  subroutine sf_test_generator_recover_x (sf_int, x, x_free)
    class(sf_test_generator_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: x
    real(default), intent(inout), optional :: x_free
    call sf_int%base_recover_x (x)
    if (present (x_free))  x_free = x_free * product (x)
  end subroutine sf_test_generator_recover_x
```

Set kinematics. Since this is a generator, just transfer input to output.

⟨*SF base: sf test generator: TBP*⟩+≡

```
procedure :: complete_kinematics => sf_test_generator_complete_kinematics
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_generator_complete_kinematics (sf_int, x, f, r, rb, map)
  class(sf_test_generator_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  real(default), dimension(:), intent(in) :: rb
  logical, intent(in) :: map
  x = r
  f = 1
  call sf_int%reduce_momenta (x)
end subroutine sf_test_generator_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF base: sf test generator: TBP*⟩+≡

```
procedure :: inverse_kinematics => sf_test_generator_inverse_kinematics
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_generator_inverse_kinematics &
     (sf_int, x, f, r, rb, map, set_momenta)
  class(sf_test_generator_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  real(default), dimension(:), intent(out) :: rb
  logical, intent(in) :: map
  logical, intent(in), optional :: set_momenta
  logical :: set_mom
  set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
  r = x
  rb= 1 - x
  f = 1
  if (set_mom)  call sf_int%reduce_momenta (x)
end subroutine sf_test_generator_inverse_kinematics
```

Apply the structure function. The matrix element becomes unity and the application always succeeds.

⟨*SF base: sf test generator: TBP*⟩+≡

```
procedure :: apply => sf_test_generator_apply
```

⟨*SF base: test auxiliary*⟩+≡

```
subroutine sf_test_generator_apply (sf_int, scale)
  class(sf_test_generator_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  call sf_int%set_matrix_element &
       (cmplx (1._default, kind=default))
  sf_int%status = SF_EVALUATED
end subroutine sf_test_generator_apply
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF base: execute tests*⟩≡

```
call test (sf_base_1, "sf_base_1", &
     "structure function configuration", &
     u, results)
```

⟨*SF base: test declarations*⟩≡

```
public :: sf_base_1
```

⟨*SF base: tests*⟩≡

```
subroutine sf_base_1 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(pdg_array_t) :: pdg_in
  type(pdg_array_t), dimension(1) :: pdg_out
  integer, dimension(:), allocatable :: pdg1
  class(sf_data_t), allocatable :: data

  write (u, "(A)")  "* Test output: sf_base_1"
  write (u, "(A)")  "*   Purpose: initialize and display &
       &test structure function data"
  write (u, "(A)")

  call model%init_test ()
  pdg_in = 25

  allocate (sf_test_data_t :: data)
  select type (data)
  type is (sf_test_data_t)
     call data%init (model, pdg_in)
  end select

  call data%write (u)

  write (u, "(A)")

  write (u, "(1x,A)")  "Outgoing particle code:"
  call data%get_pdg_out (pdg_out)
  pdg1 = pdg_out(1)
  write (u, "(2x,99(1x,I0))")  pdg1

  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_base_1"

end subroutine sf_base_1
```

Test and probe structure function

Construct and display a structure function object based on the test structure function.

⟨*SF base: execute tests*⟩+≡

```
call test (sf_base_2, "sf_base_2", &
     "structure function instance", &
     u, results)
```

⟨*SF base: test declarations*⟩+≡

```
public :: sf_base_2
```

⟨*SF base: tests*⟩+≡

```
subroutine sf_base_2 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t) :: flv
  type(pdg_array_t) :: pdg_in
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k
  type(vector4_t), dimension(2) :: q
  real(default) :: E
  real(default), dimension(:), allocatable :: r, rb, x
  real(default) :: f

  write (u, "(A)")  "* Test output: sf_base_2"
  write (u, "(A)")  "*   Purpose: initialize and fill &
       &test structure function object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call model%init_test ()
  pdg_in = 25
  call flv%init (25, model)

  call reset_interaction_counter ()

  allocate (sf_test_data_t :: data)
  select type (data)
  type is (sf_test_data_t)
     call data%init (model, pdg_in)
  end select

  write (u, "(A)")  "* Initialize structure-function object"
  write (u, "(A)")

  call data%allocate_sf_int (sf_int)
  call sf_int%init (data)
  call sf_int%set_beam_index ([1])

  call sf_int%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize incoming momentum with E=500"
  write (u, "(A)")
  E = 500
```

```
k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=1"
write (u, "(A)")

r = 1
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5"
write (u, "(A)")

r = 0.5_default
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Set kinematics with mapping for r=0.8"
write (u, "(A)")

r = 0.8_default
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.true.)
```

```
    call sf_int%write (u)

    write (u, "(A)")
    write (u, "(A,9(1x,F10.7))")  "x =", x
    write (u, "(A,9(1x,F10.7))")  "f =", f

    write (u, "(A)")
    write (u, "(A)")  "* Recover x from momenta"
    write (u, "(A)")

    q = sf_int%get_momenta (outgoing=.true.)
    call sf_int%final ()
    deallocate (sf_int)

    call data%allocate_sf_int (sf_int)
    call sf_int%init (data)
    call sf_int%set_beam_index ([1])

    call sf_int%seed_kinematics ([k])
    call sf_int%set_momenta (q, outgoing=.true.)
    call sf_int%recover_x (x)

    write (u, "(A,9(1x,F10.7))")  "x =", x

    write (u, "(A)")
    write (u, "(A)")  "* Compute inverse kinematics for x=0.64 and evaluate"
    write (u, "(A)")

    x = 0.64_default
    call sf_int%inverse_kinematics (x, f, r, rb, map=.true.)
    call sf_int%apply (scale=0._default)
    call sf_int%write (u)

    write (u, "(A)")
    write (u, "(A,9(1x,F10.7))")  "r =", r
    write (u, "(A,9(1x,F10.7))")  "f =", f

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_2"

  end subroutine sf_base_2
```

Collinear kinematics

Scan over the possibilities for mass assignment and on-shell projections, collinear case.

⟨*SF base: execute tests*⟩+≡

```
call test (sf_base_3, "sf_base_3", &
     "alternatives for collinear kinematics", &
     u, results)
```

⟨*SF base: test declarations*⟩+≡

```
public :: sf_base_3
```

⟨*SF base: tests*⟩+≡

```
subroutine sf_base_3 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(pdg_array_t) :: pdg_in
  type(flavor_t) :: flv
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k
  real(default) :: E
  real(default), dimension(:), allocatable :: r, rb, x
  real(default) :: f

  write (u, "(A)")  "* Test output: sf_base_3"
  write (u, "(A)")  "*   Purpose: check various kinematical setups"
  write (u, "(A)")  "*            for collinear structure-function splitting."
  write (u, "(A)")  "             (two masses equal, one zero)"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call model%init_test ()
  pdg_in = 25
  call flv%init (25, model)

  call reset_interaction_counter ()

  allocate (sf_test_data_t :: data)
  select type (data)
  type is (sf_test_data_t)
     call data%init (model, pdg_in)
  end select

  write (u, "(A)")  "* Initialize structure-function object"
  write (u, "(A)")

  call data%allocate_sf_int (sf_int)
  call sf_int%init (data)

  call sf_int%write (u)

  allocate (r (data%get_n_par ()))
  allocate (rb(size (r)))
  allocate (x (size (r)))

  write (u, "(A)")
```

```
write (u, "(A)")  "* Initialize incoming momentum with E=500"

E = 500
k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set radiated mass to zero"

sf_int%mr2 = 0
sf_int%mo2 = sf_int%mi2

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping energy"
write (u, "(A)")

r = 0.5_default
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping momentum"
write (u, "(A)")

r = 0.5_default
rb = 1 - r
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set outgoing mass to zero"

sf_int%mr2 = sf_int%mi2
```

```
sf_int%mo2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping energy"
write (u, "(A)")

r = 0.5_default
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping momentum"
write (u, "(A)")

r = 0.5_default
rb = 1 - r
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set incoming mass to zero"

k = vector4_moving (E, E, 3)
call sf_int%seed_kinematics ([k])

sf_int%mr2 = sf_int%mi2
sf_int%mo2 = sf_int%mi2
sf_int%mi2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping energy"
write (u, "(A)")
```

```
r = 0.5_default
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping momentum"
write (u, "(A)")

r = 0.5_default
rb = 1 - r
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set all masses to zero"

sf_int%mr2 = 0
sf_int%mo2 = 0
sf_int%mi2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5, keeping energy"
write (u, "(A)")

r = 0.5_default
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")
```

```
    call sf_int%recover_x (x)
    call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
    write (u, "(A,9(1x,F10.7))")  "x =", x
    write (u, "(A,9(1x,F10.7))")  "r =", r

    write (u, "(A)")
    write (u, "(A)")  "* Set kinematics for x=0.5, keeping momentum"
    write (u, "(A)")

    r = 0.5_default
    rb = 1 - r
    sf_int%on_shell_mode = KEEP_MOMENTUM
    call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
    call sf_int%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Recover x and r"
    write (u, "(A)")

    call sf_int%recover_x (x)
    call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
    write (u, "(A,9(1x,F10.7))")  "x =", x
    write (u, "(A,9(1x,F10.7))")  "r =", r

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_3"

  end subroutine sf_base_3
```

Non-collinear kinematics

Scan over the possibilities for mass assignment and on-shell projections, non-collinear case.

⟨*SF base: execute tests*⟩+≡

```
  call test (sf_base_4, "sf_base_4", &
       "alternatives for non-collinear kinematics", &
       u, results)
```

⟨*SF base: test declarations*⟩+≡

```
  public :: sf_base_4
```

⟨*SF base: tests*⟩+≡

```
  subroutine sf_base_4 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(pdg_array_t) :: pdg_in
```

```
type(flavor_t) :: flv
class(sf_data_t), allocatable, target :: data
class(sf_int_t), allocatable :: sf_int
type(vector4_t) :: k
real(default) :: E
real(default), dimension(:), allocatable :: r, rb, x
real(default) :: f

write (u, "(A)")  "* Test output: sf_base_4"
write (u, "(A)")  "*   Purpose: check various kinematical setups"
write (u, "(A)")  "*            for free structure-function splitting."
write (u, "(A)")  "             (two masses equal, one zero)"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call model%init_test ()
pdg_in = 25
call flv%init (25, model)

call reset_interaction_counter ()

allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
   call data%init (model, pdg_in, collinear=.false.)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)

call sf_int%write (u)

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"

E = 500
k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set radiated mass to zero"

sf_int%mr2 = 0
sf_int%mo2 = sf_int%mi2
```

```
write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping energy"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping momentum"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set outgoing mass to zero"

sf_int%mr2 = sf_int%mi2
sf_int%mo2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping energy"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)
```

```
write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping momentum"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set incoming mass to zero"

k = vector4_moving (E, E, 3)
call sf_int%seed_kinematics ([k])

sf_int%mr2 = sf_int%mi2
sf_int%mo2 = sf_int%mi2
sf_int%mi2 = 0

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping energy"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
```

```
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping momentum"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_MOMENTUM
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Recover x and r"
write (u, "(A)")

call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Set all masses to zero"

sf_int%mr2 = 0
sf_int%mo2 = 0
sf_int%mi2 = 0

write (u, "(A)")
write (u, "(A)")  "* Re-Initialize structure-function object with Q bounds"

call reset_interaction_counter ()

select type (data)
type is (sf_test_data_t)
   call data%init (model, pdg_in, collinear=.false., &
        qbounds = [1._default, 100._default])
end select

call sf_int%init (data)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping energy"
write (u, "(A)")

r = [0.5_default, 0.5_default, 0.125_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
```

```
    write (u, "(A)")  "* Recover x and r"
    write (u, "(A)")

    call sf_int%recover_x (x)
    call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
    write (u, "(A,9(1x,F10.7))")  "x =", x
    write (u, "(A,9(1x,F10.7))")  "r =", r

    write (u, "(A)")
    write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.125, keeping momentum"
    write (u, "(A)")

    r = [0.5_default, 0.5_default, 0.125_default]
    rb = 1 - r
    sf_int%on_shell_mode = KEEP_MOMENTUM
    call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
    call sf_int%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Recover x and r"
    write (u, "(A)")

    call sf_int%recover_x (x)
    call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
    write (u, "(A,9(1x,F10.7))")  "x =", x
    write (u, "(A,9(1x,F10.7))")  "r =", r

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_4"

  end subroutine sf_base_4
```

Pair spectrum

Construct and display a structure function object for a pair spectrum (a structure function involving two particles simultaneously).

⟨*SF base: execute tests*⟩+≡

```
  call test (sf_base_5, "sf_base_5", &
       "pair spectrum with radiation", &
       u, results)
```

⟨*SF base: test declarations*⟩+≡

```
  public :: sf_base_5
```

⟨*SF base: tests*⟩+≡

```
  subroutine sf_base_5 (u)
    integer, intent(in) :: u
```

```
type(model_data_t), target :: model
type(pdg_array_t) :: pdg_in
type(pdg_array_t), dimension(2) :: pdg_out
integer, dimension(:), allocatable :: pdg1, pdg2
type(flavor_t) :: flv
class(sf_data_t), allocatable, target :: data
class(sf_int_t), allocatable :: sf_int
type(vector4_t), dimension(2) :: k
type(vector4_t), dimension(4) :: q
real(default) :: E
real(default), dimension(:), allocatable :: r, rb, x
real(default) :: f

write (u, "(A)")  "* Test output: sf_base_5"
write (u, "(A)")  "*   Purpose: initialize and fill &
     &a pair spectrum object"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call model%init_test ()
call flv%init (25, model)
pdg_in = 25

call reset_interaction_counter ()

allocate (sf_test_spectrum_data_t :: data)
select type (data)
type is (sf_test_spectrum_data_t)
   call data%init (model, pdg_in, with_radiation=.true.)
end select

write (u, "(1x,A)")  "Outgoing particle codes:"
call data%get_pdg_out (pdg_out)
pdg1 = pdg_out(1)
pdg2 = pdg_out(2)
write (u, "(2x,99(1x,I0))")  pdg1, pdg2

write (u, "(A)")
write (u, "(A)")  "* Initialize spectrum object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momenta with sqrts=1000"

E = 500
k(1) = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
k(2) = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
```

```
call sf_int%seed_kinematics (k)

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.4,0.8"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = [0.4_default, 0.8_default]
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Set kinematics with mapping for r=0.6,0.8"
write (u, "(A)")

r = [0.6_default, 0.8_default]
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.true.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = sf_int%get_momenta (outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call reset_interaction_counter ()
call data%allocate_sf_int (sf_int)
call sf_int%init (data)

call sf_int%seed_kinematics (k)
call sf_int%set_momenta (q, outgoing=.true.)
call sf_int%recover_x (x)
write (u, "(A,9(1x,F10.7))")  "x =", x

write (u, "(A)")
write (u, "(A)")  "* Compute inverse kinematics for x=0.36,0.64 &
     &and evaluate"
write (u, "(A)")
```

```
    x = [0.36_default, 0.64_default]
    call sf_int%inverse_kinematics (x, f, r, rb, map=.true.)
    call sf_int%apply (scale=0._default)
    call sf_int%write (u)

    write (u, "(A)")
    write (u, "(A,9(1x,F10.7))")  "r =", r
    write (u, "(A,9(1x,F10.7))")  "f =", f

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_5"

  end subroutine sf_base_5
```

Pair spectrum without radiation

Construct and display a structure function object for a pair spectrum (a structure function involving two particles simultaneously).

⟨*SF base: execute tests*⟩+≡

```
  call test (sf_base_6, "sf_base_6", &
       "pair spectrum without radiation", &
       u, results)
```

⟨*SF base: test declarations*⟩+≡

```
  public :: sf_base_6
```

⟨*SF base: tests*⟩+≡

```
  subroutine sf_base_6 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(pdg_array_t) :: pdg_in
    type(flavor_t) :: flv
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t), dimension(2) :: k
    type(vector4_t), dimension(2) :: q
    real(default) :: E
    real(default), dimension(:), allocatable :: r, rb, x
    real(default) :: f

    write (u, "(A)")  "* Test output: sf_base_6"
    write (u, "(A)")  "*   Purpose: initialize and fill &
         &a pair spectrum object"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize configuration data"
    write (u, "(A)")
```

```
call model%init_test ()
call flv%init (25, model)
pdg_in = 25

call reset_interaction_counter ()

allocate (sf_test_spectrum_data_t :: data)
select type (data)
type is (sf_test_spectrum_data_t)
   call data%init (model, pdg_in, with_radiation=.false.)
end select

write (u, "(A)")  "* Initialize spectrum object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)

write (u, "(A)")  "* Initialize incoming momenta with sqrts=1000"

E = 500
k(1) = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
k(2) = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call sf_int%seed_kinematics (k)

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.4,0.8"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = [0.4_default, 0.8_default]
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = sf_int%get_momenta (outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call reset_interaction_counter ()
call data%allocate_sf_int (sf_int)
call sf_int%init (data)
```

```
    call sf_int%seed_kinematics (k)
    call sf_int%set_momenta (q, outgoing=.true.)
    call sf_int%recover_x (x)
    write (u, "(A,9(1x,F10.7))")  "x =", x

    write (u, "(A)")
    write (u, "(A)")  "* Compute inverse kinematics for x=0.4,0.8 &
         &and evaluate"
    write (u, "(A)")

    x = [0.4_default, 0.8_default]
    call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
    call sf_int%apply (scale=0._default)
    call sf_int%write (u)

    write (u, "(A)")
    write (u, "(A,9(1x,F10.7))")  "r =", r
    write (u, "(A,9(1x,F10.7))")  "f =", f

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_6"

  end subroutine sf_base_6
```

Direct access to structure function

Probe a structure function directly.

⟨*SF base: execute tests*⟩+≡

```
  call test (sf_base_7, "sf_base_7", &
       "direct access", &
       u, results)
```

⟨*SF base: test declarations*⟩+≡

```
  public :: sf_base_7
```

⟨*SF base: tests*⟩+≡

```
  subroutine sf_base_7 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(pdg_array_t) :: pdg_in
    type(flavor_t) :: flv
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    real(default), dimension(:), allocatable :: value

    write (u, "(A)")  "* Test output: sf_base_7"
    write (u, "(A)")  "*   Purpose: check direct access method"
```

```
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call model%init_test ()
call flv%init (25, model)
pdg_in = 25

call reset_interaction_counter ()

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
   call data%init (model, pdg_in)
end select

call data%allocate_sf_int (sf_int)
call sf_int%init (data)

write (u, "(A)")  "* Probe structure function: states"
write (u, "(A)")

write (u, "(A,I0)")  "n_states = ", sf_int%get_n_states ()
write (u, "(A,I0)")  "n_in     = ", sf_int%get_n_in ()
write (u, "(A,I0)")  "n_rad    = ", sf_int%get_n_rad ()
write (u, "(A,I0)")  "n_out    = ", sf_int%get_n_out ()
write (u, "(A)")
write (u, "(A)", advance="no")  "state(1)  = "
call quantum_numbers_write (sf_int%get_state (1), u)
write (u, *)

allocate (value (sf_int%get_n_states ()))
call sf_int%compute_values (value, &
     E=[500._default], x=[0.5_default], xb=[0.5_default], scale=0._default)

write (u, "(A)")
write (u, "(A)", advance="no")  "value (E=500, x=0.5) ="
write (u, "(9(1x," // FMT_19 // "))")  value

call sf_int%compute_values (value, &
     x=[0.1_default], xb=[0.9_default], scale=0._default)

write (u, "(A)")
write (u, "(A)", advance="no")  "value (E=500, x=0.1) ="
write (u, "(9(1x," // FMT_19 // "))")  value


write (u, "(A)")
write (u, "(A)")  "* Initialize spectrum object"
write (u, "(A)")
```

```
    deallocate (value)
    call sf_int%final ()
    deallocate (sf_int)
    deallocate (data)

    allocate (sf_test_spectrum_data_t :: data)
    select type (data)
    type is (sf_test_spectrum_data_t)
       call data%init (model, pdg_in, with_radiation=.false.)
    end select

    call data%allocate_sf_int (sf_int)
    call sf_int%init (data)

    write (u, "(A)")  "* Probe spectrum: states"
    write (u, "(A)")

    write (u, "(A,I0)")  "n_states = ", sf_int%get_n_states ()
    write (u, "(A,I0)")  "n_in     = ", sf_int%get_n_in ()
    write (u, "(A,I0)")  "n_rad    = ", sf_int%get_n_rad ()
    write (u, "(A,I0)")  "n_out    = ", sf_int%get_n_out ()
    write (u, "(A)")
    write (u, "(A)", advance="no")  "state(1)  = "
    call quantum_numbers_write (sf_int%get_state (1), u)
    write (u, *)

    allocate (value (sf_int%get_n_states ()))
    call sf_int%compute_value (1, value(1), &
         E = [500._default, 500._default], &
         x = [0.5_default, 0.6_default], &
         xb= [0.5_default, 0.4_default], &
         scale = 0._default)

    write (u, "(A)")
    write (u, "(A)", advance="no")  "value (E=500,500, x=0.5,0.6) ="
    write (u, "(9(1x," // FMT_19 // "))")  value

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_7"

  end subroutine sf_base_7
```

Structure function chain configuration

⟨*SF base: execute tests*⟩+≡

```
  call test (sf_base_8, "sf_base_8", &
```

```
         "structure function chain configuration", &
         u, results)
```

⟨*SF base: test declarations*⟩+≡

```
    public :: sf_base_8
```

⟨*SF base: tests*⟩+≡

```
    subroutine sf_base_8 (u)
      integer, intent(in) :: u
      type(model_data_t), target :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      type(beam_data_t), target :: beam_data
      class(sf_data_t), allocatable, target :: data_strfun
      class(sf_data_t), allocatable, target :: data_spectrum
      type(sf_config_t), dimension(:), allocatable :: sf_config
      type(sf_chain_t) :: sf_chain

      write (u, "(A)")  "* Test output: sf_base_8"
      write (u, "(A)")  "*   Purpose: set up a structure-function chain"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call model%init_test ()
      call flv%init (25, model)
      pdg_in = 25

      call reset_interaction_counter ()

      call beam_data%init_sqrts (1000._default, [flv, flv])

      allocate (sf_test_data_t :: data_strfun)
      select type (data_strfun)
      type is (sf_test_data_t)
         call data_strfun%init (model, pdg_in)
      end select

      allocate (sf_test_spectrum_data_t :: data_spectrum)
      select type (data_spectrum)
      type is (sf_test_spectrum_data_t)
         call data_spectrum%init (model, pdg_in, with_radiation=.true.)
      end select

      write (u, "(A)")  "* Set up chain with beams only"
      write (u, "(A)")

      call sf_chain%init (beam_data)
      call write_separator (u, 2)
      call sf_chain%write (u)
      call write_separator (u, 2)
      call sf_chain%final ()

      write (u, "(A)")
```

```
    write (u, "(A)")  "* Set up chain with structure function"
    write (u, "(A)")

    allocate (sf_config (1))
    call sf_config(1)%init ([1], data_strfun)
    call sf_chain%init (beam_data, sf_config)

    call write_separator (u, 2)
    call sf_chain%write (u)
    call write_separator (u, 2)
    call sf_chain%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Set up chain with spectrum and structure function"
    write (u, "(A)")

    deallocate (sf_config)
    allocate (sf_config (2))
    call sf_config(1)%init ([1,2], data_spectrum)
    call sf_config(2)%init ([2], data_strfun)
    call sf_chain%init (beam_data, sf_config)

    call write_separator (u, 2)
    call sf_chain%write (u)
    call write_separator (u, 2)
    call sf_chain%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_8"

  end subroutine sf_base_8
```

Structure function instance configuration

We create a structure-function chain instance which implements a configured structure-function chain. We link the momentum entries in the interactions and compute kinematics.

We do not actually connect the interactions and create evaluators. We skip this step and manually advance the status of the chain instead.

⟨*SF base: execute tests*⟩+≡

```
  call test (sf_base_9, "sf_base_9", &
       "structure function chain instance", &
       u, results)
```

⟨*SF base: test declarations*⟩+≡

```
  public :: sf_base_9
```

⟨*SF base: tests*⟩+≡

```
subroutine sf_base_9 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t) :: flv
  type(pdg_array_t) :: pdg_in
  type(beam_data_t), target :: beam_data
  class(sf_data_t), allocatable, target :: data_strfun
  class(sf_data_t), allocatable, target :: data_spectrum
  type(sf_config_t), dimension(:), allocatable, target :: sf_config
  type(sf_chain_t), target :: sf_chain
  type(sf_chain_instance_t), target :: sf_chain_instance
  type(sf_channel_t), dimension(2) :: sf_channel
  type(vector4_t), dimension(2) :: p
  integer :: j

  write (u, "(A)")  "* Test output: sf_base_9"
  write (u, "(A)")  "*   Purpose: set up a structure-function chain &
       &and create an instance"
  write (u, "(A)")  "*            compute kinematics"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call model%init_test ()
  call flv%init (25, model)
  pdg_in = 25

  call reset_interaction_counter ()

  call beam_data%init_sqrts (1000._default, [flv, flv])

  allocate (sf_test_data_t :: data_strfun)
  select type (data_strfun)
  type is (sf_test_data_t)
     call data_strfun%init (model, pdg_in)
  end select

  allocate (sf_test_spectrum_data_t :: data_spectrum)
  select type (data_spectrum)
  type is (sf_test_spectrum_data_t)
     call data_spectrum%init (model, pdg_in, with_radiation=.true.)
  end select

  write (u, "(A)")  "* Set up chain with beams only"
  write (u, "(A)")

  call sf_chain%init (beam_data)

  call sf_chain_instance%init (sf_chain, n_channel = 1)

  call sf_chain_instance%link_interactions ()
  sf_chain_instance%status = SF_DONE_CONNECTIONS
```

```
call sf_chain_instance%compute_kinematics (1, [real(default) ::])

call write_separator (u, 2)
call sf_chain%write (u)
call write_separator (u, 2)
call sf_chain_instance%write (u)
call write_separator (u, 2)

call sf_chain_instance%get_out_momenta (p)

write (u, "(A)")
write (u, "(A)")  "* Outgoing momenta:"

do j = 1, 2
   write (u, "(A)")
   call vector4_write (p(j), u)
end do

call sf_chain_instance%final ()
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")  "* Set up chain with structure function"
write (u, "(A)")

allocate (sf_config (1))
call sf_config(1)%init ([1], data_strfun)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 1)

call sf_channel(1)%init (1)
call sf_channel(1)%activate_mapping ([1])
call sf_chain_instance%set_channel (1, sf_channel(1))

call sf_chain_instance%link_interactions ()
sf_chain_instance%status = SF_DONE_CONNECTIONS
call sf_chain_instance%compute_kinematics (1, [0.8_default])

call write_separator (u, 2)
call sf_chain%write (u)
call write_separator (u, 2)
call sf_chain_instance%write (u)
call write_separator (u, 2)

call sf_chain_instance%get_out_momenta (p)

write (u, "(A)")
write (u, "(A)")  "* Outgoing momenta:"

do j = 1, 2
   write (u, "(A)")
   call vector4_write (p(j), u)
end do
```

```
  call sf_chain_instance%final ()
  call sf_chain%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Set up chain with spectrum and structure function"
  write (u, "(A)")

  deallocate (sf_config)
  allocate (sf_config (2))
  call sf_config(1)%init ([1,2], data_spectrum)
  call sf_config(2)%init ([2], data_strfun)
  call sf_chain%init (beam_data, sf_config)

  call sf_chain_instance%init (sf_chain, n_channel = 1)

  call sf_channel(2)%init (2)
  call sf_channel(2)%activate_mapping ([2])
  call sf_chain_instance%set_channel (1, sf_channel(2))

  call sf_chain_instance%link_interactions ()
  sf_chain_instance%status = SF_DONE_CONNECTIONS
  call sf_chain_instance%compute_kinematics &
       (1, [0.5_default, 0.6_default, 0.8_default])

  call write_separator (u, 2)
  call sf_chain%write (u)
  call write_separator (u, 2)
  call sf_chain_instance%write (u)
  call write_separator (u, 2)

  call sf_chain_instance%get_out_momenta (p)

  write (u, "(A)")
  write (u, "(A)")  "* Outgoing momenta:"

  do j = 1, 2
     write (u, "(A)")
     call vector4_write (p(j), u)
  end do

  call sf_chain_instance%final ()
  call sf_chain%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_base_9"

end subroutine sf_base_9
```

Structure function chain mappings

Set up a structure function chain instance with a pair of single-particle structure functions. We test different global mappings for this setup.

Again, we skip evaluators.

⟨*SF base: execute tests*⟩+≡

```
call test (sf_base_10, "sf_base_10", &
     "structure function chain mapping", &
     u, results)
```

⟨*SF base: test declarations*⟩+≡

```
public :: sf_base_10
```

⟨*SF base: tests*⟩+≡

```
subroutine sf_base_10 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t) :: flv
  type(pdg_array_t) :: pdg_in
  type(beam_data_t), target :: beam_data
  class(sf_data_t), allocatable, target :: data_strfun
  type(sf_config_t), dimension(:), allocatable, target :: sf_config
  type(sf_chain_t), target :: sf_chain
  type(sf_chain_instance_t), target :: sf_chain_instance
  type(sf_channel_t), dimension(2) :: sf_channel
  real(default), dimension(2) :: x_saved

  write (u, "(A)")  "* Test output: sf_base_10"
  write (u, "(A)")  "*   Purpose: set up a structure-function chain"
  write (u, "(A)")  "*            and check mappings"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call model%init_test ()
  call flv%init (25, model)
  pdg_in = 25

  call reset_interaction_counter ()

  call beam_data%init_sqrts (1000._default, [flv, flv])

  allocate (sf_test_data_t :: data_strfun)
  select type (data_strfun)
  type is (sf_test_data_t)
     call data_strfun%init (model, pdg_in)
  end select

  write (u, "(A)")  "* Set up chain with structure function pair &
       &and standard mapping"
  write (u, "(A)")

  allocate (sf_config (2))
  call sf_config(1)%init ([1], data_strfun)
```

```
  call sf_config(2)%init ([2], data_strfun)
  call sf_chain%init (beam_data, sf_config)

  call sf_chain_instance%init (sf_chain, n_channel = 1)

  call sf_channel(1)%init (2)
  call sf_channel(1)%set_s_mapping ([1,2])
  call sf_chain_instance%set_channel (1, sf_channel(1))

  call sf_chain_instance%link_interactions ()
  sf_chain_instance%status = SF_DONE_CONNECTIONS
  call sf_chain_instance%compute_kinematics (1, [0.8_default, 0.6_default])

  call write_separator (u, 2)
  call sf_chain_instance%write (u)
  call write_separator (u, 2)

  write (u, "(A)")
  write (u, "(A)")  "* Invert the kinematics calculation"
  write (u, "(A)")

  x_saved = sf_chain_instance%x

  call sf_chain_instance%init (sf_chain, n_channel = 1)

  call sf_channel(2)%init (2)
  call sf_channel(2)%set_s_mapping ([1, 2])
  call sf_chain_instance%set_channel (1, sf_channel(2))

  call sf_chain_instance%link_interactions ()
  sf_chain_instance%status = SF_DONE_CONNECTIONS
  call sf_chain_instance%inverse_kinematics (x_saved)

  call write_separator (u, 2)
  call sf_chain_instance%write (u)
  call write_separator (u, 2)


  call sf_chain_instance%final ()
  call sf_chain%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_base_10"

end subroutine sf_base_10
```

Structure function chain evaluation

Here, we test the complete workflow for structure-function chains. First, we create the template chain, then initialize an instance. We set up links, mask, and evaluators. Finally, we set kinematics and evaluate the matrix elements and their products.

⟨*SF base: execute tests*⟩+≡
```
call test (sf_base_11, "sf_base_11", &
     "structure function chain evaluation", &
     u, results)
```

⟨*SF base: test declarations*⟩+≡
```
public :: sf_base_11
```

⟨*SF base: tests*⟩+≡
```
subroutine sf_base_11 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t) :: flv
  type(pdg_array_t) :: pdg_in
  type(beam_data_t), target :: beam_data
  class(sf_data_t), allocatable, target :: data_strfun
  class(sf_data_t), allocatable, target :: data_spectrum
  type(sf_config_t), dimension(:), allocatable, target :: sf_config
  type(sf_chain_t), target :: sf_chain
  type(sf_chain_instance_t), target :: sf_chain_instance
  type(sf_channel_t), dimension(2) :: sf_channel
  type(particle_set_t) :: pset
  type(interaction_t), pointer :: int
  logical :: ok

  write (u, "(A)")  "* Test output: sf_base_11"
  write (u, "(A)")  "*   Purpose: set up a structure-function chain"
  write (u, "(A)")  "*            create an instance and evaluate"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call model%init_test ()
  call flv%init (25, model)
  pdg_in = 25

  call reset_interaction_counter ()

  call beam_data%init_sqrts (1000._default, [flv, flv])

  allocate (sf_test_data_t :: data_strfun)
  select type (data_strfun)
  type is (sf_test_data_t)
     call data_strfun%init (model, pdg_in)
  end select

  allocate (sf_test_spectrum_data_t :: data_spectrum)
  select type (data_spectrum)
```

```
type is (sf_test_spectrum_data_t)
   call data_spectrum%init (model, pdg_in, with_radiation=.true.)
end select

write (u, "(A)")  "* Set up chain with beams only"
write (u, "(A)")

call sf_chain%init (beam_data)

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

call sf_chain_instance%compute_kinematics (1, [real(default) ::])
call sf_chain_instance%evaluate (scale=0._default)

call write_separator (u, 2)
call sf_chain_instance%write (u)
call write_separator (u, 2)

int => sf_chain_instance%get_out_int_ptr ()
call pset%init (ok, int, int, FM_IGNORE_HELICITY, &
     [0._default, 0._default], .false., .true.)
call sf_chain_instance%final ()

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

call write_separator (u)
call pset%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Recover chain:"
write (u, "(A)")

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

int => sf_chain_instance%get_out_int_ptr ()
call pset%fill_interaction (int, 2, check_match=.false.)

call sf_chain_instance%recover_kinematics (1)
call sf_chain_instance%evaluate (scale=0._default)

call write_separator (u, 2)
call sf_chain_instance%write (u)
call write_separator (u, 2)

call pset%final ()
```

```
call sf_chain_instance%final ()
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")
write (u, "(A)")
write (u, "(A)")  "* Set up chain with structure function"
write (u, "(A)")

allocate (sf_config (1))
call sf_config(1)%init ([1], data_strfun)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_channel(1)%init (1)
call sf_channel(1)%activate_mapping ([1])
call sf_chain_instance%set_channel (1, sf_channel(1))
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

call sf_chain_instance%compute_kinematics (1, [0.8_default])
call sf_chain_instance%evaluate (scale=0._default)

call write_separator (u, 2)
call sf_chain_instance%write (u)
call write_separator (u, 2)

int => sf_chain_instance%get_out_int_ptr ()
call pset%init (ok, int, int, FM_IGNORE_HELICITY, &
     [0._default, 0._default], .false., .true.)
call sf_chain_instance%final ()

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

call write_separator (u)
call pset%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Recover chain:"
write (u, "(A)")

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_channel(1)%init (1)
call sf_channel(1)%activate_mapping ([1])
call sf_chain_instance%set_channel (1, sf_channel(1))
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

int => sf_chain_instance%get_out_int_ptr ()
```

```
call pset%fill_interaction (int, 2, check_match=.false.)

call sf_chain_instance%recover_kinematics (1)
call sf_chain_instance%evaluate (scale=0._default)

call write_separator (u, 2)
call sf_chain_instance%write (u)
call write_separator (u, 2)

call pset%final ()
call sf_chain_instance%final ()
call sf_chain%final ()

write (u, "(A)")
write (u, "(A)")
write (u, "(A)")
write (u, "(A)")  "* Set up chain with spectrum and structure function"
write (u, "(A)")

deallocate (sf_config)
allocate (sf_config (2))
call sf_config(1)%init ([1,2], data_spectrum)
call sf_config(2)%init ([2], data_strfun)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_channel(2)%init (2)
call sf_channel(2)%activate_mapping ([2])
call sf_chain_instance%set_channel (1, sf_channel(2))
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

call sf_chain_instance%compute_kinematics &
     (1, [0.5_default, 0.6_default, 0.8_default])
call sf_chain_instance%evaluate (scale=0._default)

call write_separator (u, 2)
call sf_chain_instance%write (u)
call write_separator (u, 2)

int => sf_chain_instance%get_out_int_ptr ()
call pset%init (ok, int, int, FM_IGNORE_HELICITY, &
     [0._default, 0._default], .false., .true.)
call sf_chain_instance%final ()

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

call write_separator (u)
call pset%write (u)
call write_separator (u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Recover chain:"
    write (u, "(A)")

    call sf_chain_instance%init (sf_chain, n_channel = 1)
    call sf_channel(2)%init (2)
    call sf_channel(2)%activate_mapping ([2])
    call sf_chain_instance%set_channel (1, sf_channel(2))
    call sf_chain_instance%link_interactions ()
    call sf_chain_instance%exchange_mask ()
    call sf_chain_instance%init_evaluators ()

    int => sf_chain_instance%get_out_int_ptr ()
    call pset%fill_interaction (int, 2, check_match=.false.)

    call sf_chain_instance%recover_kinematics (1)
    call sf_chain_instance%evaluate (scale=0._default)

    call write_separator (u, 2)
    call sf_chain_instance%write (u)
    call write_separator (u, 2)

    call pset%final ()
    call sf_chain_instance%final ()
    call sf_chain%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_11"

  end subroutine sf_base_11
```

Multichannel case

We set up a structure-function chain as before, but with three different parameterizations. The first instance is without mappings, the second one with single-particle mappings, and the third one with two-particle mappings.

⟨*SF base: execute tests*⟩+≡

```
  call test (sf_base_12, "sf_base_12", &
       "multi-channel structure function chain", &
       u, results)
```

⟨*SF base: test declarations*⟩+≡

```
  public :: sf_base_12
```

⟨*SF base: tests*⟩+≡

```
  subroutine sf_base_12 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
```

```
type(flavor_t) :: flv
type(pdg_array_t) :: pdg_in
type(beam_data_t), target :: beam_data
class(sf_data_t), allocatable, target :: data
type(sf_config_t), dimension(:), allocatable, target :: sf_config
type(sf_chain_t), target :: sf_chain
type(sf_chain_instance_t), target :: sf_chain_instance
real(default), dimension(2) :: x_saved
real(default), dimension(2,3) :: p_saved
type(sf_channel_t), dimension(:), allocatable :: sf_channel

write (u, "(A)")  "* Test output: sf_base_12"
write (u, "(A)")  "*   Purpose: set up and evaluate a multi-channel &
     &structure-function chain"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call model%init_test ()
call flv%init (25, model)
pdg_in = 25

call reset_interaction_counter ()

call beam_data%init_sqrts (1000._default, [flv, flv])

allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
   call data%init (model, pdg_in)
end select

write (u, "(A)")  "* Set up chain with structure function pair &
     &and three different mappings"
write (u, "(A)")

allocate (sf_config (2))
call sf_config(1)%init ([1], data)
call sf_config(2)%init ([2], data)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 3)

call allocate_sf_channels (sf_channel, n_channel = 3, n_strfun = 2)

! channel 1: no mapping
call sf_chain_instance%set_channel (1, sf_channel(1))

! channel 2: single-particle mappings
call sf_channel(2)%activate_mapping ([1,2])
! call sf_chain_instance%activate_mapping (2, [1,2])
call sf_chain_instance%set_channel (2, sf_channel(2))
```

```
! channel 3: two-particle mapping
call sf_channel(3)%set_s_mapping ([1,2])
! call sf_chain_instance%set_s_mapping (3, [1, 2])
call sf_chain_instance%set_channel (3, sf_channel(3))

call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

write (u, "(A)")  "* Compute kinematics in channel 1 and evaluate"
write (u, "(A)")

call sf_chain_instance%compute_kinematics (1, [0.8_default, 0.6_default])
call sf_chain_instance%evaluate (scale=0._default)

call write_separator (u, 2)
call sf_chain_instance%write (u)
call write_separator (u, 2)

write (u, "(A)")
write (u, "(A)")  "* Invert the kinematics calculation"
write (u, "(A)")

x_saved = sf_chain_instance%x

call sf_chain_instance%inverse_kinematics (x_saved)
call sf_chain_instance%evaluate (scale=0._default)

call write_separator (u, 2)
call sf_chain_instance%write (u)
call write_separator (u, 2)

write (u, "(A)")
write (u, "(A)")  "* Compute kinematics in channel 2 and evaluate"
write (u, "(A)")

p_saved = sf_chain_instance%p

call sf_chain_instance%compute_kinematics (2, p_saved(:,2))
call sf_chain_instance%evaluate (scale=0._default)

call write_separator (u, 2)
call sf_chain_instance%write (u)
call write_separator (u, 2)

write (u, "(A)")
write (u, "(A)")  "* Compute kinematics in channel 3 and evaluate"
write (u, "(A)")

call sf_chain_instance%compute_kinematics (3, p_saved(:,3))
call sf_chain_instance%evaluate (scale=0._default)

call write_separator (u, 2)
call sf_chain_instance%write (u)
```

```
    call write_separator (u, 2)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_chain_instance%final ()
    call sf_chain%final ()

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_12"

  end subroutine sf_base_12
```

Generated spectrum

Construct and evaluate a structure function object for a pair spectrum which is evaluated as a beam-event generator.

⟨*SF base: execute tests*⟩+≡

```
  call test (sf_base_13, "sf_base_13", &
       "pair spectrum generator", &
       u, results)
```

⟨*SF base: test declarations*⟩+≡

```
  public :: sf_base_13
```

⟨*SF base: tests*⟩+≡

```
  subroutine sf_base_13 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t) :: flv
    type(pdg_array_t) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t), dimension(2) :: k
    type(vector4_t), dimension(2) :: q
    real(default) :: E
    real(default), dimension(:), allocatable :: r, rb, x
    real(default) :: f, x_free

    write (u, "(A)")  "* Test output: sf_base_13"
    write (u, "(A)")  "*   Purpose: initialize and fill &
         &a pair generator object"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize configuration data"
    write (u, "(A)")

    call model%init_test ()
    call flv%init (25, model)
    pdg_in = 25
```

```
call reset_interaction_counter ()

allocate (sf_test_generator_data_t :: data)
select type (data)
type is (sf_test_generator_data_t)
   call data%init (model, pdg_in)
end select

write (u, "(A)")  "* Initialize generator object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

write (u, "(A)")  "* Generate free r values"
write (u, "(A)")

x_free = 1
call sf_int%generate_free (r, rb, x_free)

write (u, "(A)")  "* Initialize incoming momenta with sqrts=1000"

E = 500
k(1) = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
k(2) = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call sf_int%seed_kinematics (k)

write (u, "(A)")
write (u, "(A)")  "* Complete kinematics"
write (u, "(A)")

call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f
write (u, "(A,9(1x,F10.7))")  "xf=", x_free

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = sf_int%get_momenta (outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call reset_interaction_counter ()
call data%allocate_sf_int (sf_int)
call sf_int%init (data)
```

```
    call sf_int%seed_kinematics (k)
    call sf_int%set_momenta (q, outgoing=.true.)
    x_free = 1
    call sf_int%recover_x (x, x_free)
    write (u, "(A,9(1x,F10.7))")  "x =", x
    write (u, "(A,9(1x,F10.7))")  "xf=", x_free

    write (u, "(A)")
    write (u, "(A)")  "* Compute inverse kinematics &
         &and evaluate"
    write (u, "(A)")

    call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
    call sf_int%apply (scale=0._default)
    call sf_int%write (u)

    write (u, "(A)")
    write (u, "(A,9(1x,F10.7))")  "r =", r
    write (u, "(A,9(1x,F10.7))")  "f =", f

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_13"

  end subroutine sf_base_13
```

Structure function chain evaluation

Here, we test the complete workflow for a structure-function chain with generator. First, we create the template chain, then initialize an instance. We set up links, mask, and evaluators. Finally, we set kinematics and evaluate the matrix elements and their products.

⟨*SF base: execute tests*⟩+≡

```
  call test (sf_base_14, "sf_base_14", &
       "structure function generator evaluation", &
       u, results)
```

⟨*SF base: test declarations*⟩+≡

```
  public :: sf_base_14
```

⟨*SF base: tests*⟩+≡

```
  subroutine sf_base_14 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t) :: flv
    type(pdg_array_t) :: pdg_in
    type(beam_data_t), target :: beam_data
```

```
class(sf_data_t), allocatable, target :: data_strfun
class(sf_data_t), allocatable, target :: data_generator
type(sf_config_t), dimension(:), allocatable, target :: sf_config
real(default), dimension(:), allocatable :: p_in
type(sf_chain_t), target :: sf_chain
type(sf_chain_instance_t), target :: sf_chain_instance

write (u, "(A)")  "* Test output: sf_base_14"
write (u, "(A)")  "*   Purpose: set up a structure-function chain"
write (u, "(A)")  "*            create an instance and evaluate"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call model%init_test ()
call flv%init (25, model)
pdg_in = 25

call reset_interaction_counter ()

call beam_data%init_sqrts (1000._default, [flv, flv])

allocate (sf_test_data_t :: data_strfun)
select type (data_strfun)
type is (sf_test_data_t)
   call data_strfun%init (model, pdg_in)
end select

allocate (sf_test_generator_data_t :: data_generator)
select type (data_generator)
type is (sf_test_generator_data_t)
   call data_generator%init (model, pdg_in)
end select

write (u, "(A)")  "* Set up chain with generator and structure function"
write (u, "(A)")

allocate (sf_config (2))
call sf_config(1)%init ([1,2], data_generator)
call sf_config(2)%init ([2], data_strfun)
call sf_chain%init (beam_data, sf_config)

call sf_chain_instance%init (sf_chain, n_channel = 1)
call sf_chain_instance%link_interactions ()
call sf_chain_instance%exchange_mask ()
call sf_chain_instance%init_evaluators ()

write (u, "(A)")  "* Inject integration parameter"
write (u, "(A)")

allocate (p_in (sf_chain%get_n_bound ()), source = 0.9_default)
write (u, "(A,9(1x,F10.7))")  "p_in =", p_in
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Evaluate"
    write (u, "(A)")

    call sf_chain_instance%compute_kinematics (1, p_in)
    call sf_chain_instance%evaluate (scale=0._default)

    call sf_chain_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Extract integration parameter"
    write (u, "(A)")

    call sf_chain_instance%get_mcpar (1, p_in)
    write (u, "(A,9(1x,F10.7))")  "p_in =", p_in

    call sf_chain_instance%final ()
    call sf_chain%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_base_14"

  end subroutine sf_base_14
```

15.6 Photon radiation: ISR

⟨sf_isr.f90⟩≡
⟨*File header*⟩

```
module sf_isr

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use constants, only: pi
  use format_defs, only: FMT_15, FMT_19
  use numeric_utils
  use diagnostics
  use physics_defs, only: PHOTON
  use lorentz
  use sm_physics, only: Li2
  use pdg_arrays
  use model_data
  use flavors
  use colors
  use quantum_numbers
  use polarizations
```

```
  use sf_aux
  use sf_mappings
  use sf_base
```

⟨*Standard module head*⟩

⟨*SF isr: public*⟩

⟨*SF isr: parameters*⟩

⟨*SF isr: types*⟩

```
contains
```

⟨*SF isr: procedures*⟩

```
end module sf_isr
```

15.6.1 Physics

The ISR structure function is in the most crude approximation (LLA without α corrections, i.e. ϵ^0)

$$f_0(x) = \epsilon(1-x)^{-1+\epsilon} \qquad \text{with} \qquad \epsilon = \frac{\alpha}{\pi} q_e^2 \ln \frac{s}{m^2}, \tag{15.27}$$

where m is the mass of the incoming (and outgoing) particle, which is initially assumed on-shell.

Here, the form of ϵ results from the kinematical bounds for the momentum squared of the outgoing particle, which in the limit $m^2 \ll s$ are given by

$$t_0 = -2\bar{x}E(E+p) + m^2 \approx -\bar{x}s, \tag{15.28}$$

$$t_1 = -2\bar{x}E(E-p) + m^2 \approx xm^2, \tag{15.29}$$

so the integration over the propagator $1/(t-m^2)$ yields

$$\ln \frac{t_0 - m^2}{t_1 - m^2} = \ln \frac{s}{m^2}. \tag{15.30}$$

In $f_0(x)$, there is an integrable singularity at $x = 1$ which does not spoil the integration, but would lead to an unbounded $f_{\max}$. Therefore, we map this singularity like

$$x = 1 - (1-x')^{1/\epsilon} \tag{15.31}$$

such that

$$\int dx\, f_0(x) = \int dx' \tag{15.32}$$

The structure function has three parameters: α, m_{in} of the incoming particle and s, the hard scale. Internally, we store the exponent ϵ which is the relevant parameter. (In conventional notation, $\epsilon = \beta/2$.) As defaults, we take the actual values of α (which is probably $\alpha(s)$), the actual mass m_{in} and the squared total c.m. energy s.

Including ϵ, ϵ^2, and ϵ^3 corrections, the successive approximation of the ISR structure function read

$$f_0(x) = \epsilon(1-x)^{-1+\epsilon} \tag{15.33}$$

$$f_1(x) = g_1(\epsilon)\, f_0(x) - \frac{\epsilon}{2}(1+x) \tag{15.34}$$

$$\begin{aligned} f_2(x) = {} & g_2(\epsilon)\, f_0(x) - \frac{\epsilon}{2}(1+x) \\ & - \frac{\epsilon^2}{8}\left(\frac{1+3x^2}{1-x}\ln x + 4(1+x)\ln(1-x) + 5 + x\right) \end{aligned} \tag{15.35}$$

$$\begin{aligned} f_3(x) = {} & g_3(\epsilon)\, f_0(x) - \frac{\epsilon}{2}(1+x) \\ & - \frac{\epsilon^2}{8}\left(\frac{1+3x^2}{1-x}\ln x + 4(1+x)\ln(1-x) + 5 + x\right) \\ & - \frac{\epsilon^3}{48}\Bigg((1+x)\left[6\,\mathrm{Li}_2(x) + 12\ln^2(1-x) - 3\pi^2\right] + 6(x+5)\ln(1-x) \\ & \quad + \frac{1}{1-x}\Bigg[\frac{3}{2}(1+8x+3x^2)\ln x + 12(1+x^2)\ln x\ln(1-x) \\ & \qquad - \frac{1}{2}(1+7x^2)\ln^2 x + \frac{1}{4}(39-24x-15x^2)\Bigg]\Bigg) \end{aligned} \tag{15.36}$$

where the successive approximations to the prefactor of the leading singularity

$$g(\epsilon) = \frac{\exp\left(\epsilon(-\gamma_E + \frac{3}{4})\right)}{\Gamma(1+\epsilon)}, \tag{15.37}$$

are given by

$$g_0(\epsilon) = 1 \tag{15.38}$$

$$g_1(\epsilon) = 1 + \frac{3}{4}\epsilon \tag{15.39}$$

$$g_2(\epsilon) = 1 + \frac{3}{4}\epsilon + \frac{27-8\pi^2}{96}\epsilon^2 \tag{15.40}$$

$$g_3(\epsilon) = 1 + \frac{3}{4}\epsilon + \frac{27-8\pi^2}{96}\epsilon^2 + \frac{27-24\pi^2+128\zeta(3)}{384}\epsilon^3, \tag{15.41}$$

where, numerically

$$\zeta(3) = 1.202056903159594285399738161 51\ldots \tag{15.42}$$

Although one could calculate the function $g(\epsilon)$ exactly, truncating its Taylor expansion ensures the exact normalization of the truncated structure function at each given order:

$$\int_0^1 dx\, f_i(x) = 1 \qquad \text{for all } i. \tag{15.43}$$

Effectively, the $O(\epsilon)$ correction reduces the low-x tail of the structure function by 50% while increasing the coefficient of the singularity by $O(\epsilon)$. Relative to this, the $O(\epsilon^2)$ correction slightly enhances $x > \frac{1}{2}$ compared to $x < \frac{1}{2}$. At

$x = 0$, $f_2(x)$ introduces a logarithmic singularity which should be cut off at $x_0 = O(e^{-1/\epsilon})$: for lower x the perturbative series breaks down. The f_3 correction is slightly positive for low x values and negative near $x = 1$, where the Li_2 piece slightly softens the singularity at $x = 1$.

Instead of the definition for ϵ given above, it is customary to include a universal nonlogarithmic piece:

$$\epsilon = \frac{\alpha}{\pi} q_e^2 \left(\ln \tfrac{s}{m^2} - 1\right) \tag{15.44}$$

15.6.2 Implementation

In the concrete implementation, the zeroth order mapping (15.31) is implemented, and the Jacobian is equal to $f_i(x)/f_0(x)$. This can be written as

$$\frac{f_0(x)}{f_0(x)} = 1 \tag{15.45}$$

$$\frac{f_1(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon - \frac{1 - x^2}{2(1 - x')} \tag{15.46}$$

$$\begin{aligned}\frac{f_2(x)}{f_0(x)} &= 1 + \frac{3}{4}\epsilon + \frac{27 - 8\pi^2}{96}\epsilon^2 - \frac{1 - x^2}{2(1 - x')} \\ &\quad - \frac{(1 + 3x^2)\ln x + (1 - x)\left(4(1 + x)\ln(1 - x) + 5 + x\right)}{8(1 - x')}\epsilon\end{aligned} \tag{15.47}$$

For $x = 1$ (i.e., numerically indistinguishable from 1), this reduces to

$$\frac{f_0(x)}{f_0(x)} = 1 \tag{15.48}$$

$$\frac{f_1(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon \tag{15.49}$$

$$\frac{f_2(x)}{f_0(x)} = 1 + \frac{3}{4}\epsilon + \frac{27 - 8\pi^2}{96}\epsilon^2 \tag{15.50}$$

The last line in (15.47) is zero for

$$x_{\min} = 0.0071405332973459283954987977201 9 \tag{15.51}$$

(Mathematica result), independent of ϵ. For x values less than this we ignore this correction because of the logarithmic singularity which should in principle be resummed.

15.6.3 The ISR data block

⟨*SF isr: public*⟩≡
```
public :: isr_data_t
```

⟨*SF isr: types*⟩≡
```
type, extends (sf_data_t) :: isr_data_t
   private
   class(model_data_t), pointer :: model => null ()
   type(flavor_t), dimension(:), allocatable :: flv_in
   real(default) :: alpha = 0
```

```
    real(default) :: q_max = 0
    real(default) :: real_mass = 0
    real(default) :: mass = 0
    real(default) :: eps = 0
    real(default) :: log = 0
    logical :: recoil = .false.
    logical :: keep_energy = .true.
    integer :: order = 3
    integer :: error = NONE
  contains
  ⟨SF isr: isr data: TBP⟩
end type isr_data_t
```

Error codes

⟨*SF isr: parameters*⟩≡

```
integer, parameter :: NONE = 0
integer, parameter :: ZERO_MASS = 1
integer, parameter :: Q_MAX_TOO_SMALL = 2
integer, parameter :: EPS_TOO_LARGE = 3
integer, parameter :: INVALID_ORDER = 4
integer, parameter :: CHARGE_MIX = 5
integer, parameter :: CHARGE_ZERO = 6
integer, parameter :: MASS_MIX = 7
```

Generate flavor-dependent ISR data:

⟨*SF isr: isr data: TBP*⟩≡

```
procedure :: init => isr_data_init
```

⟨*SF isr: procedures*⟩≡

```
subroutine isr_data_init (data, model, pdg_in, alpha, q_max, &
     mass, order, recoil, keep_energy)
  class(isr_data_t), intent(out) :: data
  class(model_data_t), intent(in), target :: model
  type(pdg_array_t), intent(in) :: pdg_in
  real(default), intent(in) :: alpha
  real(default), intent(in) :: q_max
  real(default), intent(in), optional :: mass
  integer, intent(in), optional :: order
  logical, intent(in), optional :: recoil
  logical, intent(in), optional :: keep_energy
  integer :: i, n_flv
  real(default) :: charge
  data%model => model
  n_flv = pdg_array_get_length (pdg_in)
  allocate (data%flv_in (n_flv))
  do i = 1, n_flv
     call data%flv_in(i)%init (pdg_array_get (pdg_in, i), model)
  end do
  data%alpha = alpha
  data%q_max = q_max
  if (present (order)) then
     call data%set_order (order)
  end if
  if (present (recoil)) then
```

```
      data%recoil = recoil
    end if
    if (present (keep_energy)) then
      data%keep_energy = keep_energy
    end if
    data%real_mass = data%flv_in(1)%get_mass ()
    if (present (mass)) then
      if (mass > 0) then
        data%mass = mass
      else
        data%mass = data%real_mass
        if (any (data%flv_in%get_mass () /= data%mass)) then
          data%error = MASS_MIX;  return
        end if
      end if
    else
      data%mass = data%real_mass
      if (any (data%flv_in%get_mass () /= data%mass)) then
        data%error = MASS_MIX;  return
      end if
    end if
    if (vanishes (data%mass)) then
      data%error = ZERO_MASS;  return
    else if (data%mass >= data%q_max) then
      data%error = Q_MAX_TOO_SMALL;  return
    end if
    data%log = log (1 + (data%q_max / data%mass)**2)
    charge = data%flv_in(1)%get_charge ()
    if (any (abs (data%flv_in%get_charge ()) /= abs (charge))) then
      data%error = CHARGE_MIX;  return
    else if (charge == 0) then
      data%error = CHARGE_ZERO;  return
    end if
    data%eps = data%alpha / pi * charge ** 2 &
        * (2 * log (data%q_max / data%mass) - 1)
    if (data%eps > 1) then
      data%error = EPS_TOO_LARGE;  return
    end if
  end subroutine isr_data_init
```

Explicitly set ISR order

⟨*SF isr: isr data: TBP*⟩+≡

```
  procedure :: set_order => isr_data_set_order
```

⟨*SF isr: procedures*⟩+≡

```
  elemental subroutine isr_data_set_order (data, order)
    class(isr_data_t), intent(inout) :: data
    integer, intent(in) :: order
    if (order < 0 .or. order > 3) then
      data%error = INVALID_ORDER
    else
      data%order = order
    end if
  end subroutine isr_data_set_order
```

Handle error conditions. Should always be done after initialization, unless we are sure everything is ok.

⟨*SF isr: isr data: TBP*⟩+≡

```
procedure :: check => isr_data_check
```

⟨*SF isr: procedures*⟩+≡

```
subroutine isr_data_check (data)
  class(isr_data_t), intent(in) :: data
  select case (data%error)
  case (ZERO_MASS)
     call msg_fatal ("ISR: Particle mass is zero")
  case (Q_MAX_TOO_SMALL)
     call msg_fatal ("ISR: Particle mass exceeds Qmax")
  case (EPS_TOO_LARGE)
     call msg_fatal ("ISR: Expansion parameter too large, " // &
          "perturbative expansion breaks down")
  case (INVALID_ORDER)
     call msg_error ("ISR: LLA order invalid (valid values are 0,1,2,3)")
  case (MASS_MIX)
     call msg_fatal ("ISR: Incoming particle masses must be uniform")
  case (CHARGE_MIX)
     call msg_fatal ("ISR: Incoming particle charges must be uniform")
  case (CHARGE_ZERO)
     call msg_fatal ("ISR: Incoming particle must be charged")
  end select
end subroutine isr_data_check
```

Output

⟨*SF isr: isr data: TBP*⟩+≡

```
procedure :: write => isr_data_write
```

⟨*SF isr: procedures*⟩+≡

```
subroutine isr_data_write (data, unit, verbose)
  class(isr_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)") "ISR data:"
  if (allocated (data%flv_in)) then
     write (u, "(3x,A)", advance="no") "  flavor =  "
     do i = 1, size (data%flv_in)
        if (i > 1)  write (u, "(',',1x)", advance="no")
        call data%flv_in(i)%write (u)
     end do
     write (u, *)
     write (u, "(3x,A," // FMT_19 // ")") "  alpha    = ", data%alpha
     write (u, "(3x,A," // FMT_19 // ")") "  q_max    = ", data%q_max
     write (u, "(3x,A," // FMT_19 // ")") "  mass     = ", data%mass
     write (u, "(3x,A," // FMT_19 // ")") "  eps      = ", data%eps
     write (u, "(3x,A," // FMT_19 // ")") "  log      = ", data%log
     write (u, "(3x,A,I2)")      "  order    = ", data%order
     write (u, "(3x,A,L2)")      "  recoil   = ", data%recoil
```

```
      write (u, "(3x,A,L2)")      "  keep en. = ", data%keep_energy
    else
      write (u, "(3x,A)") "[undefined]"
    end if
  end subroutine isr_data_write
```

For ISR, there is the option to generate transverse momentum is generated. Hence, there can be up to three parameters, x, and two angles.

⟨*SF isr: isr data: TBP*⟩+≡

```
  procedure :: get_n_par => isr_data_get_n_par
```

⟨*SF isr: procedures*⟩+≡

```
  function isr_data_get_n_par (data) result (n)
    class(isr_data_t), intent(in) :: data
    integer :: n
    if (data%recoil) then
      n = 3
    else
      n = 1
    end if
  end function isr_data_get_n_par
```

Return the outgoing particles PDG codes. For ISR, these are identical to the incoming particles.

⟨*SF isr: isr data: TBP*⟩+≡

```
  procedure :: get_pdg_out => isr_data_get_pdg_out
```

⟨*SF isr: procedures*⟩+≡

```
  subroutine isr_data_get_pdg_out (data, pdg_out)
    class(isr_data_t), intent(in) :: data
    type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
    pdg_out(1) = data%flv_in%get_pdg ()
  end subroutine isr_data_get_pdg_out
```

Return the `eps` value. We need it for an appropriate mapping of structure-function parameters.

⟨*SF isr: isr data: TBP*⟩+≡

```
  procedure :: get_eps => isr_data_get_eps
```

⟨*SF isr: procedures*⟩+≡

```
  function isr_data_get_eps (data) result (eps)
    class(isr_data_t), intent(in) :: data
    real(default) :: eps
    eps = data%eps
  end function isr_data_get_eps
```

Allocate the interaction record.

⟨*SF isr: isr data: TBP*⟩+≡

```
  procedure :: allocate_sf_int => isr_data_allocate_sf_int
```

⟨*SF isr: procedures*⟩+≡

```
subroutine isr_data_allocate_sf_int (data, sf_int)
  class(isr_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  allocate (isr_t :: sf_int)
end subroutine isr_data_allocate_sf_int
```

15.6.4 The ISR object

The `isr_t` data type is a $1 \to 2$ interaction, i.e., we allow for single-photon emission only (but use the multi-photon resummed radiator function). The particles are ordered as (incoming, photon, outgoing).

There is no need to handle several flavors (and data blocks) in parallel, since ISR is always applied immediately after beam collision. (ISR for partons is accounted for by the PDFs themselves.) Polarization is carried through, i.e., we retain the polarization of the incoming particle and treat the emitted photon as unpolarized. Color is trivially carried through. This implies that particles 1 and 3 should be locked together. For ISR we don't need the q variable.

⟨*SF isr: public*⟩+≡

```
public :: isr_t
```

⟨*SF isr: types*⟩+≡

```
type, extends (sf_int_t) :: isr_t
   private
   type(isr_data_t), pointer :: data => null ()
   real(default) :: x = 0
   real(default) :: xb= 0
 contains
 ⟨SF isr: isr: TBP⟩
end type isr_t
```

Type string: has to be here, but there is no string variable on which ISR depends. Hence, a dummy routine.

⟨*SF isr: isr: TBP*⟩≡

```
procedure :: type_string => isr_type_string
```

⟨*SF isr: procedures*⟩+≡

```
function isr_type_string (object) result (string)
  class(isr_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "ISR: e+ e- ISR spectrum"
  else
     string = "ISR: [undefined]"
  end if
end function isr_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF isr: isr: TBP*⟩+≡

```
procedure :: write => isr_write
```

⟨*SF isr: procedures*⟩+≡

```
subroutine isr_write (object, unit, testflag)
  class(isr_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     if (object%status >= SF_DONE_KINEMATICS) then
        write (u, "(1x,A)")  "SF parameters:"
        write (u, "(3x,A," // FMT_15 // ")")  "x =", object%x
        write (u, "(3x,A," // FMT_15 // ")")  "xb=", object%xb
     end if
     call object%base_write (u, testflag)
  else
     write (u, "(1x,A)")  "ISR data: [undefined]"
  end if
end subroutine isr_write
```

Explicitly set ISR order (for unit test).

⟨*SF isr: isr: TBP*⟩+≡

```
procedure :: set_order => isr_set_order
```

⟨*SF isr: procedures*⟩+≡

```
subroutine isr_set_order (object, order)
  class(isr_t), intent(inout) :: object
  integer, intent(in) :: order
  call object%data%set_order (order)
end subroutine isr_set_order
```

15.6.5 Kinematics

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ were trivial. The ISR structure function allows for a straightforward mapping of the unit interval. So, to leading order, the structure function value is unity, but the x value is transformed. Higher orders affect the function value.

The structure function implementation applies the above mapping to the input (random) number `r` to generate the momentum fraction `x` and the function value `f`. For numerical stability reasons, we also output `xb`, which is $\bar{x} = 1 - x$.

For the ISR structure function, the mapping Jacobian cancels the structure function (to order zero). We apply the cancellation explicitly, therefore both the Jacobian `f` and the zeroth-order value (see the `apply` method) are unity if mapping is turned on. If mapping is turned off, the Jacobian `f` includes the value of the (zeroth-order) structure function, and strongly peaked.

⟨*SF isr: isr: TBP*⟩+≡

```
procedure :: complete_kinematics => isr_complete_kinematics
```

⟨*SF isr: procedures*⟩+≡

```
subroutine isr_complete_kinematics (sf_int, x, f, r, rb, map)
  class(isr_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
```

```
    real(default), intent(out) :: f
    real(default), dimension(:), intent(in) :: r
    real(default), dimension(:), intent(in) :: rb
    logical, intent(in) :: map
    real(default) :: eps
    eps = sf_int%data%eps
    if (map) then
       call map_power_1 (sf_int%xb, f, rb(1), eps)
    else
       sf_int%xb = rb(1)
       if (rb(1) > 0) then
          f = 1
       else
          f = 0
       end if
    end if
    sf_int%x = 1 - sf_int%xb
    x(1) = sf_int%x
    if (size (x) == 3)  x(2:3) = r(2:3)
    call sf_int%split_momentum (x, sf_int%xb)
    select case (sf_int%status)
    case (SF_FAILED_KINEMATICS)
       sf_int%x = 0
       sf_int%xb= 0
       f = 0
    end select
  end subroutine isr_complete_kinematics
```

Overriding the default method: we compute the `x` array from the momentum configuration. In the specific case of ISR, we also set the internally stored x and $\bar{x}$ values, so they can be used in the following routine.

Note: the extraction of $\bar{x}$ is not numerically safe, but it can't be as long as the base `recover_x` isn't.

⟨*SF isr: isr: TBP*⟩+≡

```
  procedure :: recover_x => sf_isr_recover_x
```

⟨*SF isr: procedures*⟩+≡

```
  subroutine sf_isr_recover_x (sf_int, x, x_free)
    class(isr_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: x
    real(default), intent(inout), optional :: x_free
    call sf_int%base_recover_x (x, x_free)
    sf_int%x  = x(1)
    sf_int%xb = 1 - x(1)
  end subroutine sf_isr_recover_x
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

For extracting x, we rely on the stored $\bar{x}$ value, since the x value in the argument is likely imprecise. This means that either `complete_kinematics` or `recover_x` must be called first, for the current sampling point (but maybe

another channel).

⟨*SF isr: isr: TBP*⟩+≡

```
procedure :: inverse_kinematics => isr_inverse_kinematics
```

⟨*SF isr: procedures*⟩+≡

```
subroutine isr_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
  class(isr_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  real(default), dimension(:), intent(out) :: rb
  logical, intent(in) :: map
  logical, intent(in), optional :: set_momenta
  real(default) :: eps
  logical :: set_mom
  set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
  eps = sf_int%data%eps
  if (map) then
     call map_power_inverse_1 (sf_int%xb, f, rb(1), eps)
  else
     rb(1) = sf_int%xb
     if (rb(1) > 0) then
        f = 1
     else
        f = 0
     end if
  end if
  r(1) = 1 - rb(1)
  if (size(r) == 3) then
     r(2:3) = x(2:3)
     rb(2:3)= 1 - r(2:3)
  end if
  if (set_mom) then
     call sf_int%split_momentum (x, sf_int%xb)
     select case (sf_int%status)
     case (SF_FAILED_KINEMATICS)
        r = 0
        rb= 0
        f = 0
     end select
  end if
end subroutine isr_inverse_kinematics
```

⟨*SF isr: isr: TBP*⟩+≡

```
procedure :: init => isr_init
```

⟨*SF isr: procedures*⟩+≡

```
subroutine isr_init (sf_int, data)
  class(isr_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  type(quantum_numbers_mask_t), dimension(3) :: mask
  integer, dimension(3) :: hel_lock
  type(polarization_t), target :: pol
  type(quantum_numbers_t), dimension(1) :: qn_fc
```

```
  type(flavor_t) :: flv_photon
  type(color_t) :: col_photon
  type(quantum_numbers_t) :: qn_hel, qn_photon, qn
  type(polarization_iterator_t) :: it_hel
  real(default) :: m2
  integer :: i
  mask = quantum_numbers_mask (.false., .false., &
       mask_h = [.false., .true., .false.])
  hel_lock = [3, 0, 1]
  select type (data)
  type is (isr_data_t)
     m2 = data%mass**2
     call sf_int%base_init (mask, [m2], [0._default], [m2], &
          hel_lock = hel_lock)
     sf_int%data => data
     call flv_photon%init (PHOTON, data%model)
     call col_photon%init ()
     call qn_photon%init (flv_photon, col_photon)
     call qn_photon%tag_radiated ()
     do i = 1, size (data%flv_in)
        call pol%init_generic (data%flv_in(i))
        call qn_fc(1)%init (&
             flv = data%flv_in(i), &
             col = color_from_flavor (data%flv_in(i), 1))
        call it_hel%init (pol)
        do while (it_hel%is_valid ())
           qn_hel = it_hel%get_quantum_numbers ()
           qn = qn_hel .merge. qn_fc(1)
           call sf_int%add_state ([qn, qn_photon, qn])
           call it_hel%advance ()
        end do
        ! call pol%final ()  !!! Obsolete
     end do
     call sf_int%freeze ()
     if (data%keep_energy) then
        sf_int%on_shell_mode = KEEP_ENERGY
     else
        sf_int%on_shell_mode = KEEP_MOMENTUM
     end if
     call sf_int%set_incoming ([1])
     call sf_int%set_radiated ([2])
     call sf_int%set_outgoing ([3])
     sf_int%status = SF_INITIAL
  end select
end subroutine isr_init
```

15.6.6 ISR application

For ISR, we could in principle compute kinematics and function value in a single step. In order to be able to reweight matrix elements including structure functions we split kinematics and structure function calculation. The structure function works on a single beam, assuming that the input momentum has been

set.

For the structure-function evaluation, we rely on the fact that the power mapping, which we apply in the kinematics method (if the `map` flag is set), has a Jacobian which is just the inverse lowest-order structure function. With mapping active, the two should cancel exactly.

After splitting momenta, we set the outgoing momenta on-shell. We choose to conserve momentum, so energy conservation may be violated.

⟨*SF isr: isr: TBP*⟩+≡
```
procedure :: apply => isr_apply
```

⟨*SF isr: procedures*⟩+≡
```
subroutine isr_apply (sf_int, scale)
  class(isr_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  real(default) :: f, finv, x, xb, eps, rb
  real(default) :: log_x, log_xb, x_2
  real(default), parameter :: &
       & xmin = 0.007140533297345928395498797720 19_default
  real(default), parameter :: &
       & zeta3 = 1.2020569031595942853997381615 1_default
  real(default), parameter :: &
       & g1 = 3._default / 4._default, &
       & g2 = (27 - 8*pi**2) / 96._default, &
       & g3 = (27 - 24*pi**2 + 128*zeta3) / 384._default
  associate (data => sf_int%data)
    eps = sf_int%data%eps
    x = sf_int%x
    xb = sf_int%xb
    call map_power_inverse_1 (xb, finv, rb, eps)
    if (finv > 0) then
       f = 1 / finv
    else
       f = 0
    end if
    if (f > 0 .and. data%order > 0) then
       f = f * (1 + g1 * eps)
       x_2 = x*x
       if (rb>0)  f = f * (1 - (1-x_2) / (2 * rb))
       if (data%order > 1) then
          f = f * (1 + g2 * eps**2)
          if (rb>0 .and. xb>0 .and. x>xmin) then
             log_x  = log_prec (x, xb)
             log_xb = log_prec (xb, x)
             f = f * (1 - ((1+3*x_2)*log_x + xb * (4*(1+x)*log_xb + 5 + x)) &
                  / ( 8 * rb) * eps)
          end if
          if (data%order > 2) then
             f = f * (1 + g3 * eps**3)
             if (rb > 0 .and. xb > 0 .and. x > xmin) then
                f = f * (1 - ((1+x) * xb &
                     * (6 * Li2(x) + 12 * log_xb**2 - 3 * pi**2) &
                     + 1.5_default * (1 + 8*x + 3*x_2) * log_x &
                     + 6 * (x+5) * xb * log_xb &
                     + 12 * (1+x_2) * log_x * log_xb &
```

```
                        - (1 + 7*x_2) * log_x**2 / 2 &
                        + (39 - 24*x - 15*x_2) / 4) &
                        / ( 48 * rb) * eps**2)
                end if
            end if
        end if
      end if
    end associate
    call sf_int%set_matrix_element (cmplx (f, kind=default))
    sf_int%status = SF_EVALUATED
  end subroutine isr_apply
```

15.6.7 Unit tests

Test module, followed by the corresponding implementation module.

⟨sf_isr_ut.f90⟩≡

```
⟨File header⟩

module sf_isr_ut
  use unit_tests
  use sf_isr_uti

⟨Standard module head⟩

⟨SF isr: public test⟩

contains

⟨SF isr: test driver⟩

end module sf_isr_ut
```

⟨sf_isr_uti.f90⟩≡

```
⟨File header⟩

module sf_isr_uti

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_defs, only: FMT_12
  use physics_defs, only: ELECTRON
  use lorentz
  use pdg_arrays
  use flavors
  use interactions, only: reset_interaction_counter
  use interactions, only: interaction_pacify_momenta
  use model_data
  use sf_aux, only: KEEP_ENERGY
  use sf_mappings
  use sf_base

  use sf_isr
```

⟨*Standard module head*⟩

⟨*SF isr: test declarations*⟩

```
contains
```

⟨*SF isr: tests*⟩

```
end module sf_isr_uti
```

API: driver for the unit tests below.

⟨*SF isr: public test*⟩≡

```
public :: sf_isr_test
```

⟨*SF isr: test driver*⟩≡

```
subroutine sf_isr_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```
⟨*SF isr: execute tests*⟩
```
end subroutine sf_isr_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF isr: execute tests*⟩≡

```
call test (sf_isr_1, "sf_isr_1", &
     "structure function configuration", &
     u, results)
```

⟨*SF isr: test declarations*⟩≡

```
public :: sf_isr_1
```

⟨*SF isr: tests*⟩≡

```
subroutine sf_isr_1 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(pdg_array_t) :: pdg_in
  type(pdg_array_t), dimension(1) :: pdg_out
  integer, dimension(:), allocatable :: pdg1
  class(sf_data_t), allocatable :: data

  write (u, "(A)")  "* Test output: sf_isr_1"
  write (u, "(A)")  "*   Purpose: initialize and display &
       &test structure function data"
  write (u, "(A)")

  write (u, "(A)")  "* Create empty data object"
  write (u, "(A)")

  call model%init_qed_test ()
  pdg_in = ELECTRON

  allocate (isr_data_t :: data)
  call data%write (u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Initialize"
    write (u, "(A)")

    select type (data)
    type is (isr_data_t)
       call data%init (model, pdg_in, 1./137._default, 10._default, &
            0.000511_default, order = 3, recoil = .false.)
    end select

    call data%write (u)

    write (u, "(A)")

    write (u, "(1x,A)")  "Outgoing particle codes:"
    call data%get_pdg_out (pdg_out)
    pdg1 = pdg_out(1)
    write (u, "(2x,99(1x,I0))")  pdg1

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_isr_1"

  end subroutine sf_isr_1
```

Structure function without mapping

Direct ISR evaluation. This is the use case for a double-beam structure function. The parameter pair is mapped in the calling program.

⟨*SF isr: execute tests*⟩+≡

```
  call test (sf_isr_2, "sf_isr_2", &
       "no ISR mapping", &
       u, results)
```

⟨*SF isr: test declarations*⟩+≡

```
  public :: sf_isr_2
```

⟨*SF isr: tests*⟩+≡

```
  subroutine sf_isr_2 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(pdg_array_t) :: pdg_in
    type(flavor_t) :: flv
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k
    real(default) :: E
    real(default), dimension(:), allocatable :: r, rb, x
    real(default) :: f, f_isr

    write (u, "(A)")  "* Test output: sf_isr_2"
```

```
write (u, "(A)")  "*   Purpose: initialize and fill &
     &test structure function object"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call model%init_qed_test ()
pdg_in = ELECTRON
call flv%init (ELECTRON, model)

call reset_interaction_counter ()

allocate (isr_data_t :: data)
select type (data)
type is (isr_data_t)
   call data%init (model, pdg_in, 1./137._default, 500._default, &
        0.000511_default, order = 3, recoil = .false.)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.9, no ISR mapping, &
     &collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.9_default
rb = 1 - r
write (u, "(A,9(1x," // FMT_12 // "))")  "r =", r
write (u, "(A,9(1x," // FMT_12 // "))")  "rb=", rb

call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

write (u, "(A)")
write (u, "(A,9(1x," // FMT_12 // "))")  "x =", x
write (u, "(A,9(1x," // FMT_12 // "))")  "f =", f
```

```
  write (u, "(A)")
  write (u, "(A)")  "* Invert kinematics"
  write (u, "(A)")

  call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)
  write (u, "(A,9(1x," // FMT_12 // "))")  "r =", r
  write (u, "(A,9(1x," // FMT_12 // "))")  "rb=", rb
  write (u, "(A,9(1x," // FMT_12 // "))")  "f =", f

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate ISR structure function"
  write (u, "(A)")

  call sf_int%apply (scale = 100._default)
  call sf_int%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Structure-function value, default order"
  write (u, "(A)")

  f_isr = sf_int%get_matrix_element (1)

  write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr         =", f_isr
  write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr * f_map =", f_isr * f

  write (u, "(A)")
  write (u, "(A)")  "* Re-evaluate structure function, leading order"
  write (u, "(A)")

  select type (sf_int)
  type is (isr_t)
     call sf_int%set_order (0)
  end select
  call sf_int%apply (scale = 100._default)
  f_isr = sf_int%get_matrix_element (1)

  write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr         =", f_isr
  write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr * f_map =", f_isr * f

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call sf_int%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_isr_2"

end subroutine sf_isr_2
```

Structure function with mapping

Apply the optimal ISR mapping. This is the use case for a single-beam structure function.

⟨*SF isr: execute tests*⟩+≡

```
call test (sf_isr_3, "sf_isr_3", &
     "ISR mapping", &
     u, results)
```

⟨*SF isr: test declarations*⟩+≡

```
public :: sf_isr_3
```

⟨*SF isr: tests*⟩+≡

```
subroutine sf_isr_3 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t) :: flv
  type(pdg_array_t) :: pdg_in
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k
  real(default) :: E
  real(default), dimension(:), allocatable :: r, rb, x
  real(default) :: f, f_isr

  write (u, "(A)")  "* Test output: sf_isr_3"
  write (u, "(A)")  "*   Purpose: initialize and fill &
       &test structure function object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call model%init_qed_test ()
  call flv%init (ELECTRON, model)
  pdg_in = ELECTRON

  call reset_interaction_counter ()

  allocate (isr_data_t :: data)
  select type (data)
  type is (isr_data_t)
     call data%init (model, pdg_in, 1./137._default, 500._default, &
          0.000511_default, order = 3, recoil = .false.)
  end select

  write (u, "(A)")  "* Initialize structure-function object"
  write (u, "(A)")

  call data%allocate_sf_int (sf_int)
  call sf_int%init (data)
  call sf_int%set_beam_index ([1])

  write (u, "(A)")  "* Initialize incoming momentum with E=500"
  write (u, "(A)")
```

```
E = 500
k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.7, with ISR mapping, &
     &collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.7_default
rb = 1 - r
write (u, "(A,9(1x," // FMT_12 // "))")  "r =", r
write (u, "(A,9(1x," // FMT_12 // "))")  "rb=", rb

call sf_int%complete_kinematics (x, f, r, rb, map=.true.)

write (u, "(A)")
write (u, "(A,9(1x," // FMT_12 // "))")  "x =", x
write (u, "(A,9(1x," // FMT_12 // "))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Invert kinematics"
write (u, "(A)")

call sf_int%inverse_kinematics (x, f, r, rb, map=.true.)
write (u, "(A,9(1x," // FMT_12 // "))")  "r =", r
write (u, "(A,9(1x," // FMT_12 // "))")  "rb=", rb
write (u, "(A,9(1x," // FMT_12 // "))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Evaluate ISR structure function"
write (u, "(A)")

call sf_int%apply (scale = 100._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Structure-function value, default order"
write (u, "(A)")

f_isr = sf_int%get_matrix_element (1)

write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr         =", f_isr
write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr * f_map =", f_isr * f

write (u, "(A)")
write (u, "(A)")  "* Re-evaluate structure function, leading order"
write (u, "(A)")
```

```
    select type (sf_int)
    type is (isr_t)
       call sf_int%set_order (0)
    end select
    call sf_int%apply (scale = 100._default)
    f_isr = sf_int%get_matrix_element (1)

    write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr         =", f_isr
    write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr * f_map =", f_isr * f

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_isr_3"

  end subroutine sf_isr_3
```

Non-collinear ISR splitting

Construct and display a structure function object based on the ISR structure function. We blank out numerical fluctuations for 32bit.

⟨*SF isr: execute tests*⟩+≡

```
  call test (sf_isr_4, "sf_isr_4", &
       "ISR non-collinear", &
       u, results)
```

⟨*SF isr: test declarations*⟩+≡

```
  public :: sf_isr_4
```

⟨*SF isr: tests*⟩+≡

```
  subroutine sf_isr_4 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t) :: flv
    type(pdg_array_t) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q
    real(default) :: E
    real(default), dimension(:), allocatable :: r, rb, x
    real(default) :: f, f_isr
    character(len=80) :: buffer
    integer :: u_scratch, iostat

    write (u, "(A)")  "* Test output: sf_isr_4"
    write (u, "(A)")  "*   Purpose: initialize and fill &
         &test structure function object"
```

```
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call model%init_qed_test ()
call flv%init (ELECTRON, model)
pdg_in = ELECTRON

call reset_interaction_counter ()

write (u, "(A)")
write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

allocate (isr_data_t :: data)
select type (data)
type is (isr_data_t)
   call data%init (model, pdg_in, 1./137._default, 500._default, &
        0.000511_default, order = 3, recoil = .true.)
end select

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5/0.5/0.25, with ISR mapping, "
write (u, "(A)")  "          non-coll., keeping energy"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = [0.5_default, 0.5_default, 0.25_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.true.)
call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)

write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
```

```
write (u, "(A)")  "* Recover x and r from momenta"
write (u, "(A)")

q = sf_int%get_momenta (outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%seed_kinematics ([k])
call sf_int%set_momenta (q, outgoing=.true.)
call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.true.)

write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "r =", r

write (u, "(A)")
write (u, "(A)")  "* Evaluate ISR structure function"
write (u, "(A)")

call sf_int%complete_kinematics (x, f, r, rb, map=.true.)
call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)
call sf_int%apply (scale = 10._default)
u_scratch = free_unit ()
open (u_scratch, status="scratch", action = "readwrite")
call sf_int%write (u_scratch, testflag = .true.)
rewind (u_scratch)
do
   read (u_scratch, "(A)", iostat=iostat) buffer
   if (iostat /= 0) exit
   if (buffer(1:25) == " P =   0.000000E+00  9.57") then
      buffer = replace (buffer, 26, "XXXX")
   end if
   if (buffer(1:25) == " P =   0.000000E+00 -9.57") then
      buffer = replace (buffer, 26, "XXXX")
   end if
   write (u, "(A)") buffer
end do
close (u_scratch)

write (u, "(A)")
write (u, "(A)")  "* Structure-function value"
write (u, "(A)")

f_isr = sf_int%get_matrix_element (1)

write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr         =", f_isr
write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr * f_map =", f_isr * f

write (u, "(A)")
write (u, "(A)")  "* Cleanup"
```

```
    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_isr_4"

  end subroutine sf_isr_4
```

Structure function pair with mapping

Apply the ISR mapping for a ISR pair. structure function.

⟨*SF isr: execute tests*⟩+≡

```
  call test (sf_isr_5, "sf_isr_5", &
       "ISR pair mapping", &
       u, results)
```

⟨*SF isr: test declarations*⟩+≡

```
  public :: sf_isr_5
```

⟨*SF isr: tests*⟩+≡

```
  subroutine sf_isr_5 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t) :: flv
    type(pdg_array_t) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    class(sf_mapping_t), allocatable :: mapping
    class(sf_int_t), dimension(:), allocatable :: sf_int
    type(vector4_t), dimension(2) :: k
    real(default) :: E, f_map
    real(default), dimension(:), allocatable :: p, pb, r, rb, x
    real(default), dimension(2) :: f, f_isr
    integer :: i

    write (u, "(A)")  "* Test output: sf_isr_5"
    write (u, "(A)")  "*   Purpose: initialize and fill &
         &test structure function object"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize configuration data"
    write (u, "(A)")

    call model%init_qed_test ()
    call flv%init (ELECTRON, model)
    pdg_in = ELECTRON

    call reset_interaction_counter ()

    allocate (isr_data_t :: data)
    select type (data)
    type is (isr_data_t)
       call data%init (model, pdg_in, 1./137._default, 500._default, &
            0.000511_default, order = 3, recoil = .false.)
```

```
end select

allocate (sf_ip_mapping_t :: mapping)
select type (mapping)
type is (sf_ip_mapping_t)
   select type (data)
   type is (isr_data_t)
      call mapping%init (eps = data%get_eps ())
   end select
   call mapping%set_index (1, 1)
   call mapping%set_index (2, 2)
end select

call mapping%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

allocate (isr_t :: sf_int (2))

do i = 1, 2
   call sf_int(i)%init (data)
   call sf_int(i)%set_beam_index ([i])
end do

write (u, "(A)")  "* Initialize incoming momenta with E=500"
write (u, "(A)")
E = 500
k(1) = vector4_moving (E,   sqrt (E**2 - flv%get_mass ()**2), 3)
k(2) = vector4_moving (E, - sqrt (E**2 - flv%get_mass ()**2), 3)
call pacify (k, 1e-10_default)
do i = 1, 2
   call vector4_write (k(i), u)
   call sf_int(i)%seed_kinematics (k(i:i))
end do

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for p=[0.7,0.4], collinear"
write (u, "(A)")

allocate (p (2 * data%get_n_par ()))
allocate (pb(size (p)))
allocate (r (size (p)))
allocate (rb(size (p)))
allocate (x (size (p)))

p = [0.7_default, 0.4_default]
pb= 1 - p
call mapping%compute (r, rb, f_map, p, pb)

write (u, "(A,9(1x," // FMT_12 // "))")  "p =", p
write (u, "(A,9(1x," // FMT_12 // "))")  "pb=", pb
write (u, "(A,9(1x," // FMT_12 // "))")  "r =", r
```

```
write (u, "(A,9(1x," // FMT_12 // "))")  "rb=", rb
write (u, "(A,9(1x," // FMT_12 // "))")  "fm=", f_map

do i = 1, 2
   call sf_int(i)%complete_kinematics (x(i:i), f(i), r(i:i), rb(i:i), &
        map=.false.)
end do

write (u, "(A)")
write (u, "(A,9(1x," // FMT_12 // "))")  "x =", x
write (u, "(A,9(1x," // FMT_12 // "))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Invert kinematics"
write (u, "(A)")

do i = 1, 2
   call sf_int(i)%inverse_kinematics (x(i:i), f(i), r(i:i), rb(i:i), &
        map=.false.)
end do
call mapping%inverse (r, rb, f_map, p, pb)

write (u, "(A,9(1x," // FMT_12 // "))")  "p =", p
write (u, "(A,9(1x," // FMT_12 // "))")  "pb=", pb
write (u, "(A,9(1x," // FMT_12 // "))")  "r =", r
write (u, "(A,9(1x," // FMT_12 // "))")  "rb=", rb
write (u, "(A,9(1x," // FMT_12 // "))")  "fm=", f_map

write (u, "(A)")
write (u, "(A)")  "* Evaluate ISR structure function"

call sf_int(1)%apply (scale = 100._default)
call sf_int(2)%apply (scale = 100._default)

write (u, "(A)")
write (u, "(A)")  "* Structure function #1"
write (u, "(A)")
call sf_int(1)%write (u, testflag = .true.)

write (u, "(A)")
write (u, "(A)")  "* Structure function #2"
write (u, "(A)")
call sf_int(2)%write (u, testflag = .true.)

write (u, "(A)")
write (u, "(A)")  "* Structure-function value, default order"
write (u, "(A)")

do i = 1, 2
   f_isr(i) = sf_int(i)%get_matrix_element (1)
end do

write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr         =", &
     product (f_isr)
```

```
  write (u, "(A,9(1x," // FMT_12 // "))")  "f_isr * f_map =", &
       product (f_isr * f) * f_map

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  do i = 1, 2
     call sf_int(i)%final ()
  end do
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_isr_5"

end subroutine sf_isr_5
```

15.7 EPA

⟨sf_epa.f90⟩≡
⟨*File header*⟩

```
module sf_epa
```

⟨*Use kinds*⟩
⟨*Use strings*⟩

```
  use io_units
  use constants, only: pi
  use format_defs, only: FMT_17, FMT_19
  use numeric_utils
  use diagnostics
  use physics_defs, only: PHOTON
  use lorentz
  use pdg_arrays
  use model_data
  use flavors
  use colors
  use quantum_numbers
  use state_matrices
  use polarizations
  use interactions
  use sf_aux
  use sf_base
```

⟨*Standard module head*⟩

⟨*SF epa: public*⟩

⟨*SF epa: parameters*⟩

⟨*SF epa: types*⟩

```
contains
```

⟨*SF epa: procedures*⟩

```
end module sf_epa
```

15.7.1 Physics

The EPA structure function for a photon inside an (elementary) particle p with energy E, mass m and charge q_p (e.g., electron) is given by ($\bar{x} \equiv 1 - x$)

$$f(x) = \frac{\alpha}{\pi}\, q_p^2\, \frac{1}{x} \left[\left(\bar{x} + \frac{x^2}{2} \right) \ln \frac{Q^2_{\max}}{Q^2_{\min}} - \left(1 - \frac{x}{2}\right)^2 \ln \frac{x^2 + \frac{Q^2_{\max}}{E^2}}{x^2 + \frac{Q^2_{\min}}{E^2}} - x^2 \frac{m^2}{Q^2_{\min}} \left(1 - \frac{Q^2_{\min}}{Q^2_{\max}} \right) \right]. \quad (15.52)$$

If no explicit Q bounds are provided, the kinematical bounds are

$$-Q^2_{\max} = t_0 = -2\bar{x}(E^2 + p\bar{p}) + 2m^2 \approx -4\bar{x}E^2, \tag{15.53}$$

$$-Q^2_{\min} = t_1 = -2\bar{x}(E^2 - p\bar{p}) + 2m^2 \approx -\frac{x^2}{\bar{x}}m^2. \tag{15.54}$$

The second and third terms in (15.52) are negative definite (and subleading). Noting that $\bar{x}+x^2/2$ is bounded between 1/2 and 1, we derive that $f(x)$ is always smaller than

$$\bar{f}(x) = \frac{\alpha}{\pi}\, q_p^2\, \frac{L - 2\ln x}{x} \qquad \text{where} \qquad L = \ln \frac{\min(4E^2_{\max}, Q^2_{\max})}{\max(m^2, Q^2_{\min})}, \tag{15.55}$$

where we allow for explicit Q bounds that narrow the kinematical range. Therefore, we generate this distribution:

$$\int_{x_0}^{x_1} dx\, \bar{f}(x) = C(x_0, x_1) \int_0^1 dx' \tag{15.56}$$

We set

$$\ln x = \frac{1}{2}\left\{L - \sqrt{L^2 - 4\left[x' \ln x_1 (L - \ln x_1) + \bar{x}' \ln x_0 (L - \ln x_0)\right]}\right\} \tag{15.57}$$

such that $x(0) = x_0$ and $x(1) = x_1$ and

$$\frac{dx}{dx'} = \left(\frac{\alpha}{\pi} q_p^2\right)^{-1} x \frac{C(x_0, x_1)}{L - 2\ln x} \tag{15.58}$$

with

$$C(x_0, x_1) = \frac{\alpha}{\pi} q_p^2 \left[\ln x_1 (L - \ln x_1) - \ln x_0 (L - \ln x_0)\right] \tag{15.59}$$

such that (15.56) is satisfied. Finally, we have

$$\int_{x_0}^{x_1} dx\, f(x) = C(x_0, x_1) \int_0^1 dx'\, \frac{f(x(x'))}{\bar{f}(x(x'))} \tag{15.60}$$

where x' is calculated from x via (15.57).

The structure of the mapping is most obvious from:

$$x'(x) = \frac{\log x (L - \log x) - \log x_0 (L - \log x_0)}{\log x_1 (L - \log x_1) - \log x_0 (L - \log x_0)}\ . \tag{15.61}$$

15.7.2 The EPA data block

The EPA parameters are: α, $E_{\max}$, m, $Q_{\min}$, and $x_{\min}$. Instead of m we can use the incoming particle PDG code as input; from this we can deduce the mass and charge.

Internally we store in addition $C_{0/1} = \frac{\alpha}{\pi} q_e^2 \ln x_{0/1}(L - \ln x_{0/1})$, the c.m. energy squared and the incoming particle mass.

⟨*SF epa: public*⟩≡

```
public :: epa_data_t
```

⟨*SF epa: types*⟩≡

```
type, extends(sf_data_t) :: epa_data_t
   private
   class(model_data_t), pointer :: model => null ()
   type(flavor_t), dimension(:), allocatable :: flv_in
   real(default) :: alpha
   real(default) :: x_min
   real(default) :: x_max
   real(default) :: q_min
   real(default) :: q_max
   real(default) :: E_max
   real(default) :: mass
   real(default) :: log
   real(default) :: a
   real(default) :: c0
   real(default) :: c1
   real(default) :: dc
   integer :: error = NONE
   logical :: recoil = .false.
   logical :: keep_energy = .true.
 contains
 ⟨SF epa: epa data: TBP⟩
end type epa_data_t
```

Error codes

⟨*SF epa: parameters*⟩≡

```
integer, parameter :: NONE = 0
integer, parameter :: ZERO_QMIN = 1
integer, parameter :: Q_MAX_TOO_SMALL = 2
integer, parameter :: ZERO_XMIN = 3
integer, parameter :: MASS_MIX = 4
integer, parameter :: NO_EPA = 5
```

⟨*SF epa: epa data: TBP*⟩≡

```
procedure :: init => epa_data_init
```

⟨*SF epa: procedures*⟩≡

```
subroutine epa_data_init (data, model, pdg_in, alpha, &
     x_min, q_min, E_max, mass, recoil, keep_energy)
  class(epa_data_t), intent(inout) :: data
  class(model_data_t), intent(in), target :: model
  type(pdg_array_t), intent(in) :: pdg_in
  real(default), intent(in) :: alpha, x_min, q_min, E_max
  real(default), intent(in), optional :: mass
  logical, intent(in), optional :: recoil
  logical, intent(in), optional :: keep_energy
  integer :: n_flv, i
  data%model => model
  n_flv = pdg_array_get_length (pdg_in)
  allocate (data%flv_in (n_flv))
  do i = 1, n_flv
     call data%flv_in(i)%init (pdg_array_get (pdg_in, i), model)
  end do
  data%alpha = alpha
  data%E_max = E_max
```

```
    data%x_min = x_min
    data%x_max = 1
    if (vanishes (data%x_min)) then
       data%error = ZERO_XMIN;  return
    end if
    data%q_min = q_min
    data%q_max = 2 * data%E_max
    select case (char (data%model%get_name ()))
    case ("QCD","Test")
       data%error = NO_EPA;  return
    end select
    if (present (recoil)) then
       data%recoil = recoil
    end if
    if (present (keep_energy)) then
       data%keep_energy = keep_energy
    end if
    if (present (mass)) then
       data%mass = mass
    else
       data%mass = data%flv_in(1)%get_mass ()
       if (any (data%flv_in%get_mass () /= data%mass)) then
          data%error = MASS_MIX;  return
       end if
    end if
    if (max (data%mass, data%q_min) == 0) then
       data%error = ZERO_QMIN;  return
    else if (max (data%mass, data%q_min) >= data%E_max) then
       data%error = Q_MAX_TOO_SMALL;  return
    end if
    data%log = log (4 * (data%E_max / max (data%mass, data%q_min)) ** 2 )
    data%a  = data%alpha / pi
    data%c0 = log (data%x_min) * (data%log - log (data%x_min))
    data%c1 = log (data%x_max) * (data%log - log (data%x_max))
    data%dc = data%c1 - data%c0
  end subroutine epa_data_init
```

Handle error conditions. Should always be done after initialization, unless we are sure everything is ok.

⟨*SF epa: epa data: TBP*⟩+≡

```
  procedure :: check => epa_data_check
```

⟨*SF epa: procedures*⟩+≡

```
  subroutine epa_data_check (data)
    class(epa_data_t), intent(in) :: data
    select case (data%error)
    case (NO_EPA)
       call msg_fatal ("EPA structure function not available for model " &
            // char (data%model%get_name ()) // ".")
    case (ZERO_QMIN)
       call msg_fatal ("EPA: Particle mass is zero")
    case (Q_MAX_TOO_SMALL)
       call msg_fatal ("EPA: Particle mass exceeds Qmax")
    case (ZERO_XMIN)
```

```
      call msg_fatal ("EPA: x_min must be larger than zero")
   case (MASS_MIX)
      call msg_fatal ("EPA: incoming particle masses must be uniform")
   end select
end subroutine epa_data_check
```

Output

⟨*SF epa: epa data: TBP*⟩+≡

```
procedure :: write => epa_data_write
```

⟨*SF epa: procedures*⟩+≡

```
subroutine epa_data_write (data, unit, verbose)
   class(epa_data_t), intent(in) :: data
   integer, intent(in), optional :: unit
   logical, intent(in), optional :: verbose
   integer :: u, i
   u = given_output_unit (unit);  if (u < 0)  return
   write (u, "(1x,A)") "EPA data:"
   if (allocated (data%flv_in)) then
      write (u, "(3x,A)", advance="no") "  flavor =  "
      do i = 1, size (data%flv_in)
         if (i > 1)  write (u, "(',',1x)", advance="no")
         call data%flv_in(i)%write (u)
      end do
      write (u, *)
      write (u, "(3x,A," // FMT_19 // ")") "  alpha    = ", data%alpha
      write (u, "(3x,A," // FMT_19 // ")") "  x_min    = ", data%x_min
      write (u, "(3x,A," // FMT_19 // ")") "  x_max    = ", data%x_max
      write (u, "(3x,A," // FMT_19 // ")") "  q_min    = ", data%q_min
      write (u, "(3x,A," // FMT_19 // ")") "  q_max    = ", data%q_max
      write (u, "(3x,A," // FMT_19 // ")") "  E_max    = ", data%e_max
      write (u, "(3x,A," // FMT_19 // ")") "  mass     = ", data%mass
      write (u, "(3x,A," // FMT_19 // ")") "  a        = ", data%a
      write (u, "(3x,A," // FMT_19 // ")") "  c0       = ", data%c0
      write (u, "(3x,A," // FMT_19 // ")") "  c1       = ", data%c1
      write (u, "(3x,A," // FMT_19 // ")") "  log      = ", data%log
      write (u, "(3x,A,L2)")      "  recoil   = ", data%recoil
      write (u, "(3x,A,L2)")      "  keep en. = ", data%keep_energy
   else
      write (u, "(3x,A)") "[undefined]"
   end if
end subroutine epa_data_write
```

The number of kinematic parameters.

⟨*SF epa: epa data: TBP*⟩+≡

```
procedure :: get_n_par => epa_data_get_n_par
```

⟨*SF epa: procedures*⟩+≡

```
function epa_data_get_n_par (data) result (n)
   class(epa_data_t), intent(in) :: data
   integer :: n
   if (data%recoil) then
      n = 3
   else
```

```
      n = 1
    end if
  end function epa_data_get_n_par
```

Return the outgoing particles PDG codes. The outgoing particle is always the photon while the radiated particle is identical to the incoming one.

⟨*SF epa: epa data: TBP*⟩+≡

```
  procedure :: get_pdg_out => epa_data_get_pdg_out
```

⟨*SF epa: procedures*⟩+≡

```
  subroutine epa_data_get_pdg_out (data, pdg_out)
    class(epa_data_t), intent(in) :: data
    type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
    pdg_out(1) = PHOTON
  end subroutine epa_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF epa: epa data: TBP*⟩+≡

```
  procedure :: allocate_sf_int => epa_data_allocate_sf_int
```

⟨*SF epa: procedures*⟩+≡

```
  subroutine epa_data_allocate_sf_int (data, sf_int)
    class(epa_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (epa_t :: sf_int)
  end subroutine epa_data_allocate_sf_int
```

15.7.3 The EPA object

The `epa_t` data type is a $1 \to 2$ interaction. We should be able to handle several flavors in parallel, since EPA is not necessarily applied immediately after beam collision: Photons may be radiated from quarks. In that case, the partons are massless and $q_{\min}$ applies instead, so we do not need to generate several kinematical configurations in parallel.

The squared charge values multiply the matrix elements, depending on the flavour. We scan the interaction after building it, so we have the correct assignments.

The particles are ordered as (incoming, radiated, photon), where the photon initiates the hard interaction.

We generate an unpolarized photon and transfer initial polarization to the radiated parton. Color is transferred in the same way.

⟨*SF epa: types*⟩+≡

```
  type, extends (sf_int_t) :: epa_t
     type(epa_data_t), pointer :: data => null ()
     real(default) :: x  = 0
     real(default) :: xb = 0
     real(default) :: E  = 0
     real(default), dimension(:), allocatable :: charge2
   contains
   ⟨SF epa: epa: TBP⟩
  end type epa_t
```

Type string: has to be here, but there is no string variable on which EPA depends. Hence, a dummy routine.

⟨*SF epa: epa: TBP*⟩≡
```
procedure :: type_string => epa_type_string
```

⟨*SF epa: procedures*⟩+≡
```
function epa_type_string (object) result (string)
  class(epa_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "EPA: equivalent photon approx."
  else
     string = "EPA: [undefined]"
  end if
end function epa_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF epa: epa: TBP*⟩+≡
```
procedure :: write => epa_write
```

⟨*SF epa: procedures*⟩+≡
```
subroutine epa_write (object, unit, testflag)
  class(epa_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     if (object%status >= SF_DONE_KINEMATICS) then
        write (u, "(1x,A)")  "SF parameters:"
        write (u, "(3x,A," // FMT_17 // ")")  "x =", object%x
        if (object%status >= SF_FAILED_EVALUATION) then
           write (u, "(3x,A," // FMT_17 // ")")  "E =", object%E
        end if
     end if
     call object%base_write (u, testflag)
  else
     write (u, "(1x,A)")  "EPA data: [undefined]"
  end if
end subroutine epa_write
```

Prepare the interaction object. We have to construct transition matrix elements for all flavor and helicity combinations.

⟨*SF epa: epa: TBP*⟩+≡
```
procedure :: init => epa_init
```

⟨*SF epa: procedures*⟩+≡
```
subroutine epa_init (sf_int, data)
  class(epa_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  type(quantum_numbers_mask_t), dimension(3) :: mask
  integer, dimension(3) :: hel_lock
  type(polarization_t), target :: pol
```

```
    type(quantum_numbers_t), dimension(1) :: qn_fc
    type(flavor_t) :: flv_photon
    type(color_t) :: col_photon
    type(quantum_numbers_t) :: qn_hel, qn_photon, qn, qn_rad
    type(polarization_iterator_t) :: it_hel
    integer :: i
    mask = quantum_numbers_mask (.false., .false., &
         mask_h = [.false., .false., .true.])
    hel_lock = [2, 1, 0]
    select type (data)
    type is (epa_data_t)
       call sf_int%base_init (mask, [data%mass**2], &
            [data%mass**2], [0._default], hel_lock = hel_lock)
       sf_int%data => data
       call flv_photon%init (PHOTON, data%model)
       call col_photon%init ()
       call qn_photon%init (flv_photon, col_photon)
       do i = 1, size (data%flv_in)
          call pol%init_generic (data%flv_in(i))
          call qn_fc(1)%init ( &
               flv = data%flv_in(i), &
               col = color_from_flavor (data%flv_in(i), 1))
          call it_hel%init (pol)
          do while (it_hel%is_valid ())
             qn_hel = it_hel%get_quantum_numbers ()
             qn = qn_hel .merge. qn_fc(1)
             qn_rad = qn
             call qn_rad%tag_radiated ()
             call sf_int%add_state ([qn, qn_rad, qn_photon])
             call it_hel%advance ()
          end do
          !  call pol%final ()
       end do
       call sf_int%freeze ()
       if (data%keep_energy) then
          sf_int%on_shell_mode = KEEP_ENERGY
       else
          sf_int%on_shell_mode = KEEP_MOMENTUM
       end if
       call sf_int%set_incoming ([1])
       call sf_int%set_radiated ([2])
       call sf_int%set_outgoing ([3])
    end select
  end subroutine epa_init
```

Prepare the charge array. This is separate from the previous routine since the state matrix may be helicity-contracted.

⟨*SF epa: epa: TBP*⟩+≡

```
  procedure :: setup_constants => epa_setup_constants
```

⟨*SF epa: procedures*⟩+≡

```
  subroutine epa_setup_constants (sf_int)
    class(epa_t), intent(inout), target :: sf_int
    type(state_iterator_t) :: it
```

```
    type(flavor_t) :: flv
    integer :: i, n_me
    n_me = sf_int%get_n_matrix_elements ()
    allocate (sf_int%charge2 (n_me))
    call it%init (sf_int%interaction_t%get_state_matrix_ptr ())
    do while (it%is_valid ())
       i = it%get_me_index ()
       flv = it%get_flavor (1)
       sf_int%charge2(i) = flv%get_charge () ** 2
       call it%advance ()
    end do
    sf_int%status = SF_INITIAL
  end subroutine epa_setup_constants
```

15.7.4 Kinematics

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

The EPA structure function allows for a straightforward mapping of the unit interval. The x value is transformed, and the mapped structure function becomes unity at its upper boundary.

The structure function implementation applies the above mapping to the input (random) number `r` to generate the momentum fraction `x` and the function value `f`. For numerical stability reasons, we also output `xb`, which is $\bar{x} = 1 - x$.

⟨*SF epa: epa: TBP*⟩+≡

```
  procedure :: complete_kinematics => epa_complete_kinematics
```

⟨*SF epa: procedures*⟩+≡

```
  subroutine epa_complete_kinematics (sf_int, x, f, r, rb, map)
    class(epa_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: x
    real(default), intent(out) :: f
    real(default), dimension(:), intent(in) :: r
    real(default), dimension(:), intent(in) :: rb
    logical, intent(in) :: map
    real(default) :: xb1
    real(default) :: delta, sqrt_delta, lx
    if (map) then
       associate (data => sf_int%data)
         delta = data%log ** 2 -  4 * (r(1) * data%c1 + rb(1) * data%c0)
         if (delta > 0) then
            sqrt_delta = sqrt (delta)
            lx = (data%log - sqrt_delta) / 2
         else
            sf_int%status = SF_FAILED_KINEMATICS
            f = 0
            return
         end if
         x(1) = exp (lx)
         f = x(1) * data%dc / sqrt_delta
       end associate
    else
```

```
      x(1) = r(1)
      if (sf_int%data%x_min < x(1) .and. x(1) < sf_int%data%x_max) then
         f = 1
      else
         sf_int%status = SF_FAILED_KINEMATICS
         f = 0
         return
      end if
   end if
   xb1 = 1 - x(1)
   if (size(x) == 3)  x(2:3) = r(2:3)
   call sf_int%split_momentum (x, xb1)
   select case (sf_int%status)
   case (SF_DONE_KINEMATICS)
      sf_int%x = x(1)
      sf_int%xb= xb1
      sf_int%E  = energy (sf_int%get_momentum (1))
   case (SF_FAILED_KINEMATICS)
      sf_int%x = 0
      sf_int%xb= 0
      f = 0
   end select
end subroutine epa_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF epa: epa: TBP*⟩+≡

```
procedure :: inverse_kinematics => epa_inverse_kinematics
```

⟨*SF epa: procedures*⟩+≡

```
subroutine epa_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
  class(epa_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  real(default), dimension(:), intent(out) :: rb
  logical, intent(in) :: map
  logical, intent(in), optional :: set_momenta
  real(default) :: lx, delta, sqrt_delta, c
  logical :: set_mom
  set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
  if (map) then
     associate (data => sf_int%data)
       lx = log (x(1))
       sqrt_delta = data%log - 2 * lx
       delta = sqrt_delta ** 2
       c = (data%log ** 2 - delta) / 4
       r (1) = (c - data%c0) / data%dc
       rb(1) = (data%c1 - c) / data%dc
       f = x(1) * data%dc / sqrt_delta
     end associate
  else
     r (1) = x(1)
```

```
      rb(1) = 1 - x(1)
      if (sf_int%data%x_min < x(1) .and. x(1) < sf_int%data%x_max) then
         f = 1
      else
         f = 0
      end if
   end if
   if (size(r) == 3) then
      r (2:3) = x(2:3)
      rb(2:3) = 1 - x(2:3)
   end if
   if (set_mom) then
      call sf_int%split_momentum (x, sf_int%xb)
      select case (sf_int%status)
      case (SF_DONE_KINEMATICS)
         sf_int%x  = x(1)
         sf_int%xb = 1 - x(1)
         sf_int%E  = energy (sf_int%get_momentum (1))
      case (SF_FAILED_KINEMATICS)
         sf_int%x = 0
         f = 0
      end select
   end if
end subroutine epa_inverse_kinematics
```

Overriding the default method: we compute the `x` array from the momentum configuration. In the specific case of EPA, we also set the internally stored x and $\bar{x}$ values, so they can be used in the following routine.

Note: the extraction of $\bar{x}$ is not numerically safe, but it can't be as long as the base `recover_x` isn't.

⟨*SF epa: epa: TBP*⟩+≡

```
procedure :: recover_x => sf_epa_recover_x
```

⟨*SF epa: procedures*⟩+≡

```
subroutine sf_epa_recover_x (sf_int, x, x_free)
  class(epa_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(inout), optional :: x_free
  call sf_int%base_recover_x (x, x_free)
  sf_int%x  = x(1)
  sf_int%xb = 1 - x(1)
end subroutine sf_epa_recover_x
```

15.7.5 EPA application

For EPA, we can in principle compute kinematics and function value in a single step. In order to be able to reweight events, kinematics and structure function application are separated. This function works on a single beam, assuming that the input momentum has been set. We need three random numbers as input: one for x, and two for the polar and azimuthal angles. Alternatively, for the no-recoil case, we can skip p_T generation; in this case, we only need one.

For obtaining splitting kinematics, we rely on the assumption that all in-particles are mass-degenerate (or there is only one), so the generated x values are identical.

⟨*SF epa: epa: TBP*⟩+≡
```
procedure :: apply => epa_apply
```

⟨*SF epa: procedures*⟩+≡
```
subroutine epa_apply (sf_int, scale)
  class(epa_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  real(default) :: x, xb, qminsq, qmaxsq, f, E
  associate (data => sf_int%data)
    x = sf_int%x
    xb= sf_int%xb
    E = sf_int%E
    qminsq = max (x ** 2 / xb * data%mass ** 2, data%q_min ** 2)
    qmaxsq = min (4 * xb * E ** 2, data%q_max ** 2)
    if (qminsq < qmaxsq) then
       f = data%a / x &
            * ((xb + x ** 2 / 2) * log (qmaxsq / qminsq) &
            - (1 - x / 2) ** 2 &
            * log ((x**2 + qmaxsq / E ** 2) / (x**2 + qminsq / E ** 2)) &
            - x ** 2 * data%mass ** 2 / qminsq * (1 - qminsq / qmaxsq))
    else
       f = 0
    end if
    call sf_int%set_matrix_element &
         (cmplx (f, kind=default) * sf_int%charge2)
  end associate
  sf_int%status = SF_EVALUATED
end subroutine epa_apply
```

15.7.6 Unit tests

Test module, followed by the corresponding implementation module.

⟨`sf_epa_ut.f90`⟩≡
```
⟨File header⟩

module sf_epa_ut
  use unit_tests
  use sf_epa_uti

⟨Standard module head⟩

⟨SF epa: public test⟩

contains

⟨SF epa: test driver⟩

end module sf_epa_ut
```

⟨sf_epa_uti.f90⟩≡
⟨*File header*⟩

```
module sf_epa_uti
```

⟨*Use kinds*⟩

```
  use physics_defs, only: ELECTRON
  use lorentz
  use pdg_arrays
  use flavors
  use interactions, only: reset_interaction_counter
  use interactions, only: interaction_pacify_momenta
  use model_data
  use sf_aux
  use sf_base

  use sf_epa
```

⟨*Standard module head*⟩

⟨*SF epa: test declarations*⟩

```
contains
```

⟨*SF epa: tests*⟩

```
end module sf_epa_uti
```

API: driver for the unit tests below.

⟨*SF epa: public test*⟩≡

```
  public :: sf_epa_test
```

⟨*SF epa: test driver*⟩≡

```
  subroutine sf_epa_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```

⟨*SF epa: execute tests*⟩

```
  end subroutine sf_epa_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF epa: execute tests*⟩≡

```
  call test (sf_epa_1, "sf_epa_1", &
       "structure function configuration", &
       u, results)
```

⟨*SF epa: test declarations*⟩≡

```
  public :: sf_epa_1
```

⟨*SF epa: tests*⟩≡

```
  subroutine sf_epa_1 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(pdg_array_t) :: pdg_in
```

```
    type(pdg_array_t), dimension(1) :: pdg_out
    integer, dimension(:), allocatable :: pdg1
    class(sf_data_t), allocatable :: data

    write (u, "(A)")  "* Test output: sf_epa_1"
    write (u, "(A)")  "*   Purpose: initialize and display &
         &test structure function data"
    write (u, "(A)")

    write (u, "(A)")  "* Create empty data object"
    write (u, "(A)")

    call model%init_qed_test ()
    pdg_in = ELECTRON

    allocate (epa_data_t :: data)
    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize"
    write (u, "(A)")

    select type (data)
    type is (epa_data_t)
       call data%init (model, pdg_in, 1./137._default, 0.01_default, &
            10._default, 50._default, 0.000511_default, recoil = .false.)
    end select

    call data%write (u)

    write (u, "(A)")

    write (u, "(1x,A)")  "Outgoing particle codes:"
    call data%get_pdg_out (pdg_out)
    pdg1 = pdg_out(1)
    write (u, "(2x,99(1x,I0))")  pdg1

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_epa_1"

  end subroutine sf_epa_1
```

Test and probe structure function

Construct and display a structure function object based on the EPA structure function.

⟨*SF epa: execute tests*⟩+≡

```
  call test (sf_epa_2, "sf_epa_2", &
       "structure function instance", &
       u, results)
```

⟨*SF epa: test declarations*⟩+≡

```
public :: sf_epa_2
```

⟨*SF epa: tests*⟩+≡

```
subroutine sf_epa_2 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t) :: flv
  type(pdg_array_t) :: pdg_in
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k
  type(vector4_t), dimension(2) :: q
  real(default) :: E
  real(default), dimension(:), allocatable :: r, rb, x
  real(default) :: f

  write (u, "(A)")  "* Test output: sf_epa_2"
  write (u, "(A)")  "*   Purpose: initialize and fill &
       &test structure function object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call model%init_qed_test ()
  call flv%init (ELECTRON, model)
  pdg_in = ELECTRON

  call reset_interaction_counter ()

  allocate (epa_data_t :: data)
  select type (data)
  type is (epa_data_t)
     call data%init (model, pdg_in, 1./137._default, 0.01_default, &
          10._default, 50._default, 0.000511_default, recoil = .false.)
  end select

  write (u, "(A)")  "* Initialize structure-function object"
  write (u, "(A)")

  call data%allocate_sf_int (sf_int)
  call sf_int%init (data)
  call sf_int%set_beam_index ([1])
  call sf_int%setup_constants ()

  write (u, "(A)")  "* Initialize incoming momentum with E=500"
  write (u, "(A)")
  E = 500
  k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
  call pacify (k, 1e-10_default)
  call vector4_write (k, u)
  call sf_int%seed_kinematics ([k])

  write (u, "(A)")
```

```
write (u, "(A)")  "* Set kinematics for r=0.4, no EPA mapping, collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.4_default
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = sf_int%get_momenta (outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

call sf_int%seed_kinematics ([k])
call sf_int%set_momenta (q, outgoing=.true.)
call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false., &
     set_momenta=.true.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Evaluate EPA structure function"
write (u, "(A)")

call sf_int%apply (scale = 100._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_int%final ()
call model%final ()

write (u, "(A)")
```

```
    write (u, "(A)")  "* Test output end: sf_epa_2"

  end subroutine sf_epa_2
```

Standard mapping

Construct and display a structure function object based on the EPA structure function, applying the standard single-particle mapping.

⟨*SF epa: execute tests*⟩+≡

```
  call test (sf_epa_3, "sf_epa_3", &
       "apply mapping", &
       u, results)
```

⟨*SF epa: test declarations*⟩+≡

```
  public :: sf_epa_3
```

⟨*SF epa: tests*⟩+≡

```
  subroutine sf_epa_3 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t) :: flv
    type(pdg_array_t) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q
    real(default) :: E
    real(default), dimension(:), allocatable :: r, rb, x
    real(default) :: f

    write (u, "(A)")  "* Test output: sf_epa_3"
    write (u, "(A)")  "*   Purpose: initialize and fill &
         &test structure function object"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize configuration data"
    write (u, "(A)")

    call model%init_qed_test ()
    call flv%init (ELECTRON, model)
    pdg_in = ELECTRON

    call reset_interaction_counter ()

    allocate (epa_data_t :: data)
    select type (data)
    type is (epa_data_t)
       call data%init (model, pdg_in, 1./137._default, 0.01_default, &
            10._default, 50._default, 0.000511_default, recoil = .false.)
    end select

    write (u, "(A)")  "* Initialize structure-function object"
    write (u, "(A)")
```

```
call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.4, with EPA mapping, collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.4_default
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.true.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = sf_int%get_momenta (outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

call sf_int%seed_kinematics ([k])
call sf_int%set_momenta (q, outgoing=.true.)
call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.true., &
     set_momenta=.true.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Evaluate EPA structure function"
    write (u, "(A)")

    call sf_int%apply (scale = 100._default)
    call sf_int%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_epa_3"

  end subroutine sf_epa_3
```

Non-collinear case

Construct and display a structure function object based on the EPA structure function.

⟨*SF epa: execute tests*⟩+≡

```
  call test (sf_epa_4, "sf_epa_4", &
       "non-collinear", &
       u, results)
```

⟨*SF epa: test declarations*⟩+≡

```
  public :: sf_epa_4
```

⟨*SF epa: tests*⟩+≡

```
  subroutine sf_epa_4 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t) :: flv
    type(pdg_array_t) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q
    real(default) :: E, m
    real(default), dimension(:), allocatable :: r, rb, x
    real(default) :: f

    write (u, "(A)")  "* Test output: sf_epa_4"
    write (u, "(A)")  "*   Purpose: initialize and fill &
         &test structure function object"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize configuration data"
    write (u, "(A)")
```

```
call model%init_qed_test ()
call flv%init (ELECTRON, model)
pdg_in = ELECTRON

call reset_interaction_counter ()

allocate (epa_data_t :: data)
select type (data)
type is (epa_data_t)
   call data%init (model, pdg_in, 1./137._default, 0.01_default, &
        10._default, 50._default, 5.0_default, recoil = .true.)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

write (u, "(A)")  "* Initialize incoming momentum with E=500, me = 5 GeV"
write (u, "(A)")
E = 500
m = 5
k = vector4_moving (E, sqrt (E**2 - m**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.5/0.5/0.25, with EPA mapping, "
write (u, "(A)")  "          non-coll., keeping energy, me = 5 GeV"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = [0.5_default, 0.5_default, 0.25_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.true.)
call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x and r from momenta"
write (u, "(A)")
```

```
    q = sf_int%get_momenta (outgoing=.true.)
    call sf_int%final ()
    deallocate (sf_int)

    call data%allocate_sf_int (sf_int)
    call sf_int%init (data)
    call sf_int%set_beam_index ([1])
    call sf_int%setup_constants ()

    call sf_int%seed_kinematics ([k])
    call sf_int%set_momenta (q, outgoing=.true.)
    call sf_int%recover_x (x)
    call sf_int%inverse_kinematics (x, f, r, rb, map=.true., &
         set_momenta=.true.)
    call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)

    write (u, "(A,9(1x,F10.7))")  "r =", r
    write (u, "(A,9(1x,F10.7))")  "rb=", rb
    write (u, "(A,9(1x,F10.7))")  "x =", x
    write (u, "(A,9(1x,F10.7))")  "f =", f

    write (u, "(A)")
    write (u, "(A)")  "* Evaluate EPA structure function"
    write (u, "(A)")

    call sf_int%apply (scale = 100._default)
    call sf_int%write (u, testflag = .true.)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_epa_4"

  end subroutine sf_epa_4
```

Structure function for multiple flavors

Construct and display a structure function object based on the EPA structure function. The incoming state has multiple particles with non-uniform charge.

⟨*SF epa: execute tests*⟩+≡

```
  call test (sf_epa_5, "sf_epa_5", &
       "multiple flavors", &
       u, results)
```

⟨*SF epa: test declarations*⟩+≡

```
  public :: sf_epa_5
```

⟨*SF epa: tests*⟩+≡

```
  subroutine sf_epa_5 (u)
```

```
integer, intent(in) :: u
type(model_data_t), target :: model
type(flavor_t) :: flv
type(pdg_array_t) :: pdg_in
class(sf_data_t), allocatable, target :: data
class(sf_int_t), allocatable :: sf_int
type(vector4_t) :: k
real(default) :: E
real(default), dimension(:), allocatable :: r, rb, x
real(default) :: f

write (u, "(A)")  "* Test output: sf_epa_5"
write (u, "(A)")  "*   Purpose: initialize and fill &
     &test structure function object"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call model%init_sm_test ()
call flv%init (1, model)
pdg_in = [1, 2, -1, -2]

call reset_interaction_counter ()

allocate (epa_data_t :: data)
select type (data)
type is (epa_data_t)
   call data%init (model, pdg_in, 1./137._default, 0.01_default, &
        10._default, 50._default, 0.000511_default, recoil = .false.)
   call data%check ()
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.4, no EPA mapping, collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
```

```
  allocate (rb(size (r)))
  allocate (x (size (r)))

  r = 0.4_default
  rb = 1 - r
  call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

  write (u, "(A,9(1x,F10.7))")  "r =", r
  write (u, "(A,9(1x,F10.7))")  "rb=", rb
  write (u, "(A,9(1x,F10.7))")  "x =", x
  write (u, "(A,9(1x,F10.7))")  "f =", f

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate EPA structure function"
  write (u, "(A)")

  call sf_int%apply (scale = 100._default)
  call sf_int%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call sf_int%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_epa_5"

end subroutine sf_epa_5
```

15.8 EWA

⟨sf_ewa.f90⟩≡
⟨*File header*⟩

```
module sf_ewa
```

⟨*Use kinds*⟩
⟨*Use strings*⟩

```
  use io_units
  use constants, only: pi
  use format_defs, only: FMT_17, FMT_19
  use numeric_utils
  use diagnostics
  use physics_defs, only: W_BOSON, Z_BOSON
  use lorentz
  use pdg_arrays
  use model_data
  use flavors
  use colors
  use quantum_numbers
  use state_matrices
  use polarizations
  use interactions
  use sf_aux
  use sf_base
```

⟨*Standard module head*⟩

⟨*SF ewa: public*⟩

⟨*SF ewa: parameters*⟩

⟨*SF ewa: types*⟩

```
contains
```

⟨*SF ewa: procedures*⟩

```
end module sf_ewa
```

15.8.1 Physics

The EWA structure function for a Z or W inside a fermion (lepton or quark) depends on the vector-boson polarization. We distinguish transversal ($\pm$) and

longitudinal (0) polarization.

$$F_+(x) = \frac{1}{16\pi^2}\,\frac{(v-a)^2+(v+a)^2\bar{x}^2}{x}\left[\ln\left(\frac{p_{\perp,\max}^2+\bar{x}M^2}{\bar{x}M^2}\right)-\frac{p_{\perp,\max}^2}{p_{\perp,\max}^2+\bar{x}M^2}\right] \tag{15.62}$$

$$F_-(x) = \frac{1}{16\pi^2}\,\frac{(v+a)^2+(v-a)^2\bar{x}^2}{x}\left[\ln\left(\frac{p_{\perp,\max}^2+\bar{x}M^2}{\bar{x}M^2}\right)-\frac{p_{\perp,\max}^2}{p_{\perp,\max}^2+\bar{x}M^2}\right] \tag{15.63}$$

$$F_0(x) = \frac{v^2+a^2}{8\pi^2}\,\frac{2\bar{x}}{x}\,\frac{p_{\perp,\max}^2}{p_{\perp,\max}^2+\bar{x}M^2} \tag{15.64}$$

where $p_{\perp,\max}$ is the cutoff in transversal momentum, M is the vector-boson mass, v and a are the vector and axial-vector couplings, and $\bar{x} \equiv 1-x$. Note that the longitudinal structure function is finite for large cutoff, while the transversal structure function is logarithmically divergent.

The maximal transverse momentum is given by the kinematical limit, it is

$$p_{\perp,\max} = \bar{x}\sqrt{s}/2. \tag{15.65}$$

The vector and axial couplings for a fermion branching into a W are

$$v_W = \frac{g}{2\sqrt{2}}, \qquad a_W = \frac{g}{2\sqrt{2}}. \tag{15.66}$$

For Z emission, this is replaced by

$$v_Z = \frac{g}{2\cos\theta_w}\left(t_3 - 2q\sin^2\theta_w\right), \qquad a_Z = \frac{g}{2\cos\theta_w}t_3, \tag{15.67}$$

where $t_3 = \pm\frac{1}{2}$ is the fermion isospin, and q its charge.

For an initial antifermion, the signs of the axial couplings are inverted. Note that a common sign change of v and a is irrelevant.

The EWA depends on the parameters g, $\sin^2\theta_w$, M_W, and M_Z. These can all be taken from the SM input, and the prefactors are calculated from those and the incoming particle type.

Since these structure functions have a $1/x$ singularity (which is not really relevant in practice, however, since the vector boson mass is finite), we map this singularity allowing for nontrivial x bounds:

$$x = \exp(\bar{r}\ln x_0 + r\ln x_1) \tag{15.68}$$

such that

$$\int_{x_0}^{x_1}\frac{dx}{x} = (\ln x_1 - \ln x_0)\int_0^1 dr. \tag{15.69}$$

As a user parameter, we have the cutoff $p_{\perp,\max}$. The divergence $1/x$ also requires a x_0 cutoff; and for completeness we introduce a corresponding x_1. Physically, the minimal sensible value of x is M^2/s, although the approximation loses its value already at higher x values.

15.8.2 The EWA data block

The EWA parameters are: $p_{T,\max}$, c_V, c_A, and m. Instead of m we can use the incoming particle PDG code as input; from this we can deduce the mass and charges. In the initialization phase it is not yet determined whether a W or a Z is radiated, hence we set the vector and axial-vector couplings equal to the common prefactors $g/2 = e/2/\sin\theta_W$.

In principle, for EWA it would make sense to allow the user to also set the upper bound for x, $x_{\max}$, but we fix it to one here.

⟨*SF ewa: public*⟩≡
```
public :: ewa_data_t
```

⟨*SF ewa: types*⟩≡
```
type, extends(sf_data_t) :: ewa_data_t
   private
   class(model_data_t), pointer :: model => null ()
   type(flavor_t), dimension(:), allocatable :: flv_in
   type(flavor_t), dimension(:), allocatable :: flv_out
   real(default) :: pt_max
   real(default) :: sqrts
   real(default) :: x_min
   real(default) :: x_max
   real(default) :: mass
   real(default) :: m_out
   real(default) :: q_min
   real(default) :: cv
   real(default) :: ca
   real(default) :: costhw
   real(default) :: sinthw
   real(default) :: mW
   real(default) :: mZ
   real(default) :: coeff
   logical :: mass_set = .false.
   logical :: recoil = .false.
   logical :: keep_energy = .false.
   integer :: id = 0
   integer :: error = NONE
 contains
 ⟨SF ewa: ewa data: TBP⟩
end type ewa_data_t
```

Error codes

⟨*SF ewa: parameters*⟩≡
```
integer, parameter :: NONE = 0
integer, parameter :: ZERO_QMIN = 1
integer, parameter :: Q_MAX_TOO_SMALL = 2
integer, parameter :: ZERO_XMIN = 3
integer, parameter :: MASS_MIX = 4
integer, parameter :: ZERO_SW = 5
integer, parameter :: ISOSPIN_MIX = 6
integer, parameter :: WRONG_PRT = 7
integer, parameter :: MASS_MIX_OUT = 8
integer, parameter :: NO_EWA = 9
```

⟨*SF ewa: ewa data: TBP*⟩≡

```
procedure :: init => ewa_data_init
```

⟨*SF ewa: procedures*⟩≡

```
subroutine ewa_data_init (data, model, pdg_in, x_min, pt_max, &
      sqrts, recoil, keep_energy, mass)
  class(ewa_data_t), intent(inout) :: data
  class(model_data_t), intent(in), target :: model
  type(pdg_array_t), intent(in) :: pdg_in
  real(default), intent(in) :: x_min, pt_max, sqrts
  logical, intent(in) :: recoil, keep_energy
  real(default), intent(in), optional :: mass
  real(default) :: g, ee
  integer :: n_flv, i
  data%model => model
  if (.not. any (pdg_in .match. &
       [1,2,3,4,5,6,11,13,15,-1,-2,-3,-4,-5,-6,-11,-13,-15])) then
     data%error = WRONG_PRT;  return
  end if
  n_flv = pdg_array_get_length (pdg_in)
  allocate (data%flv_in (n_flv))
  allocate (data%flv_out(n_flv))
  do i = 1, n_flv
     call data%flv_in(i)%init (pdg_array_get (pdg_in, i), model)
  end do
  data%pt_max = pt_max
  data%sqrts = sqrts
  data%x_min = x_min
  data%x_max = 1
  if (vanishes (data%x_min)) then
     data%error = ZERO_XMIN;  return
  end if
  select case (char (data%model%get_name ()))
  case ("QCD","QED","Test")
     data%error = NO_EWA;  return
  end select
  ee = data%model%get_real (var_str ("ee"))
  data%sinthw = data%model%get_real (var_str ("sw"))
  data%costhw = data%model%get_real (var_str ("cw"))
  data%mZ = data%model%get_real (var_str ("mZ"))
  data%mW = data%model%get_real (var_str ("mW"))
  if (data%sinthw /= 0) then
     g = ee / data%sinthw
  else
     data%error = ZERO_SW;  return
  end if
  data%cv = g / 2._default
  data%ca = g / 2._default
  data%coeff = 1._default / (8._default * PI**2)
  data%recoil = recoil
  data%keep_energy = keep_energy
  if (present (mass)) then
     data%mass = mass
     data%m_out = mass
     data%mass_set = .true.
```

```
    else
       data%mass = data%flv_in(1)%get_mass ()
       if (any (data%flv_in%get_mass () /= data%mass)) then
          data%error = MASS_MIX;  return
       end if
    end if
  end subroutine ewa_data_init
```

Set the vector boson ID for distinguishing W and Z bosons.

⟨*SF ewa: ewa data: TBP*⟩+≡

```
  procedure :: set_id => ewa_set_id
```

⟨*SF ewa: procedures*⟩+≡

```
  subroutine ewa_set_id (data, id)
    class(ewa_data_t), intent(inout) :: data
    integer, intent(in) :: id
    integer :: i, isospin, pdg
    if (.not. allocated (data%flv_in)) &
         call msg_bug ("EWA: incoming particles not set")
    data%id = id
    select case (data%id)
    case (23)
       data%m_out = data%mass
       data%flv_out = data%flv_in
    case (24)
       do i = 1, size (data%flv_in)
          pdg = data%flv_in(i)%get_pdg ()
          isospin = data%flv_in(i)%get_isospin_type ()
          if (isospin > 0) then
             !!! up-type quark or neutrinos
             if (data%flv_in(i)%is_antiparticle ()) then
                call data%flv_out(i)%init (pdg + 1, data%model)
             else
                call data%flv_out(i)%init (pdg - 1, data%model)
             end if
          else
             !!! down-type quark or lepton
             if (data%flv_in(i)%is_antiparticle ()) then
                call data%flv_out(i)%init (pdg - 1, data%model)
             else
                call data%flv_out(i)%init (pdg + 1, data%model)
             end if
          end if
       end do
       if (.not. data%mass_set) then
          data%m_out = data%flv_out(1)%get_mass ()
          if (any (data%flv_out%get_mass () /= data%m_out)) then
             data%error = MASS_MIX_OUT;  return
          end if
       end if
    end select
  end subroutine ewa_set_id
```

Handle error conditions. Should always be done after initialization, unless we are sure everything is ok.

⟨*SF ewa: ewa data: TBP*⟩+≡
```
procedure :: check => ewa_data_check
```

⟨*SF ewa: procedures*⟩+≡
```
subroutine ewa_data_check (data)
  class(ewa_data_t), intent(in) :: data
  select case (data%error)
  case (WRONG_PRT)
     call msg_fatal ("EWA structure function only accessible for " &
          // "SM quarks and leptons.")
  case (NO_EWA)
     call msg_fatal ("EWA structure function not available for model " &
          // char (data%model%get_name ()))
  case (ZERO_SW)
     call msg_fatal ("EWA: Vanishing value of sin(theta_w)")
  case (ZERO_QMIN)
     call msg_fatal ("EWA: Particle mass is zero")
  case (Q_MAX_TOO_SMALL)
     call msg_fatal ("EWA: Particle mass exceeds Qmax")
  case (ZERO_XMIN)
     call msg_fatal ("EWA: x_min must be larger than zero")
  case (MASS_MIX)
     call msg_fatal ("EWA: incoming particle masses must be uniform")
  case (MASS_MIX_OUT)
     call msg_fatal ("EWA: outgoing particle masses must be uniform")
  case (ISOSPIN_MIX)
     call msg_fatal ("EWA: incoming particle isospins must be uniform")
  end select
end subroutine ewa_data_check
```

Output

⟨*SF ewa: ewa data: TBP*⟩+≡
```
procedure :: write => ewa_data_write
```

⟨*SF ewa: procedures*⟩+≡
```
subroutine ewa_data_write (data, unit, verbose)
  class(ewa_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)") "EWA data:"
  if (allocated (data%flv_in) .and. allocated (data%flv_out)) then
     write (u, "(3x,A)", advance="no") "  flavor(in)  =  "
     do i = 1, size (data%flv_in)
        if (i > 1)  write (u, "(',',1x)", advance="no")
        call data%flv_in(i)%write (u)
     end do
     write (u, *)
     write (u, "(3x,A)", advance="no") "  flavor(out) =  "
     do i = 1, size (data%flv_out)
        if (i > 1)  write (u, "(',',1x)", advance="no")
```

```
      call data%flv_out(i)%write (u)
   end do
   write (u, *)
   write (u, "(3x,A," // FMT_19 // ")") "  x_min     = ", data%x_min
   write (u, "(3x,A," // FMT_19 // ")") "  x_max     = ", data%x_max
   write (u, "(3x,A," // FMT_19 // ")") "  pt_max    = ", data%pt_max
   write (u, "(3x,A," // FMT_19 // ")") "  sqrts     = ", data%sqrts
   write (u, "(3x,A," // FMT_19 // ")") "  mass      = ", data%mass
   write (u, "(3x,A," // FMT_19 // ")") "  cv        = ", data%cv
   write (u, "(3x,A," // FMT_19 // ")") "  ca        = ", data%ca
   write (u, "(3x,A," // FMT_19 // ")") "  coeff     = ", data%coeff
   write (u, "(3x,A," // FMT_19 // ")") "  costhw    = ", data%costhw
   write (u, "(3x,A," // FMT_19 // ")") "  sinthw    = ", data%sinthw
   write (u, "(3x,A," // FMT_19 // ")") "  mZ        = ", data%mZ
   write (u, "(3x,A," // FMT_19 // ")") "  mW        = ", data%mW
   write (u, "(3x,A,L2)")      "  recoil    = ", data%recoil
   write (u, "(3x,A,L2)")      "  keep en.  = ", data%keep_energy
   write (u, "(3x,A,I2)")      "  PDG (VB)  = ", data%id
else
   write (u, "(3x,A)") "[undefined]"
end if
end subroutine ewa_data_write
```

The number of parameters is one for collinear splitting, in case the `recoil` option is set, we take the recoil into account.

⟨*SF ewa: ewa data: TBP*⟩+≡

```
procedure :: get_n_par => ewa_data_get_n_par
```

⟨*SF ewa: procedures*⟩+≡

```
function ewa_data_get_n_par (data) result (n)
  class(ewa_data_t), intent(in) :: data
  integer :: n
  if (data%recoil) then
     n = 3
  else
     n = 1
  end if
end function ewa_data_get_n_par
```

Return the outgoing particles PDG codes. This depends, whether this is a charged-current or neutral-current interaction.

⟨*SF ewa: ewa data: TBP*⟩+≡

```
procedure :: get_pdg_out => ewa_data_get_pdg_out
```

⟨*SF ewa: procedures*⟩+≡

```
subroutine ewa_data_get_pdg_out (data, pdg_out)
  class(ewa_data_t), intent(in) :: data
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
  integer, dimension(:), allocatable :: pdg1
  integer :: i, n_flv
  if (allocated (data%flv_out)) then
     n_flv = size (data%flv_out)
  else
     n_flv = 0
```

```
    end if
    allocate (pdg1 (n_flv))
    do i = 1, n_flv
       pdg1(i) = data%flv_out(i)%get_pdg ()
    end do
    pdg_out(1) = pdg1
  end subroutine ewa_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF ewa: ewa data: TBP*⟩+≡

```
  procedure :: allocate_sf_int => ewa_data_allocate_sf_int
```

⟨*SF ewa: procedures*⟩+≡

```
  subroutine ewa_data_allocate_sf_int (data, sf_int)
    class(ewa_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (ewa_t :: sf_int)
  end subroutine ewa_data_allocate_sf_int
```

15.8.3 The EWA object

The `ewa_t` data type is a $1 \to 2$ interaction. We should be able to handle several flavors in parallel, since EWA is not necessarily applied immediately after beam collision: W/Z bosons may be radiated from quarks. In that case, the partons are massless and $q_{\min}$ applies instead, so we do not need to generate several kinematical configurations in parallel.

The particles are ordered as (incoming, radiated, W/Z), where the W/Z initiates the hard interaction.

In the case of EPA, we generated an unpolarized photon and transferred initial polarization to the radiated parton. Color is transferred in the same way. I do not know whether the same can/should be done for EWA, as the structure functions depend on the W/Z polarization. If we are having Z bosons, both up- and down-type fermions can participate. Otherwise, with a W^+ an up-type fermion is transferred to a down-type fermion, and the other way round.

⟨*SF ewa: types*⟩+≡

```
  type, extends (sf_int_t) :: ewa_t
     type(ewa_data_t), pointer :: data => null ()
     real(default) :: x  = 0
     real(default) :: xb = 0
     integer :: n_me = 0
     real(default), dimension(:), allocatable :: cv
     real(default), dimension(:), allocatable :: ca
   contains
   ⟨SF ewa: ewa: TBP⟩
  end type ewa_t
```

Type string: has to be here, but there is no string variable on which EWA depends. Hence, a dummy routine.

⟨*SF ewa: ewa: TBP*⟩≡

```
  procedure :: type_string => ewa_type_string
```

⟨*SF ewa: procedures*⟩+≡

```
function ewa_type_string (object) result (string)
  class(ewa_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "EWA: equivalent W/Z approx."
  else
     string = "EWA: [undefined]"
  end if
end function ewa_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF ewa: ewa: TBP*⟩+≡

```
procedure :: write => ewa_write
```

⟨*SF ewa: procedures*⟩+≡

```
subroutine ewa_write (object, unit, testflag)
  class(ewa_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     if (object%status >= SF_DONE_KINEMATICS) then
        write (u, "(1x,A)")  "SF parameters:"
        write (u, "(3x,A," // FMT_17 // ")")  "x =", object%x
     end if
     call object%base_write (u, testflag)
  else
     write (u, "(1x,A)")  "EWA data: [undefined]"
  end if
end subroutine ewa_write
```

The current implementation requires uniform isospin for all incoming particles, therefore we need to probe only the first one.

⟨*SF ewa: ewa: TBP*⟩+≡

```
procedure :: init => ewa_init
```

⟨*SF ewa: procedures*⟩+≡

```
subroutine ewa_init (sf_int, data)
  class(ewa_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  type(quantum_numbers_mask_t), dimension(3) :: mask
  integer, dimension(3) :: hel_lock
  type(polarization_t), target :: pol
  type(quantum_numbers_t), dimension(1) :: qn_fc, qn_fc_fin
  type(flavor_t) :: flv_z, flv_wp, flv_wm
  type(color_t) :: col0
  type(quantum_numbers_t) :: qn_hel, qn_z, qn_wp, qn_wm, qn, qn_rad, qn_w
  type(polarization_iterator_t) :: it_hel
  integer :: i, isospin
  select type (data)
  type is (ewa_data_t)
```

```
mask = quantum_numbers_mask (.false., .false., &
     mask_h = [.false., .false., .true.])
hel_lock = [2, 1, 0]
call col0%init ()
select case (data%id)
case (23)
   !!! Z boson, flavor is not changing
   call sf_int%base_init (mask, [data%mass**2], [data%mass**2], &
        [data%mZ**2], hel_lock = hel_lock)
   sf_int%data => data
   call flv_z%init (Z_BOSON, data%model)
   call qn_z%init (flv_z, col0)
   do i = 1, size (data%flv_in)
      call pol%init_generic (data%flv_in(i))
      call qn_fc(1)%init ( &
           flv = data%flv_in(i), &
           col = color_from_flavor (data%flv_in(i), 1))
      call it_hel%init (pol)
      do while (it_hel%is_valid ())
         qn_hel = it_hel%get_quantum_numbers ()
         qn = qn_hel .merge. qn_fc(1)
         qn_rad = qn
         call qn_rad%tag_radiated ()
         call sf_int%add_state ([qn, qn_rad, qn_z])
         call it_hel%advance ()
      end do
      !  call pol%final ()
   end do
case (24)
   call sf_int%base_init (mask, [data%mass**2], [data%m_out**2], &
        [data%mW**2], hel_lock = hel_lock)
   sf_int%data => data
   call flv_wp%init (W_BOSON, data%model)
   call flv_wm%init (- W_BOSON, data%model)
   call qn_wp%init (flv_wp, col0)
   call qn_wm%init (flv_wm, col0)
   do i = 1, size (data%flv_in)
      isospin = data%flv_in(i)%get_isospin_type ()
      if (isospin > 0) then
         !!! up-type quark or neutrinos
         if (data%flv_in(i)%is_antiparticle ()) then
            qn_w = qn_wm
         else
            qn_w = qn_wp
         end if
      else
         !!! down-type quark or lepton
         if (data%flv_in(i)%is_antiparticle ()) then
            qn_w = qn_wp
         else
            qn_w = qn_wm
         end if
      end if
      call pol%init_generic (data%flv_in(i))
```

```
            call qn_fc(1)%init ( &
                 flv = data%flv_in(i), &
                 col = color_from_flavor (data%flv_in(i), 1))
            call qn_fc_fin(1)%init ( &
                 flv = data%flv_out(i), &
                 col = color_from_flavor (data%flv_out(i), 1))
            call it_hel%init (pol)
            do while (it_hel%is_valid ())
               qn_hel = it_hel%get_quantum_numbers ()
               qn = qn_hel .merge. qn_fc(1)
               qn_rad = qn_hel .merge. qn_fc_fin(1)
               call qn_rad%tag_radiated ()
               call sf_int%add_state ([qn, qn_rad, qn_w])
               call it_hel%advance ()
            end do
            ! call pol%final ()
         end do
      case default
         call msg_fatal ("EWA initialization failed: wrong particle type.")
      end select
      call sf_int%freeze ()
      if (data%keep_energy) then
         sf_int%on_shell_mode = KEEP_ENERGY
      else
         sf_int%on_shell_mode = KEEP_MOMENTUM
      end if
      call sf_int%set_incoming ([1])
      call sf_int%set_radiated ([2])
      call sf_int%set_outgoing ([3])
   end select
end subroutine ewa_init
```

Prepare the coupling arrays. This is separate from the previous routine since the state matrix may be helicity-contracted.

⟨*SF ewa: ewa: TBP*⟩+≡

```
procedure :: setup_constants => ewa_setup_constants
```

⟨*SF ewa: procedures*⟩+≡

```
subroutine ewa_setup_constants (sf_int)
  class(ewa_t), intent(inout), target :: sf_int
  type(state_iterator_t) :: it
  type(flavor_t) :: flv
  real(default) :: q, t3
  integer :: i
  sf_int%n_me = sf_int%get_n_matrix_elements ()
  allocate (sf_int%cv (sf_int%n_me))
  allocate (sf_int%ca (sf_int%n_me))
  associate (data => sf_int%data)
    select case (data%id)
    case (23)
       call it%init (sf_int%interaction_t%get_state_matrix_ptr ())
       do while (it%is_valid ())
          i = it%get_me_index ()
          flv = it%get_flavor (1)
```

```
            q = flv%get_charge ()
            t3 = flv%get_isospin ()
            if (flv%is_antiparticle ()) then
               sf_int%cv(i) = - data%cv &
                    * (t3 - 2._default * q * data%sinthw**2) / data%costhw
               sf_int%ca(i) = data%ca *  t3 / data%costhw
            else
               sf_int%cv(i) = data%cv &
                    * (t3 - 2._default * q * data%sinthw**2) / data%costhw
               sf_int%ca(i) = data%ca *  t3 / data%costhw
            end if
            call it%advance ()
         end do
      case (24)
         call it%init (sf_int%interaction_t%get_state_matrix_ptr ())
         do while (it%is_valid ())
            i = it%get_me_index ()
            flv = it%get_flavor (1)
            if (flv%is_antiparticle ()) then
               sf_int%cv(i) = data%cv / sqrt(2._default)
               sf_int%ca(i) = - data%ca / sqrt(2._default)
            else
               sf_int%cv(i) = data%cv / sqrt(2._default)
               sf_int%ca(i) = data%ca / sqrt(2._default)
            end if
            call it%advance ()
         end do
      end select
    end associate
    sf_int%status = SF_INITIAL
  end subroutine ewa_setup_constants
```

15.8.4 Kinematics

Set kinematics. The EWA structure function allows for a straightforward mapping of the unit interval. So, to leading order, the structure function value is unity, but the x value is transformed. Higher orders affect the function value. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

If `map` is set, the exponential mapping for the $1/x$ singularity discussed above is applied.

⟨*SF ewa: ewa: TBP*⟩+≡

```
procedure :: complete_kinematics => ewa_complete_kinematics
```

⟨*SF ewa: procedures*⟩+≡

```
subroutine ewa_complete_kinematics (sf_int, x, f, r, rb, map)
  class(ewa_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  real(default), dimension(:), intent(in) :: rb
  logical, intent(in) :: map
  real(default) :: xb1, e_1
```

```
    real(default) :: x0, x1, lx0, lx1, lx
    e_1 = energy (sf_int%get_momentum (1))
    if (sf_int%data%recoil) then
       select case (sf_int%data%id)
       case (23)
          x0 = max (sf_int%data%x_min, sf_int%data%mz / e_1)
       case (24)
          x0 = max (sf_int%data%x_min, sf_int%data%mw / e_1)
       end select
    else
       x0 = sf_int%data%x_min
    end if
    x1 = sf_int%data%x_max
    if ( x0 >= x1) then
       f = 0
       sf_int%status = SF_FAILED_KINEMATICS
       return
    end if
    if (map) then
       lx0 = log (x0)
       lx1 = log (x1)
       lx = lx1 * r(1) + lx0 * rb(1)
       x(1) = exp(lx)
       f = x(1) * (lx1 - lx0)
    else
       x(1) = r(1)
       if (x0 < x(1) .and. x(1) < x1) then
          f = 1
       else
          sf_int%status = SF_FAILED_KINEMATICS
          f = 0
          return
       end if
    end if
    xb1 = 1 - x(1)
    if (size(x) == 3)  x(2:3) = r(2:3)
    call sf_int%split_momentum (x, xb1)
    select case (sf_int%status)
    case (SF_DONE_KINEMATICS)
       sf_int%x  = x(1)
       sf_int%xb = xb1
    case (SF_FAILED_KINEMATICS)
       sf_int%x  = 0
       sf_int%xb = 0
       f = 0
    end select
  end subroutine ewa_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF ewa: ewa: TBP*⟩+≡

```
  procedure :: inverse_kinematics => ewa_inverse_kinematics
```

⟨*SF ewa: procedures*⟩+≡

```
subroutine ewa_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
  class(ewa_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  real(default), dimension(:), intent(out) :: rb
  logical, intent(in) :: map
  logical, intent(in), optional :: set_momenta
  real(default) :: x0, x1, lx0, lx1, lx, e_1
  logical :: set_mom
  set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
  e_1 = energy (sf_int%get_momentum (1))
  if (sf_int%data%recoil) then
     select case (sf_int%data%id)
     case (23)
        x0 = max (sf_int%data%x_min, sf_int%data%mz / e_1)
     case (24)
        x0 = max (sf_int%data%x_min, sf_int%data%mw / e_1)
     end select
  else
     x0 = sf_int%data%x_min
  end if
  x1 = sf_int%data%x_max
  if (map) then
     lx0 = log (x0)
     lx1 = log (x1)
     lx = log (x(1))
     r(1)  = (lx - lx0) / (lx1 - lx0)
     rb(1) = (lx1 - lx) / (lx1 - lx0)
     f = x(1) * (lx1 - lx0)
  else
     r (1) = x(1)
     rb(1) = 1 - x(1)
     if (x0 < x(1) .and. x(1) < x1) then
        f = 1
     else
        f = 0
     end if
  end if
  if (size(r) == 3) then
     r (2:3) = x(2:3)
     rb(2:3) = 1 - x(2:3)
  end if
  if (set_mom) then
     call sf_int%split_momentum (x, sf_int%xb)
     select case (sf_int%status)
     case (SF_DONE_KINEMATICS)
        sf_int%x  = x(1)
        sf_int%xb = 1 - x(1)
     case (SF_FAILED_KINEMATICS)
        sf_int%x = 0
        f = 0
     end select
```

```
    end if
  end subroutine ewa_inverse_kinematics
```

15.8.5 EWA application

For EWA, we can compute kinematics and function value in a single step. This function works on a single beam, assuming that the input momentum has been set. We need four random numbers as input: one for x, one for Q^2, and two for the polar and azimuthal angles. Alternatively, we can skip p_T generation; in this case, we only need one.

For obtaining splitting kinematics, we rely on the assumption that all in-particles are mass-degenerate (or there is only one), so the generated x values are identical.

⟨*SF ewa: ewa: TBP*⟩+≡

```
  procedure :: apply => ewa_apply
```

⟨*SF ewa: procedures*⟩+≡

```
  subroutine ewa_apply (sf_int, scale)
    class(ewa_t), intent(inout) :: sf_int
    real(default), intent(in) :: scale
    real(default) :: x, xb, pt2, c1, c2
    real(default) :: cv, ca
    real(default) :: f, fm, fp, fL
    integer :: i
    associate (data => sf_int%data)
      x  = sf_int%x
      xb = sf_int%xb
      pt2 = min ((data%pt_max)**2, (xb * data%sqrts / 2)**2)
      select case (data%id)
      case (23)
         !!! Z boson structure function
         c1 = log (1 + pt2 / (xb * (data%mZ)**2))
         c2 = 1 / (1 + (xb * (data%mZ)**2) / pt2)
      case (24)
         !!! W boson structure function
         c1 = log (1 + pt2 / (xb * (data%mW)**2))
         c2 = 1 / (1 + (xb * (data%mW)**2) / pt2)
      end select
      do i = 1, sf_int%n_me
         cv = sf_int%cv(i)
         ca = sf_int%ca(i)
         fm = data%coeff * &
              ((cv + ca)**2 + ((cv - ca) * xb)**2) * (c1 - c2) / (2 * x)
         fp = data%coeff * &
              ((cv - ca)**2 + ((cv + ca) * xb)**2) * (c1 - c2) / (2 * x)
         fL = data%coeff * &
              (cv**2 + ca**2) * (2 * xb / x) * c2
         f = fp + fm + fL
         if (.not. vanishes (f)) then
            fp = fp / f
            fm = fm / f
            fL = fL / f
```

```
        end if
        call sf_int%set_matrix_element (i, cmplx (f, kind=default))
      end do
    end associate
    sf_int%status = SF_EVALUATED
  end subroutine ewa_apply
```

15.8.6 Unit tests

Test module, followed by the corresponding implementation module.

⟨sf_ewa_ut.f90⟩≡

```
⟨File header⟩

module sf_ewa_ut
  use unit_tests
  use sf_ewa_uti

⟨Standard module head⟩

⟨SF ewa: public test⟩

contains

⟨SF ewa: test driver⟩

end module sf_ewa_ut
```

⟨sf_ewa_uti.f90⟩≡

```
⟨File header⟩

module sf_ewa_uti

⟨Use kinds⟩
  use lorentz
  use pdg_arrays
  use flavors
  use interactions, only: reset_interaction_counter
  use interactions, only: interaction_pacify_momenta
  use model_data
  use sf_aux
  use sf_base

  use sf_ewa

⟨Standard module head⟩

⟨SF ewa: test declarations⟩

contains

⟨SF ewa: tests⟩

end module sf_ewa_uti
```

API: driver for the unit tests below.

⟨*SF ewa: public test*⟩≡
```
public :: sf_ewa_test
```

⟨*SF ewa: test driver*⟩≡
```
subroutine sf_ewa_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨SF ewa: execute tests⟩
end subroutine sf_ewa_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF ewa: execute tests*⟩≡
```
call test (sf_ewa_1, "sf_ewa_1", &
     "structure function configuration", &
     u, results)
```

⟨*SF ewa: test declarations*⟩≡
```
public :: sf_ewa_1
```

⟨*SF ewa: tests*⟩≡
```
subroutine sf_ewa_1 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(pdg_array_t) :: pdg_in
  type(pdg_array_t), dimension(1) :: pdg_out
  integer, dimension(:), allocatable :: pdg1
  class(sf_data_t), allocatable :: data

  write (u, "(A)")  "* Test output: sf_ewa_1"
  write (u, "(A)")  "*   Purpose: initialize and display &
       &test structure function data"
  write (u, "(A)")

  write (u, "(A)")  "* Create empty data object"
  write (u, "(A)")

  call model%init_sm_test ()
  pdg_in = 2

  allocate (ewa_data_t :: data)
  call data%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize for Z boson"
  write (u, "(A)")

  select type (data)
  type is (ewa_data_t)
     call data%init (model, pdg_in, 0.01_default, &
          500._default, 5000._default, .false., .false.)
     call data%set_id (23)
```

```
    end select

    call data%write (u)

    write (u, "(A)")

    write (u, "(1x,A)")  "Outgoing particle codes:"
    call data%get_pdg_out (pdg_out)
    pdg1 = pdg_out(1)
    write (u, "(2x,99(1x,I0))")  pdg1

    write (u, "(A)")
    write (u, "(A)")  "* Initialize for W boson"
    write (u, "(A)")

    deallocate (data)
    allocate (ewa_data_t :: data)
    select type (data)
    type is (ewa_data_t)
       call data%init (model, pdg_in, 0.01_default, &
            500._default, 5000._default, .false., .false.)
       call data%set_id (24)
    end select

    call data%write (u)

    write (u, "(A)")

    write (u, "(1x,A)")  "Outgoing particle codes:"
    call data%get_pdg_out (pdg_out)
    pdg1 = pdg_out(1)
    write (u, "(2x,99(1x,I0))")  pdg1

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_ewa_1"

  end subroutine sf_ewa_1
```

Test and probe structure function

Construct and display a structure function object based on the EWA structure function.

⟨*SF ewa: execute tests*⟩+≡

```
  call test (sf_ewa_2, "sf_ewa_2", &
       "structure function instance", &
       u, results)
```

⟨*SF ewa: test declarations*⟩+≡

```
  public :: sf_ewa_2
```

⟨*SF ewa: tests*⟩+≡

```
subroutine sf_ewa_2 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t) :: flv
  type(pdg_array_t) :: pdg_in
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k
  type(vector4_t), dimension(2) :: q
  real(default) :: E
  real(default), dimension(:), allocatable :: r, rb, x
  real(default) :: f

  write (u, "(A)")  "* Test output: sf_ewa_2"
  write (u, "(A)")  "*   Purpose: initialize and fill &
       &test structure function object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call model%init_sm_test ()
  call flv%init (2, model)
  pdg_in = 2

  call reset_interaction_counter ()

  allocate (ewa_data_t :: data)
  select type (data)
  type is (ewa_data_t)
     call data%init (model, pdg_in, 0.01_default, &
          500._default, 3000._default, .false., .true.)
     call data%set_id (24)
  end select

  write (u, "(A)")  "* Initialize structure-function object"
  write (u, "(A)")

  call data%allocate_sf_int (sf_int)
  call sf_int%init (data)
  call sf_int%set_beam_index ([1])
  call sf_int%setup_constants ()

  call sf_int%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize incoming momentum with E=1500"
  write (u, "(A)")
  E = 1500
  k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
  call pacify (k, 1e-10_default)
  call vector4_write (k, u)
  call sf_int%seed_kinematics ([k])
```

```
write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.4, no EWA mapping, collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.4_default
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = sf_int%get_momenta (outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

call sf_int%seed_kinematics ([k])
call sf_int%set_momenta (q, outgoing=.true.)
call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.false., &
     set_momenta=.true.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Evaluate EWA structure function"
write (u, "(A)")

call sf_int%apply (scale = 100._default)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call sf_int%final ()
call model%final ()
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_ewa_2"

  end subroutine sf_ewa_2
```

Standard mapping

Construct and display a structure function object based on the EWA structure function, applying the standard single-particle mapping.

⟨*SF ewa: execute tests*⟩+≡

```
  call test (sf_ewa_3, "sf_ewa_3", &
       "apply mapping", &
       u, results)
```

⟨*SF ewa: test declarations*⟩+≡

```
  public :: sf_ewa_3
```

⟨*SF ewa: tests*⟩+≡

```
  subroutine sf_ewa_3 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t) :: flv
    type(pdg_array_t) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q
    real(default) :: E
    real(default), dimension(:), allocatable :: r, rb, x
    real(default) :: f

    write (u, "(A)")  "* Test output: sf_ewa_3"
    write (u, "(A)")  "*   Purpose: initialize and fill &
         &test structure function object"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize configuration data"
    write (u, "(A)")

    call model%init_sm_test ()
    call flv%init (2, model)
    pdg_in = 2

    call reset_interaction_counter ()

    allocate (ewa_data_t :: data)
    select type (data)
    type is (ewa_data_t)
       call data%init (model, pdg_in, 0.01_default, &
            500._default, 3000._default, .false., .true.)
       call data%set_id (24)
    end select
```

```
write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=1500"
write (u, "(A)")
E = 1500
k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.4, with EWA mapping, collinear"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.4_default
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.true.)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = sf_int%get_momenta (outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

call sf_int%seed_kinematics ([k])
call sf_int%set_momenta (q, outgoing=.true.)
call sf_int%recover_x (x)
call sf_int%inverse_kinematics (x, f, r, rb, map=.true., &
     set_momenta=.true.)
```

```
    write (u, "(A,9(1x,F10.7))")  "r =", r
    write (u, "(A,9(1x,F10.7))")  "rb=", rb
    write (u, "(A,9(1x,F10.7))")  "x =", x
    write (u, "(A,9(1x,F10.7))")  "f =", f

    write (u, "(A)")
    write (u, "(A)")  "* Evaluate EWA structure function"
    write (u, "(A)")

    call sf_int%apply (scale = 100._default)
    call sf_int%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_ewa_3"

  end subroutine sf_ewa_3
```

Non-collinear case

Construct and display a structure function object based on the EPA structure function.

⟨*SF ewa: execute tests*⟩+≡

```
  call test (sf_ewa_4, "sf_ewa_4", &
       "non-collinear", &
       u, results)
```

⟨*SF ewa: test declarations*⟩+≡

```
  public :: sf_ewa_4
```

⟨*SF ewa: tests*⟩+≡

```
  subroutine sf_ewa_4 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t) :: flv
    type(pdg_array_t) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k
    type(vector4_t), dimension(2) :: q
    real(default) :: E
    real(default), dimension(:), allocatable :: r, rb, x
    real(default) :: f

    write (u, "(A)")  "* Test output: sf_ewa_4"
    write (u, "(A)")  "*   Purpose: initialize and fill &
         &test structure function object"
```

```
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call modeL%init_sm_test ()
call flv%init (2, model)
pdg_in = 2

call reset_interaction_counter ()

allocate (ewa_data_t :: data)
select type (data)
type is (ewa_data_t)
   call data%init (model, pdg_in, 0.01_default, &
        500._default, 3000.0_default, .true., .true.)
   call data%set_id (24)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])
call sf_int%setup_constants ()

write (u, "(A)")  "* Initialize incoming momentum with E=1500"
write (u, "(A)")
E = 1500
k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call pacify (k, 1e-10_default)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for r=0.5/0.5/0.25, with EWA mapping, "
write (u, "(A)")  "          non-coll., keeping energy"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = [0.5_default, 0.5_default, 0.25_default]
rb = 1 - r
sf_int%on_shell_mode = KEEP_ENERGY
call sf_int%complete_kinematics (x, f, r, rb, map=.true.)
call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f
```

```
  write (u, "(A)")
  write (u, "(A)")  "* Recover x and r from momenta"
  write (u, "(A)")

  q = sf_int%get_momenta (outgoing=.true.)
  call sf_int%final ()
  deallocate (sf_int)

  call data%allocate_sf_int (sf_int)
  call sf_int%init (data)
  call sf_int%set_beam_index ([1])
  call sf_int%setup_constants ()

  call sf_int%seed_kinematics ([k])
  call sf_int%set_momenta (q, outgoing=.true.)
  call sf_int%recover_x (x)
  call sf_int%inverse_kinematics (x, f, r, rb, map=.true., &
       set_momenta=.true.)
  call interaction_pacify_momenta (sf_int%interaction_t, 1e-10_default)

  write (u, "(A,9(1x,F10.7))")  "r =", r
  write (u, "(A,9(1x,F10.7))")  "rb=", rb
  write (u, "(A,9(1x,F10.7))")  "x =", x
  write (u, "(A,9(1x,F10.7))")  "f =", f

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate EWA structure function"
  write (u, "(A)")

  call sf_int%apply (scale = 1500._default)
  call sf_int%write (u, testflag = .true.)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call sf_int%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_ewa_4"

end subroutine sf_ewa_4
```

Structure function for multiple flavors

Construct and display a structure function object based on the EWA structure function. The incoming state has multiple particles with non-uniform quantum numbers.

⟨*SF ewa: execute tests*⟩+≡

```
call test (sf_ewa_5, "sf_ewa_5", &
     "structure function instance", &
```

```
        u, results)
```

⟨*SF ewa: test declarations*⟩+≡

```
    public :: sf_ewa_5
```

⟨*SF ewa: tests*⟩+≡

```
    subroutine sf_ewa_5 (u)
      integer, intent(in) :: u
      type(model_data_t), target :: model
      type(flavor_t) :: flv
      type(pdg_array_t) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t) :: k
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: f

      write (u, "(A)")  "* Test output: sf_ewa_5"
      write (u, "(A)")  "*   Purpose: initialize and fill &
           &test structure function object"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call model%init_sm_test ()
      call flv%init (2, model)
      pdg_in = [1, 2, -1, -2]

      call reset_interaction_counter ()

      allocate (ewa_data_t :: data)
      select type (data)
      type is (ewa_data_t)
         call data%init (model, pdg_in, 0.01_default, &
              500._default, 3000._default, .false., .true.)
         call data%set_id (24)
      end select

      write (u, "(A)")  "* Initialize structure-function object"
      write (u, "(A)")

      call data%allocate_sf_int (sf_int)
      call sf_int%init (data)
      call sf_int%set_beam_index ([1])
      call sf_int%setup_constants ()

      call sf_int%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize incoming momentum with E=1500"
      write (u, "(A)")
      E = 1500
      k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
```

```
  call pacify (k, 1e-10_default)
  call vector4_write (k, u)
  call sf_int%seed_kinematics ([k])

  write (u, "(A)")
  write (u, "(A)")  "* Set kinematics for r=0.4, no EWA mapping, collinear"
  write (u, "(A)")

  allocate (r (data%get_n_par ()))
  allocate (rb(size (r)))
  allocate (x (size (r)))

  r = 0.4_default
  rb = 1 - r
  call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

  write (u, "(A,9(1x,F10.7))")  "r =", r
  write (u, "(A,9(1x,F10.7))")  "rb=", rb
  write (u, "(A,9(1x,F10.7))")  "x =", x
  write (u, "(A,9(1x,F10.7))")  "f =", f

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate EWA structure function"
  write (u, "(A)")

  call sf_int%apply (scale = 100._default)
  call sf_int%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call sf_int%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_ewa_5"

end subroutine sf_ewa_5
```

15.9 Energy-scan spectrum

This spectrum is actually a trick that allows us to plot the c.m. energy dependence of a cross section without scanning the input energy. We start with the observation that a spectrum $f(x)$, applied to one of the incoming beams only, results in a cross section

$$\sigma = \int dx\, f(x)\, \hat{\sigma}(xs). \tag{15.70}$$

We want to compute the distribution of $E = \sqrt{\hat{s}} = \sqrt{xs}$, i.e.,

$$\frac{d\sigma}{dE} = \frac{2\sqrt{x}}{\sqrt{s}}\,\frac{d\sigma}{dx} = \frac{2\sqrt{x}}{\sqrt{s}}\, f(x)\, \hat{\sigma}(xs), \tag{15.71}$$

so if we set

$$f(x) = \frac{\sqrt{s}}{2\sqrt{x}}, \tag{15.72}$$

we get the distribution

$$\frac{d\sigma}{dE} = \hat{\sigma}(\hat{s} = E^2). \tag{15.73}$$

We implement this as a spectrum with a single parameter x. The parameters for the individual beams are computed as $x_i = \sqrt{x}$, so they are equal and the kinematics is always symmetric.

⟨sf_escan.f90⟩≡

```
⟨File header⟩

module sf_escan

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_defs, only: FMT_12
  use numeric_utils
  use diagnostics
  use lorentz
  use pdg_arrays
  use model_data
  use flavors
  use quantum_numbers
  use state_matrices
  use polarizations
  use sf_base

⟨Standard module head⟩

⟨SF escan: public⟩

⟨SF escan: types⟩

contains

⟨SF escan: procedures⟩

end module sf_escan
```

15.9.1 Data type

The `norm` is unity if the total cross section should be normalized to one, and $\sqrt{s}$ if it should be normalized to the total energy. In the latter case, the differential distribution $d\sigma/d\sqrt{\hat{s}}$ coincides with the partonic cross section $\hat{\sigma}$ as a function of $\sqrt{\hat{s}}$.

⟨*SF escan: public*⟩≡
```
public :: escan_data_t
```

⟨*SF escan: types*⟩≡
```
type, extends(sf_data_t) :: escan_data_t
   private
   type(flavor_t), dimension(:,:), allocatable :: flv_in
   integer, dimension(2) :: n_flv = 0
   real(default) :: norm = 1
 contains
 ⟨SF escan: escan data: TBP⟩
end type escan_data_t
```

⟨*SF escan: escan data: TBP*⟩≡
```
procedure :: init => escan_data_init
```

⟨*SF escan: procedures*⟩≡
```
subroutine escan_data_init (data, model, pdg_in, norm)
  class(escan_data_t), intent(out) :: data
  class(model_data_t), intent(in), target :: model
  type(pdg_array_t), dimension(2), intent(in) :: pdg_in
  real(default), intent(in), optional :: norm
  real(default), dimension(2) :: m2
  integer :: i, j
  data%n_flv = pdg_array_get_length (pdg_in)
  allocate (data%flv_in (maxval (data%n_flv), 2))
  do i = 1, 2
     do j = 1, data%n_flv(i)
        call data%flv_in(j, i)%init (pdg_array_get (pdg_in(i), j), model)
     end do
  end do
  m2 = data%flv_in(1,:)%get_mass ()
  do i = 1, 2
     if (.not. any (nearly_equal (data%flv_in(1:data%n_flv(i),i)%get_mass (), m2(i)))) then
        call msg_fatal ("Energy scan: incoming particle mass must be uniform")
     end if
  end do
  if (present (norm))  data%norm = norm
end subroutine escan_data_init
```

Output

⟨*SF escan: escan data: TBP*⟩+≡
```
procedure :: write => escan_data_write
```

⟨*SF escan: procedures*⟩+≡
```
subroutine escan_data_write (data, unit, verbose)
  class(escan_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
```

```
    logical, intent(in), optional :: verbose
    integer :: u, i, j
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(1x,A)") "Energy-scan data:"
    write (u, "(3x,A)", advance="no")  "prt_in = "
    do i = 1, 2
       if (i > 1)  write (u, "(',',1x)", advance="no")
       do j = 1, data%n_flv(i)
          if (j > 1)  write (u, "(':')", advance="no")
          write (u, "(A)", advance="no")  char (data%flv_in(j,i)%get_name ())
       end do
    end do
    write (u, *)
    write (u, "(3x,A," // FMT_12 // ")")  "norm   =", data%norm
  end subroutine escan_data_write
```

Kinematics is completely collinear, hence there is only one parameter for a pair spectrum.

⟨*SF escan: escan data: TBP*⟩+≡

```
  procedure :: get_n_par => escan_data_get_n_par
```

⟨*SF escan: procedures*⟩+≡

```
  function escan_data_get_n_par (data) result (n)
    class(escan_data_t), intent(in) :: data
    integer :: n
    n = 1
  end function escan_data_get_n_par
```

Return the outgoing particles PDG codes. This is always the same as the incoming particle, where we use two indices for the two beams.

⟨*SF escan: escan data: TBP*⟩+≡

```
  procedure :: get_pdg_out => escan_data_get_pdg_out
```

⟨*SF escan: procedures*⟩+≡

```
  subroutine escan_data_get_pdg_out (data, pdg_out)
    class(escan_data_t), intent(in) :: data
    type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
    integer :: i, n
    n = 2
    do i = 1, n
       pdg_out(i) = data%flv_in(1:data%n_flv(i),i)%get_pdg ()
    end do
  end subroutine escan_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF escan: escan data: TBP*⟩+≡

```
  procedure :: allocate_sf_int => escan_data_allocate_sf_int
```

⟨*SF escan: procedures*⟩+≡

```
  subroutine escan_data_allocate_sf_int (data, sf_int)
    class(escan_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (escan_t :: sf_int)
```

```
end subroutine escan_data_allocate_sf_int
```

15.9.2 The Energy-scan object

This is a spectrum, not a radiation. We create an interaction with two incoming and two outgoing particles, flavor, color, and helicity being carried through. x nevertheless is only one-dimensional, as we are always using only one beam parameter.

⟨*SF escan: types*⟩+≡

```
type, extends (sf_int_t) :: escan_t
   type(escan_data_t), pointer :: data => null ()
 contains
 ⟨SF escan: escan: TBP⟩
end type escan_t
```

Type string: for the energy scan this is just a dummy function.

⟨*SF escan: escan: TBP*⟩≡

```
procedure :: type_string => escan_type_string
```

⟨*SF escan: procedures*⟩+≡

```
function escan_type_string (object) result (string)
  class(escan_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "Escan: energy scan"
  else
     string = "Escan: [undefined]"
  end if
end function escan_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF escan: escan: TBP*⟩+≡

```
procedure :: write => escan_write
```

⟨*SF escan: procedures*⟩+≡

```
subroutine escan_write (object, unit, testflag)
  class(escan_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     call object%base_write (u, testflag)
  else
     write (u, "(1x,A)")  "Energy scan data: [undefined]"
  end if
end subroutine escan_write
```

⟨*SF escan: escan: TBP*⟩+≡

```
procedure :: init => escan_init
```

⟨*SF escan: procedures*⟩+≡

```
subroutine escan_init (sf_int, data)
  class(escan_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  type(quantum_numbers_mask_t), dimension(4) :: mask
  integer, dimension(4) :: hel_lock
  real(default), dimension(2) :: m2
  real(default), dimension(0) :: mr2
  type(quantum_numbers_t), dimension(4) :: qn_fc, qn_hel, qn
  type(polarization_t), target :: pol1, pol2
  type(polarization_iterator_t) :: it_hel1, it_hel2
  integer :: j1, j2
  select type (data)
  type is (escan_data_t)
     hel_lock = [3, 4, 1, 2]
     m2 = data%flv_in(1,:)%get_mass ()
     call sf_int%base_init (mask, m2, mr2, m2, hel_lock = hel_lock)
     sf_int%data => data
     do j1 = 1, data%n_flv(1)
        call qn_fc(1)%init ( &
             flv = data%flv_in(j1,1), &
             col = color_from_flavor (data%flv_in(j1,1)))
        call qn_fc(3)%init ( &
             flv = data%flv_in(j1,1), &
             col = color_from_flavor (data%flv_in(j1,1)))
        call pol1%init_generic (data%flv_in(j1,1))
        do j2 = 1, data%n_flv(2)
           call qn_fc(2)%init ( &
                flv = data%flv_in(j2,2), &
                col = color_from_flavor (data%flv_in(j2,2)))
           call qn_fc(4)%init ( &
                flv = data%flv_in(j2,2), &
                col = color_from_flavor (data%flv_in(j2,2)))
           call pol2%init_generic (data%flv_in(j2,2))
           call it_hel1%init (pol1)
           do while (it_hel1%is_valid ())
              qn_hel(1) = it_hel1%get_quantum_numbers ()
              qn_hel(3) = it_hel1%get_quantum_numbers ()
              call it_hel2%init (pol2)
              do while (it_hel2%is_valid ())
                 qn_hel(2) = it_hel2%get_quantum_numbers ()
                 qn_hel(4) = it_hel2%get_quantum_numbers ()
                 qn = qn_hel .merge. qn_fc
                 call sf_int%add_state (qn)
                 call it_hel2%advance ()
              end do
              call it_hel1%advance ()
           end do
           ! call pol2%final ()
        end do
        ! call pol1%final ()
     end do
     call sf_int%set_incoming ([1,2])
     call sf_int%set_outgoing ([3,4])
```

```
      call sf_int%freeze ()
      sf_int%status = SF_INITIAL
    end select
  end subroutine escan_init
```

15.9.3 Kinematics

Set kinematics. We have a single parameter, but reduce both beams. The `map` flag is ignored.

⟨*SF escan: escan: TBP*⟩+≡

```
  procedure :: complete_kinematics => escan_complete_kinematics
```

⟨*SF escan: procedures*⟩+≡

```
  subroutine escan_complete_kinematics (sf_int, x, f, r, rb, map)
    class(escan_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: x
    real(default), intent(out) :: f
    real(default) :: sqrt_x
    real(default), dimension(:), intent(in) :: r
    real(default), dimension(:), intent(in) :: rb
    logical, intent(in) :: map
    x = r
    sqrt_x = sqrt (x(1))
    if (sqrt_x > 0) then
       f = 1 / (2 * sqrt_x)
    else
       f = 0
       sf_int%status = SF_FAILED_KINEMATICS
       return
    end if
    call sf_int%reduce_momenta ([sqrt_x, sqrt_x])
  end subroutine escan_complete_kinematics
```

Recover x. The base procedure should return two momentum fractions for the two beams, while we have only one parameter. This is the product of the extracted momentum fractions.

⟨*SF escan: escan: TBP*⟩+≡

```
  procedure :: recover_x => escan_recover_x
```

⟨*SF escan: procedures*⟩+≡

```
  subroutine escan_recover_x (sf_int, x, x_free)
    class(escan_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: x
    real(default), intent(inout), optional :: x_free
    real(default), dimension(2) :: xi
    call sf_int%base_recover_x (xi, x_free)
    x = product (xi)
  end subroutine escan_recover_x
```

Compute inverse kinematics.

⟨*SF escan: escan: TBP*⟩+≡

```
  procedure :: inverse_kinematics => escan_inverse_kinematics
```

⟨*SF escan: procedures*⟩+≡

```
subroutine escan_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
  class(escan_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  real(default), dimension(:), intent(out) :: rb
  logical, intent(in) :: map
  logical, intent(in), optional :: set_momenta
  real(default) :: sqrt_x
  logical :: set_mom
  set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
  sqrt_x = sqrt (x(1))
  if (sqrt_x > 0) then
     f = 1 / (2 * sqrt_x)
  else
     f = 0
     sf_int%status = SF_FAILED_KINEMATICS
     return
  end if
  r = x
  rb = 1 - r
  if (set_mom) then
     call sf_int%reduce_momenta ([sqrt_x, sqrt_x])
  end if
end subroutine escan_inverse_kinematics
```

15.9.4 Energy scan application

Here, we insert the predefined norm.

⟨*SF escan: escan: TBP*⟩+≡

```
procedure :: apply => escan_apply
```

⟨*SF escan: procedures*⟩+≡

```
subroutine escan_apply (sf_int, scale)
  class(escan_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  real(default) :: f
  associate (data => sf_int%data)
    f = data%norm
  end associate
  call sf_int%set_matrix_element (cmplx (f, kind=default))
  sf_int%status = SF_EVALUATED
end subroutine escan_apply
```

15.9.5 Unit tests

Test module, followed by the corresponding implementation module.

⟨`sf_escan_ut.f90`⟩≡

⟨*File header*⟩

```
module sf_escan_ut
  use unit_tests
  use sf_escan_uti

⟨Standard module head⟩

⟨SF escan: public test⟩

contains

⟨SF escan: test driver⟩

end module sf_escan_ut
```

⟨sf_escan_uti.f90⟩≡

```
⟨File header⟩

module sf_escan_uti

⟨Use kinds⟩
  use physics_defs, only: ELECTRON
  use lorentz
  use pdg_arrays
  use flavors
  use interactions, only: reset_interaction_counter
  use model_data
  use sf_aux
  use sf_base

  use sf_escan

⟨Standard module head⟩

⟨SF escan: test declarations⟩

contains

⟨SF escan: tests⟩

end module sf_escan_uti
```

API: driver for the unit tests below.

⟨*SF escan: public test*⟩≡

```
  public :: sf_escan_test
```

⟨*SF escan: test driver*⟩≡

```
  subroutine sf_escan_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨SF escan: execute tests⟩
  end subroutine sf_escan_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF escan: execute tests*⟩≡

```
call test (sf_escan_1, "sf_escan_1", &
     "structure function configuration", &
     u, results)
```

⟨*SF escan: test declarations*⟩≡

```
public :: sf_escan_1
```

⟨*SF escan: tests*⟩≡

```
subroutine sf_escan_1 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(pdg_array_t), dimension(2) :: pdg_in
  type(pdg_array_t), dimension(2) :: pdg_out
  integer, dimension(:), allocatable :: pdg1, pdg2
  class(sf_data_t), allocatable :: data

  write (u, "(A)")  "* Test output: sf_escan_1"
  write (u, "(A)")  "*   Purpose: initialize and display &
       &energy-scan structure function data"
  write (u, "(A)")

  call model%init_qed_test ()
  pdg_in(1) = ELECTRON
  pdg_in(2) = -ELECTRON

  allocate (escan_data_t :: data)
  select type (data)
  type is (escan_data_t)
     call data%init (model, pdg_in, norm = 2._default)
  end select

  call data%write (u)

  write (u, "(A)")

  write (u, "(1x,A)")  "Outgoing particle codes:"
  call data%get_pdg_out (pdg_out)
  pdg1 = pdg_out(1)
  pdg2 = pdg_out(2)
  write (u, "(2x,99(1x,I0))")  pdg1, pdg2

  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_escan_1"

end subroutine sf_escan_1
```

g@

Probe the structure-function object

Active the beam event reader, generate an event.

⟨*SF escan: execute tests*⟩+≡

```
call test (sf_escan_2, "sf_escan_2", &
     "generate event", &
     u, results)
```

⟨*SF escan: test declarations*⟩+≡

```
public :: sf_escan_2
```

⟨*SF escan: tests*⟩+≡

```
subroutine sf_escan_2 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t), dimension(2) :: flv
  type(pdg_array_t), dimension(2) :: pdg_in
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k1, k2
  real(default) :: E
  real(default), dimension(:), allocatable :: r, rb, x
  real(default) :: x_free, f

  write (u, "(A)")  "* Test output: sf_escan_2"
  write (u, "(A)")  "*   Purpose: initialize and display &
       &beam-events structure function data"
  write (u, "(A)")

  call model%init_qed_test ()
  call flv(1)%init (ELECTRON, model)
  call flv(2)%init (-ELECTRON, model)
  pdg_in(1) = ELECTRON
  pdg_in(2) = -ELECTRON

  call reset_interaction_counter ()

  allocate (escan_data_t :: data)
  select type (data)
  type is (escan_data_t)
     call data%init (model, pdg_in)
  end select

  write (u, "(A)")  "* Initialize structure-function object"
  write (u, "(A)")

  call data%allocate_sf_int (sf_int)
  call sf_int%init (data)
  call sf_int%set_beam_index ([1,2])

  write (u, "(A)")  "* Initialize incoming momentum with E=500"
  write (u, "(A)")
  E = 250
  k1 = vector4_moving (E, sqrt (E**2 - flv(1)%get_mass ()**2), 3)
  k2 = vector4_moving (E,-sqrt (E**2 - flv(2)%get_mass ()**2), 3)
  call vector4_write (k1, u)
  call vector4_write (k2, u)
  call sf_int%seed_kinematics ([k1, k2])
```

```
  write (u, "(A)")
  write (u, "(A)")  "* Set dummy parameters and generate x"
  write (u, "(A)")

  allocate (r (data%get_n_par ()))
  allocate (rb(size (r)))
  allocate (x (size (r)))

  r  = 0.8
  rb = 1 - r
  x_free = 1

  call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

  write (u, "(A,9(1x,F10.7))")  "r =", r
  write (u, "(A,9(1x,F10.7))")  "rb=", rb
  write (u, "(A,9(1x,F10.7))")  "x =", x
  write (u, "(A,9(1x,F10.7))")  "f =", f
  write (u, "(A,9(1x,F10.7))")  "xf=", x_free

  write (u, "(A)")
  write (u, "(A)")  "* Inverse kinematics"
  write (u, "(A)")

  call sf_int%recover_x (x, x_free)
  call sf_int%inverse_kinematics (x, f, r, rb, map=.false.)

  write (u, "(A,9(1x,F10.7))")  "r =", r
  write (u, "(A,9(1x,F10.7))")  "rb=", rb
  write (u, "(A,9(1x,F10.7))")  "x =", x
  write (u, "(A,9(1x,F10.7))")  "f =", f
  write (u, "(A,9(1x,F10.7))")  "xf=", x_free

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate"
  write (u, "(A)")

  call sf_int%apply (scale = 0._default)
  call sf_int%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call sf_int%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_escan_2"

end subroutine sf_escan_2
```

15.10 Gaussian beam spread

Instead of an analytic beam description, beam data may be provided in form of an event file. In its most simple form, the event file contains pairs of x values, relative to nominal beam energies. More advanced formats may include polarization, etc. The current implementation carries beam polarization through, if specified.

The code is very similar to the energy scan described above.

However, we must include a file-handle manager for the beam-event files. Two different processes may access a given beam-event file at the same time (i.e., serially but alternating). Accessing an open file from two different units is non-standard and not supported by all compilers. Therefore, we keep a global registry of open files, associated units, and reference counts. The `gaussian_t` objects act as proxies to this registry.

⟨`sf_gaussian.f90`⟩≡

```
⟨File header⟩

module sf_gaussian

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_defs, only: FMT_12
  use file_registries
  use diagnostics
  use lorentz
  use rng_base
  use pdg_arrays
  use model_data
  use flavors
  use quantum_numbers
  use state_matrices
  use polarizations
  use sf_base

⟨Standard module head⟩

⟨SF gaussian: public⟩

⟨SF gaussian: types⟩

contains

⟨SF gaussian: procedures⟩

end module sf_gaussian
```

15.10.1 The beam-data file registry

We manage data files via the `file_registries` module. To this end, we keep the registry as a private module variable here.

⟨*CCC SF gaussian: variables*⟩≡

```
type(file_registry_t), save :: beam_file_registry
```

15.10.2 Data type

We store the spread for each beam, as a relative number related to the beam energy. For the actual generation, we include an (abstract) random-number generator factory.

⟨*SF gaussian: public*⟩≡
```
public :: gaussian_data_t
```

⟨*SF gaussian: types*⟩≡
```
type, extends(sf_data_t) :: gaussian_data_t
   private
   type(flavor_t), dimension(2) :: flv_in
   real(default), dimension(2) :: spread
   class(rng_factory_t), allocatable :: rng_factory
 contains
 ⟨SF gaussian: gaussian data: TBP⟩
end type gaussian_data_t
```

⟨*SF gaussian: gaussian data: TBP*⟩≡
```
procedure :: init => gaussian_data_init
```

⟨*SF gaussian: procedures*⟩≡
```
subroutine gaussian_data_init (data, model, pdg_in, spread, rng_factory)
  class(gaussian_data_t), intent(out) :: data
  class(model_data_t), intent(in), target :: model
  type(pdg_array_t), dimension(2), intent(in) :: pdg_in
  real(default), dimension(2), intent(in) :: spread
  class(rng_factory_t), intent(inout), allocatable :: rng_factory
  if (any (spread < 0)) then
     call msg_fatal ("Gaussian beam spread: must not be negative")
  end if
  call data%flv_in(1)%init (pdg_array_get (pdg_in(1), 1), model)
  call data%flv_in(2)%init (pdg_array_get (pdg_in(2), 1), model)
  data%spread = spread
  call move_alloc (from = rng_factory, to = data%rng_factory)
end subroutine gaussian_data_init
```

Return true since this spectrum is always in generator mode.

⟨*SF gaussian: gaussian data: TBP*⟩+≡
```
procedure :: is_generator => gaussian_data_is_generator
```

⟨*SF gaussian: procedures*⟩+≡
```
function gaussian_data_is_generator (data) result (flag)
  class(gaussian_data_t), intent(in) :: data
  logical :: flag
  flag = .true.
end function gaussian_data_is_generator
```

The number of parameters is two. They are free parameters.

⟨*SF gaussian: gaussian data: TBP*⟩+≡
```
procedure :: get_n_par => gaussian_data_get_n_par
```

⟨*SF gaussian: procedures*⟩+≡

```
function gaussian_data_get_n_par (data) result (n)
  class(gaussian_data_t), intent(in) :: data
  integer :: n
  n = 2
end function gaussian_data_get_n_par
```

⟨*SF gaussian: gaussian data: TBP*⟩+≡

```
procedure :: get_pdg_out => gaussian_data_get_pdg_out
```

⟨*SF gaussian: procedures*⟩+≡

```
subroutine gaussian_data_get_pdg_out (data, pdg_out)
  class(gaussian_data_t), intent(in) :: data
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
  integer :: i, n
  n = 2
  do i = 1, n
     pdg_out(i) = data%flv_in(i)%get_pdg ()
  end do
end subroutine gaussian_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF gaussian: gaussian data: TBP*⟩+≡

```
procedure :: allocate_sf_int => gaussian_data_allocate_sf_int
```

⟨*SF gaussian: procedures*⟩+≡

```
subroutine gaussian_data_allocate_sf_int (data, sf_int)
  class(gaussian_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  allocate (gaussian_t :: sf_int)
end subroutine gaussian_data_allocate_sf_int
```

Output

⟨*SF gaussian: gaussian data: TBP*⟩+≡

```
procedure :: write => gaussian_data_write
```

⟨*SF gaussian: procedures*⟩+≡

```
subroutine gaussian_data_write (data, unit, verbose)
  class(gaussian_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)") "Gaussian beam spread data:"
  write (u, "(3x,A,A,A,A)") "prt_in = ", &
       char (data%flv_in(1)%get_name ()), &
       ", ", char (data%flv_in(2)%get_name ())
  write (u, "(3x,A,2(1x," // FMT_12 // "))") "spread =", data%spread
  call data%rng_factory%write (u)
end subroutine gaussian_data_write
```

15.10.3 The gaussian object

Flavor and polarization carried through, no radiated particles. The generator needs a random-number generator, obviously.

⟨*SF gaussian: public*⟩+≡
```
public :: gaussian_t
```

⟨*SF gaussian: types*⟩+≡
```
type, extends (sf_int_t) :: gaussian_t
   type(gaussian_data_t), pointer :: data => null ()
   class(rng_t), allocatable :: rng
 contains
 ⟨SF gaussian: gaussian: TBP⟩
end type gaussian_t
```

Type string: show gaussian file.

⟨*SF gaussian: gaussian: TBP*⟩≡
```
procedure :: type_string => gaussian_type_string
```

⟨*SF gaussian: procedures*⟩+≡
```
function gaussian_type_string (object) result (string)
  class(gaussian_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "Gaussian: gaussian beam-energy spread"
  else
     string = "Gaussian: [undefined]"
  end if
end function gaussian_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF gaussian: gaussian: TBP*⟩+≡
```
procedure :: write => gaussian_write
```

⟨*SF gaussian: procedures*⟩+≡
```
subroutine gaussian_write (object, unit, testflag)
  class(gaussian_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     call object%rng%write (u)
     call object%base_write (u, testflag)
  else
     write (u, "(1x,A)")  "gaussian data: [undefined]"
  end if
end subroutine gaussian_write
```

⟨*SF gaussian: gaussian: TBP*⟩+≡
```
procedure :: init => gaussian_init
```

⟨*SF gaussian: procedures*⟩+≡

```
subroutine gaussian_init (sf_int, data)
  class(gaussian_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  real(default), dimension(2) :: m2
  real(default), dimension(0) :: mr2
  type(quantum_numbers_mask_t), dimension(4) :: mask
  integer, dimension(4) :: hel_lock
  type(quantum_numbers_t), dimension(4) :: qn_fc, qn_hel, qn
  type(polarization_t), target :: pol1, pol2
  type(polarization_iterator_t) :: it_hel1, it_hel2
  integer :: i
  select type (data)
  type is (gaussian_data_t)
     m2 = data%flv_in%get_mass () ** 2
     hel_lock = [3, 4, 1, 2]
     mask = quantum_numbers_mask (.false., .false., .false.)
     call sf_int%base_init (mask, m2, mr2, m2, hel_lock = hel_lock)
     sf_int%data => data
     do i = 1, 2
        call qn_fc(i)%init ( &
             flv = data%flv_in(i), &
             col = color_from_flavor (data%flv_in(i)))
        call qn_fc(i+2)%init ( &
             flv = data%flv_in(i), &
             col = color_from_flavor (data%flv_in(i)))
     end do
     call pol1%init_generic (data%flv_in(1))
     call it_hel1%init (pol1)
     do while (it_hel1%is_valid ())
        qn_hel(1) = it_hel1%get_quantum_numbers ()
        qn_hel(3) = it_hel1%get_quantum_numbers ()
        call pol2%init_generic (data%flv_in(2))
        call it_hel2%init (pol2)
        do while (it_hel2%is_valid ())
           qn_hel(2) = it_hel2%get_quantum_numbers ()
           qn_hel(4) = it_hel2%get_quantum_numbers ()
           qn = qn_hel .merge. qn_fc
           call sf_int%add_state (qn)
           call it_hel2%advance ()
        end do
        ! call pol2%final ()
        call it_hel1%advance ()
     end do
     ! call pol1%final ()
     call sf_int%freeze ()
     call sf_int%set_incoming ([1,2])
     call sf_int%set_outgoing ([3,4])
     sf_int%status = SF_INITIAL
  end select
  call sf_int%data%rng_factory%make (sf_int%rng)
end subroutine gaussian_init
```

This spectrum type needs a finalizer, which closes the data file.

⟨*SF gaussian: gaussian: TBP*⟩+≡
```
procedure :: final => sf_gaussian_final
```

⟨*SF gaussian: procedures*⟩+≡
```
subroutine sf_gaussian_final (object)
  class(gaussian_t), intent(inout) :: object
  call object%interaction_t%final ()
end subroutine sf_gaussian_final
```

15.10.4 Kinematics

Refer to the `data` component.

⟨*SF gaussian: gaussian: TBP*⟩+≡
```
procedure :: is_generator => gaussian_is_generator
```

⟨*SF gaussian: procedures*⟩+≡
```
function gaussian_is_generator (sf_int) result (flag)
  class(gaussian_t), intent(in) :: sf_int
  logical :: flag
  flag = sf_int%data%is_generator ()
end function gaussian_is_generator
```

Generate free parameters. The x value should be distributed with mean 1 and σ given by the spread. We reject negative x values. (This cut slightly biases the distribution, but for reasonable (small) spreads negative r should not occur.

⟨*SF gaussian: gaussian: TBP*⟩+≡
```
procedure :: generate_free => gaussian_generate_free
```

⟨*SF gaussian: procedures*⟩+≡
```
subroutine gaussian_generate_free (sf_int, r, rb, x_free)
  class(gaussian_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(inout) :: x_free
  real(default), dimension(size(r)) :: z
  associate (data => sf_int%data)
    do
       call sf_int%rng%generate_gaussian (z)
       rb = z * data%spread
       r = 1 - rb
       x_free = x_free * product (r)
       if (all (r > 0))  exit
    end do
  end associate
end subroutine gaussian_generate_free
```

Set kinematics. Trivial transfer since this is a pure generator. The `map` flag doesn't apply.

⟨*SF gaussian: gaussian: TBP*⟩+≡
```
procedure :: complete_kinematics => gaussian_complete_kinematics
```

⟨*SF gaussian: procedures*⟩+≡

```
subroutine gaussian_complete_kinematics (sf_int, x, f, r, rb, map)
  class(gaussian_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  real(default), dimension(:), intent(in) :: rb
  logical, intent(in) :: map
  if (map) then
     call msg_fatal ("gaussian: map flag not supported")
  else
     x = r
     f = 1
  end if
  call sf_int%reduce_momenta (x)
end subroutine gaussian_complete_kinematics
```

Compute inverse kinematics. Trivial in this case.

⟨*SF gaussian: gaussian: TBP*⟩+≡

```
procedure :: inverse_kinematics => gaussian_inverse_kinematics
```

⟨*SF gaussian: procedures*⟩+≡

```
subroutine gaussian_inverse_kinematics &
     (sf_int, x, f, r, rb, map, set_momenta)
  class(gaussian_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  real(default), dimension(:), intent(out) :: rb
  logical, intent(in) :: map
  logical, intent(in), optional :: set_momenta
  logical :: set_mom
  set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
  if (map) then
     call msg_fatal ("gaussian: map flag not supported")
  else
     r = x
     f = 1
  end if
  rb = 1 - r
  if (set_mom) then
     call sf_int%reduce_momenta (x)
  end if
end subroutine gaussian_inverse_kinematics
```

15.10.5 gaussian application

Trivial, just set the unit weight.

⟨*SF gaussian: gaussian: TBP*⟩+≡

```
procedure :: apply => gaussian_apply
```

⟨*SF gaussian: procedures*⟩+≡

```
subroutine gaussian_apply (sf_int, scale)
  class(gaussian_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  real(default) :: f
  f = 1
  call sf_int%set_matrix_element (cmplx (f, kind=default))
  sf_int%status = SF_EVALUATED
end subroutine gaussian_apply
```

15.10.6 Unit tests

Test module, followed by the corresponding implementation module.

⟨sf_gaussian_ut.f90⟩≡

```
⟨File header⟩

module sf_gaussian_ut
  use unit_tests
  use sf_gaussian_uti

⟨Standard module head⟩

⟨SF gaussian: public test⟩

contains

⟨SF gaussian: test driver⟩

end module sf_gaussian_ut
```

⟨sf_gaussian_uti.f90⟩≡

```
⟨File header⟩

module sf_gaussian_uti

⟨Use kinds⟩
  use physics_defs, only: ELECTRON
  use lorentz
  use pdg_arrays
  use flavors
  use interactions, only: reset_interaction_counter
  use model_data
  use rng_base
  use sf_aux
  use sf_base

  use sf_gaussian

  use rng_base_ut, only: rng_test_factory_t

⟨Standard module head⟩

⟨SF gaussian: test declarations⟩
```

```
contains

⟨SF gaussian: tests⟩

end module sf_gaussian_uti
```

API: driver for the unit tests below.

⟨*SF gaussian: public test*⟩≡
```
public :: sf_gaussian_test
```

⟨*SF gaussian: test driver*⟩≡
```
subroutine sf_gaussian_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨SF gaussian: execute tests⟩
end subroutine sf_gaussian_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF gaussian: execute tests*⟩≡
```
call test (sf_gaussian_1, "sf_gaussian_1", &
     "structure function configuration", &
     u, results)
```

⟨*SF gaussian: test declarations*⟩≡
```
public :: sf_gaussian_1
```

⟨*SF gaussian: tests*⟩≡
```
subroutine sf_gaussian_1 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(pdg_array_t), dimension(2) :: pdg_in
  type(pdg_array_t), dimension(2) :: pdg_out
  integer, dimension(:), allocatable :: pdg1, pdg2
  class(sf_data_t), allocatable :: data
  class(rng_factory_t), allocatable :: rng_factory

  write (u, "(A)")  "* Test output: sf_gaussian_1"
  write (u, "(A)")  "*   Purpose: initialize and display &
       &gaussian-spread structure function data"
  write (u, "(A)")

  call model%init_qed_test ()
  pdg_in(1) = ELECTRON
  pdg_in(2) = -ELECTRON

  allocate (gaussian_data_t :: data)
  allocate (rng_test_factory_t :: rng_factory)
  select type (data)
  type is (gaussian_data_t)
     call data%init (model, pdg_in, [1e-2_default, 2e-2_default], rng_factory)
  end select
```

```
    call data%write (u)

    write (u, "(A)")

    write (u, "(1x,A)")  "Outgoing particle codes:"
    call data%get_pdg_out (pdg_out)
    pdg1 = pdg_out(1)
    pdg2 = pdg_out(2)
    write (u, "(2x,99(1x,I0))")  pdg1, pdg2

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_gaussian_1"

  end subroutine sf_gaussian_1
```

Probe the structure-function object

Active the beam event reader, generate an event.

⟨*SF gaussian: execute tests*⟩+≡

```
  call test (sf_gaussian_2, "sf_gaussian_2", &
       "generate event", &
       u, results)
```

⟨*SF gaussian: test declarations*⟩+≡

```
  public :: sf_gaussian_2
```

⟨*SF gaussian: tests*⟩+≡

```
  subroutine sf_gaussian_2 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t), dimension(2) :: flv
    type(pdg_array_t), dimension(2) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    class(rng_factory_t), allocatable :: rng_factory
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k1, k2
    real(default) :: E
    real(default), dimension(:), allocatable :: r, rb, x
    real(default) :: x_free, f
    integer :: i

    write (u, "(A)")  "* Test output: sf_gaussian_2"
    write (u, "(A)")  "*   Purpose: initialize and display &
         &gaussian-spread structure function data"
    write (u, "(A)")

    call model%init_qed_test ()
    call flv(1)%init (ELECTRON, model)
    call flv(2)%init (-ELECTRON, model)
    pdg_in(1) = ELECTRON
    pdg_in(2) = -ELECTRON
```

```
call reset_interaction_counter ()

allocate (gaussian_data_t :: data)
allocate (rng_test_factory_t :: rng_factory)
select type (data)
type is (gaussian_data_t)
   call data%init (model, pdg_in, [1e-2_default, 2e-2_default], rng_factory)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1,2])

write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 250
k1 = vector4_moving (E, sqrt (E**2 - flv(1)%get_mass ()**2), 3)
k2 = vector4_moving (E,-sqrt (E**2 - flv(2)%get_mass ()**2), 3)
call vector4_write (k1, u)
call vector4_write (k2, u)
call sf_int%seed_kinematics ([k1, k2])

write (u, "(A)")
write (u, "(A)")  "* Set dummy parameters and generate x."
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r  = 0
rb = 0
x_free = 1
call sf_int%generate_free (r, rb, x_free)
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call pacify (rb, 1.e-8_default)

write (u, "(A,9(1x,F10.7))")  "r =", r
write (u, "(A,9(1x,F10.7))")  "rb=", rb
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f
write (u, "(A,9(1x,F10.7))")  "xf=", x_free

write (u, "(A)")
write (u, "(A)")  "* Evaluate"
write (u, "(A)")

call sf_int%apply (scale = 0._default)
call sf_int%write (u)

write (u, "(A)")
```

```
  write (u, "(A)")  "* Generate more events"
  write (u, "(A)")

  select type (sf_int)
  type is (gaussian_t)
     do i = 1, 3
        call sf_int%generate_free (r, rb, x_free)
        write (u, "(A,9(1x,F10.7))")  "r =", r
     end do
  end select

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call sf_int%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_gaussian_2"

end subroutine sf_gaussian_2
```

15.11 Using beam event data

Instead of an analytic beam description, beam data may be provided in form of an event file. In its most simple form, the event file contains pairs of x values, relative to nominal beam energies. More advanced formats may include polarization, etc. The current implementation carries beam polarization through, if specified.

The code is very similar to the energy scan described above.

However, we must include a file-handle manager for the beam-event files. Two different processes may access a given beam-event file at the same time (i.e., serially but alternating). Accessing an open file from two different units is non-standard and not supported by all compilers. Therefore, we keep a global registry of open files, associated units, and reference counts. The `beam_events_t` objects act as proxies to this registry.

⟨`sf_beam_events.f90`⟩≡

```
⟨File header⟩

module sf_beam_events

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use file_registries
  use diagnostics
  use lorentz
  use pdg_arrays
  use model_data
  use flavors
  use quantum_numbers
  use state_matrices
  use polarizations
  use sf_base

⟨Standard module head⟩

⟨SF beam events: public⟩

⟨SF beam events: types⟩

⟨SF beam events: variables⟩

contains

⟨SF beam events: procedures⟩

end module sf_beam_events
```

15.11.1 The beam-data file registry

We manage data files via the `file_registries` module. To this end, we keep the registry as a private module variable here.

This is public only for the unit tests.

⟨*SF beam events: public*⟩≡
```
public :: beam_file_registry
```

⟨*SF beam events: variables*⟩≡
```
type(file_registry_t), save :: beam_file_registry
```

15.11.2 Data type

⟨*SF beam events: public*⟩+≡
```
public :: beam_events_data_t
```

⟨*SF beam events: types*⟩≡
```
type, extends(sf_data_t) :: beam_events_data_t
   private
   type(flavor_t), dimension(2) :: flv_in
   type(string_t) :: dir
   type(string_t) :: file
   type(string_t) :: fqn
   integer :: unit = 0
   logical :: warn_eof = .true.
 contains
 ⟨SF beam events: beam events data: TBP⟩
end type beam_events_data_t
```

⟨*SF beam events: beam events data: TBP*⟩≡
```
procedure :: init => beam_events_data_init
```

⟨*SF beam events: procedures*⟩≡
```
subroutine beam_events_data_init (data, model, pdg_in, dir, file, warn_eof)
  class(beam_events_data_t), intent(out) :: data
  class(model_data_t), intent(in), target :: model
  type(pdg_array_t), dimension(2), intent(in) :: pdg_in
  type(string_t), intent(in) :: dir
  type(string_t), intent(in) :: file
  logical, intent(in), optional :: warn_eof
  if (any (pdg_array_get_length (pdg_in) /= 1)) then
     call msg_fatal ("Beam events: incoming beam particles must be unique")
  end if
  call data%flv_in(1)%init (pdg_array_get (pdg_in(1), 1), model)
  call data%flv_in(2)%init (pdg_array_get (pdg_in(2), 1), model)
  data%dir = dir
  data%file = file
  if (present (warn_eof))  data%warn_eof = warn_eof
end subroutine beam_events_data_init
```

Return true since this spectrum is always in generator mode.

⟨*SF beam events: beam events data: TBP*⟩+≡
```
procedure :: is_generator => beam_events_data_is_generator
```

⟨*SF beam events: procedures*⟩+≡
```
function beam_events_data_is_generator (data) result (flag)
  class(beam_events_data_t), intent(in) :: data
  logical :: flag
  flag = .true.
end function beam_events_data_is_generator
```

The number of parameters is two. They are free parameters.

⟨*SF beam events: beam events data: TBP*⟩+≡
```
procedure :: get_n_par => beam_events_data_get_n_par
```

⟨*SF beam events: procedures*⟩+≡
```
function beam_events_data_get_n_par (data) result (n)
  class(beam_events_data_t), intent(in) :: data
  integer :: n
  n = 2
end function beam_events_data_get_n_par
```

⟨*SF beam events: beam events data: TBP*⟩+≡
```
procedure :: get_pdg_out => beam_events_data_get_pdg_out
```

⟨*SF beam events: procedures*⟩+≡
```
subroutine beam_events_data_get_pdg_out (data, pdg_out)
  class(beam_events_data_t), intent(in) :: data
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
  integer :: i, n
  n = 2
  do i = 1, n
     pdg_out(i) = data%flv_in(i)%get_pdg ()
  end do
end subroutine beam_events_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF beam events: beam events data: TBP*⟩+≡
```
procedure :: allocate_sf_int => beam_events_data_allocate_sf_int
```

⟨*SF beam events: procedures*⟩+≡
```
subroutine beam_events_data_allocate_sf_int (data, sf_int)
  class(beam_events_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  allocate (beam_events_t :: sf_int)
end subroutine beam_events_data_allocate_sf_int
```

Output

⟨*SF beam events: beam events data: TBP*⟩+≡
```
procedure :: write => beam_events_data_write
```

⟨*SF beam events: procedures*⟩+≡
```
subroutine beam_events_data_write (data, unit, verbose)
  class(beam_events_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
```

```
  write (u, "(1x,A)") "Beam-event file data:"
  write (u, "(3x,A,A,A,A)") "prt_in = ", &
       char (data%flv_in(1)%get_name ()), &
       ", ", char (data%flv_in(2)%get_name ())
  write (u, "(3x,A,A,A)") "file   = '", char (data%file), "'"
  write (u, "(3x,A,I0)")  "unit   = ", data%unit
  write (u, "(3x,A,L1)")  "warn   = ", data%warn_eof
end subroutine beam_events_data_write
```

The data file needs to be opened and closed explicitly. The open/close message is communicated to the file handle registry, which does the actual work.

We determine first whether to look in the local directory or in the given system directory.

⟨*SF beam events: beam events data: TBP*⟩+≡

```
procedure :: open => beam_events_data_open
procedure :: close => beam_events_data_close
```

⟨*SF beam events: procedures*⟩+≡

```
subroutine beam_events_data_open (data)
  class(beam_events_data_t), intent(inout) :: data
  logical :: exist
  if (data%unit == 0) then
     data%fqn = data%file
     if (data%fqn == "") &
          call msg_fatal ("Beam events: $beam_events_file is not set")
     inquire (file = char (data%fqn), exist = exist)
     if (.not. exist) then
        data%fqn = data%dir // "/" // data%file
        inquire (file = char (data%fqn), exist = exist)
        if (.not. exist) then
           data%fqn = ""
           call msg_fatal ("Beam events: file '" &
                // char (data%file) // "' not found")
           return
        end if
     end if
     call msg_message ("Beam events: reading from file '" &
          // char (data%file) // "'")
     call beam_file_registry%open (data%fqn, data%unit)
  else
     call msg_bug ("Beam events: file '" &
       // char (data%file) // "' is already open")
  end if
end subroutine beam_events_data_open

subroutine beam_events_data_close (data)
  class(beam_events_data_t), intent(inout) :: data
  if (data%unit /= 0) then
     call beam_file_registry%close (data%fqn)
     call msg_message ("Beam events: closed file '" &
       // char (data%file) // "'")
     data%unit = 0
  end if
end subroutine beam_events_data_close
```

Return the beam event file.

⟨*SF beam events: beam events data: TBP*⟩+≡
```
procedure :: get_beam_file => beam_events_data_get_beam_file
```

⟨*SF beam events: procedures*⟩+≡
```
function beam_events_data_get_beam_file (data) result (file)
  class(beam_events_data_t), intent(in) :: data
  type(string_t) :: file
  file = "Beam events: " // data%file
end function beam_events_data_get_beam_file
```

15.11.3 The beam events object

Flavor and polarization carried through, no radiated particles.

⟨*SF beam events: public*⟩+≡
```
public :: beam_events_t
```

⟨*SF beam events: types*⟩+≡
```
type, extends (sf_int_t) :: beam_events_t
   type(beam_events_data_t), pointer :: data => null ()
   integer :: count = 0
 contains
 ⟨SF beam events: beam events: TBP⟩
end type beam_events_t
```

Type string: show beam events file.

⟨*SF beam events: beam events: TBP*⟩≡
```
procedure :: type_string => beam_events_type_string
```

⟨*SF beam events: procedures*⟩+≡
```
function beam_events_type_string (object) result (string)
  class(beam_events_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "Beam events: " // object%data%file
  else
     string = "Beam events: [undefined]"
  end if
end function beam_events_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF beam events: beam events: TBP*⟩+≡
```
procedure :: write => beam_events_write
```

⟨*SF beam events: procedures*⟩+≡
```
subroutine beam_events_write (object, unit, testflag)
  class(beam_events_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
```

```
    if (associated (object%data)) then
       call object%data%write (u)
       call object%base_write (u, testflag)
    else
       write (u, "(1x,A)")  "Beam events data: [undefined]"
    end if
  end subroutine beam_events_write
```

⟨*SF beam events: beam events: TBP*⟩+≡

```
  procedure :: init => beam_events_init
```

⟨*SF beam events: procedures*⟩+≡

```
  subroutine beam_events_init (sf_int, data)
    class(beam_events_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    real(default), dimension(2) :: m2
    real(default), dimension(0) :: mr2
    type(quantum_numbers_mask_t), dimension(4) :: mask
    integer, dimension(4) :: hel_lock
    type(quantum_numbers_t), dimension(4) :: qn_fc, qn_hel, qn
    type(polarization_t), target :: pol1, pol2
    type(polarization_iterator_t) :: it_hel1, it_hel2
    integer :: i
    select type (data)
    type is (beam_events_data_t)
       m2 = data%flv_in%get_mass () ** 2
       hel_lock = [3, 4, 1, 2]
       mask = quantum_numbers_mask (.false., .false., .false.)
       call sf_int%base_init (mask, m2, mr2, m2, hel_lock = hel_lock)
       sf_int%data => data
       do i = 1, 2
          call qn_fc(i)%init ( &
               flv = data%flv_in(i), &
               col = color_from_flavor (data%flv_in(i)))
          call qn_fc(i+2)%init ( &
               flv = data%flv_in(i), &
               col = color_from_flavor (data%flv_in(i)))
       end do
       call pol1%init_generic (data%flv_in(1))
       call it_hel1%init (pol1)
       do while (it_hel1%is_valid ())
          qn_hel(1) = it_hel1%get_quantum_numbers ()
          qn_hel(3) = it_hel1%get_quantum_numbers ()
          call pol2%init_generic (data%flv_in(2))
          call it_hel2%init (pol2)
          do while (it_hel2%is_valid ())
             qn_hel(2) = it_hel2%get_quantum_numbers ()
             qn_hel(4) = it_hel2%get_quantum_numbers ()
             qn = qn_hel .merge. qn_fc
             call sf_int%add_state (qn)
             call it_hel2%advance ()
          end do
          ! call pol2%final ()
          call it_hel1%advance ()
```

```
      end do
      ! call pol1%final ()
      call sf_int%freeze ()
      call sf_int%set_incoming ([1,2])
      call sf_int%set_outgoing ([3,4])
      call sf_int%data%open ()
      sf_int%status = SF_INITIAL
    end select
end subroutine beam_events_init
```

This spectrum type needs a finalizer, which closes the data file.

⟨*SF beam events: beam events: TBP*⟩+≡

```
procedure :: final => sf_beam_events_final
```

⟨*SF beam events: procedures*⟩+≡

```
subroutine sf_beam_events_final (object)
  class(beam_events_t), intent(inout) :: object
  call object%data%close ()
  call object%interaction_t%final ()
end subroutine sf_beam_events_final
```

15.11.4 Kinematics

Refer to the `data` component.

⟨*SF beam events: beam events: TBP*⟩+≡

```
procedure :: is_generator => beam_events_is_generator
```

⟨*SF beam events: procedures*⟩+≡

```
function beam_events_is_generator (sf_int) result (flag)
  class(beam_events_t), intent(in) :: sf_int
  logical :: flag
  flag = sf_int%data%is_generator ()
end function beam_events_is_generator
```

Generate free parameters. We read them from file.

⟨*SF beam events: beam events: TBP*⟩+≡

```
procedure :: generate_free => beam_events_generate_free
```

⟨*SF beam events: procedures*⟩+≡

```
recursive subroutine beam_events_generate_free (sf_int, r, rb,  x_free)
  class(beam_events_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(inout) :: x_free
  integer :: iostat
  associate (data => sf_int%data)
    if (data%unit /= 0) then
       read (data%unit, fmt=*, iostat=iostat)  r
       if (iostat > 0) then
          write (msg_buffer, "(A,IO,A)") &
               "Beam events: I/O error after reading ", sf_int%count, &
               " events"
          call msg_fatal ()
```

```
      else if (iostat < 0) then
         if (sf_int%count == 0) then
            call msg_fatal ("Beam events: file is empty")
         else if (sf_int%data%warn_eof) then
            write (msg_buffer, "(A,I0,A)") &
                 "Beam events: End of file after reading ", sf_int%count, &
                 " events, rewinding"
            call msg_warning ()
         end if
         rewind (data%unit)
         sf_int%count = 0
         call sf_int%generate_free (r, rb, x_free)
      else
         sf_int%count = sf_int%count + 1
         rb = 1 - r
         x_free = x_free * product (r)
      end if
   else
      call msg_bug ("Beam events: file is not open for reading")
   end if
  end associate
end subroutine beam_events_generate_free
```

Set kinematics. Trivial transfer since this is a pure generator. The `map` flag doesn't apply.

⟨*SF beam events: beam events: TBP*⟩+≡

```
procedure :: complete_kinematics => beam_events_complete_kinematics
```

⟨*SF beam events: procedures*⟩+≡

```
subroutine beam_events_complete_kinematics (sf_int, x, f, r, rb, map)
  class(beam_events_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  real(default), dimension(:), intent(in) :: rb
  logical, intent(in) :: map
  if (map) then
     call msg_fatal ("Beam events: map flag not supported")
  else
     x = r
     f = 1
  end if
  call sf_int%reduce_momenta (x)
end subroutine beam_events_complete_kinematics
```

Compute inverse kinematics. Trivial in this case.

⟨*SF beam events: beam events: TBP*⟩+≡

```
procedure :: inverse_kinematics => beam_events_inverse_kinematics
```

⟨*SF beam events: procedures*⟩+≡

```
subroutine beam_events_inverse_kinematics &
     (sf_int, x, f, r, rb, map, set_momenta)
  class(beam_events_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
```

```
    real(default), intent(out) :: f
    real(default), dimension(:), intent(out) :: r
    real(default), dimension(:), intent(out) :: rb
    logical, intent(in) :: map
    logical, intent(in), optional :: set_momenta
    logical :: set_mom
    set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
    if (map) then
       call msg_fatal ("Beam events: map flag not supported")
    else
       r = x
       f = 1
    end if
    rb = 1 - r
    if (set_mom) then
       call sf_int%reduce_momenta (x)
    end if
  end subroutine beam_events_inverse_kinematics
```

15.11.5 Beam events application

Trivial, just set the unit weight.

⟨*SF beam events: beam events: TBP*⟩+≡

```
  procedure :: apply => beam_events_apply
```

⟨*SF beam events: procedures*⟩+≡

```
  subroutine beam_events_apply (sf_int, scale)
    class(beam_events_t), intent(inout) :: sf_int
    real(default), intent(in) :: scale
    real(default) :: f
    f = 1
    call sf_int%set_matrix_element (cmplx (f, kind=default))
    sf_int%status = SF_EVALUATED
  end subroutine beam_events_apply
```

15.11.6 Unit tests

Test module, followed by the corresponding implementation module.

⟨sf_beam_events_ut.f90⟩≡

```
  ⟨File header⟩

  module sf_beam_events_ut
    use unit_tests
    use sf_beam_events_uti

  ⟨Standard module head⟩

  ⟨SF beam events: public test⟩

  contains
```

```
  ⟨SF beam events: test driver⟩

  end module sf_beam_events_ut
```

⟨sf_beam_events_uti.f90⟩≡

```
  ⟨File header⟩

  module sf_beam_events_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use io_units
    use physics_defs, only: ELECTRON
    use lorentz
    use pdg_arrays
    use flavors
    use interactions, only: reset_interaction_counter
    use model_data
    use sf_aux
    use sf_base

    use sf_beam_events

  ⟨Standard module head⟩

  ⟨SF beam events: test declarations⟩

  contains

  ⟨SF beam events: tests⟩

  end module sf_beam_events_uti
```

API: driver for the unit tests below.

⟨*SF beam events: public test*⟩≡

```
    public :: sf_beam_events_test
```

⟨*SF beam events: test driver*⟩≡

```
    subroutine sf_beam_events_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨SF beam events: execute tests⟩
    end subroutine sf_beam_events_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF beam events: execute tests*⟩≡

```
    call test (sf_beam_events_1, "sf_beam_events_1", &
         "structure function configuration", &
         u, results)
```

⟨*SF beam events: test declarations*⟩≡

```
    public :: sf_beam_events_1
```

⟨*SF beam events: tests*⟩≡

```
subroutine sf_beam_events_1 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(pdg_array_t), dimension(2) :: pdg_in
  type(pdg_array_t), dimension(2) :: pdg_out
  integer, dimension(:), allocatable :: pdg1, pdg2
  class(sf_data_t), allocatable :: data

  write (u, "(A)")  "* Test output: sf_beam_events_1"
  write (u, "(A)")  "*   Purpose: initialize and display &
       &beam-events structure function data"
  write (u, "(A)")

  call model%init_qed_test ()
  pdg_in(1) = ELECTRON
  pdg_in(2) = -ELECTRON

  allocate (beam_events_data_t :: data)
  select type (data)
  type is (beam_events_data_t)
     call data%init (model, pdg_in, var_str (""), var_str ("beam_events.dat"))
  end select

  call data%write (u)

  write (u, "(A)")

  write (u, "(1x,A)")  "Outgoing particle codes:"
  call data%get_pdg_out (pdg_out)
  pdg1 = pdg_out(1)
  pdg2 = pdg_out(2)
  write (u, "(2x,99(1x,I0))")  pdg1, pdg2

  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_beam_events_1"

end subroutine sf_beam_events_1
```

Probe the structure-function object

Active the beam event reader, generate an event.

⟨*SF beam events: execute tests*⟩+≡

```
call test (sf_beam_events_2, "sf_beam_events_2", &
     "generate event", &
     u, results)
```

⟨*SF beam events: test declarations*⟩+≡

```
public :: sf_beam_events_2
```

⟨*SF beam events: tests*⟩+≡

```
subroutine sf_beam_events_2 (u)
```

```
integer, intent(in) :: u
type(model_data_t), target :: model
type(flavor_t), dimension(2) :: flv
type(pdg_array_t), dimension(2) :: pdg_in
class(sf_data_t), allocatable, target :: data
class(sf_int_t), allocatable :: sf_int
type(vector4_t) :: k1, k2
real(default) :: E
real(default), dimension(:), allocatable :: r, rb, x
real(default) :: x_free, f
integer :: i

write (u, "(A)")  "* Test output: sf_beam_events_2"
write (u, "(A)")  "*   Purpose: initialize and display &
     &beam-events structure function data"
write (u, "(A)")

call model%init_qed_test ()
call flv(1)%init (ELECTRON, model)
call flv(2)%init (-ELECTRON, model)
pdg_in(1) = ELECTRON
pdg_in(2) = -ELECTRON

call reset_interaction_counter ()

allocate (beam_events_data_t :: data)
select type (data)
type is (beam_events_data_t)
   call data%init (model, pdg_in, &
        var_str (""), var_str ("test_beam_events.dat"))
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1,2])

write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 250
k1 = vector4_moving (E, sqrt (E**2 - flv(1)%get_mass ()**2), 3)
k2 = vector4_moving (E,-sqrt (E**2 - flv(2)%get_mass ()**2), 3)
call vector4_write (k1, u)
call vector4_write (k2, u)
call sf_int%seed_kinematics ([k1, k2])

write (u, "(A)")
write (u, "(A)")  "* Set dummy parameters and generate x."
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
```

```
   allocate (x (size (r)))

   r  = 0
   rb = 0
   x_free = 1
   call sf_int%generate_free (r, rb, x_free)
   call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

   write (u, "(A,9(1x,F10.7))")  "r =", r
   write (u, "(A,9(1x,F10.7))")  "rb=", rb
   write (u, "(A,9(1x,F10.7))")  "x =", x
   write (u, "(A,9(1x,F10.7))")  "f =", f
   write (u, "(A,9(1x,F10.7))")  "xf=", x_free
   select type (sf_int)
   type is (beam_events_t)
      write (u, "(A,1x,I0)")  "count =", sf_int%count
   end select

   write (u, "(A)")
   write (u, "(A)")  "* Evaluate"
   write (u, "(A)")

   call sf_int%apply (scale = 0._default)
   call sf_int%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Generate more events, rewind"
   write (u, "(A)")

   select type (sf_int)
   type is (beam_events_t)
      do i = 1, 3
         call sf_int%generate_free (r, rb, x_free)
         write (u, "(A,9(1x,F10.7))")  "r =", r
         write (u, "(A,1x,I0)")  "count =", sf_int%count
      end do
   end select

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call sf_int%final ()
   call model%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: sf_beam_events_2"

end subroutine sf_beam_events_2
```

Check the file handle registry

Open and close some files, checking the registry contents.
⟨*SF beam events: execute tests*⟩+≡

```
call test (sf_beam_events_3, "sf_beam_events_3", &
     "check registry", &
     u, results)
```

⟨*SF beam events: test declarations*⟩+≡

```
public :: sf_beam_events_3
```

⟨*SF beam events: tests*⟩+≡

```
subroutine sf_beam_events_3 (u)
  integer, intent(in) :: u
  integer :: u1

  write (u, "(A)")  "* Test output: sf_beam_events_2"
  write (u, "(A)")  "*   Purpose: check file handle registry"
  write (u, "(A)")

  write (u, "(A)")  "* Create some empty files"
  write (u, "(A)")

  u1 = free_unit ()
  open (u1, file = "sf_beam_events_f1.tmp", action="write", status="new")
  close (u1)
  open (u1, file = "sf_beam_events_f2.tmp", action="write", status="new")
  close (u1)
  open (u1, file = "sf_beam_events_f3.tmp", action="write", status="new")
  close (u1)

  write (u, "(A)")  "* Empty registry"
  write (u, "(A)")

  call beam_file_registry%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Insert three entries"
  write (u, "(A)")

  call beam_file_registry%open (var_str ("sf_beam_events_f3.tmp"))
  call beam_file_registry%open (var_str ("sf_beam_events_f2.tmp"))
  call beam_file_registry%open (var_str ("sf_beam_events_f1.tmp"))
  call beam_file_registry%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Open a second channel"
  write (u, "(A)")

  call beam_file_registry%open (var_str ("sf_beam_events_f2.tmp"))
  call beam_file_registry%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Close second entry twice"
  write (u, "(A)")

  call beam_file_registry%close (var_str ("sf_beam_events_f2.tmp"))
  call beam_file_registry%close (var_str ("sf_beam_events_f2.tmp"))
  call beam_file_registry%write (u)
```

```
  write (u, "(A)")
  write (u, "(A)")  "* Close last entry"
  write (u, "(A)")

  call beam_file_registry%close (var_str ("sf_beam_events_f3.tmp"))
  call beam_file_registry%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Close remaining entry"
  write (u, "(A)")

  call beam_file_registry%close (var_str ("sf_beam_events_f1.tmp"))
  call beam_file_registry%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  open (u1, file = "sf_beam_events_f1.tmp", action="write")
  close (u1, status = "delete")
  open (u1, file = "sf_beam_events_f2.tmp", action="write")
  close (u1, status = "delete")
  open (u1, file = "sf_beam_events_f3.tmp", action="write")
  close (u1, status = "delete")

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_beam_events_3"

end subroutine sf_beam_events_3
```

15.12 Lepton collider beamstrahlung: CIRCE1

⟨sf_circe1.f90⟩≡

```
⟨File header⟩

module sf_circe1

⟨Use kinds⟩
  use kinds, only: double
⟨Use strings⟩
  use io_units
  use format_defs, only: FMT_17, FMT_19
  use diagnostics
  use physics_defs, only: ELECTRON, PHOTON
  use lorentz
  use rng_base
  use pdg_arrays
  use model_data
  use flavors
  use colors
  use quantum_numbers
  use state_matrices
  use polarizations
  use sf_mappings
  use sf_base
  use circe1, circe1_rng_t => rng_type !NODEP!

⟨Standard module head⟩

⟨SF circe1: public⟩

⟨SF circe1: types⟩

contains

⟨SF circe1: procedures⟩

end module sf_circe1
```

15.12.1 Physics

Beamstrahlung is applied before ISR. The `CIRCE1` implementation has a single structure function for both beams (which makes sense since it has to be switched on or off for both beams simultaneously). Nevertheless it is factorized:

The functional form in the `CIRCE1` parameterization is defined for electrons or photons

$$f(x) = \alpha\, x^{\beta}\, (1-x)^{\gamma} \tag{15.74}$$

for $x < 1 - \epsilon$ (resp. $x > \epsilon$ in the photon case). In the remaining interval, the standard form is zero, with a delta singularity at $x = 1$ (resp. $x = 0$). Equivalently, the delta part may be distributed uniformly among this interval. This latter form is implemented in the `kirke` version of the `CIRCE1` subroutines, and is used here.

The parameter circe1_eps sets the peak mapping of the CIRCE1 structure function. Its default value is 10^{-5}. The other parameters are the parameterization version and revision number, the accelerator type, and the $\sqrt{s}$ value used by CIRCE1. The chattiness can also be set.

Since the energy is distributed in a narrow region around unity (for electrons) or zero (for photons), it is advantageous to map the interval first. The mapping is controlled by the parameter circe1_epsilon which is taken from the CIRCE1 internal data structure.

The $\sqrt{s}$ value, if not explicitly set, is taken from the process data. Note that interpolating $\sqrt{s}$ is not recommended; one should rather choose one of the distinct values known to CIRCE1.

15.12.2 The CIRCE1 data block

The CIRCE1 parameters are: The incoming flavors, the flags whether the photon or the lepton is the parton in the hard interaction, the flags for the generation mode (generator/mapping/no mapping), the mapping parameter ϵ, $\sqrt{s}$ and several steering parameters: ver, rev, acc, chat.

In generator mode, the x values are actually discarded and a random number generator is used instead.

⟨*SF circe1: public*⟩≡
```
public :: circe1_data_t
```

⟨*SF circe1: types*⟩≡
```
type, extends (sf_data_t) :: circe1_data_t
   private
   class(model_data_t), pointer :: model => null ()
   type(flavor_t), dimension(2) :: flv_in
   integer, dimension(2) :: pdg_in
   real(default), dimension(2) :: m_in = 0
   logical, dimension(2) :: photon = .false.
   logical :: generate = .false.
   class(rng_factory_t), allocatable :: rng_factory
   real(default) :: sqrts = 0
   real(default) :: eps = 0
   integer :: ver = 0
   integer :: rev = 0
   character(6) :: acc = "?"
   integer :: chat = 0
   logical :: with_radiation = .false.
 contains
   ⟨SF circe1: circe1 data: TBP⟩
end type circe1_data_t
```

⟨*SF circe1: circe1 data: TBP*⟩≡
```
procedure :: init => circe1_data_init
```

⟨*SF circe1: procedures*⟩≡
```
subroutine circe1_data_init &
     (data, model, pdg_in, sqrts, eps, out_photon, &
      ver, rev, acc, chat, with_radiation)
  class(circe1_data_t), intent(out) :: data
  class(model_data_t), intent(in), target :: model
```

```
    type(pdg_array_t), dimension(2), intent(in) :: pdg_in
    real(default), intent(in) :: sqrts
    real(default), intent(in) :: eps
    logical, dimension(2), intent(in) :: out_photon
    character(*), intent(in) :: acc
    integer, intent(in) :: ver, rev, chat
    logical, intent(in) :: with_radiation
    data%model => model
    if (any (pdg_array_get_length (pdg_in) /= 1)) then
       call msg_fatal ("CIRCE1: incoming beam particles must be unique")
    end if
    call data%flv_in(1)%init (pdg_array_get (pdg_in(1), 1), model)
    call data%flv_in(2)%init (pdg_array_get (pdg_in(2), 1), model)
    data%pdg_in = data%flv_in%get_pdg ()
    data%m_in = data%flv_in%get_mass ()
    data%sqrts = sqrts
    data%eps = eps
    data%photon = out_photon
    data%ver = ver
    data%rev = rev
    data%acc = acc
    data%chat = chat
    data%with_radiation = with_radiation
    call data%check ()
    call circex (0.d0, 0.d0, dble (data%sqrts), &
         data%acc, data%ver, data%rev, data%chat)
  end subroutine circe1_data_init
```

Activate the generator mode. We import a RNG factory into the data type, which can then spawn RNG generator objects.

⟨*SF circe1: circe1 data: TBP*⟩+≡

```
  procedure :: set_generator_mode => circe1_data_set_generator_mode
```

⟨*SF circe1: procedures*⟩+≡

```
  subroutine circe1_data_set_generator_mode (data, rng_factory)
    class(circe1_data_t), intent(inout) :: data
    class(rng_factory_t), intent(inout), allocatable :: rng_factory
    data%generate = .true.
    call move_alloc (from = rng_factory, to = data%rng_factory)
  end subroutine circe1_data_set_generator_mode
```

Handle error conditions.

⟨*SF circe1: circe1 data: TBP*⟩+≡

```
  procedure :: check => circe1_data_check
```

⟨*SF circe1: procedures*⟩+≡

```
  subroutine circe1_data_check (data)
    class(circe1_data_t), intent(in) :: data
    type(flavor_t) :: flv_electron, flv_photon
    call flv_electron%init (ELECTRON, data%model)
    call flv_photon%init (PHOTON, data%model)
    if (.not. flv_electron%is_defined () &
         .or. .not. flv_photon%is_defined ()) then
       call msg_fatal ("CIRCE1: model must contain photon and electron")
```

```
    end if
    if (any (abs (data%pdg_in) /= ELECTRON) &
         .or. (data%pdg_in(1) /= - data%pdg_in(2))) then
       call msg_fatal ("CIRCE1: applicable only for e+e- or e-e+ collisions")
    end if
    if (data%eps <= 0) then
       call msg_error ("CIRCE1: circe1_eps = 0: integration will &
            &miss x=1 peak")
    end if
  end subroutine circe1_data_check
```

Output

⟨*SF circe1: circe1 data: TBP*⟩+≡

```
  procedure :: write => circe1_data_write
```

⟨*SF circe1: procedures*⟩+≡

```
  subroutine circe1_data_write (data, unit, verbose)
    class(circe1_data_t), intent(in) :: data
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(1x,A)") "CIRCE1 data:"
    write (u, "(3x,A,2(1x,A))") "prt_in   =", &
         char (data%flv_in(1)%get_name ()), &
         char (data%flv_in(2)%get_name ())
    write (u, "(3x,A,2(1x,L1))")  "photon   =", data%photon
    write (u, "(3x,A,L1)")        "generate = ", data%generate
    write (u, "(3x,A,2(1x," // FMT_19 // "))") "m_in     =", data%m_in
    write (u, "(3x,A," // FMT_19 // ")") "sqrts    = ", data%sqrts
    write (u, "(3x,A," // FMT_19 // ")") "eps      = ", data%eps
    write (u, "(3x,A,I0)") "ver      = ", data%ver
    write (u, "(3x,A,I0)") "rev      = ", data%rev
    write (u, "(3x,A,A)")  "acc      = ", data%acc
    write (u, "(3x,A,I0)") "chat     = ", data%chat
    write (u, "(3x,A,L1)") "with rad.= ", data%with_radiation
    if (data%generate)  call data%rng_factory%write (u)
  end subroutine circe1_data_write
```

Return true if this structure function is in generator mode. In that case, all parameters are free, otherwise bound. (We do not support mixed cases.) Default is: no generator.

⟨*SF circe1: circe1 data: TBP*⟩+≡

```
  procedure :: is_generator => circe1_data_is_generator
```

⟨*SF circe1: procedures*⟩+≡

```
  function circe1_data_is_generator (data) result (flag)
    class(circe1_data_t), intent(in) :: data
    logical :: flag
    flag = data%generate
  end function circe1_data_is_generator
```

The number of parameters is two, collinear splitting for the two beams.

⟨*SF circe1: circe1 data: TBP*⟩+≡
```
procedure :: get_n_par => circe1_data_get_n_par
```

⟨*SF circe1: procedures*⟩+≡
```
function circe1_data_get_n_par (data) result (n)
  class(circe1_data_t), intent(in) :: data
  integer :: n
  n = 2
end function circe1_data_get_n_par
```

Return the outgoing particles PDG codes. This is either the incoming particle (if a photon is radiated), or the photon if that is the particle of the hard interaction. The latter is determined via the `photon` flag. There are two entries for the two beams.

⟨*SF circe1: circe1 data: TBP*⟩+≡
```
procedure :: get_pdg_out => circe1_data_get_pdg_out
```

⟨*SF circe1: procedures*⟩+≡
```
subroutine circe1_data_get_pdg_out (data, pdg_out)
  class(circe1_data_t), intent(in) :: data
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
  integer :: i, n
  n = 2
  do i = 1, n
     if (data%photon(i)) then
        pdg_out(i) = PHOTON
     else
        pdg_out(i) = data%pdg_in(i)
     end if
  end do
end subroutine circe1_data_get_pdg_out
```

This variant is not inherited, it returns integers.

⟨*SF circe1: circe1 data: TBP*⟩+≡
```
procedure :: get_pdg_int => circe1_data_get_pdg_int
```

⟨*SF circe1: procedures*⟩+≡
```
function circe1_data_get_pdg_int (data) result (pdg)
  class(circe1_data_t), intent(in) :: data
  integer, dimension(2) :: pdg
  integer :: i
  do i = 1, 2
     if (data%photon(i)) then
        pdg(i) = PHOTON
     else
        pdg(i) = data%pdg_in(i)
     end if
  end do
end function circe1_data_get_pdg_int
```

Allocate the interaction record.

⟨*SF circe1: circe1 data: TBP*⟩+≡
```
procedure :: allocate_sf_int => circe1_data_allocate_sf_int
```

⟨*SF circe1: procedures*⟩+≡

```
subroutine circe1_data_allocate_sf_int (data, sf_int)
  class(circe1_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  allocate (circe1_t :: sf_int)
end subroutine circe1_data_allocate_sf_int
```

Return the accelerator type.

⟨*SF circe1: circe1 data: TBP*⟩+≡

```
procedure :: get_beam_file => circe1_data_get_beam_file
```

⟨*SF circe1: procedures*⟩+≡

```
function circe1_data_get_beam_file (data) result (file)
  class(circe1_data_t), intent(in) :: data
  type(string_t) :: file
  file = "CIRCE1: " // data%acc
end function circe1_data_get_beam_file
```

15.12.3 Random Number Generator for CIRCE

The CIRCE implementation now supports a generic random-number generator object that allows for a local state as a component. To support this, we must extend the abstract type provided by CIRCE and delegate the generator call to the (also abstract) RNG used by WHIZARD.

⟨*SF circe1: types*⟩+≡

```
type, extends (circe1_rng_t) :: rng_obj_t
   class(rng_t), allocatable :: rng
 contains
   procedure :: generate => rng_obj_generate
end type rng_obj_t
```

⟨*SF circe1: procedures*⟩+≡

```
subroutine rng_obj_generate (rng_obj, u)
  class(rng_obj_t), intent(inout) :: rng_obj
  real(double), intent(out) :: u
  real(default) :: x
  call rng_obj%rng%generate (x)
  u = x
end subroutine rng_obj_generate
```

15.12.4 The CIRCE1 object

This is a $2 \to 4$ interaction, where, depending on the parameters, any two of the four outgoing particles are connected to the hard interactions, the others are radiated. Knowing that all particles are colorless, we do not have to deal with color.

The flavors are sorted such that the first two particles are the incoming leptons, the next two are the radiated particles, and the last two are the partons initiating the hard interaction.

CIRCE1 does not support polarized beams explicitly. For simplicity, we nevertheless carry beam polarization through to the outgoing electrons and make the photons unpolarized.

In the case that no radiated particle is kept (which actually is the default), polarization is always transferred to the electrons, too. If there is a recoil photon in the event, the radiated particles are 3 and 4, respectively, and 5 and 6 are the outgoing ones (triggering the hard scattering process), while in the case of no radiation, the outgoing particles are 3 and 4, respectively. In the case of the electron being the radiated particle, helicity is not kept.

⟨*SF circe1: public*⟩+≡
```
public :: circe1_t
```

⟨*SF circe1: types*⟩+≡
```
type, extends (sf_int_t) :: circe1_t
   type(circe1_data_t), pointer :: data => null ()
   real(default), dimension(2) :: x = 0
   real(default) :: f = 0
   logical, dimension(2) :: continuum = .true.
   logical, dimension(2) :: peak = .true.
   type(rng_obj_t) :: rng_obj
 contains
 ⟨SF circe1: circe1: TBP⟩
end type circe1_t
```

Type string: has to be here, but there is no string variable on which CIRCE1 depends. Hence, a dummy routine.

⟨*SF circe1: circe1: TBP*⟩≡
```
procedure :: type_string => circe1_type_string
```

⟨*SF circe1: procedures*⟩+≡
```
function circe1_type_string (object) result (string)
  class(circe1_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "CIRCE1: beamstrahlung"
  else
     string = "CIRCE1: [undefined]"
  end if
end function circe1_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF circe1: circe1: TBP*⟩+≡
```
procedure :: write => circe1_write
```

⟨*SF circe1: procedures*⟩+≡
```
subroutine circe1_write (object, unit, testflag)
  class(circe1_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
```

```
      if (object%data%generate)  call object%rng_obj%rng%write (u)
      if (object%status >= SF_DONE_KINEMATICS) then
         write (u, "(3x,A,2(1x," // FMT_17 // "))")  "x =", object%x
         if (object%status >= SF_FAILED_EVALUATION) then
            write (u, "(3x,A,1x," // FMT_17 // ")")  "f =", object%f
         end if
      end if
      call object%base_write (u, testflag)
   else
      write (u, "(1x,A)")  "CIRCE1 data: [undefined]"
   end if
end subroutine circe1_write
```

⟨*SF circe1: circe1: TBP*⟩+≡

```
procedure :: init => circe1_init
```

⟨*SF circe1: procedures*⟩+≡

```
subroutine circe1_init (sf_int, data)
  class(circe1_t), intent(out) :: sf_int
  class(sf_data_t), intent(in), target :: data
  logical, dimension(6) :: mask_h
  type(quantum_numbers_mask_t), dimension(6) :: mask
  integer, dimension(6) :: hel_lock
  type(polarization_t), target :: pol1, pol2
  type(quantum_numbers_t), dimension(1) :: qn_fc1, qn_fc2
  type(flavor_t) :: flv_photon
  type(color_t) :: col0
  real(default), dimension(2) :: mi2, mr2, mo2
  type(quantum_numbers_t) :: qn_hel1, qn_hel2, qn_photon, qn1, qn2
  type(quantum_numbers_t), dimension(6) :: qn
  type(polarization_iterator_t) :: it_hel1, it_hel2
  hel_lock = 0
  mask_h = .false.
  select type (data)
  type is (circe1_data_t)
     mi2 = data%m_in**2
     if (data%with_radiation) then
        if (data%photon(1)) then
           hel_lock(1) = 3;  hel_lock(3) = 1;  mask_h(5) = .true.
           mr2(1) = mi2(1)
           mo2(1) = 0._default
        else
           hel_lock(1) = 5;  hel_lock(5) = 1;  mask_h(3) = .true.
           mr2(1) = 0._default
           mo2(1) = mi2(1)
        end if
        if (data%photon(2)) then
           hel_lock(2) = 4;  hel_lock(4) = 2;  mask_h(6) = .true.
           mr2(2) = mi2(2)
           mo2(2) = 0._default
        else
           hel_lock(2) = 6;  hel_lock(6) = 2;  mask_h(4) = .true.
           mr2(2) = 0._default
           mo2(2) = mi2(2)
```

```
         end if
         mask = quantum_numbers_mask (.false., .false., mask_h)
         call sf_int%base_init (mask, mi2, mr2, mo2, &
              hel_lock = hel_lock)
         sf_int%data => data
         call flv_photon%init (PHOTON, data%model)
         call col0%init ()
         call qn_photon%init (flv_photon, col0)
         call pol1%init_generic (data%flv_in(1))
         call qn_fc1(1)%init (flv = data%flv_in(1), col = col0)
         call pol2%init_generic (data%flv_in(2))
         call qn_fc2(1)%init (flv = data%flv_in(2), col = col0)
         call it_hel1%init (pol1)

         do while (it_hel1%is_valid ())
            qn_hel1 = it_hel1%get_quantum_numbers ()
            qn1 = qn_hel1 .merge. qn_fc1(1)
            qn(1) = qn1
            if (data%photon(1)) then
               qn(3) = qn1;  qn(5) = qn_photon
            else
               qn(3) = qn_photon;  qn(5) = qn1
            end if
            call it_hel2%init (pol2)
            do while (it_hel2%is_valid ())
               qn_hel2 = it_hel2%get_quantum_numbers ()
               qn2 = qn_hel2 .merge. qn_fc2(1)
               qn(2) = qn2
               if (data%photon(2)) then
                  qn(4) = qn2;  qn(6) = qn_photon
               else
                  qn(4) = qn_photon;  qn(6) = qn2
               end if
               call qn(3:4)%tag_radiated ()
               call sf_int%add_state (qn)
               call it_hel2%advance ()
            end do
            call it_hel1%advance ()
         end do
!          call pol1%final ()
!          call pol2%final ()
         call sf_int%freeze ()
         call sf_int%set_incoming ([1,2])
         call sf_int%set_radiated ([3,4])
         call sf_int%set_outgoing ([5,6])
      else
         if (data%photon(1)) then
            mask_h(3) = .true.
            mo2(1) = 0._default
         else
            hel_lock(1) = 3;  hel_lock(3) = 1
            mo2(1) = mi2(1)
         end if
         if (data%photon(2)) then
```

```
            mask_h(4) = .true.
            mo2(2) = 0._default
         else
            hel_lock(2) = 4;  hel_lock(4) = 2
            mo2(2) = mi2(2)
         end if
         mask = quantum_numbers_mask (.false., .false., mask_h)
         call sf_int%base_init (mask(1:4), mi2, [real(default) :: ], mo2, &
              hel_lock = hel_lock(1:4))
         sf_int%data => data
         call flv_photon%init (PHOTON, data%model)
         call col0%init ()
         call qn_photon%init (flv_photon, col0)
         call pol1%init_generic (data%flv_in(1))
         call qn_fc1(1)%init (flv = data%flv_in(1), col = col0)
         call pol2%init_generic (data%flv_in(2))
         call qn_fc2(1)%init (flv = data%flv_in(2), col = col0)
         call it_hel1%init (pol1)

         do while (it_hel1%is_valid ())
            qn_hel1 = it_hel1%get_quantum_numbers ()
            qn1 = qn_hel1 .merge. qn_fc1(1)
            qn(1) = qn1
            if (data%photon(1)) then
               qn(3) = qn_photon
            else
               qn(3) = qn1
            end if
            call it_hel2%init (pol2)
            do while (it_hel2%is_valid ())
               qn_hel2 = it_hel2%get_quantum_numbers ()
               qn2 = qn_hel2 .merge. qn_fc2(1)
               qn(2) = qn2
               if (data%photon(2)) then
                  qn(4) = qn_photon
               else
                  qn(4) = qn2
               end if
               call sf_int%add_state (qn(1:4))
               call it_hel2%advance ()
            end do
            call it_hel1%advance ()
         end do
!          call pol1%final ()
!          call pol2%final ()
         call sf_int%freeze ()
         call sf_int%set_incoming ([1,2])
         call sf_int%set_outgoing ([3,4])
      end if
      sf_int%status = SF_INITIAL
   end select
   if (sf_int%data%generate) then
      call sf_int%data%rng_factory%make (sf_int%rng_obj%rng)
   end if
```

```
end subroutine circe1_init
```

15.12.5 Kinematics

Refer to the `data` component.

⟨*SF circe1: circe1: TBP*⟩+≡

```
procedure :: is_generator => circe1_is_generator
```

⟨*SF circe1: procedures*⟩+≡

```
function circe1_is_generator (sf_int) result (flag)
  class(circe1_t), intent(in) :: sf_int
  logical :: flag
  flag = sf_int%data%is_generator ()
end function circe1_is_generator
```

Generate free parameters, if generator mode is on. Otherwise, the parameters will be discarded.

⟨*SF circe1: circe1: TBP*⟩+≡

```
procedure :: generate_free => circe1_generate_free
```

⟨*SF circe1: procedures*⟩+≡

```
subroutine circe1_generate_free (sf_int, r, rb,  x_free)
  class(circe1_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(inout) :: x_free

  if (sf_int%data%generate) then
     call circe_generate (r, sf_int%data%get_pdg_int (), sf_int%rng_obj)
     rb = 1 - r
     x_free = x_free * product (r)
  else
     r = 0
     rb= 1
  end if
end subroutine circe1_generate_free
```

Generator mode: depending on the particle codes, call one of the available `girce` generators. Illegal particle code combinations should have been caught during data initialization.

⟨*SF circe1: procedures*⟩+≡

```
subroutine circe_generate (x, pdg, rng_obj)
  real(default), dimension(2), intent(out) :: x
  integer, dimension(2), intent(in) :: pdg
  class(rng_obj_t), intent(inout) :: rng_obj
  real(double) :: xc1, xc2
  select case (abs (pdg(1)))
  case (ELECTRON)
     select case (abs (pdg(2)))
     case (ELECTRON)
        call gircee (xc1, xc2, rng_obj = rng_obj)
     case (PHOTON)
        call girceg (xc1, xc2, rng_obj = rng_obj)
```

```
      end select
   case (PHOTON)
      select case (abs (pdg(2)))
      case (ELECTRON)
         call girceg (xc2, xc1, rng_obj = rng_obj)
      case (PHOTON)
         call gircgg (xc1, xc2, rng_obj = rng_obj)
      end select
   end select
   x = [xc1, xc2]
end subroutine circe_generate
```

Set kinematics. The r values (either from integration or from the generator call above) are copied to x unchanged, and f is unity. We store the x values, so we can use them for the evaluation later.

⟨*SF circe1: circe1: TBP*⟩+≡

```
procedure :: complete_kinematics => circe1_complete_kinematics
```

⟨*SF circe1: procedures*⟩+≡

```
subroutine circe1_complete_kinematics (sf_int, x, f, r, rb, map)
  class(circe1_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  real(default), dimension(:), intent(in) :: rb
  logical, intent(in) :: map
  real(default), dimension(2) :: xb1
  x = r
  sf_int%x = x
  f = 1
  if (sf_int%data%with_radiation) then
     xb1 = 1 - x
     call sf_int%split_momenta (x, xb1)
  else
     call sf_int%reduce_momenta (x)
  end if
  select case (sf_int%status)
  case (SF_FAILED_KINEMATICS);  f = 0
  end select
end subroutine circe1_complete_kinematics
```

Compute inverse kinematics. In generator mode, the r values are meaningless, but we copy them anyway.

⟨*SF circe1: circe1: TBP*⟩+≡

```
procedure :: inverse_kinematics => circe1_inverse_kinematics
```

⟨*SF circe1: procedures*⟩+≡

```
subroutine circe1_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
  class(circe1_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  real(default), dimension(:), intent(out) :: rb
  logical, intent(in) :: map
```

```
    logical, intent(in), optional :: set_momenta
    real(default), dimension(2) :: xb1
    logical :: set_mom
    set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
    r = x
    rb = 1 - x
    sf_int%x = x
    f = 1
    if (set_mom) then
       xb1 = 1 - x
       call sf_int%split_momenta (x, xb1)
       select case (sf_int%status)
       case (SF_FAILED_KINEMATICS);  f = 0
       end select
    end if
  end subroutine circe1_inverse_kinematics
```

15.12.6 CIRCE1 application

CIRCE is applied for the two beams at once. We can safely assume that no structure functions are applied before this, so the incoming particles are on-shell electrons/positrons.

The scale is ignored.

⟨*SF circe1: circe1: TBP*⟩+≡

```
  procedure :: apply => circe1_apply
```

⟨*SF circe1: procedures*⟩+≡

```
  subroutine circe1_apply (sf_int, scale)
    class(circe1_t), intent(inout) :: sf_int
    real(default), intent(in) :: scale
    real(default), dimension(2) :: xb
    real(double), dimension(2) :: xc
    real(double), parameter :: one = 1
    associate (data => sf_int%data)
      xc = sf_int%x
      xb = 1 - sf_int%x
      if (data%generate) then
         sf_int%f = 1
      else
         sf_int%f = 0
         if (all (sf_int%continuum)) then
            sf_int%f = circe (xc(1), xc(2), data%pdg_in(1), data%pdg_in(2))
         end if
         if (sf_int%continuum(2) .and. sf_int%peak(1)) then
            sf_int%f = sf_int%f &
                 + circe (one, xc(2), data%pdg_in(1), data%pdg_in(2)) &
                 * peak (xb(1), data%eps)
         end if
         if (sf_int%continuum(1) .and. sf_int%peak(2)) then
            sf_int%f = sf_int%f &
                 + circe (xc(1), one, data%pdg_in(1), data%pdg_in(2)) &
                 * peak (xb(2), data%eps)
         end if
```

```
          if (all (sf_int%peak)) then
             sf_int%f = sf_int%f &
                  + circe (one, one, data%pdg_in(1), data%pdg_in(2)) &
                  * peak (xb(1), data%eps) * peak (xb(2), data%eps)
          end if
        end if
      end associate
      call sf_int%set_matrix_element (cmplx (sf_int%f, kind=default))
      sf_int%status = SF_EVALUATED
    end subroutine circe1_apply
```

This is a smeared delta peak at zero, as an endpoint singularity. We choose an exponentially decreasing function, starting at zero, with integral (from 0 to 1) $1 - e^{-1/\epsilon}$. For small ϵ, this reduces to one.

⟨*SF circe1: procedures*⟩+≡

```
    function peak (x, eps) result (f)
      real(default), intent(in) :: x, eps
      real(default) :: f
      f = exp (-x / eps) / eps
    end function peak
```

15.12.7 Unit tests

Test module, followed by the corresponding implementation module.

⟨sf_circe1_ut.f90⟩≡

```
  ⟨File header⟩

  module sf_circe1_ut
    use unit_tests
    use sf_circe1_uti

  ⟨Standard module head⟩

  ⟨SF circe1: public test⟩

  contains

  ⟨SF circe1: test driver⟩

  end module sf_circe1_ut
```

⟨sf_circe1_uti.f90⟩≡

```
  ⟨File header⟩

  module sf_circe1_uti

  ⟨Use kinds⟩
    use physics_defs, only: ELECTRON
    use lorentz
    use pdg_arrays
    use flavors
    use interactions, only: reset_interaction_counter
```

```
  use model_data
  use rng_base
  use sf_aux
  use sf_base

  use sf_circe1

  use rng_base_ut, only: rng_test_factory_t

⟨Standard module head⟩

⟨SF circe1: test declarations⟩

contains

⟨SF circe1: tests⟩

end module sf_circe1_uti
```

API: driver for the unit tests below.

⟨*SF circe1: public test*⟩≡
```
  public :: sf_circe1_test
```

⟨*SF circe1: test driver*⟩≡
```
  subroutine sf_circe1_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨SF circe1: execute tests⟩
  end subroutine sf_circe1_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF circe1: execute tests*⟩≡
```
  call test (sf_circe1_1, "sf_circe1_1", &
       "structure function configuration", &
       u, results)
```

⟨*SF circe1: test declarations*⟩≡
```
  public :: sf_circe1_1
```

⟨*SF circe1: tests*⟩≡
```
  subroutine sf_circe1_1 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(pdg_array_t), dimension(2) :: pdg_in
    type(pdg_array_t), dimension(2) :: pdg_out
    integer, dimension(:), allocatable :: pdg1, pdg2
    class(sf_data_t), allocatable :: data

    write (u, "(A)")  "* Test output: sf_circe1_1"
    write (u, "(A)")  "*   Purpose: initialize and display &
         &CIRCE structure function data"
    write (u, "(A)")
```

```
    write (u, "(A)")  "* Create empty data object"
    write (u, "(A)")

    call model%init_qed_test ()
    pdg_in(1) = ELECTRON
    pdg_in(2) = -ELECTRON

    allocate (circe1_data_t :: data)
    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize"
    write (u, "(A)")

    select type (data)
    type is (circe1_data_t)
       call data%init (model, pdg_in, &
            sqrts = 500._default, &
            eps = 1e-6_default, &
            out_photon = [.false., .false.], &
            ver = 0, &
            rev = 0, &
            acc = "SBAND", &
            chat = 0, &
            with_radiation = .true.)
    end select

    call data%write (u)

    write (u, "(A)")

    write (u, "(1x,A)")  "Outgoing particle codes:"
    call data%get_pdg_out (pdg_out)
    pdg1 = pdg_out(1)
    pdg2 = pdg_out(2)
    write (u, "(2x,99(1x,I0))")  pdg1, pdg2

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_circe1_1"

  end subroutine sf_circe1_1
```

Test and probe structure function

Construct and display a structure function object based on the PDF builtin structure function.

⟨*SF circe1: execute tests*⟩+≡

```
  call test (sf_circe1_2, "sf_circe1_2", &
       "structure function instance", &
       u, results)
```

⟨*SF circe1: test declarations*⟩+≡

```
public :: sf_circe1_2
```

⟨*SF circe1: tests*⟩+≡

```
subroutine sf_circe1_2 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t), dimension(2) :: flv
  type(pdg_array_t), dimension(2) :: pdg_in
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k1, k2
  type(vector4_t), dimension(4) :: q
  real(default) :: E
  real(default), dimension(:), allocatable :: r, rb, x
  real(default) :: f

  write (u, "(A)")  "* Test output: sf_circe1_2"
  write (u, "(A)")  "*   Purpose: initialize and fill &
       &circe1 structure function object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call model%init_qed_test ()
  call flv(1)%init (ELECTRON, model)
  call flv(2)%init (-ELECTRON, model)
  pdg_in(1) = ELECTRON
  pdg_in(2) = -ELECTRON

  call reset_interaction_counter ()

  allocate (circe1_data_t :: data)
  select type (data)
  type is (circe1_data_t)
     call data%init (model, pdg_in, &
          sqrts = 500._default, &
          eps = 1e-6_default, &
          out_photon = [.false., .false.], &
          ver = 0, &
          rev = 0, &
          acc = "SBAND", &
          chat = 0, &
          with_radiation = .true.)
  end select

  write (u, "(A)")  "* Initialize structure-function object"
  write (u, "(A)")

  call data%allocate_sf_int (sf_int)
  call sf_int%init (data)
  call sf_int%set_beam_index ([1,2])

  call sf_int%write (u)
```

```
write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 250
k1 = vector4_moving (E, sqrt (E**2 - flv(1)%get_mass ()**2), 3)
k2 = vector4_moving (E,-sqrt (E**2 - flv(2)%get_mass ()**2), 3)
call vector4_write (k1, u)
call vector4_write (k2, u)
call sf_int%seed_kinematics ([k1, k2])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.95,0.85."
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = [0.9_default, 0.8_default]
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = sf_int%get_momenta (outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1, 2])

call sf_int%seed_kinematics ([k1, k2])
call sf_int%set_momenta (q, outgoing=.true.)
call sf_int%recover_x (x)

write (u, "(A,9(1x,F10.7))")  "x =", x

write (u, "(A)")
write (u, "(A)")  "* Evaluate"
write (u, "(A)")

call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%apply (scale = 0._default)
call sf_int%write (u)

write (u, "(A)")
```

```
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_circe1_2"

  end subroutine sf_circe1_2
```

Generator mode

Construct and evaluate a structure function object in generator mode.

⟨*SF circe1: execute tests*⟩+≡

```
  call test (sf_circe1_3, "sf_circe1_3", &
       "generator mode", &
       u, results)
```

⟨*SF circe1: test declarations*⟩+≡

```
  public :: sf_circe1_3
```

⟨*SF circe1: tests*⟩+≡

```
  subroutine sf_circe1_3 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t), dimension(2) :: flv
    type(pdg_array_t), dimension(2) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    class(rng_factory_t), allocatable :: rng_factory
    class(sf_int_t), allocatable :: sf_int
    type(vector4_t) :: k1, k2
    real(default) :: E
    real(default), dimension(:), allocatable :: r, rb, x
    real(default) :: f, x_free

    write (u, "(A)")  "* Test output: sf_circe1_3"
    write (u, "(A)")  "*   Purpose: initialize and fill &
         &circe1 structure function object"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize configuration data"
    write (u, "(A)")

    call model%init_qed_test ()
    call flv(1)%init (ELECTRON, model)
    call flv(2)%init (-ELECTRON, model)
    pdg_in(1) = ELECTRON
    pdg_in(2) = -ELECTRON

    call reset_interaction_counter ()

    allocate (circe1_data_t :: data)
    allocate (rng_test_factory_t :: rng_factory)
    select type (data)
```

```
type is (circe1_data_t)
   call data%init (model, pdg_in, &
        sqrts = 500._default, &
        eps = 1e-6_default, &
        out_photon = [.false., .false.], &
        ver = 0, &
        rev = 0, &
        acc = "SBAND", &
        chat = 0, &
        with_radiation = .true.)
   call data%set_generator_mode (rng_factory)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1,2])
select type (sf_int)
type is (circe1_t)
   call sf_int%rng_obj%rng%init (3)
end select

write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 250
k1 = vector4_moving (E, sqrt (E**2 - flv(1)%get_mass ()**2), 3)
k2 = vector4_moving (E,-sqrt (E**2 - flv(2)%get_mass ()**2), 3)
call vector4_write (k1, u)
call vector4_write (k2, u)
call sf_int%seed_kinematics ([k1, k2])

write (u, "(A)")
write (u, "(A)")  "* Generate x"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r  = 0
rb = 0
x_free = 1
call sf_int%generate_free (r, rb, x_free)
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f
write (u, "(A,9(1x,F10.7))")  "xf=", x_free

write (u, "(A)")
write (u, "(A)")  "* Evaluate"
write (u, "(A)")
```

```
  call sf_int%apply (scale = 0._default)
  call sf_int%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call sf_int%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_circe1_3"

end subroutine sf_circe1_3
```

15.13 Lepton Collider Beamstrahlung and Photon collider: CIRCE2

⟨sf_circe2.f90⟩≡

```
⟨File header⟩

module sf_circe2

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_defs, only: FMT_19
  use numeric_utils
  use diagnostics
  use os_interface
  use physics_defs, only: PHOTON, ELECTRON
  use lorentz
  use rng_base
  use selectors
  use pdg_arrays
  use model_data
  use flavors
  use colors
  use helicities
  use quantum_numbers
  use state_matrices
  use polarizations
  use sf_base
  use circe2, circe2_rng_t => rng_type !NODEP!

⟨Standard module head⟩

⟨SF circe2: public⟩

⟨SF circe2: types⟩

contains

⟨SF circe2: procedures⟩

end module sf_circe2
```

15.13.1 Physics

CIRCE2 describes photon spectra Beamstrahlung is applied before ISR. The CIRCE2 implementation has a single structure function for both beams (which makes sense since it has to be switched on or off for both beams simultaneously).

15.13.2 The CIRCE2 data block

The CIRCE2 parameters are: file and collider specification, incoming (= outgoing) particles. The luminosity is returned by circe2_luminosity.

⟨*SF circe2: public*⟩≡
```
public :: circe2_data_t
```

⟨*SF circe2: types*⟩≡
```
type, extends (sf_data_t) :: circe2_data_t
   private
   class(model_data_t), pointer :: model => null ()
   type(flavor_t), dimension(2) :: flv_in
   integer, dimension(2) :: pdg_in
   real(default) :: sqrts = 0
   logical :: polarized = .false.
   logical :: beams_polarized = .false.
   class(rng_factory_t), allocatable :: rng_factory
   type(string_t) :: filename
   type(string_t) :: file
   type(string_t) :: design
   real(default) :: lumi = 0
   real(default), dimension(4) :: lumi_hel_frac = 0
   integer, dimension(0:4) :: h1 = [0, -1, -1, 1, 1]
   integer, dimension(0:4) :: h2 = [0, -1,  1,-1, 1]
   integer :: error = 1
 contains
   ⟨SF circe2: circe2 data: TBP⟩
end type circe2_data_t
```

⟨*SF circe2: types*⟩+≡
```
type(circe2_state) :: circe2_global_state
```

⟨*SF circe2: circe2 data: TBP*⟩≡
```
procedure :: init => circe2_data_init
```

⟨*SF circe2: procedures*⟩≡
```
subroutine circe2_data_init (data, os_data, model, pdg_in, &
     sqrts, polarized, beam_pol, file, design)
  class(circe2_data_t), intent(out) :: data
  type(os_data_t), intent(in) :: os_data
  class(model_data_t), intent(in), target :: model
  type(pdg_array_t), dimension(2), intent(in) :: pdg_in
  real(default), intent(in) :: sqrts
  logical, intent(in) :: polarized, beam_pol
  type(string_t), intent(in) :: file, design
  integer :: h
  data%model => model
  if (any (pdg_array_get_length (pdg_in) /= 1)) then
     call msg_fatal ("CIRCE2: incoming beam particles must be unique")
  end if
  call data%flv_in(1)%init (pdg_array_get (pdg_in(1), 1), model)
  call data%flv_in(2)%init (pdg_array_get (pdg_in(2), 1), model)
  data%pdg_in = data%flv_in%get_pdg ()
  data%sqrts = sqrts
  data%polarized = polarized
  data%beams_polarized = beam_pol
  data%filename = file
  data%design = design
```

```
    call data%check_file (os_data)
    call circe2_load (circe2_global_state, trim (char(data%file)), &
            trim (char(data%design)), data%sqrts, data%error)
    call data%check ()
    data%lumi = circe2_luminosity (circe2_global_state, data%pdg_in, [0, 0])
    if (vanishes (data%lumi)) then
       call msg_fatal ("CIRCE2: luminosity vanishes for specified beams.")
    end if
    if (data%polarized) then
       do h = 1, 4
          data%lumi_hel_frac(h) = &
               circe2_luminosity (circe2_global_state, data%pdg_in, &
                                  [data%h1(h), data%h2(h)]) &
               / data%lumi
       end do
    end if
  end subroutine circe2_data_init
```

Activate the generator mode. We import a RNG factory into the data type, which can then spawn RNG generator objects.

⟨*SF circe2: circe2 data: TBP*⟩+≡

```
  procedure :: set_generator_mode => circe2_data_set_generator_mode
```

⟨*SF circe2: procedures*⟩+≡

```
  subroutine circe2_data_set_generator_mode (data, rng_factory)
    class(circe2_data_t), intent(inout) :: data
    class(rng_factory_t), intent(inout), allocatable :: rng_factory
    call move_alloc (from = rng_factory, to = data%rng_factory)
  end subroutine circe2_data_set_generator_mode
```

Check whether the requested data file is in the system directory or in the current directory.

⟨*SF circe2: circe2 data: TBP*⟩+≡

```
  procedure :: check_file => circe2_check_file
```

⟨*SF circe2: procedures*⟩+≡

```
  subroutine circe2_check_file (data, os_data)
    class(circe2_data_t), intent(inout) :: data
    type(os_data_t), intent(in) :: os_data
    logical :: exist
    type(string_t) :: file
    file = data%filename
    if (file == "") &
         call msg_fatal ("CIRCE2: $circe2_file is not set")
    inquire (file = char (file), exist = exist)
    if (exist) then
       data%file = file
    else
       file = os_data%whizard_circe2path // "/" // data%filename
       inquire (file = char (file), exist = exist)
       if (exist) then
          data%file = file
       else
          call msg_fatal ("CIRCE2: data file '" // char (data%filename) &
```

```
                // "' not found")
         end if
      end if
   end subroutine circe2_check_file
```

Handle error conditions.

⟨*SF circe2: circe2 data: TBP*⟩+≡

```
   procedure :: check => circe2_data_check
```

⟨*SF circe2: procedures*⟩+≡

```
   subroutine circe2_data_check (data)
      class(circe2_data_t), intent(in) :: data
      type(flavor_t) :: flv_photon, flv_electron
      call flv_photon%init (PHOTON, data%model)
      if (.not. flv_photon%is_defined ()) then
         call msg_fatal ("CIRCE2: model must contain photon")
      end if
      call flv_electron%init (ELECTRON, data%model)
      if (.not. flv_electron%is_defined ()) then
         call msg_fatal ("CIRCE2: model must contain electron")
      end if
      if (any (abs (data%pdg_in) /= PHOTON .and. abs (data%pdg_in) /= ELECTRON)) &
           then
         call msg_fatal ("CIRCE2: applicable only for e+e- or photon collisions")
      end if
      select case (data%error)
      case (-1)
         call msg_fatal ("CIRCE2: data file not found.")
      case (-2)
         call msg_fatal ("CIRCE2: beam setup does not match data file.")
      case (-3)
         call msg_fatal ("CIRCE2: invalid format of data file.")
      case (-4)
         call msg_fatal ("CIRCE2: data file too large.")
      end select
   end subroutine circe2_data_check
```

Output

⟨*SF circe2: circe2 data: TBP*⟩+≡

```
   procedure :: write => circe2_data_write
```

⟨*SF circe2: procedures*⟩+≡

```
   subroutine circe2_data_write (data, unit, verbose)
      class(circe2_data_t), intent(in) :: data
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: verbose
      integer :: u, h
      u = given_output_unit (unit)
      write (u, "(1x,A)") "CIRCE2 data:"
      write (u, "(3x,A,A)")       "file   = ", char(data%filename)
      write (u, "(3x,A,A)")       "design = ", char(data%design)
      write (u, "(3x,A," // FMT_19 // ")") "sqrts  = ", data%sqrts
      write (u, "(3x,A,A,A,A)")   "prt_in = ", &
           char (data%flv_in(1)%get_name ()), &
```

```
        ", ", char (data%flv_in(2)%get_name ())
    write (u, "(3x,A,L1)")      "polarized  = ", data%polarized
    write (u, "(3x,A,L1)")      "beams pol. = ", data%beams_polarized
    write (u, "(3x,A," // FMT_19 // ")") "luminosity = ", data%lumi
    if (data%polarized) then
       do h = 1, 4
          write (u, "(6x,'(',I2,1x,I2,')',1x,'=',1x)", advance="no") &
               data%h1(h), data%h2(h)
          write (u, "(6x, " // FMT_19 // ")")  data%lumi_hel_frac(h)
       end do
    end if
    call data%rng_factory%write (u)
  end subroutine circe2_data_write
```

This is always in generator mode.

⟨*SF circe2: circe2 data: TBP*⟩+≡

```
  procedure :: is_generator => circe2_data_is_generator
```

⟨*SF circe2: procedures*⟩+≡

```
  function circe2_data_is_generator (data) result (flag)
    class(circe2_data_t), intent(in) :: data
    logical :: flag
    flag = .true.
  end function circe2_data_is_generator
```

The number of parameters is two, collinear splitting for the two beams.

⟨*SF circe2: circe2 data: TBP*⟩+≡

```
  procedure :: get_n_par => circe2_data_get_n_par
```

⟨*SF circe2: procedures*⟩+≡

```
  function circe2_data_get_n_par (data) result (n)
    class(circe2_data_t), intent(in) :: data
    integer :: n
    n = 2
  end function circe2_data_get_n_par
```

Return the outgoing particles PDG codes. They are equal to the incoming ones.

⟨*SF circe2: circe2 data: TBP*⟩+≡

```
  procedure :: get_pdg_out => circe2_data_get_pdg_out
```

⟨*SF circe2: procedures*⟩+≡

```
  subroutine circe2_data_get_pdg_out (data, pdg_out)
    class(circe2_data_t), intent(in) :: data
    type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
    integer :: i, n
    n = 2
    do i = 1, n
       pdg_out(i) = data%pdg_in(i)
    end do
  end subroutine circe2_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF circe2: circe2 data: TBP*⟩+≡

```
  procedure :: allocate_sf_int => circe2_data_allocate_sf_int
```

⟨*SF circe2: procedures*⟩+≡

```
subroutine circe2_data_allocate_sf_int (data, sf_int)
  class(circe2_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  allocate (circe2_t :: sf_int)
end subroutine circe2_data_allocate_sf_int
```

Return the beam file.

⟨*SF circe2: circe2 data: TBP*⟩+≡

```
procedure :: get_beam_file => circe2_data_get_beam_file
```

⟨*SF circe2: procedures*⟩+≡

```
function circe2_data_get_beam_file (data) result (file)
  class(circe2_data_t), intent(in) :: data
  type(string_t) :: file
  file = "CIRCE2: " // data%filename
end function circe2_data_get_beam_file
```

15.13.3 Random Number Generator for CIRCE

The CIRCE implementation now supports a generic random-number generator object that allows for a local state as a component. To support this, we must extend the abstract type provided by CIRCE and delegate the generator call to the (also abstract) RNG used by WHIZARD.

⟨*SF circe2: types*⟩+≡

```
type, extends (circe2_rng_t) :: rng_obj_t
   class(rng_t), allocatable :: rng
 contains
   procedure :: generate => rng_obj_generate
end type rng_obj_t
```

⟨*SF circe2: procedures*⟩+≡

```
subroutine rng_obj_generate (rng_obj, u)
  class(rng_obj_t), intent(inout) :: rng_obj
  real(default), intent(out) :: u
  real(default) :: x
  call rng_obj%rng%generate (x)
  u = x
end subroutine rng_obj_generate
```

15.13.4 The CIRCE2 object

For CIRCE2 spectra it does not make sense to describe the state matrix as a radiation interaction, even if photons originate from laser backscattering. Instead, it is a $2 \to 2$ interaction where the incoming particles are identical to the outgoing ones.

The current implementation of CIRCE2 does support polarization and classical correlations, but no entanglement, so the density matrix of the outgoing

particles is diagonal. The incoming particles are unpolarized (user-defined polarization for beams is meaningless, since polarization is described by the data file). The outgoing particles are polarized or polarization-averaged, depending on user request.

When assigning matrix elements, we scan the previously initialized state matrix. For each entry, we extract helicity and call the structure function. In the unpolarized case, the helicity is undefined and replaced by value zero. In the polarized case, there are four entries. If the generator is used, only one entry is nonzero in each call. Which one, is determined by comparing with a previously (randomly, distributed by relative luminosity) selected pair of helicities.

⟨*SF circe2: public*⟩+≡
```
public :: circe2_t
```

⟨*SF circe2: types*⟩+≡
```
type, extends (sf_int_t) :: circe2_t
   type(circe2_data_t), pointer :: data => null ()
   type(rng_obj_t) :: rng_obj
   type(selector_t) :: selector
   integer :: h_sel = 0
 contains
 ⟨SF circe2: circe2: TBP⟩
end type circe2_t
```

Type string: show file and design of `CIRCE2` structure function.

⟨*SF circe2: circe2: TBP*⟩≡
```
procedure :: type_string => circe2_type_string
```

⟨*SF circe2: procedures*⟩+≡
```
function circe2_type_string (object) result (string)
  class(circe2_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "CIRCE2: " // object%data%design
  else
     string = "CIRCE2: [undefined]"
  end if
end function circe2_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF circe2: circe2: TBP*⟩+≡
```
procedure :: write => circe2_write
```

⟨*SF circe2: procedures*⟩+≡
```
subroutine circe2_write (object, unit, testflag)
  class(circe2_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  if (associated (object%data)) then
     call object%data%write (u)
     call object%base_write (u, testflag)
  else
```

```
      write (u, "(1x,A)")  "CIRCE2 data: [undefined]"
    end if
  end subroutine circe2_write
```

⟨*SF circe2: circe2: TBP*⟩+≡

```
  procedure :: init => circe2_init
```

⟨*SF circe2: procedures*⟩+≡

```
  subroutine circe2_init (sf_int, data)
    class(circe2_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    logical, dimension(4) :: mask_h
    real(default), dimension(0) :: null_array
    type(quantum_numbers_mask_t), dimension(4) :: mask
    type(quantum_numbers_t), dimension(4) :: qn
    type(helicity_t) :: hel
    type(color_t) :: col0
    integer :: h
    select type (data)
    type is (circe2_data_t)
       if (data%polarized .and. data%beams_polarized) then
          call msg_fatal ("CIRCE2: Beam polarization can't be set &
               &for polarized data file")
       else if (data%beams_polarized) then
          call msg_warning ("CIRCE2: User-defined beam polarization set &
               &for unpolarized CIRCE2 data file")
       end if
       mask_h(1:2) = .not. data%beams_polarized
       mask_h(3:4) = .not. (data%polarized .or. data%beams_polarized)
       mask = quantum_numbers_mask (.false., .false., mask_h)
       call sf_int%base_init (mask, [0._default, 0._default], &
            null_array, [0._default, 0._default])
       sf_int%data => data
       if (data%polarized) then
          if (vanishes (sum (data%lumi_hel_frac)) .or. &
               any (data%lumi_hel_frac < 0)) then
             call msg_fatal ("CIRCE2: Helicity-dependent lumi " &
                  // "fractions all vanish or",  &
                  [var_str ("are negative: Please inspect the " &
                  // "CIRCE2 file or "), &
                   var_str ("switch off the polarized" // &
                  " option for CIRCE2.")])
          else
             call sf_int%selector%init (data%lumi_hel_frac)
          end if
       end if
       call col0%init ()
       if (data%beams_polarized) then
          do h = 1, 4
             call hel%init (data%h1(h))
             call qn(1)%init &
                  (flv = data%flv_in(1), col = col0, hel = hel)
             call qn(3)%init &
                  (flv = data%flv_in(1), col = col0, hel = hel)
```

```
            call hel%init (data%h2(h))
            call qn(2)%init &
                 (flv = data%flv_in(2), col = col0, hel = hel)
            call qn(4)%init &
                 (flv = data%flv_in(2), col = col0, hel = hel)
            call sf_int%add_state (qn)
         end do
      else if (data%polarized) then
         call qn(1)%init (flv = data%flv_in(1), col = col0)
         call qn(2)%init (flv = data%flv_in(2), col = col0)
         do h = 1, 4
            call hel%init (data%h1(h))
            call qn(3)%init &
                 (flv = data%flv_in(1), col = col0, hel = hel)
            call hel%init (data%h2(h))
            call qn(4)%init &
                 (flv = data%flv_in(2), col = col0, hel = hel)
            call sf_int%add_state (qn)
         end do
      else
         call qn(1)%init (flv = data%flv_in(1), col = col0)
         call qn(2)%init (flv = data%flv_in(2), col = col0)
         call qn(3)%init (flv = data%flv_in(1), col = col0)
         call qn(4)%init (flv = data%flv_in(2), col = col0)
         call sf_int%add_state (qn)
      end if
      call sf_int%freeze ()
      call sf_int%set_incoming ([1,2])
      call sf_int%set_outgoing ([3,4])
      call sf_int%data%rng_factory%make (sf_int%rng_obj%rng)
      sf_int%status = SF_INITIAL
    end select
  end subroutine circe2_init
```

15.13.5 Kinematics

Refer to the `data` component.

⟨*SF circe2: circe2: TBP*⟩+≡

```
  procedure :: is_generator => circe2_is_generator
```

⟨*SF circe2: procedures*⟩+≡

```
  function circe2_is_generator (sf_int) result (flag)
    class(circe2_t), intent(in) :: sf_int
    logical :: flag
    flag = sf_int%data%is_generator ()
  end function circe2_is_generator
```

Generate free parameters. We first select a helicity, which we have to store, then generate x values for that helicity.

⟨*SF circe2: circe2: TBP*⟩+≡

```
  procedure :: generate_free => circe2_generate_whizard_free
```

⟨*SF circe2: procedures*⟩+≡

```
subroutine circe2_generate_whizard_free (sf_int, r, rb, x_free)
  class(circe2_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: r, rb
  real(default), intent(inout) :: x_free
  integer :: h_sel
  if (sf_int%data%polarized) then
     call sf_int%selector%generate (sf_int%rng_obj%rng, h_sel)
  else
     h_sel = 0
  end if
  sf_int%h_sel = h_sel
  call circe2_generate_whizard (r, sf_int%data%pdg_in, &
       [sf_int%data%h1(h_sel), sf_int%data%h2(h_sel)], &
       sf_int%rng_obj)
  rb = 1 - r
  x_free = x_free * product (r)
end subroutine circe2_generate_whizard_free
```

Generator mode: call the CIRCE2 generator for the given particles and helicities. (For unpolarized generation, helicities are zero.)

⟨*SF circe2: procedures*⟩+≡

```
subroutine circe2_generate_whizard (x, pdg, hel, rng_obj)
  real(default), dimension(2), intent(out) :: x
  integer, dimension(2), intent(in) :: pdg
  integer, dimension(2), intent(in) :: hel
  class(rng_obj_t), intent(inout) :: rng_obj
  call circe2_generate (circe2_global_state, rng_obj, x, pdg, hel)
end subroutine circe2_generate_whizard
```

Set kinematics. Trivial here.

⟨*SF circe2: circe2: TBP*⟩+≡

```
procedure :: complete_kinematics => circe2_complete_kinematics
```

⟨*SF circe2: procedures*⟩+≡

```
subroutine circe2_complete_kinematics (sf_int, x, f, r, rb, map)
  class(circe2_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  real(default), dimension(:), intent(in) :: rb
  logical, intent(in) :: map
  if (map) then
     call msg_fatal ("CIRCE2: map flag not supported")
  else
     x = r
     f = 1
  end if
  call sf_int%reduce_momenta (x)
end subroutine circe2_complete_kinematics
```

Compute inverse kinematics.

⟨*SF circe2: circe2: TBP*⟩+≡

```
procedure :: inverse_kinematics => circe2_inverse_kinematics
```

⟨*SF circe2: procedures*⟩+≡

```
subroutine circe2_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
  class(circe2_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  real(default), dimension(:), intent(out) :: rb
  logical, intent(in) :: map
  logical, intent(in), optional :: set_momenta
  logical :: set_mom
  set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
  if (map) then
     call msg_fatal ("CIRCE2: map flag not supported")
  else
     r = x
     rb= 1 - r
     f = 1
  end if
  if (set_mom) then
     call sf_int%reduce_momenta (x)
  end if
end subroutine circe2_inverse_kinematics
```

15.13.6 CIRCE2 application

This function works on both beams. In polarized mode, we set only the selected helicity. In unpolarized mode, the interaction has only one entry, and the factor is unity.

⟨*SF circe2: circe2: TBP*⟩+≡

```
procedure :: apply => circe2_apply
```

⟨*SF circe2: procedures*⟩+≡

```
subroutine circe2_apply (sf_int, scale)
  class(circe2_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  complex(default) :: f
  associate (data => sf_int%data)
    f = 1
    if (data%beams_polarized) then
       call sf_int%set_matrix_element (f)
    else if (data%polarized) then
       call sf_int%set_matrix_element (sf_int%h_sel, f)
    else
       call sf_int%set_matrix_element (1, f)
    end if
  end associate
  sf_int%status = SF_EVALUATED
end subroutine circe2_apply
```

15.13.7 Unit tests

Test module, followed by the corresponding implementation module.

⟨sf_circe2_ut.f90⟩≡

```
⟨File header⟩

module sf_circe2_ut
  use unit_tests
  use sf_circe2_uti

⟨Standard module head⟩

⟨SF circe2: public test⟩

contains

⟨SF circe2: test driver⟩

end module sf_circe2_ut
```

⟨sf_circe2_uti.f90⟩≡

```
⟨File header⟩

module sf_circe2_uti

⟨Use kinds⟩
⟨Use strings⟩
  use os_interface
  use physics_defs, only: PHOTON
  use lorentz
  use pdg_arrays
  use flavors
  use interactions, only: reset_interaction_counter
  use model_data
  use rng_base
  use sf_aux
  use sf_base

  use sf_circe2

  use rng_base_ut, only: rng_test_factory_t

⟨Standard module head⟩

⟨SF circe2: test declarations⟩

contains

⟨SF circe2: tests⟩

end module sf_circe2_uti
```

API: driver for the unit tests below.

⟨*SF circe2: public test*⟩≡

```
  public :: sf_circe2_test
```

⟨*SF circe2: test driver*⟩≡

```
subroutine sf_circe2_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨SF circe2: execute tests⟩
end subroutine sf_circe2_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF circe2: execute tests*⟩≡

```
call test (sf_circe2_1, "sf_circe2_1", &
     "structure function configuration", &
     u, results)
```

⟨*SF circe2: test declarations*⟩≡

```
public :: sf_circe2_1
```

⟨*SF circe2: tests*⟩≡

```
subroutine sf_circe2_1 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_data_t), target :: model
  type(pdg_array_t), dimension(2) :: pdg_in
  type(pdg_array_t), dimension(2) :: pdg_out
  integer, dimension(:), allocatable :: pdg1, pdg2
  class(sf_data_t), allocatable :: data
  class(rng_factory_t), allocatable :: rng_factory

  write (u, "(A)")  "* Test output: sf_circe2_1"
  write (u, "(A)")  "*   Purpose: initialize and display &
       &CIRCE structure function data"
  write (u, "(A)")

  write (u, "(A)")  "* Create empty data object"
  write (u, "(A)")

  call os_data_init (os_data)
  call model%init_qed_test ()
  pdg_in(1) = PHOTON
  pdg_in(2) = PHOTON

  allocate (circe2_data_t :: data)
  allocate (rng_test_factory_t :: rng_factory)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize (unpolarized)"
  write (u, "(A)")

  select type (data)
  type is (circe2_data_t)
     call data%init (os_data, model, pdg_in, &
          sqrts = 500._default, &
          polarized = .false., &
```

```
            beam_pol = .false., &
            file = var_str ("teslagg_500_polavg.circe"), &
            design = var_str ("TESLA/GG"))
         call data%set_generator_mode (rng_factory)
      end select

      call data%write (u)

      write (u, "(A)")

      write (u, "(1x,A)")  "Outgoing particle codes:"
      call data%get_pdg_out (pdg_out)
      pdg1 = pdg_out(1)
      pdg2 = pdg_out(2)
      write (u, "(2x,99(1x,I0))")  pdg1, pdg2

      write (u, "(A)")
      write (u, "(A)")  "* Initialize (polarized)"
      write (u, "(A)")

      allocate (rng_test_factory_t :: rng_factory)

      select type (data)
      type is (circe2_data_t)
         call data%init (os_data, model, pdg_in, &
            sqrts = 500._default, &
            polarized = .true., &
            beam_pol = .false., &
            file = var_str ("teslagg_500.circe"), &
            design = var_str ("TESLA/GG"))
         call data%set_generator_mode (rng_factory)
      end select

      call data%write (u)

      call model%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_circe2_1"

   end subroutine sf_circe2_1
```

Generator mode, unpolarized

Construct and evaluate a structure function object in generator mode.

⟨*SF circe2: execute tests*⟩+≡

```
   call test (sf_circe2_2, "sf_circe2_2", &
        "generator, unpolarized", &
        u, results)
```

⟨*SF circe2: test declarations*⟩+≡

```
   public :: sf_circe2_2
```

⟨*SF circe2: tests*⟩+≡

```
subroutine sf_circe2_2 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_data_t), target :: model
  type(flavor_t), dimension(2) :: flv
  type(pdg_array_t), dimension(2) :: pdg_in
  class(sf_data_t), allocatable, target :: data
  class(rng_factory_t), allocatable :: rng_factory
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k1, k2
  real(default) :: E
  real(default), dimension(:), allocatable :: r, rb, x
  real(default) :: f, x_free

  write (u, "(A)")  "* Test output: sf_circe2_2"
  write (u, "(A)")  "*   Purpose: initialize and fill &
       &circe2 structure function object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)
  call model%init_qed_test ()
  call flv(1)%init (PHOTON, model)
  call flv(2)%init (PHOTON, model)
  pdg_in(1) = PHOTON
  pdg_in(2) = PHOTON

  call reset_interaction_counter ()

  allocate (circe2_data_t :: data)
  allocate (rng_test_factory_t :: rng_factory)
  select type (data)
  type is (circe2_data_t)
     call data%init (os_data, model, pdg_in, &
          sqrts = 500._default, &
          polarized = .false., &
          beam_pol = .false., &
          file = var_str ("teslagg_500_polavg.circe"), &
          design = var_str ("TESLA/GG"))
     call data%set_generator_mode (rng_factory)
  end select

  write (u, "(A)")  "* Initialize structure-function object"
  write (u, "(A)")

  call data%allocate_sf_int (sf_int)
  call sf_int%init (data)
  call sf_int%set_beam_index ([1,2])
  select type (sf_int)
  type is (circe2_t)
     call sf_int%rng_obj%rng%init (3)
```

```
    end select

    write (u, "(A)")  "* Initialize incoming momentum with E=500"
    write (u, "(A)")
    E = 250
    k1 = vector4_moving (E, sqrt (E**2 - flv(1)%get_mass ()**2), 3)
    k2 = vector4_moving (E,-sqrt (E**2 - flv(2)%get_mass ()**2), 3)
    call vector4_write (k1, u)
    call vector4_write (k2, u)
    call sf_int%seed_kinematics ([k1, k2])

    write (u, "(A)")
    write (u, "(A)")  "* Generate x"
    write (u, "(A)")

    allocate (r (data%get_n_par ()))
    allocate (rb(size (r)))
    allocate (x (size (r)))

    r  = 0
    rb = 0
    x_free = 1
    call sf_int%generate_free (r, rb, x_free)
    call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

    write (u, "(A,9(1x,F10.7))")  "x =", x
    write (u, "(A,9(1x,F10.7))")  "f =", f
    write (u, "(A,9(1x,F10.7))")  "xf=", x_free

    write (u, "(A)")
    write (u, "(A)")  "* Evaluate"
    write (u, "(A)")

    call sf_int%apply (scale = 0._default)
    call sf_int%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call sf_int%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_circe2_2"

  end subroutine sf_circe2_2
```

Generator mode, polarized

Construct and evaluate a structure function object in generator mode.

⟨*SF circe2: execute tests*⟩+≡

```
  call test (sf_circe2_3, "sf_circe2_3", &
```

```
         "generator, polarized", &
         u, results)
```

⟨*SF circe2: test declarations*⟩+≡

```
    public :: sf_circe2_3
```

⟨*SF circe2: tests*⟩+≡

```
    subroutine sf_circe2_3 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_data_t), target :: model
      type(flavor_t), dimension(2) :: flv
      type(pdg_array_t), dimension(2) :: pdg_in
      class(sf_data_t), allocatable, target :: data
      class(rng_factory_t), allocatable :: rng_factory
      class(sf_int_t), allocatable :: sf_int
      type(vector4_t) :: k1, k2
      real(default) :: E
      real(default), dimension(:), allocatable :: r, rb, x
      real(default) :: f, x_free

      write (u, "(A)")  "* Test output: sf_circe2_3"
      write (u, "(A)")  "*   Purpose: initialize and fill &
           &circe2 structure function object"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call os_data_init (os_data)
      call model%init_qed_test ()
      call flv(1)%init (PHOTON, model)
      call flv(2)%init (PHOTON, model)
      pdg_in(1) = PHOTON
      pdg_in(2) = PHOTON

      call reset_interaction_counter ()

      allocate (circe2_data_t :: data)
      allocate (rng_test_factory_t :: rng_factory)
      select type (data)
      type is (circe2_data_t)
         call data%init (os_data, model, pdg_in, &
              sqrts = 500._default, &
              polarized = .true., &
              beam_pol = .false., &
              file = var_str ("teslagg_500.circe"), &
              design = var_str ("TESLA/GG"))
         call data%set_generator_mode (rng_factory)
      end select

      write (u, "(A)")  "* Initialize structure-function object"
      write (u, "(A)")

      call data%allocate_sf_int (sf_int)
```

```
  call sf_int%init (data)
  call sf_int%set_beam_index ([1,2])
  select type (sf_int)
  type is (circe2_t)
     call sf_int%rng_obj%rng%init (3)
  end select

  write (u, "(A)")  "* Initialize incoming momentum with E=500"
  write (u, "(A)")
  E = 250
  k1 = vector4_moving (E, sqrt (E**2 - flv(1)%get_mass ()**2), 3)
  k2 = vector4_moving (E,-sqrt (E**2 - flv(2)%get_mass ()**2), 3)
  call vector4_write (k1, u)
  call vector4_write (k2, u)
  call sf_int%seed_kinematics ([k1, k2])

  write (u, "(A)")
  write (u, "(A)")  "* Generate x"
  write (u, "(A)")

  allocate (r (data%get_n_par ()))
  allocate (rb(size (r)))
  allocate (x (size (r)))

  r  = 0
  rb = 0
  x_free = 1
  call sf_int%generate_free (r, rb, x_free)
  call sf_int%complete_kinematics (x, f, r, rb, map=.false.)

  write (u, "(A,9(1x,F10.7))")  "x =", x
  write (u, "(A,9(1x,F10.7))")  "f =", f
  write (u, "(A,9(1x,F10.7))")  "xf=", x_free

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate"
  write (u, "(A)")

  call sf_int%apply (scale = 0._default)
  call sf_int%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call sf_int%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_circe2_3"

end subroutine sf_circe2_3
```

15.14 HOPPET interface

Interface to the HOPPET wrapper necessary to perform the LO vs. NLO matching of processes containing an initial b quark.

⟨hoppet_interface.f90⟩≡

```
⟨File header⟩

module hoppet_interface
  use lhapdf !NODEP!

⟨Standard module head⟩

  public :: hoppet_init, hoppet_eval

contains

  subroutine hoppet_init (pdf_builtin, pdf, pdf_id)
    logical, intent(in) :: pdf_builtin
    type(lhapdf_pdf_t), intent(inout), optional :: pdf
    integer, intent(in), optional :: pdf_id
    external InitForWhizard
    call InitForWhizard (pdf_builtin, pdf, pdf_id)
  end subroutine hoppet_init

  subroutine hoppet_eval (x, q, f)
    double precision, intent(in)  :: x, q
    double precision, intent(out) :: f(-6:6)
    external EvalForWhizard
    call EvalForWhizard (x, q, f)
  end subroutine hoppet_eval

end module hoppet_interface
```

15.15 Builtin PDF sets

For convenience in order not to depend on the external package LHAPDF, we ship some PDFs with WHIZARD.

15.15.1 The module

⟨sf_pdf_builtin.f90⟩≡

```
⟨File header⟩

module sf_pdf_builtin

⟨Use kinds⟩
  use kinds, only: double
⟨Use strings⟩
  use io_units
  use format_defs, only: FMT_17
  use diagnostics
  use os_interface
  use physics_defs, only: PROTON, PHOTON, GLUON
  use physics_defs, only: HADRON_REMNANT_SINGLET
  use physics_defs, only: HADRON_REMNANT_TRIPLET
  use physics_defs, only: HADRON_REMNANT_OCTET
  use sm_qcd
  use lorentz
  use pdg_arrays
  use model_data
  use flavors
  use colors
  use quantum_numbers
  use state_matrices
  use polarizations
  use sf_base
  use pdf_builtin !NODEP!
  use hoppet_interface

⟨Standard module head⟩

⟨SF pdf builtin: public⟩

⟨SF pdf builtin: types⟩

⟨SF pdf builtin: parameters⟩

contains

⟨SF pdf builtin: procedures⟩

end module sf_pdf_builtin
```

15.15.2 Codes for default PDF sets

⟨*SF pdf builtin: parameters*⟩≡

```
character(*), parameter :: PDF_BUILTIN_DEFAULT_PROTON = "CTEQ6L"
character(*), parameter :: PDF_BUILTIN_DEFAULT_PION   = "NONE"
character(*), parameter :: PDF_BUILTIN_DEFAULT_PHOTON = "MRST2004QEDp"
```

15.15.3 The PDF builtin data block

The data block holds the incoming flavor (which has to be proton, pion, or photon), the corresponding pointer to the global access data (1, 2, or 3), the flag `invert` which is set for an antiproton, the bounds as returned by LHAPDF for the specified set, and a mask that determines which partons will be actually in use.

⟨*SF pdf builtin: public*⟩≡

```
public :: pdf_builtin_data_t
```

⟨*SF pdf builtin: types*⟩≡

```
type, extends (sf_data_t) :: pdf_builtin_data_t
   private
   integer :: id = -1
   type (string_t) :: name
   class(model_data_t), pointer :: model => null ()
   type(flavor_t) :: flv_in
   logical :: invert
   logical :: has_photon
   logical :: photon
   logical, dimension(-6:6) :: mask
   logical :: mask_photon
   logical :: hoppet_b_matching = .false.
 contains
 ⟨SF pdf builtin: pdf builtin data: TBP⟩
end type pdf_builtin_data_t
```

Generate PDF data and initialize the requested set. Pion and photon PDFs are disabled at the moment until we ship appropiate structure functions. needed.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩≡

```
procedure :: init => pdf_builtin_data_init
```

⟨*SF pdf builtin: procedures*⟩≡

```
subroutine pdf_builtin_data_init (data, &
     model, pdg_in, name, path, hoppet_b_matching)
  class(pdf_builtin_data_t), intent(out) :: data
  class(model_data_t), intent(in), target :: model
  type(pdg_array_t), intent(in) :: pdg_in
  type(string_t), intent(in) :: name
  type(string_t), intent(in) :: path
  logical, intent(in), optional :: hoppet_b_matching
  data%model => model
  if (pdg_array_get_length (pdg_in) /= 1) &
       call msg_fatal ("PDF: incoming particle must be unique")
  call data%flv_in%init (pdg_array_get (pdg_in, 1), model)
  data%mask = .true.
  data%mask_photon = .true.
  select case (pdg_array_get (pdg_in, 1))
  case (PROTON)
```

```
      data%name = var_str (PDF_BUILTIN_DEFAULT_PROTON)
      data%invert = .false.
      data%photon = .false.
   case (-PROTON)
      data%name = var_str (PDF_BUILTIN_DEFAULT_PROTON)
      data%invert = .true.
      data%photon = .false.
      ! case (PIPLUS)
      !    data%name = var_str (PDF_BUILTIN_DEFAULT_PION)
      !    data%invert = .false.
      !    data%photon = .false.
      ! case (-PIPLUS)
      !    data%name = var_str (PDF_BUILTIN_DEFAULT_PION)
      !    data%invert = .true.
      !    data%photon = .false.
      ! case (PHOTON)
      !    data%name = var_str (PDF_BUILTIN_DEFAULT_PHOTON)
      !    data%invert = .false.
      !    data%photon = .true.
   case default
      call msg_fatal ("PDF: " &
           // "incoming particle must either proton or antiproton.")
      return
   end select
   data%name = name
   data%id = pdf_get_id (data%name)
   if (data%id < 0) call msg_fatal ("unknown PDF set " // char (data%name))
   data%has_photon = pdf_provides_photon (data%id)
   if (present (hoppet_b_matching))  data%hoppet_b_matching = hoppet_b_matching
   call pdf_init (data%id, path)
   if (data%hoppet_b_matching)  call hoppet_init (.true., pdf_id = data%id)
end subroutine pdf_builtin_data_init
```

Enable/disable partons explicitly. If a mask entry is true, applying the PDF will generate the corresponding flavor on output.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩+≡

```
procedure :: set_mask => pdf_builtin_data_set_mask
```

⟨*SF pdf builtin: procedures*⟩+≡

```
subroutine pdf_builtin_data_set_mask (data, mask)
  class(pdf_builtin_data_t), intent(inout) :: data
  logical, dimension(-6:6), intent(in) :: mask
  data%mask = mask
end subroutine pdf_builtin_data_set_mask
```

Output.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩+≡

```
procedure :: write => pdf_builtin_data_write
```

⟨*SF pdf builtin: procedures*⟩+≡

```
subroutine pdf_builtin_data_write (data, unit, verbose)
  class(pdf_builtin_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
```

```
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)")  "PDF builtin data:"
  if (data%id < 0) then
     write (u, "(3x,A)") "[undefined]"
     return
  end if
  write (u, "(3x,A)", advance="no") "flavor       = "
  call data%flv_in%write (u);  write (u, *)
  write (u, "(3x,A,A)")   "name         = ", char (data%name)
  write (u, "(3x,A,L1)")  "invert       = ", data%invert
  write (u, "(3x,A,L1)")  "has photon   = ", data%has_photon
  write (u, "(3x,A,6(1x,L1),1x,A,1x,L1,1x,A,6(1x,L1))") &
       "mask         =", &
       data%mask(-6:-1), "*", data%mask(0), "*", data%mask(1:6)
  write (u, "(3x,A,L1)") "photon mask  = ", data%mask_photon
  write (u, "(3x,A,L1)") "hoppet_b     = ", data%hoppet_b_matching
end subroutine pdf_builtin_data_write
```

The number of parameters is one. We do not generate transverse momentum.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩+≡

```
procedure :: get_n_par => pdf_builtin_data_get_n_par
```

⟨*SF pdf builtin: procedures*⟩+≡

```
function pdf_builtin_data_get_n_par (data) result (n)
  class(pdf_builtin_data_t), intent(in) :: data
  integer :: n
  n = 1
end function pdf_builtin_data_get_n_par
```

Return the outgoing particle PDG codes. This is based on the mask.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩+≡

```
procedure :: get_pdg_out => pdf_builtin_data_get_pdg_out
```

⟨*SF pdf builtin: procedures*⟩+≡

```
subroutine pdf_builtin_data_get_pdg_out (data, pdg_out)
  class(pdf_builtin_data_t), intent(in) :: data
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
  integer, dimension(:), allocatable :: pdg1
  integer :: n, np, i
  n = count (data%mask)
  np = 0;  if (data%has_photon .and. data%mask_photon)  np = 1
  allocate (pdg1 (n + np))
  pdg1(1:n) = pack ([(i, i = -6, 6)], data%mask)
  if (np == 1)  pdg1(n+np) = PHOTON
  pdg_out(1) = pdg1
end subroutine pdf_builtin_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩+≡

```
procedure :: allocate_sf_int => pdf_builtin_data_allocate_sf_int
```

⟨*SF pdf builtin: procedures*⟩+≡

```
subroutine pdf_builtin_data_allocate_sf_int (data, sf_int)
  class(pdf_builtin_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  allocate (pdf_builtin_t :: sf_int)
end subroutine pdf_builtin_data_allocate_sf_int
```

Return the numerical PDF set index.

⟨*SF pdf builtin: pdf builtin data: TBP*⟩+≡

```
procedure :: get_pdf_set => pdf_builtin_data_get_pdf_set
```

⟨*SF pdf builtin: procedures*⟩+≡

```
function pdf_builtin_data_get_pdf_set (data) result (pdf_set)
  class(pdf_builtin_data_t), intent(in) :: data
  integer :: pdf_set
  pdf_set = data%id
end function pdf_builtin_data_get_pdf_set
```

15.15.4 The PDF object

The PDF $1 \to 2$ interaction which describes the splitting of an (anti)proton into a parton and a beam remnant. We stay in the strict forward-splitting limit, but allow some invariant mass for the beam remnant such that the outgoing parton is exactly massless. For a real event, we would replace this by a parton cascade, where the outgoing partons have virtuality as dictated by parton-shower kinematics, and transverse momentum is generated.

The PDF application is a $1 \to 2$ splitting process, where the particles are ordered as (hadron, remnant, parton).

Polarization is ignored completely. The beam particle is colorless, while partons and beam remnant carry color. The remnant gets a special flavor code.

⟨*SF pdf builtin: public*⟩+≡

```
public :: pdf_builtin_t
```

⟨*SF pdf builtin: types*⟩+≡

```
type, extends (sf_int_t) :: pdf_builtin_t
   type(pdf_builtin_data_t), pointer :: data => null ()
   real(default) :: x = 0
   real(default) :: q = 0
 contains
 ⟨SF pdf builtin: pdf builtin: TBP⟩
end type pdf_builtin_t
```

Type string: display the chosen PDF set.

⟨*SF pdf builtin: pdf builtin: TBP*⟩≡

```
procedure :: type_string => pdf_builtin_type_string
```

⟨*SF pdf builtin: procedures*⟩+≡

```
function pdf_builtin_type_string (object) result (string)
  class(pdf_builtin_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "PDF builtin: " // object%data%name
```

```
    else
       string = "PDF builtin: [undefined]"
    end if
  end function pdf_builtin_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF pdf builtin: pdf builtin: TBP*⟩+≡

```
  procedure :: write => pdf_builtin_write
```

⟨*SF pdf builtin: procedures*⟩+≡

```
  subroutine pdf_builtin_write (object, unit, testflag)
    class(pdf_builtin_t), intent(in) :: object
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: testflag
    integer :: u
    u = given_output_unit (unit)
    if (associated (object%data)) then
       call object%data%write (u)
       if (object%status >= SF_DONE_KINEMATICS) then
          write (u, "(1x,A)")  "SF parameters:"
          write (u, "(3x,A," // FMT_17 // ")")  "x =", object%x
          if (object%status >= SF_FAILED_EVALUATION) then
             write (u, "(3x,A," // FMT_17 // ")")  "Q =", object%q
          end if
       end if
       call object%base_write (u, testflag)
    else
       write (u, "(1x,A)")  "PDF builtin data: [undefined]"
    end if
  end subroutine pdf_builtin_write
```

Initialize. We know that `data` will be of concrete type `sf_test_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass, but keep the radiated mass zero.

Optionally, we can provide minimum and maximum values for the momentum transfer.

⟨*SF pdf builtin: pdf builtin: TBP*⟩+≡

```
  procedure :: init => pdf_builtin_init
```

⟨*SF pdf builtin: procedures*⟩+≡

```
  subroutine pdf_builtin_init (sf_int, data)
    class(pdf_builtin_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    type(quantum_numbers_mask_t), dimension(3) :: mask
    type(flavor_t) :: flv, flv_remnant
    type(color_t) :: col0
    type(quantum_numbers_t), dimension(3) :: qn
    integer :: i
    select type (data)
    type is (pdf_builtin_data_t)
       mask = quantum_numbers_mask (.false., .false., .true.)
       call col0%init ()
```

```
      call sf_int%base_init (mask, [0._default], [0._default], [0._default])
      sf_int%data => data
      do i = -6, 6
         if (data%mask(i)) then
            call qn(1)%init (data%flv_in, col = col0)
            if (i == 0) then
               call flv%init (GLUON, data%model)
               call flv_remnant%init (HADRON_REMNANT_OCTET, data%model)
            else
               call flv%init (i, data%model)
               call flv_remnant%init &
                    (sign (HADRON_REMNANT_TRIPLET, -i), data%model)
            end if
            call qn(2)%init ( &
                 flv = flv_remnant, col = color_from_flavor (flv_remnant, 1))
            call qn(2)%tag_radiated ()
            call qn(3)%init ( &
                 flv = flv, col = color_from_flavor (flv, 1, reverse=.true.))
            call sf_int%add_state (qn)
         end if
      end do
      if (data%has_photon .and. data%mask_photon) then
         call flv%init (PHOTON, data%model)
         call flv_remnant%init (HADRON_REMNANT_SINGLET, data%model)
         call qn(2)%init (flv = flv_remnant, &
              col = color_from_flavor (flv_remnant, 1))
         call qn(2)%tag_radiated ()
         call qn(3)%init (flv = flv, &
              col = color_from_flavor (flv, 1, reverse = .true.))
         call sf_int%add_state (qn)
      end if
      call sf_int%freeze ()
      call sf_int%set_incoming ([1])
      call sf_int%set_radiated ([2])
      call sf_int%set_outgoing ([3])
      sf_int%status = SF_INITIAL
    end select
  end subroutine pdf_builtin_init
```

15.15.5 Kinematics

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set $x = r^2$ and consequently $f(r) = 2r$.

⟨*SF pdf builtin: pdf builtin: TBP*⟩+≡

```
  procedure :: complete_kinematics => pdf_builtin_complete_kinematics
```

⟨*SF pdf builtin: procedures*⟩+≡

```
  subroutine pdf_builtin_complete_kinematics (sf_int, x, f, r, rb, map)
    class(pdf_builtin_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(out) :: x
    real(default), intent(out) :: f
```

```
    real(default), dimension(:), intent(in) :: r
    real(default), dimension(:), intent(in) :: rb
    logical, intent(in) :: map
    real(default) :: xb1
    if (map) then
       call msg_fatal ("PDF builtin: map flag not supported")
    else
       x(1) = r(1)
       f = 1
    end if
    xb1 = 1 - x(1)
    call sf_int%split_momentum (x, xb1)
    select case (sf_int%status)
    case (SF_DONE_KINEMATICS)
       sf_int%x = x(1)
    case (SF_FAILED_KINEMATICS)
       sf_int%x = 0
       f = 0
    end select
  end subroutine pdf_builtin_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF pdf builtin: pdf builtin: TBP*⟩+≡

```
  procedure :: inverse_kinematics => pdf_builtin_inverse_kinematics
```

⟨*SF pdf builtin: procedures*⟩+≡

```
  subroutine pdf_builtin_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
    class(pdf_builtin_t), intent(inout) :: sf_int
    real(default), dimension(:), intent(in) :: x
    real(default), intent(out) :: f
    real(default), dimension(:), intent(out) :: r
    real(default), dimension(:), intent(out) :: rb
    logical, intent(in) :: map
    logical, intent(in), optional :: set_momenta
    real(default) :: xb1
    logical :: set_mom
    set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
    if (map) then
       call msg_fatal ("PDF builtin: map flag not supported")
    else
       r(1) = x(1)
       f = 1
    end if
    xb1 = 1 - x(1)
    rb = 1 - r
    if (set_mom) then
       call sf_int%split_momentum (x, xb1)
       select case (sf_int%status)
       case (SF_DONE_KINEMATICS)
          sf_int%x = x(1)
       case (SF_FAILED_KINEMATICS)
          sf_int%x = 0
```

```
        f = 0
      end select
    end if
  end subroutine pdf_builtin_inverse_kinematics
```

15.15.6 Structure function

Once the scale is also known, we can actually call the PDF and set the values. Contrary to LHAPDF, the wrapper already takes care of adjusting to the x and Q bounds. Account for the Jacobian.

⟨*SF pdf builtin: pdf builtin: TBP*⟩+≡

```
  procedure :: apply => pdf_builtin_apply
```

⟨*SF pdf builtin: procedures*⟩+≡

```
  subroutine pdf_builtin_apply (sf_int, scale)
    class(pdf_builtin_t), intent(inout) :: sf_int
    real(default), intent(in) :: scale
    real(default), dimension(-6:6) :: ff
    real(double), dimension(-6:6) :: ff_dbl
    real(default) :: x, fph
    real(double) :: xx, qq
    complex(default), dimension(:), allocatable :: fc
    associate (data => sf_int%data)
      sf_int%q = scale
      x = sf_int%x
      xx = x
      qq = scale
      if (data%invert) then
         if (data%has_photon) then
            call pdf_evolve (data%id, x, scale, ff(6:-6:-1), fph)
         else
            if (data%hoppet_b_matching) then
               call hoppet_eval (xx, qq, ff_dbl(6:-6:-1))
               ff = ff_dbl
            else
               call pdf_evolve (data%id, x, scale, ff(6:-6:-1))
            end if
         end if
      else
         if (data%has_photon) then
            call pdf_evolve (data%id, x, scale, ff, fph)
         else
            if (data%hoppet_b_matching) then
               call hoppet_eval (xx, qq, ff_dbl)
               ff = ff_dbl
            else
               call pdf_evolve (data%id, x, scale, ff)
            end if
         end if
      end if
      if (data%has_photon) then
         allocate (fc (count ([data%mask, data%mask_photon])))
         fc = max (pack ([ff, fph], &
```

```
            [data%mask, data%mask_photon]), 0._default)
      else
         allocate (fc (count (data%mask)))
         fc = max (pack (ff, data%mask), 0._default)
      end if
    end associate
    call sf_int%set_matrix_element (fc)
    sf_int%status = SF_EVALUATED
  end subroutine pdf_builtin_apply
```

15.15.7 Strong Coupling

Since the PDF codes provide a function for computing the running α_s value, we make this available as an implementation of the abstract `alpha_qcd_t` type, which is used for matrix element evaluation.

⟨*SF pdf builtin: public*⟩+≡

```
  public :: alpha_qcd_pdf_builtin_t
```

⟨*SF pdf builtin: types*⟩+≡

```
  type, extends (alpha_qcd_t) :: alpha_qcd_pdf_builtin_t
     type(string_t) :: pdfset_name
     integer :: pdfset_id = -1
   contains
   ⟨SF pdf builtin: alpha qcd: TBP⟩
  end type alpha_qcd_pdf_builtin_t
```

Output.

⟨*SF pdf builtin: alpha qcd: TBP*⟩≡

```
  procedure :: write => alpha_qcd_pdf_builtin_write
```

⟨*SF pdf builtin: procedures*⟩+≡

```
  subroutine alpha_qcd_pdf_builtin_write (object, unit)
    class(alpha_qcd_pdf_builtin_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(3x,A)")  "QCD parameters (pdf_builtin):"
    write (u, "(5x,A,A)")  "PDF set = ", char (object%pdfset_name)
    write (u, "(5x,A,I0)") "PDF ID  = ", object%pdfset_id
  end subroutine alpha_qcd_pdf_builtin_write
```

Calculation: the numeric ID selects the correct PDF set, which must be properly initialized.

⟨*SF pdf builtin: alpha qcd: TBP*⟩+≡

```
  procedure :: get => alpha_qcd_pdf_builtin_get
```

⟨*SF pdf builtin: procedures*⟩+≡

```
  function alpha_qcd_pdf_builtin_get (alpha_qcd, scale) result (alpha)
    class(alpha_qcd_pdf_builtin_t), intent(in) :: alpha_qcd
    real(default), intent(in) :: scale
    real(default) :: alpha
    alpha = pdf_alphas (alpha_qcd%pdfset_id, scale)
```

```
    end function alpha_qcd_pdf_builtin_get
```

Initialization. We need to access the global initialization status.

⟨*SF pdf builtin: alpha qcd: TBP*⟩+≡

```
    procedure :: init => alpha_qcd_pdf_builtin_init
```

⟨*SF pdf builtin: procedures*⟩+≡

```
    subroutine alpha_qcd_pdf_builtin_init (alpha_qcd, name, path)
      class(alpha_qcd_pdf_builtin_t), intent(out) :: alpha_qcd
      type(string_t), intent(in) :: name
      type(string_t), intent(in) :: path
      alpha_qcd%pdfset_name = name
      alpha_qcd%pdfset_id = pdf_get_id (name)
      if (alpha_qcd%pdfset_id < 0) &
           call msg_fatal ("QCD parameter initialization: PDF set " &
           // char (name) // " is unknown")
      call pdf_init (alpha_qcd%pdfset_id, path)
    end subroutine alpha_qcd_pdf_builtin_init
```

15.15.8 Unit tests

Test module, followed by the corresponding implementation module.

⟨sf_pdf_builtin_ut.f90⟩≡

```
  ⟨File header⟩

  module sf_pdf_builtin_ut
    use unit_tests
    use sf_pdf_builtin_uti

  ⟨Standard module head⟩

  ⟨SF pdf builtin: public test⟩

  contains

  ⟨SF pdf builtin: test driver⟩

  end module sf_pdf_builtin_ut
```

⟨sf_pdf_builtin_uti.f90⟩≡

```
  ⟨File header⟩

  module sf_pdf_builtin_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use os_interface
    use physics_defs, only: PROTON
    use sm_qcd
    use lorentz
    use pdg_arrays
    use flavors
    use interactions, only: reset_interaction_counter
```

```
  use model_data
  use sf_base

  use sf_pdf_builtin
```

⟨*Standard module head*⟩

⟨*SF pdf builtin: test declarations*⟩

```
contains
```

⟨*SF pdf builtin: tests*⟩

```
end module sf_pdf_builtin_uti
```

API: driver for the unit tests below.

⟨*SF pdf builtin: public test*⟩≡

```
  public :: sf_pdf_builtin_test
```

⟨*SF pdf builtin: test driver*⟩≡

```
  subroutine sf_pdf_builtin_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```
⟨*SF pdf builtin: execute tests*⟩
```
  end subroutine sf_pdf_builtin_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF pdf builtin: execute tests*⟩≡

```
  call test (sf_pdf_builtin_1, "sf_pdf_builtin_1", &
       "structure function configuration", &
       u, results)
```

⟨*SF pdf builtin: test declarations*⟩≡

```
  public :: sf_pdf_builtin_1
```

⟨*SF pdf builtin: tests*⟩≡

```
  subroutine sf_pdf_builtin_1 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_data_t), target :: model
    type(pdg_array_t) :: pdg_in
    type(pdg_array_t), dimension(1) :: pdg_out
    integer, dimension(:), allocatable :: pdg1
    class(sf_data_t), allocatable :: data
    type(string_t) :: name

    write (u, "(A)")  "* Test output: sf_pdf_builtin_1"
    write (u, "(A)")  "*   Purpose: initialize and display &
         &test structure function data"
    write (u, "(A)")

    write (u, "(A)")  "* Create empty data object"
    write (u, "(A)")
```

```
    call os_data_init (os_data)

    call model%init_sm_test ()
    pdg_in = PROTON

    allocate (pdf_builtin_data_t :: data)
    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize"
    write (u, "(A)")

    name = "CTEQ6L"

    select type (data)
    type is (pdf_builtin_data_t)
       call data%init (model, pdg_in, name, &
            os_data%pdf_builtin_datapath)
    end select

    call data%write (u)

    write (u, "(A)")

    write (u, "(1x,A)")  "Outgoing particle codes:"
    call data%get_pdg_out (pdg_out)
    pdg1 = pdg_out(1)
    write (u, "(2x,99(1x,I0))")  pdg1

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_pdf_builtin_1"

  end subroutine sf_pdf_builtin_1
```

Test and probe structure function

Construct and display a structure function object based on the PDF builtin structure function.

⟨*SF pdf builtin: execute tests*⟩+≡

```
  call test (sf_pdf_builtin_2, "sf_pdf_builtin_2", &
       "structure function instance", &
       u, results)
```

⟨*SF pdf builtin: test declarations*⟩+≡

```
  public :: sf_pdf_builtin_2
```

⟨*SF pdf builtin: tests*⟩+≡

```
  subroutine sf_pdf_builtin_2 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
```

```
type(model_data_t), target :: model
type(flavor_t) :: flv
type(pdg_array_t) :: pdg_in
class(sf_data_t), allocatable, target :: data
class(sf_int_t), allocatable :: sf_int
type(string_t) :: name
type(vector4_t) :: k
type(vector4_t), dimension(2) :: q
real(default) :: E
real(default), dimension(:), allocatable :: r, rb, x
real(default) :: f

write (u, "(A)")  "* Test output: sf_pdf_builtin_2"
write (u, "(A)")  "*   Purpose: initialize and fill &
     &test structure function object"
write (u, "(A)")

write (u, "(A)")  "* Initialize configuration data"
write (u, "(A)")

call os_data_init (os_data)
call model%init_sm_test ()
call flv%init (PROTON, model)
pdg_in = PROTON

call reset_interaction_counter ()

name = "CTEQ6L"

allocate (pdf_builtin_data_t :: data)
select type (data)
type is (pdf_builtin_data_t)
   call data%init (model, pdg_in, name, &
        os_data%pdf_builtin_datapath)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
```

```
  write (u, "(A)")  "* Set kinematics for x=0.5"
  write (u, "(A)")

  allocate (r (data%get_n_par ()))
  allocate (rb(size (r)))
  allocate (x (size (r)))

  r = 0.5_default
  rb = 1 - r
  call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
  call sf_int%write (u)

  write (u, "(A)")
  write (u, "(A,9(1x,F10.7))")  "x =", x
  write (u, "(A,9(1x,F10.7))")  "f =", f

  write (u, "(A)")
  write (u, "(A)")  "* Recover x from momenta"
  write (u, "(A)")

  q = sf_int%get_momenta (outgoing=.true.)
  call sf_int%final ()
  deallocate (sf_int)

  call data%allocate_sf_int (sf_int)
  call sf_int%init (data)
  call sf_int%set_beam_index ([1])

  call sf_int%seed_kinematics ([k])
  call sf_int%set_momenta (q, outgoing=.true.)
  call sf_int%recover_x (x)

  write (u, "(A,9(1x,F10.7))")  "x =", x

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate for Q = 100 GeV"
  write (u, "(A)")

  call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
  call sf_int%apply (scale = 100._default)
  call sf_int%write (u)


  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call sf_int%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_pdf_builtin_2"

end subroutine sf_pdf_builtin_2
```

Strong Coupling

Test α_s as an implementation of the `alpha_qcd_t` abstract type.

⟨*SF pdf builtin: execute tests*⟩+≡

```
call test (sf_pdf_builtin_3, "sf_pdf_builtin_3", &
     "running alpha_s", &
     u, results)
```

⟨*SF pdf builtin: test declarations*⟩+≡

```
public :: sf_pdf_builtin_3
```

⟨*SF pdf builtin: tests*⟩+≡

```
subroutine sf_pdf_builtin_3 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(qcd_t) :: qcd
  type(string_t) :: name

  write (u, "(A)")  "* Test output: sf_pdf_builtin_3"
  write (u, "(A)")  "*   Purpose: initialize and evaluate alpha_s"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call os_data_init (os_data)

  name = "CTEQ6L"

  write (u, "(A)")  "* Initialize qcd object"
  write (u, "(A)")

  allocate (alpha_qcd_pdf_builtin_t :: qcd%alpha)
  select type (alpha => qcd%alpha)
  type is (alpha_qcd_pdf_builtin_t)
     call alpha%init (name, os_data%pdf_builtin_datapath)
  end select
  call qcd%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate for Q = 100"
  write (u, "(A)")

  write (u, "(1x,A,F8.5)")  "alpha = ", qcd%alpha%get (100._default)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_pdf_builtin_3"

end subroutine sf_pdf_builtin_3
```

15.16 LHAPDF

Parton distribution functions (PDFs) are available via an interface to the LHAPDF standard library.

15.16.1 The module

⟨sf_lhapdf.f90⟩≡

```
⟨File header⟩

module sf_lhapdf

⟨Use kinds⟩
⟨Use strings⟩
  use format_defs, only: FMT_17, FMT_19
  use io_units
  use system_dependencies, only: LHAPDF_PDFSETS_PATH
  use system_dependencies, only: LHAPDF5_AVAILABLE
  use system_dependencies, only: LHAPDF6_AVAILABLE
  use diagnostics
  use physics_defs, only: PROTON, PHOTON, PIPLUS, GLUON
  use physics_defs, only: HADRON_REMNANT_SINGLET
  use physics_defs, only: HADRON_REMNANT_TRIPLET
  use physics_defs, only: HADRON_REMNANT_OCTET
  use lorentz
  use sm_qcd
  use pdg_arrays
  use model_data
  use flavors
  use colors
  use quantum_numbers
  use state_matrices
  use polarizations
  use sf_base
  use lhapdf !NODEP!
  use hoppet_interface

⟨Standard module head⟩

⟨SF lhapdf: public⟩

⟨SF lhapdf: types⟩

⟨SF lhapdf: parameters⟩

⟨SF lhapdf: variables⟩

⟨SF lhapdf: interfaces⟩

contains

⟨SF lhapdf: procedures⟩

end module sf_lhapdf
```

15.16.2 Codes for default PDF sets

The default PDF for protons set is chosen to be CTEQ6ll (LO fit with LO α_s).

⟨*SF lhapdf: parameters*⟩≡
```
character(*), parameter :: LHAPDF5_DEFAULT_PROTON = "cteq6ll.LHpdf"
character(*), parameter :: LHAPDF5_DEFAULT_PION   = "ABFKWPI.LHgrid"
character(*), parameter :: LHAPDF5_DEFAULT_PHOTON = "GSG960.LHgrid"
character(*), parameter :: LHAPDF6_DEFAULT_PROTON = "CT10"
```

15.16.3 LHAPDF library interface

Here we specify explicit interfaces for all LHAPDF routines that we use below.

⟨*SF lhapdf: interfaces*⟩≡
```
interface
   subroutine InitPDFsetM (set, file)
     integer, intent(in) :: set
     character(*), intent(in) :: file
   end subroutine InitPDFsetM
end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
interface
   subroutine InitPDFM (set, mem)
     integer, intent(in) :: set, mem
   end subroutine InitPDFM
end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
interface
   subroutine numberPDFM (set, n_members)
     integer, intent(in) :: set
     integer, intent(out) :: n_members
   end subroutine numberPDFM
end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
interface
   subroutine evolvePDFM (set, x, q, ff)
     integer, intent(in) :: set
     double precision, intent(in) :: x, q
     double precision, dimension(-6:6), intent(out) :: ff
   end subroutine evolvePDFM
end interface
```

⟨*SF lhapdf: interfaces*⟩+≡
```
interface
   subroutine evolvePDFphotonM (set, x, q, ff, fphot)
     integer, intent(in) :: set
     double precision, intent(in) :: x, q
     double precision, dimension(-6:6), intent(out) :: ff
     double precision, intent(out) :: fphot
   end subroutine evolvePDFphotonM
```

```
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡

```
    interface
       subroutine evolvePDFpM (set, x, q, s, scheme, ff)
         integer, intent(in) :: set
         double precision, intent(in) :: x, q, s
         integer, intent(in) :: scheme
         double precision, dimension(-6:6), intent(out) :: ff
       end subroutine evolvePDFpM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡

```
    interface
       subroutine GetXminM (set, mem, xmin)
         integer, intent(in) :: set, mem
         double precision, intent(out) :: xmin
       end subroutine GetXminM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡

```
    interface
       subroutine GetXmaxM (set, mem, xmax)
         integer, intent(in) :: set, mem
         double precision, intent(out) :: xmax
       end subroutine GetXmaxM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡

```
    interface
       subroutine GetQ2minM (set, mem, q2min)
         integer, intent(in) :: set, mem
         double precision, intent(out) :: q2min
       end subroutine GetQ2minM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡

```
    interface
       subroutine GetQ2maxM (set, mem, q2max)
         integer, intent(in) :: set, mem
         double precision, intent(out) :: q2max
       end subroutine GetQ2maxM
    end interface
```

⟨*SF lhapdf: interfaces*⟩+≡

```
    interface
       function has_photon () result(flag)
          logical :: flag
       end function has_photon
    end interface
```

15.16.4 The LHAPDF status

This type holds the initialization status of the LHAPDF system. Entry 1 is for proton PDFs, entry 2 for pion PDFs, entry 3 for photon PDFs.

Since it is connected to the external LHAPDF library, this is a truly global object. We implement it as a a private module variable. To access it from elsewhere, the caller has to create and initialize an object of type `lhapdf_status_t`, which acts as a proxy.

⟨*SF lhapdf: types*⟩≡
```
type :: lhapdf_global_status_t
   private
   logical, dimension(3) :: initialized = .false.
end type lhapdf_global_status_t
```

⟨*SF lhapdf: variables*⟩≡
```
type(lhapdf_global_status_t), save :: lhapdf_global_status
```

⟨*SF lhapdf: procedures*⟩≡
```
function lhapdf_global_status_is_initialized (set) result (flag)
  logical :: flag
  integer, intent(in), optional :: set
  if (present (set)) then
     select case (set)
     case (1:3);    flag = lhapdf_global_status%initialized(set)
     case default;  flag = .false.
     end select
  else
     flag = any (lhapdf_global_status%initialized)
  end if
end function lhapdf_global_status_is_initialized
```

⟨*SF lhapdf: procedures*⟩+≡
```
subroutine lhapdf_global_status_set_initialized (set)
  integer, intent(in) :: set
  lhapdf_global_status%initialized(set) = .true.
end subroutine lhapdf_global_status_set_initialized
```

This is the only public procedure, it tells the system to forget about previous initialization, allowing for changing the chosen PDF set. Note that such a feature works only if the global program flow is serial, so no two distinct sets are accessed simultaneously. But this applies to LHAPDF anyway.

⟨*SF lhapdf: public*⟩≡
```
public :: lhapdf_global_reset
```

⟨*SF lhapdf: procedures*⟩+≡
```
subroutine lhapdf_global_reset ()
  lhapdf_global_status%initialized = .false.
end subroutine lhapdf_global_reset
```

15.16.5 LHAPDF initialization

Before using LHAPDF, we have to initialize it with a particular data set and member. This applies not just if we use structure functions, but also if we just use an α_s formula. The integer `set` should be 1 for proton, 2 for pion, and 3 for photon, but this is just convention.

It appears as if LHAPDF does not allow for multiple data sets being used concurrently (?), so multi-threaded usage with different sets (e.g., a scan) is excluded. The current setup with a global flag that indicates initialization is fine as long as Whizard itself is run in serial mode at the Sindarin level. If we introduce multithreading in any form from Sindarin, we have to rethink the implementation of the LHAPDF interface. (The same considerations apply to builtin PDFs.)

If the particular set has already been initialized, do nothing. This implies that whenever we want to change the setup for a particular set, we have to reset the LHAPDF status. `lhapdf_initialize` has an obvious name clash with `lhapdf_init`, the reason it works for `pdf_builtin` is that there things are outsourced to a separate module (inc. `lhapdf_status` etc.).

⟨*SF lhapdf: public*⟩+≡
```
public :: lhapdf_initialize
```

⟨*SF lhapdf: procedures*⟩+≡
```
subroutine lhapdf_initialize (set, prefix, file, member, pdf, b_match)
  integer, intent(in) :: set
  type(string_t), intent(inout) :: prefix
  type(string_t), intent(inout) :: file
  type(lhapdf_pdf_t), intent(inout), optional :: pdf
  integer, intent(inout) :: member
  logical, intent(in), optional :: b_match
  if (prefix == "")  prefix = LHAPDF_PDFSETS_PATH
  if (LHAPDF5_AVAILABLE) then
     if (lhapdf_global_status_is_initialized (set))  return
     if (file == "") then
        select case (set)
        case (1);  file = LHAPDF5_DEFAULT_PROTON
        case (2);  file = LHAPDF5_DEFAULT_PION
        case (3);  file = LHAPDF5_DEFAULT_PHOTON
        end select
     end if
     if (data_file_exists (prefix // "/" // file)) then
        call InitPDFsetM (set, char (prefix // "/" // file))
     else
        call msg_fatal ("LHAPDF: Data file '" &
             // char (file) // "' not found in '" // char (prefix) // "'.")
        return
     end if
     if (.not. dataset_member_exists (set, member)) then
        call msg_error (" LHAPDF: Chosen member does not exist for set '" &
             // char (file) // "', using default.")
        member = 0
     end if
     call InitPDFM (set, member)
  else if (LHAPDF6_AVAILABLE) then
```

```
  ! TODO: (bcn 2015-07-07) we should have a closer look why this global
  !                        check must not be executed
  !   if (lhapdf_global_status_is_initialized (set) .and. &
  !        pdf%is_associated ()) return
     if (file == "") then
        select case (set)
        case (1);  file = LHAPDF6_DEFAULT_PROTON
        case (2);
           call msg_fatal ("LHAPDF6: no pion PDFs supported")
        case (3);
           call msg_fatal ("LHAPDF6: no photon PDFs supported")
        end select
     end if
     if (data_file_exists (prefix // "/" // file // "/" // file // ".info")) then
        call pdf%init (char (file), member)
     else
        call msg_fatal ("LHAPDF: Data file '" &
             // char (file) // "' not found in '" // char (prefix) // "'.")
        return
     end if
  end if
  if (present (b_match)) then
     if (b_match) then
        if (LHAPDF5_AVAILABLE) then
           call hoppet_init (.false.)
        else if (LHAPDF6_AVAILABLE) then
           call hoppet_init (.false., pdf)
        end if
     end if
  end if
  call lhapdf_global_status_set_initialized (set)
contains
  function data_file_exists (fq_name) result (exist)
    type(string_t), intent(in) :: fq_name
    logical :: exist
    inquire (file = char(fq_name), exist = exist)
  end function data_file_exists
  function dataset_member_exists (set, member) result (exist)
    integer, intent(in) :: set, member
    logical :: exist
    integer :: n_members
    call numberPDFM (set, n_members)
    exist = member >= 0 .and. member <= n_members
  end function dataset_member_exists
end subroutine lhapdf_initialize
```

15.16.6 Kinematics

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set $x = r^2$ and consequently $f(r) = 2r$.

⟨*SF lhapdf: lhapdf: TBP*⟩≡

```
procedure :: complete_kinematics => lhapdf_complete_kinematics
```

⟨*SF lhapdf: procedures*⟩+≡

```
subroutine lhapdf_complete_kinematics (sf_int, x, f, r, rb, map)
  class(lhapdf_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  real(default), dimension(:), intent(in) :: rb
  logical, intent(in) :: map
  real(default) :: xb1
  if (map) then
     call msg_fatal ("LHAPDF: map flag not supported")
  else
     x(1) = r(1)
     f = 1
  end if
  xb1 = 1 - x(1)
  call sf_int%split_momentum (x, xb1)
  select case (sf_int%status)
  case (SF_DONE_KINEMATICS)
     sf_int%x = x(1)
  case (SF_FAILED_KINEMATICS)
     sf_int%x = 0
     f = 0
  end select
end subroutine lhapdf_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF lhapdf: lhapdf: TBP*⟩+≡

```
procedure :: inverse_kinematics => lhapdf_inverse_kinematics
```

⟨*SF lhapdf: procedures*⟩+≡

```
subroutine lhapdf_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
  class(lhapdf_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  real(default), dimension(:), intent(out) :: rb
  logical, intent(in) :: map
  logical, intent(in), optional :: set_momenta
  real(default) :: xb1
  logical :: set_mom
  set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
  if (map) then
     call msg_fatal ("LHAPDF: map flag not supported")
  else
     r(1) = x(1)
     f = 1
  end if
  xb1 = 1 - x(1)
```

```
    rb = 1 - r
    if (set_mom) then
       call sf_int%split_momentum (x, xb1)
       select case (sf_int%status)
       case (SF_DONE_KINEMATICS)
          sf_int%x = x(1)
       case (SF_FAILED_KINEMATICS)
          sf_int%x = 0
          f = 0
       end select
    end if
  end subroutine lhapdf_inverse_kinematics
```

15.16.7 The LHAPDF data block

The data block holds the incoming flavor (which has to be proton, pion, or photon), the corresponding pointer to the global access data (1, 2, or 3), the flag `invert` which is set for an antiproton, the bounds as returned by LHAPDF for the specified set, and a mask that determines which partons will be actually in use.

⟨*SF lhapdf: public*⟩+≡

```
  public :: lhapdf_data_t
```

⟨*SF lhapdf: types*⟩+≡

```
  type, extends (sf_data_t) :: lhapdf_data_t
     private
     type(string_t) :: prefix
     type(string_t) :: file
     type(lhapdf_pdf_t) :: pdf
     integer :: member = 0
     class(model_data_t), pointer :: model => null ()
     type(flavor_t) :: flv_in
     integer :: set = 0
     logical :: invert = .false.
     logical :: photon = .false.
     logical :: has_photon = .false.
     integer :: photon_scheme = 0
     real(default) :: xmin = 0, xmax = 0
     real(default) :: qmin = 0, qmax = 0
     logical, dimension(-6:6) :: mask = .true.
     logical :: mask_photon = .true.
     logical :: hoppet_b_matching = .false.
   contains
     ⟨SF lhapdf: lhapdf data: TBP⟩
  end type lhapdf_data_t
```

Generate PDF data. This is provided as a function, but it has the side-effect of initializing the requested PDF set. A finalizer is not needed.

The library uses double precision, so since the default precision may be extended or quadruple, we use auxiliary variables for type casting.

⟨*SF lhapdf: lhapdf data: TBP*⟩≡

```
  procedure :: init => lhapdf_data_init
```

⟨*SF lhapdf: procedures*⟩+≡

```
subroutine lhapdf_data_init &
     (data, model, pdg_in, prefix, file, member, photon_scheme, &
          hoppet_b_matching)
  class(lhapdf_data_t), intent(out) :: data
  class(model_data_t), intent(in), target :: model
  type(pdg_array_t), intent(in) :: pdg_in
  type(string_t), intent(in), optional :: prefix, file
  integer, intent(in), optional :: member
  integer, intent(in), optional :: photon_scheme
  logical, intent(in), optional :: hoppet_b_matching
  double precision :: xmin, xmax, q2min, q2max
  external :: InitPDFsetM, InitPDFM, numberPDFM
  external :: GetXminM, GetXmaxM, GetQ2minM, GetQ2maxM
  if (.not. LHAPDF5_AVAILABLE .and. .not. LHAPDF6_AVAILABLE) then
     call msg_fatal ("LHAPDF requested but library is not linked")
     return
  end if
  data%model => model
  if (pdg_array_get_length (pdg_in) /= 1) &
       call msg_fatal ("PDF: incoming particle must be unique")
  call data%flv_in%init (pdg_array_get (pdg_in, 1), model)
  select case (pdg_array_get (pdg_in, 1))
  case (PROTON)
     data%set = 1
  case (-PROTON)
     data%set = 1
     data%invert = .true.
  case (PIPLUS)
     data%set = 2
  case (-PIPLUS)
     data%set = 2
     data%invert = .true.
  case (PHOTON)
     data%set = 3
     data%photon = .true.
     if (present (photon_scheme))  data%photon_scheme = photon_scheme
  case default
     call msg_fatal (" LHAPDF: " &
          // "incoming particle must be (anti)proton, pion, or photon.")
     return
  end select
  if (present (prefix)) then
     data%prefix = prefix
  else
     data%prefix = ""
  end if
  if (present (file)) then
     data%file = file
  else
     data%file = ""
  end if
  if (present (hoppet_b_matching))  data%hoppet_b_matching = hoppet_b_matching
  if (LHAPDF5_AVAILABLE) then
```

```
      call lhapdf_initialize &
           (data%set, data%prefix, data%file, data%member, &
           b_match = data%hoppet_b_matching)
      call GetXminM (data%set, data%member, xmin)
      call GetXmaxM (data%set, data%member, xmax)
      call GetQ2minM (data%set, data%member, q2min)
      call GetQ2maxM (data%set, data%member, q2max)
      data%xmin = xmin
      data%xmax = xmax
      data%qmin = sqrt (q2min)
      data%qmax = sqrt (q2max)
      data%has_photon = has_photon ()
    else if (LHAPDF6_AVAILABLE) then
      call lhapdf_initialize &
           (data%set, data%prefix, data%file, data%member, &
           data%pdf, data%hoppet_b_matching)
      data%xmin = data%pdf%getxmin ()
      data%xmax = data%pdf%getxmax ()
      data%qmin = sqrt(data%pdf%getq2min ())
      data%qmax = sqrt(data%pdf%getq2max ())
      data%has_photon = data%pdf%has_photon ()
    end if
  end subroutine lhapdf_data_init
```

Enable/disable partons explicitly. If a mask entry is true, applying the PDF will generate the corresponding flavor on output.

⟨*LHAPDF: lhapdf data: TBP*⟩≡

```
  procedure :: set_mask => lhapdf_data_set_mask
```

⟨*LHAPDF: procedures*⟩≡

```
  subroutine lhapdf_data_set_mask (data, mask)
    class(lhapdf_data_t), intent(inout) :: data
    logical, dimension(-6:6), intent(in) :: mask
    data%mask = mask
  end subroutine lhapdf_data_set_mask
```

Return the public part of the data set.

⟨*LHAPDF: public*⟩≡

```
  public :: lhapdf_data_get_public_info
```

⟨*LHAPDF: procedures*⟩+≡

```
  subroutine lhapdf_data_get_public_info &
      (data, lhapdf_dir, lhapdf_file, lhapdf_member)
    type(lhapdf_data_t), intent(in) :: data
    type(string_t), intent(out) :: lhapdf_dir, lhapdf_file
    integer, intent(out) :: lhapdf_member
    lhapdf_dir = data%prefix
    lhapdf_file = data%file
    lhapdf_member = data%member
  end subroutine lhapdf_data_get_public_info
```

Return the number of the member of the data set.

⟨*LHAPDF: public*⟩+≡

```
  public :: lhapdf_data_get_set
```

⟨*LHAPDF: procedures*⟩+≡

```
function lhapdf_data_get_set(data) result(set)
  type(lhapdf_data_t), intent(in) :: data
  integer :: set
  set = data%set
end function lhapdf_data_get_set
```

Output

⟨*SF lhapdf: lhapdf data: TBP*⟩+≡

```
procedure :: write => lhapdf_data_write
```

⟨*SF lhapdf: procedures*⟩+≡

```
subroutine lhapdf_data_write (data, unit, verbose)
  class(lhapdf_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  logical :: verb
  integer :: u
  if (present (verbose)) then
     verb = verbose
  else
     verb = .false.
  end if
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)") "LHAPDF data:"
  if (data%set /= 0) then
     write (u, "(3x,A)", advance="no") "flavor       =  "
     call data%flv_in%write (u);  write (u, *)
     if (verb) then
        write (u, "(3x,A,A)")       "  prefix       =  ", char (data%prefix)
     else
        write (u, "(3x,A,A)")       "  prefix       = ", &
             " <empty (non-verbose version)>"
     end if
     write (u, "(3x,A,A)")       "  file         =  ", char (data%file)
     write (u, "(3x,A,I3)")      "  member       = ", data%member
     write (u, "(3x,A," // FMT_19 // ")") "  x(min)       = ", data%xmin
     write (u, "(3x,A," // FMT_19 // ")") "  x(max)       = ", data%xmax
     write (u, "(3x,A," // FMT_19 // ")") "  Q(min)       = ", data%qmin
     write (u, "(3x,A," // FMT_19 // ")") "  Q(max)       = ", data%qmax
     write (u, "(3x,A,L1)")      "  invert       =  ", data%invert
     if (data%photon)  write (u, "(3x,A,I3)") &
          "  IP2 (scheme) = ", data%photon_scheme
        write (u, "(3x,A,6(1x,L1),1x,A,1x,L1,1x,A,6(1x,L1))") &
             "  mask         = ", &
             data%mask(-6:-1), "*", data%mask(0), "*", data%mask(1:6)
        write (u, "(3x,A,L1)")   "  photon mask  =  ", data%mask_photon
     if (data%set == 1)  write (u, "(3x,A,L1)") &
          "  hoppet_b     =  ", data%hoppet_b_matching
  else
     write (u, "(3x,A)") "[undefined]"
  end if
end subroutine lhapdf_data_write
```

The number of parameters is one. We do not generate transverse momentum.

⟨*SF lhapdf: lhapdf data: TBP*⟩+≡
```
procedure :: get_n_par => lhapdf_data_get_n_par
```

⟨*SF lhapdf: procedures*⟩+≡
```
function lhapdf_data_get_n_par (data) result (n)
  class(lhapdf_data_t), intent(in) :: data
  integer :: n
  n = 1
end function lhapdf_data_get_n_par
```

Return the outgoing particle PDG codes. This is based on the mask.

⟨*SF lhapdf: lhapdf data: TBP*⟩+≡
```
procedure :: get_pdg_out => lhapdf_data_get_pdg_out
```

⟨*SF lhapdf: procedures*⟩+≡
```
subroutine lhapdf_data_get_pdg_out (data, pdg_out)
  class(lhapdf_data_t), intent(in) :: data
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
  integer, dimension(:), allocatable :: pdg1
  integer :: n, np, i
  n = count (data%mask)
  np = 0;  if (data%has_photon .and. data%mask_photon)  np = 1
  allocate (pdg1 (n + np))
  pdg1(1:n) = pack ([(i, i = -6, 6)], data%mask)
  if (np == 1)  pdg1(n+np) = PHOTON
  pdg_out(1) = pdg1
end subroutine lhapdf_data_get_pdg_out
```

Allocate the interaction record.

⟨*SF lhapdf: lhapdf data: TBP*⟩+≡
```
procedure :: allocate_sf_int => lhapdf_data_allocate_sf_int
```

⟨*SF lhapdf: procedures*⟩+≡
```
subroutine lhapdf_data_allocate_sf_int (data, sf_int)
  class(lhapdf_data_t), intent(in) :: data
  class(sf_int_t), intent(inout), allocatable :: sf_int
  allocate (lhapdf_t :: sf_int)
end subroutine lhapdf_data_allocate_sf_int
```

Return the numerical PDF set index.

⟨*SF lhapdf: lhapdf data: TBP*⟩+≡
```
procedure :: get_pdf_set => lhapdf_data_get_pdf_set
```

⟨*SF lhapdf: procedures*⟩+≡
```
function lhapdf_data_get_pdf_set (data) result (pdf_set)
  class(lhapdf_data_t), intent(in) :: data
  integer :: pdf_set
  pdf_set = data%set
end function lhapdf_data_get_pdf_set
```

15.16.8 The LHAPDF object

The `lhapdf_t` data type is a $1 \to 2$ interaction which describes the splitting of an (anti)proton into a parton and a beam remnant. We stay in the strict forward-splitting limit, but allow some invariant mass for the beam remnant such that the outgoing parton is exactly massless. For a real event, we would replace this by a parton cascade, where the outgoing partons have virtuality as dictated by parton-shower kinematics, and transverse momentum is generated.

This is the LHAPDF object which holds input data together with the interaction. We also store the x momentum fraction and the scale, since kinematics and function value are requested at different times.

The PDF application is a $1 \to 2$ splitting process, where the particles are ordered as (hadron, remnant, parton).

Polarization is ignored completely. The beam particle is colorless, while partons and beam remnant carry color. The remnant gets a special flavor code.

⟨*SF lhapdf: public*⟩+≡
```
public :: lhapdf_t
```

⟨*SF lhapdf: types*⟩+≡
```
type, extends (sf_int_t) :: lhapdf_t
   type(lhapdf_data_t), pointer :: data => null ()
   real(default) :: x = 0
   real(default) :: q = 0
   real(default) :: s = 0
 contains
 ⟨SF lhapdf: lhapdf: TBP⟩
end type lhapdf_t
```

Type string: display the chosen PDF set.

⟨*SF lhapdf: lhapdf: TBP*⟩+≡
```
procedure :: type_string => lhapdf_type_string
```

⟨*SF lhapdf: procedures*⟩+≡
```
function lhapdf_type_string (object) result (string)
  class(lhapdf_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "LHAPDF: " // object%data%file
  else
     string = "LHAPDF: [undefined]"
  end if
end function lhapdf_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF lhapdf: lhapdf: TBP*⟩+≡
```
procedure :: write => lhapdf_write
```

⟨*SF lhapdf: procedures*⟩+≡
```
subroutine lhapdf_write (object, unit, testflag)
  class(lhapdf_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
```

```
    u = given_output_unit (unit)
    if (associated (object%data)) then
       call object%data%write (u)
       if (object%status >= SF_DONE_KINEMATICS) then
          write (u, "(1x,A)")  "SF parameters:"
          write (u, "(3x,A," // FMT_17 // ")")  "x =", object%x
          if (object%status >= SF_FAILED_EVALUATION) then
             write (u, "(3x,A," // FMT_17 // ")")  "Q =", object%q
          end if
       end if
       call object%base_write (u, testflag)
    else
       write (u, "(1x,A)")  "LHAPDF data: [undefined]"
    end if
  end subroutine lhapdf_write
```

Initialize. We know that `data` will be of concrete type `sf_lhapdf_data_t`, but we have to cast this explicitly.

For this implementation, we set the incoming and outgoing masses equal to the physical particle mass, but keep the radiated mass zero.

⟨*SF lhapdf: lhapdf: TBP*⟩+≡

```
  procedure :: init => lhapdf_init
```

⟨*SF lhapdf: procedures*⟩+≡

```
  subroutine lhapdf_init (sf_int, data)
    class(lhapdf_t), intent(out) :: sf_int
    class(sf_data_t), intent(in), target :: data
    type(quantum_numbers_mask_t), dimension(3) :: mask
    type(flavor_t) :: flv, flv_remnant
    type(color_t) :: col0
    type(quantum_numbers_t), dimension(3) :: qn
    integer :: i
    select type (data)
    type is (lhapdf_data_t)
       mask = quantum_numbers_mask (.false., .false., .true.)
       call col0%init ()
       call sf_int%base_init (mask, [0._default], [0._default], [0._default])
       sf_int%data => data
       do i = -6, 6
          if (data%mask(i)) then
             call qn(1)%init (data%flv_in, col = col0)
             if (i == 0) then
                call flv%init (GLUON, data%model)
                call flv_remnant%init (HADRON_REMNANT_OCTET, data%model)
             else
                call flv%init (i, data%model)
                call flv_remnant%init &
                     (sign (HADRON_REMNANT_TRIPLET, -i), data%model)
             end if
             call qn(2)%init ( &
                  flv = flv_remnant, col = color_from_flavor (flv_remnant, 1))
             call qn(2)%tag_radiated ()
             call qn(3)%init ( &
                  flv = flv, col = color_from_flavor (flv, 1, reverse=.true.))
```

```
            call sf_int%add_state (qn)
         end if
      end do
      if (data%has_photon .and. data%mask_photon) then
         call flv%init (PHOTON, data%model)
         call flv_remnant%init (HADRON_REMNANT_SINGLET, data%model)
         call qn(2)%init (flv = flv_remnant, &
              col = color_from_flavor (flv_remnant, 1))
         call qn(2)%tag_radiated ()
         call qn(3)%init (flv = flv, &
              col = color_from_flavor (flv, 1, reverse=.true.))
         call sf_int%add_state (qn)
      end if
      call sf_int%freeze ()
      call sf_int%set_incoming ([1])
      call sf_int%set_radiated ([2])
      call sf_int%set_outgoing ([3])
      sf_int%status = SF_INITIAL
   end select
end subroutine lhapdf_init
```

15.16.9 Structure function

We have to cast the LHAPDF arguments to/from double precision (possibly from/to extended/quadruple precision), if necessary. Furthermore, some structure functions can yield negative results (sea quarks close to $x = 1$). We set these unphysical values to zero.

⟨*SF lhapdf: lhapdf: TBP*⟩+≡

```
procedure :: apply => lhapdf_apply
```

⟨*SF lhapdf: procedures*⟩+≡

```
subroutine lhapdf_apply (sf_int, scale)
  class(lhapdf_t), intent(inout) :: sf_int
  real(default), intent(in) :: scale
  real(default) :: x, s
  double precision :: xx, qq, ss
  double precision, dimension(-6:6) :: ff
  double precision :: fphot
  complex(default), dimension(:), allocatable :: fc
  external :: evolvePDFM, evolvePDFpM
  associate (data => sf_int%data)
    sf_int%q = scale
    x = sf_int%x
    s = sf_int%s
    xx = x
    qq = min (data% qmax, scale)
    qq = max (data% qmin, qq)
    if (.not. data% photon) then
       if (data% invert) then
          if (data%has_photon) then
             if (LHAPDF5_AVAILABLE) then
                call evolvePDFphotonM &
                     (data% set, xx, qq, ff(6:-6:-1), fphot)
```

```
            else if (LHAPDF6_AVAILABLE) then
               call data%pdf%evolve_pdfphotonm &
                    (xx, qq, ff(6:-6:-1), fphot)
            end if
         else
            if (data%hoppet_b_matching) then
               call hoppet_eval (xx, qq, ff(6:-6:-1))
            else
               if (LHAPDF5_AVAILABLE) then
                  call evolvePDFM (data% set, xx, qq, ff(6:-6:-1))
               else if (LHAPDF6_AVAILABLE) then
                  call data%pdf%evolve_pdfm (xx, qq, ff(6:-6:-1))
               end if
            end if
         end if
      else
         if (data%has_photon) then
            if (LHAPDF5_AVAILABLE) then
               call evolvePDFphotonM (data% set, xx, qq, ff, fphot)
            else if (LHAPDF6_AVAILABLE) then
               call data%pdf%evolve_pdfphotonm (xx, qq, ff, fphot)
            end if
         else
            if (data%hoppet_b_matching) then
               call hoppet_eval (xx, qq, ff)
            else
               if (LHAPDF5_AVAILABLE) then
                  call evolvePDFM (data% set, xx, qq, ff)
               else if (LHAPDF6_AVAILABLE) then
                  call data%pdf%evolve_pdfm (xx, qq, ff)
               end if
            end if
         end if
      end if
   else
      ss = s
      if (LHAPDF5_AVAILABLE) then
         call evolvePDFpM (data% set, xx, qq, &
              ss, data% photon_scheme, ff)
      else if (LHAPDF6_AVAILABLE) then
         call data%pdf%evolve_pdfpm (xx, qq, ss, &
              data%photon_scheme, ff)
      end if
   end if
   if (data%has_photon) then
      allocate (fc (count ([data%mask, data%mask_photon])))
      fc = max (pack ([ff, fphot] / x, &
           [data% mask, data%mask_photon]), 0._default)
   else
      allocate (fc (count (data%mask)))
      fc = max (pack (ff / x, data%mask), 0._default)
   end if
end associate
call sf_int%set_matrix_element (fc)
```

```
    sf_int%status = SF_EVALUATED
  end subroutine lhapdf_apply
```

15.16.10 Strong Coupling

Since the PDF codes provide a function for computing the running α_s value, we make this available as an implementation of the abstract `alpha_qcd_t` type, which is used for matrix element evaluation.

⟨*SF lhapdf: public*⟩+≡
```
  public :: alpha_qcd_lhapdf_t
```

⟨*SF lhapdf: types*⟩+≡
```
  type, extends (alpha_qcd_t) :: alpha_qcd_lhapdf_t
     type(string_t) :: pdfset_dir
     type(string_t) :: pdfset_file
     integer :: pdfset_member = -1
     type(lhapdf_pdf_t) :: pdf
   contains
   ⟨SF lhapdf: alpha qcd: TBP⟩
  end type alpha_qcd_lhapdf_t
```

Output. As in earlier versions we leave the LHAPDF path out.

⟨*SF lhapdf: alpha qcd: TBP*⟩≡
```
  procedure :: write => alpha_qcd_lhapdf_write
```

⟨*SF lhapdf: procedures*⟩+≡
```
  subroutine alpha_qcd_lhapdf_write (object, unit)
    class(alpha_qcd_lhapdf_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(3x,A)")  "QCD parameters (lhapdf):"
    write (u, "(5x,A,A)")  "PDF set    = ", char (object%pdfset_file)
    write (u, "(5x,A,I0)") "PDF member = ", object%pdfset_member
  end subroutine alpha_qcd_lhapdf_write
```

Calculation: the numeric member ID selects the correct PDF set, which must be properly initialized.

⟨*SF lhapdf: interfaces*⟩+≡
```
  interface
     double precision function alphasPDF (Q)
       double precision, intent(in) :: Q
     end function alphasPDF
  end interface
```

⟨*SF lhapdf: alpha qcd: TBP*⟩+≡
```
  procedure :: get => alpha_qcd_lhapdf_get
```

⟨*SF lhapdf: procedures*⟩+≡
```
function alpha_qcd_lhapdf_get (alpha_qcd, scale) result (alpha)
  class(alpha_qcd_lhapdf_t), intent(in) :: alpha_qcd
  real(default), intent(in) :: scale
  real(default) :: alpha
  if (LHAPDF5_AVAILABLE) then
     alpha = alphasPDF (dble (scale))
  else if (LHAPDF6_AVAILABLE) then
     alpha = alpha_qcd%pdf%alphas_pdf (dble (scale))
  end if
end function alpha_qcd_lhapdf_get
```

Initialization. We need to access the (quasi-global) initialization status.

⟨*SF lhapdf: alpha qcd: TBP*⟩+≡
```
procedure :: init => alpha_qcd_lhapdf_init
```

⟨*SF lhapdf: procedures*⟩+≡
```
subroutine alpha_qcd_lhapdf_init (alpha_qcd, file, member, path)
  class(alpha_qcd_lhapdf_t), intent(out) :: alpha_qcd
  type(string_t), intent(inout) :: file
  integer, intent(inout) :: member
  type(string_t), intent(inout) :: path
  alpha_qcd%pdfset_file = file
  alpha_qcd%pdfset_member = member
  if (alpha_qcd%pdfset_member < 0) &
       call msg_fatal ("QCD parameter initialization: PDF set " &
       // char (file) // " is unknown")
  if (LHAPDF5_AVAILABLE) then
     call lhapdf_initialize (1, path, file, member)
  else if (LHAPDF6_AVAILABLE) then
     call lhapdf_initialize &
          (1, path, file, member, alpha_qcd%pdf)
  end if
end subroutine alpha_qcd_lhapdf_init
```

15.16.11 Unit tests

Test module, followed by the corresponding implementation module.

⟨sf_lhapdf_ut.f90⟩≡
```
⟨File header⟩

module sf_lhapdf_ut
  use unit_tests
  use system_dependencies, only: LHAPDF5_AVAILABLE
  use system_dependencies, only: LHAPDF6_AVAILABLE
  use sf_lhapdf_uti

⟨Standard module head⟩

⟨SF lhapdf: public test⟩

contains
```

```
  ⟨SF lhapdf: test driver⟩

  end module sf_lhapdf_ut
```

⟨sf_lhapdf_uti.f90⟩≡

```
  ⟨File header⟩

  module sf_lhapdf_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use system_dependencies, only: LHAPDF5_AVAILABLE
    use system_dependencies, only: LHAPDF6_AVAILABLE
    use os_interface
    use physics_defs, only: PROTON
    use sm_qcd
    use lorentz
    use pdg_arrays
    use flavors
    use interactions, only: reset_interaction_counter
    use model_data
    use sf_base

    use sf_lhapdf

  ⟨Standard module head⟩

  ⟨SF lhapdf: test declarations⟩

  contains

  ⟨SF lhapdf: tests⟩

  end module sf_lhapdf_uti
```

API: driver for the unit tests below.

⟨*SF lhapdf: public test*⟩≡

```
    public :: sf_lhapdf_test
```

⟨*SF lhapdf: test driver*⟩≡

```
    subroutine sf_lhapdf_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨SF lhapdf: execute tests⟩
    end subroutine sf_lhapdf_test
```

Test structure function data

Construct and display a test structure function data object.

⟨*SF lhapdf: execute tests*⟩≡

```
    if (LHAPDF5_AVAILABLE) then
       call test (sf_lhapdf_1, "sf_lhapdf5_1", &
            "structure function configuration", &
```

```
        u, results)
else if (LHAPDF6_AVAILABLE) then
   call test (sf_lhapdf_1, "sf_lhapdf6_1", &
        "structure function configuration", &
        u, results)
end if
```

⟨*SF lhapdf: test declarations*⟩≡

```
public :: sf_lhapdf_1
```

⟨*SF lhapdf: tests*⟩≡

```
subroutine sf_lhapdf_1 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(pdg_array_t) :: pdg_in
  type(pdg_array_t), dimension(1) :: pdg_out
  integer, dimension(:), allocatable :: pdg1
  class(sf_data_t), allocatable :: data

  write (u, "(A)")  "* Test output: sf_lhapdf_1"
  write (u, "(A)")  "*   Purpose: initialize and display &
       &test structure function data"
  write (u, "(A)")

  write (u, "(A)")  "* Create empty data object"
  write (u, "(A)")

  call model%init_sm_test ()
  pdg_in = PROTON

  allocate (lhapdf_data_t :: data)
  call data%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize"
  write (u, "(A)")

  select type (data)
  type is (lhapdf_data_t)
     call data%init (model, pdg_in)
  end select

  call data%write (u)

  write (u, "(A)")

  write (u, "(1x,A)")  "Outgoing particle codes:"
  call data%get_pdg_out (pdg_out)
  pdg1 = pdg_out(1)
  write (u, "(2x,99(1x,I0))")  pdg1

  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: sf_lhapdf_1"
```

```
end subroutine sf_lhapdf_1
```

Test and probe structure function

Construct and display a structure function object based on the PDF builtin structure function.

⟨*SF lhapdf: execute tests*⟩+≡

```
if (LHAPDF5_AVAILABLE) then
   call test (sf_lhapdf_2, "sf_lhapdf5_2", &
        "structure function instance", &
        u, results)
else if (LHAPDF6_AVAILABLE) then
   call test (sf_lhapdf_2, "sf_lhapdf6_2", &
        "structure function instance", &
        u, results)
end if
```

⟨*SF lhapdf: test declarations*⟩+≡

```
public :: sf_lhapdf_2
```

⟨*SF lhapdf: tests*⟩+≡

```
subroutine sf_lhapdf_2 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t) :: flv
  type(pdg_array_t) :: pdg_in
  class(sf_data_t), allocatable, target :: data
  class(sf_int_t), allocatable :: sf_int
  type(vector4_t) :: k
  type(vector4_t), dimension(2) :: q
  real(default) :: E
  real(default), dimension(:), allocatable :: r, rb, x
  real(default) :: f

  write (u, "(A)")  "* Test output: sf_lhapdf_2"
  write (u, "(A)")  "*   Purpose: initialize and fill &
       &test structure function object"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize configuration data"
  write (u, "(A)")

  call model%init_sm_test ()
  call flv%init (PROTON, model)
  pdg_in = PROTON
  call lhapdf_global_reset ()

  call reset_interaction_counter ()

  allocate (lhapdf_data_t :: data)
  select type (data)
  type is (lhapdf_data_t)
```

```
   call data%init (model, pdg_in)
end select

write (u, "(A)")  "* Initialize structure-function object"
write (u, "(A)")

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize incoming momentum with E=500"
write (u, "(A)")
E = 500
k = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
call vector4_write (k, u)
call sf_int%seed_kinematics ([k])

write (u, "(A)")
write (u, "(A)")  "* Set kinematics for x=0.5"
write (u, "(A)")

allocate (r (data%get_n_par ()))
allocate (rb(size (r)))
allocate (x (size (r)))

r = 0.5_default
rb = 1 - r
call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
call sf_int%write (u)

write (u, "(A)")
write (u, "(A,9(1x,F10.7))")  "x =", x
write (u, "(A,9(1x,F10.7))")  "f =", f

write (u, "(A)")
write (u, "(A)")  "* Recover x from momenta"
write (u, "(A)")

q = sf_int%get_momenta (outgoing=.true.)
call sf_int%final ()
deallocate (sf_int)

call data%allocate_sf_int (sf_int)
call sf_int%init (data)
call sf_int%set_beam_index ([1])

call sf_int%seed_kinematics ([k])
call sf_int%set_momenta (q, outgoing=.true.)
call sf_int%recover_x (x)

write (u, "(A,9(1x,F10.7))")  "x =", x
```

```
      write (u, "(A)")
      write (u, "(A)")  "* Evaluate for Q = 100 GeV"
      write (u, "(A)")

      call sf_int%complete_kinematics (x, f, r, rb, map=.false.)
      call sf_int%apply (scale = 100._default)
      call sf_int%write (u, testflag = .true.)


      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call sf_int%final ()
      call model%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: sf_lhapdf_2"

    end subroutine sf_lhapdf_2
```

Strong Coupling

Test α_s as an implementation of the `alpha_qcd_t` abstract type.

⟨*SF lhapdf: execute tests*⟩+≡

```
    if (LHAPDF5_AVAILABLE) then
       call test (sf_lhapdf_3, "sf_lhapdf5_3", &
            "running alpha_s", &
            u, results)
    else if (LHAPDF6_AVAILABLE) then
       call test (sf_lhapdf_3, "sf_lhapdf6_3", &
            "running alpha_s", &
            u, results)
    end if
```

⟨*SF lhapdf: test declarations*⟩+≡

```
    public :: sf_lhapdf_3
```

⟨*SF lhapdf: tests*⟩+≡

```
    subroutine sf_lhapdf_3 (u)
      integer, intent(in) :: u
      type(qcd_t) :: qcd
      type(string_t) :: name, path
      integer :: member

      write (u, "(A)")  "* Test output: sf_lhapdf_3"
      write (u, "(A)")  "*   Purpose: initialize and evaluate alpha_s"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize configuration data"
      write (u, "(A)")

      call lhapdf_global_reset ()
```

```
    if (LHAPDF5_AVAILABLE) then
       name = "cteq6ll.LHpdf"
       member = 1
       path = ""
    else if (LHAPDF6_AVAILABLE) then
       name = "CT10"
       member = 1
       path = ""
    end if

    write (u, "(A)")  "* Initialize qcd object"
    write (u, "(A)")

    allocate (alpha_qcd_lhapdf_t :: qcd%alpha)
    select type (alpha => qcd%alpha)
    type is (alpha_qcd_lhapdf_t)
       call alpha%init (name, member, path)
    end select
    call qcd%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Evaluate for Q = 100"
    write (u, "(A)")

    write (u, "(1x,A,F8.5)")  "alpha = ", qcd%alpha%get (100._default)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: sf_lhapdf_3"

  end subroutine sf_lhapdf_3
```

15.17 Easy PDF Access

For the shower, subtraction and matching, it is very useful to have direct access to $f(x, Q)$ independently of the used library.

⟨pdf.f90⟩≡

```
  ⟨File header⟩

  module pdf

  ⟨Use kinds with double⟩
    use io_units
    use system_dependencies, only: LHAPDF5_AVAILABLE, LHAPDF6_AVAILABLE
    use diagnostics
    use beam_structures
    use lhapdf !NODEP!
    use pdf_builtin !NODEP!
```

⟨*Standard module head*⟩

⟨*PDF: public*⟩

⟨*PDF: parameters*⟩

⟨*PDF: types*⟩

```
contains
```

⟨*PDF: procedures*⟩

```
end module pdf
```

We support the following implementations:

⟨*PDF: parameters*⟩≡

```
integer, parameter, public :: STRF_NONE = 0
integer, parameter, public :: STRF_LHAPDF6 = 1
integer, parameter, public :: STRF_LHAPDF5 = 2
integer, parameter, public :: STRF_PDF_BUILTIN = 3
```

A container to bundle all necessary PDF data. Could be moved to a more central location.

⟨*PDF: public*⟩≡

```
public :: pdf_data_t
```

⟨*PDF: types*⟩≡

```
type :: pdf_data_t
  type(lhapdf_pdf_t) :: pdf
  real(default) :: xmin, xmax, qmin, qmax
  integer :: type = STRF_NONE
  integer :: set = 0
 contains
 ⟨PDF: pdf data: TBP⟩
end type pdf_data_t
```

⟨*PDF: pdf data: TBP*⟩≡

```
procedure :: init => pdf_data_init
```

⟨*PDF: procedures*⟩≡

```
subroutine pdf_data_init (pdf_data, pdf_data_in)
  class(pdf_data_t), intent(out) :: pdf_data
  type(pdf_data_t), target, intent(in) :: pdf_data_in
  pdf_data%xmin = pdf_data_in%xmin
  pdf_data%xmax = pdf_data_in%xmax
  pdf_data%qmin = pdf_data_in%qmin
  pdf_data%qmax = pdf_data_in%qmax
  pdf_data%set = pdf_data_in%set
  pdf_data%type = pdf_data_in%type
  if (pdf_data%type == STRF_LHAPDF6) then
     if (pdf_data_in%pdf%is_associated ()) then
        call lhapdf_copy_pointer (pdf_data_in%pdf, pdf_data%pdf)
     else
        call msg_bug ('pdf_data_init: pdf_data%pdf was not associated!')
```

```
      end if
    end if
  end subroutine pdf_data_init
```

⟨*PDF: pdf data: TBP*⟩+≡

```
  procedure :: write => pdf_data_write
```

⟨*PDF: procedures*⟩+≡

```
  subroutine pdf_data_write (pdf_data, unit)
    class(pdf_data_t), intent(in) :: pdf_data
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit); if (u < 0) return
    write (u, "(3x,A,I0)") "PDF set  = ", pdf_data%set
    write (u, "(3x,A,I0)") "PDF type = ", pdf_data%type
  end subroutine pdf_data_write
```

⟨*PDF: pdf data: TBP*⟩+≡

```
  procedure :: setup => pdf_data_setup
```

⟨*PDF: procedures*⟩+≡

```
  subroutine pdf_data_setup (pdf_data, caller, beam_structure, lhapdf_member, set)
    class(pdf_data_t), intent(inout) :: pdf_data
    character(len=*), intent(in) :: caller
    type(beam_structure_t), intent(in) :: beam_structure
    integer, intent(in) :: lhapdf_member, set
    real(default) :: xmin, xmax, q2min, q2max
    pdf_data%set = set
    if (beam_structure%contains ("lhapdf")) then
       if (LHAPDF6_AVAILABLE) then
          pdf_data%type = STRF_LHAPDF6
       else if (LHAPDF5_AVAILABLE) then
          pdf_data%type = STRF_LHAPDF5
       end if
       write (msg_buffer, "(A,I0)")  caller &
            // ": interfacing LHAPDF set #", pdf_data%set
       call msg_message ()
    else if (beam_structure%contains ("pdf_builtin")) then
       pdf_data%type = STRF_PDF_BUILTIN
       write (msg_buffer, "(A,I0)")  caller &
            // ": interfacing PDF builtin set #", pdf_data%set
       call msg_message ()
    end if
    select case (pdf_data%type)
    case (STRF_LHAPDF6)
       pdf_data%xmin = pdf_data%pdf%getxmin ()
       pdf_data%xmax = pdf_data%pdf%getxmax ()
       pdf_data%qmin = sqrt(pdf_data%pdf%getq2min ())
       pdf_data%qmax = sqrt(pdf_data%pdf%getq2max ())
    case (STRF_LHAPDF5)
       call GetXminM (1, lhapdf_member, xmin)
       call GetXmaxM (1, lhapdf_member, xmax)
       call GetQ2minM (1, lhapdf_member, q2min)
       call GetQ2maxM (1, lhapdf_member, q2max)
```

```
      pdf_data%xmin = xmin
      pdf_data%xmax = xmax
      pdf_data%qmin = sqrt(q2min)
      pdf_data%qmax = sqrt(q2max)
    end select
  end subroutine pdf_data_setup
```

This could be overloaded with a version that only asks for a specific flavor as it is supported by LHAPDF6.

⟨*PDF: pdf data: TBP*⟩+≡

```
  procedure :: evolve => pdf_data_evolve
```

⟨*PDF: procedures*⟩+≡

```
  subroutine pdf_data_evolve (pdf_data, x, q_in, f)
    class(pdf_data_t), intent(inout) :: pdf_data
    real(double), intent(in) :: x, q_in
    real(double), dimension(-6:6), intent(out) :: f
    real(double) :: q
    select case (pdf_data%type)
    case (STRF_PDF_BUILTIN)
       call pdf_evolve_LHAPDF (pdf_data%set, x, q_in, f)
    case (STRF_LHAPDF6)
       q = min (pdf_data%qmax, q_in)
       q = max (pdf_data%qmin, q)
       call pdf_data%pdf%evolve_pdfm (x, q, f)
    case (STRF_LHAPDF5)
       q = min (pdf_data%qmax, q_in)
       q = max (pdf_data%qmin, q)
       call evolvePDFM (pdf_data%set, x, q, f)
    case default
       call msg_fatal ("PDF function: unknown PDF method.")
    end select
  end subroutine pdf_data_evolve
```

15.18 Dispatch

⟨dispatch_beams.f90⟩≡

```
  ⟨File header⟩

  module dispatch_beams

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use diagnostics
    use os_interface, only: os_data_t
    use variables, only: var_list_t
    use constants, only: PI
    use numeric_utils, only: vanishes
    use physics_defs, only: PHOTON
    use rng_base, only: rng_factory_t
    use pdg_arrays
    use model_data, only: model_data_t
```

```
  use dispatch_rng, only: dispatch_rng_factory
  use flavors, only: flavor_t
  use sm_qcd, only: qcd_t, alpha_qcd_fixed_t, alpha_qcd_from_scale_t
  use sm_qcd, only: alpha_qcd_from_lambda_t
  use physics_defs, only: MZ_REF, ALPHA_QCD_MZ_REF

  use beam_structures
  use sf_base
  use sf_mappings
  use sf_isr
  use sf_epa
  use sf_ewa
  use sf_escan
  use sf_gaussian
  use sf_beam_events
  use sf_circe1
  use sf_circe2
  use sf_pdf_builtin
  use sf_lhapdf

⟨Standard module head⟩

⟨Dispatch beams: public⟩

⟨Dispatch beams: types⟩

⟨Dispatch beams: variables⟩

contains

⟨Dispatch beams: procedures⟩

end module dispatch_beams
```

This data type is a container for transferring structure-function specific data from the `dispatch_sf_data` to the `dispatch_sf_channels` subroutine.

⟨*Dispatch beams: public*⟩≡
```
  public :: sf_prop_t
```

⟨*Dispatch beams: types*⟩≡
```
  type :: sf_prop_t
     real(default), dimension(2) :: isr_eps = 1
  end type sf_prop_t
```

Allocate a structure-function configuration object according to the `sf_method` string.

The `sf_prop` object can be used to transfer structure-function specific data up and to the `dispatch_sf_channels` subroutine below, so they can be used for particular mappings.

The `var_list` object is intent(inout) because the RNG generator seed may change during initialization.

The `pdg_in` array is the array of incoming flavors, corresponding to the upstream structure function or the beam array. This will be checked for the structure function in question and replaced by the outgoing flavors. The `pdg_prc`

array is the array of incoming flavors (beam index, component index) for the hard process.

⟨*Dispatch beams: public*⟩+≡

```
public :: dispatch_sf_data
```

⟨*Dispatch beams: procedures*⟩≡

```
subroutine dispatch_sf_data (data, sf_method, i_beam, sf_prop, &
       var_list, var_list_global, model, &
       os_data, sqrts, pdg_in, pdg_prc, polarized)
  class(sf_data_t), allocatable, intent(inout) :: data
  type(string_t), intent(in) :: sf_method
  integer, dimension(:), intent(in) :: i_beam
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_in
  type(pdg_array_t), dimension(:,:), intent(in) :: pdg_prc
  type(sf_prop_t), intent(inout) :: sf_prop
  type(var_list_t), intent(in) :: var_list
  type(var_list_t), intent(inout) :: var_list_global
  class(model_data_t), target, intent(in) :: model
  type(os_data_t), intent(in) :: os_data
  real(default), intent(in) :: sqrts
  logical, intent(in) :: polarized
  type(pdg_array_t), dimension(:), allocatable :: pdg_out
  real(default) :: isr_alpha, isr_q_max, isr_mass
  integer :: isr_order
  logical :: isr_recoil, isr_keep_energy
  real(default) :: epa_alpha, epa_x_min, epa_q_min, epa_e_max, epa_mass
  logical :: epa_recoil, epa_keep_energy
  real(default) :: ewa_x_min, ewa_pt_max, ewa_mass
  logical :: ewa_recoil, ewa_keep_energy
  type(pdg_array_t), dimension(:), allocatable :: pdg_prc1
  integer :: ewa_id
  type(string_t) :: pdf_name
  type(string_t) :: lhapdf_dir, lhapdf_file
  type(string_t), dimension(13) :: lhapdf_photon_sets
  integer :: lhapdf_member, lhapdf_photon_scheme
  logical :: hoppet_b_matching
  class(rng_factory_t), allocatable :: rng_factory
  logical :: circe1_photon1, circe1_photon2, circe1_generate, &
       circe1_with_radiation
  real(default) :: circe1_sqrts, circe1_eps
  integer :: circe1_version, circe1_chattiness, &
       circe1_revision
  character(6) :: circe1_accelerator
  logical :: circe2_polarized
  type(string_t) :: circe2_design, circe2_file
  real(default), dimension(2) :: gaussian_spread
  logical :: beam_events_warn_eof
  type(string_t) :: beam_events_dir, beam_events_file
  logical :: escan_normalize
  lhapdf_photon_sets = [var_str ("DOG0.LHgrid"), var_str ("DOG1.LHgrid"), &
       var_str ("DGG.LHgrid"), var_str ("LACG.LHgrid"), &
       var_str ("GSG0.LHgrid"), var_str ("GSG1.LHgrid"), &
       var_str ("GSG960.LHgrid"), var_str ("GSG961.LHgrid"), &
       var_str ("GRVG0.LHgrid"), var_str ("GRVG1.LHgrid"), &
```

```
     var_str ("ACFGPG.LHgrid"), var_str ("WHITG.LHgrid"), &
     var_str ("SASG.LHgrid")]
select case (char (sf_method))
case ("pdf_builtin")
   allocate (pdf_builtin_data_t :: data)
   select type (data)
   type is (pdf_builtin_data_t)
      pdf_name = &
           var_list%get_sval (var_str ("$pdf_builtin_set"))
      hoppet_b_matching = &
           var_list%get_lval (var_str ("?hoppet_b_matching"))
      call data%init ( &
           model, pdg_in(i_beam(1)), &
           name = pdf_name, &
           path = os_data%pdf_builtin_datapath, &
           hoppet_b_matching = hoppet_b_matching)
   end select
case ("pdf_builtin_photon")
   call msg_fatal ("Currently, there are no photon PDFs built into WHIZARD,", &
        [var_str ("for the photon content inside a proton or neutron use"), &
         var_str ("the 'lhapdf_photon' structure function.")])
case ("lhapdf")
   allocate (lhapdf_data_t :: data)
   if (pdg_array_get (pdg_in(i_beam(1)), 1) == PHOTON) then
      call msg_fatal ("The 'lhapdf' structure is intended only for protons and", &
           [var_str ("pions, please use 'lhapdf_photon' for photon beams.")])
   end if
   lhapdf_dir = &
        var_list%get_sval (var_str ("$lhapdf_dir"))
   lhapdf_file = &
        var_list%get_sval (var_str ("$lhapdf_file"))
   lhapdf_member = &
        var_list%get_ival (var_str ("lhapdf_member"))
   lhapdf_photon_scheme = &
        var_list%get_ival (var_str ("lhapdf_photon_scheme"))
   hoppet_b_matching = &
        var_list%get_lval (var_str ("?hoppet_b_matching"))
   select type (data)
   type is (lhapdf_data_t)
      call data%init &
           (model, pdg_in(i_beam(1)), &
            lhapdf_dir, lhapdf_file, lhapdf_member, &
            lhapdf_photon_scheme, hoppet_b_matching)
   end select
case ("lhapdf_photon")
   allocate (lhapdf_data_t :: data)
   if (pdg_array_get_length (pdg_in(i_beam(1))) /= 1 .or. &
        pdg_array_get (pdg_in(i_beam(1)), 1) /= PHOTON) then
      call msg_fatal ("The 'lhapdf_photon' structure function is exclusively for", &
           [var_str ("photon PDFs, i.e. for photons as beam particles")])
   end if
   lhapdf_dir = &
        var_list%get_sval (var_str ("$lhapdf_dir"))
   lhapdf_file = &
```

```
         var_list%get_sval (var_str ("$lhapdf_photon_file"))
   lhapdf_member = &
         var_list%get_ival (var_str ("lhapdf_member"))
   lhapdf_photon_scheme = &
         var_list%get_ival (var_str ("lhapdf_photon_scheme"))
   if (.not. any (lhapdf_photon_sets == lhapdf_file)) then
      call msg_fatal ("This PDF set is not supported or not " // &
           "intended for photon beams.")
   end if
   select type (data)
   type is (lhapdf_data_t)
      call data%init &
           (model, pdg_in(i_beam(1)), &
            lhapdf_dir, lhapdf_file, lhapdf_member, &
            lhapdf_photon_scheme)
   end select
case ("isr")
   allocate (isr_data_t :: data)
   isr_alpha = &
         var_list%get_rval (var_str ("isr_alpha"))
   if (vanishes (isr_alpha)) then
      isr_alpha = (var_list%get_rval (var_str ("ee"))) &
           ** 2 / (4 * PI)
   end if
   isr_q_max = &
         var_list%get_rval (var_str ("isr_q_max"))
   if (vanishes (isr_q_max)) then
      isr_q_max = sqrts
   end if
   isr_mass   = var_list%get_rval (var_str ("isr_mass"))
   isr_order  = var_list%get_ival (var_str ("isr_order"))
   isr_recoil = var_list%get_lval (var_str ("?isr_recoil"))
   isr_keep_energy = var_list%get_lval (var_str ("?isr_keep_energy"))
   select type (data)
   type is (isr_data_t)
      call data%init &
           (model, pdg_in (i_beam(1)), isr_alpha, isr_q_max, &
           isr_mass, isr_order, recoil = isr_recoil, keep_energy = &
           isr_keep_energy)
      call data%check ()
      sf_prop%isr_eps(i_beam(1)) = data%get_eps ()
   end select
case ("epa")
   allocate (epa_data_t :: data)
   epa_alpha = var_list%get_rval (var_str ("epa_alpha"))
   if (vanishes (epa_alpha)) then
      epa_alpha = (var_list%get_rval (var_str ("ee"))) &
           ** 2 / (4 * PI)
   end if
   epa_x_min = var_list%get_rval (var_str ("epa_x_min"))
   epa_q_min = var_list%get_rval (var_str ("epa_q_min"))
   epa_e_max = var_list%get_rval (var_str ("epa_e_max"))
   if (vanishes (epa_e_max)) then
      epa_e_max = sqrts
```

```
   end if
   epa_mass   = var_list%get_rval (var_str ("epa_mass"))
   epa_recoil = var_list%get_lval (var_str ("?epa_recoil"))
   epa_keep_energy = var_list%get_lval (var_str ("?epa_keep_energy"))
   select type (data)
   type is (epa_data_t)
      call data%init &
           (model, pdg_in (i_beam(1)), epa_alpha, epa_x_min, &
           epa_q_min, epa_e_max, epa_mass, recoil = epa_recoil, &
           keep_energy = epa_keep_energy)
      call data%check ()
   end select
case ("ewa")
   allocate (ewa_data_t :: data)
   allocate (pdg_prc1 (size (pdg_prc, 2)))
   pdg_prc1 = pdg_prc(i_beam(1),:)
   if (any (pdg_array_get_length (pdg_prc1) /= 1) &
        .or. any (pdg_prc1 /= pdg_prc1(1))) then
      call msg_fatal &
           ("EWA: process incoming particle (W/Z) must be unique")
   end if
   ewa_id = abs (pdg_array_get (pdg_prc1(1), 1))
   ewa_x_min = var_list%get_rval (var_str ("ewa_x_min"))
   ewa_pt_max = var_list%get_rval (var_str ("ewa_pt_max"))
   if (vanishes (ewa_pt_max)) then
      ewa_pt_max = sqrts
   end if
   ewa_mass = var_list%get_rval (var_str ("ewa_mass"))
   ewa_recoil = var_list%get_lval (&
        var_str ("?ewa_recoil"))
   ewa_keep_energy = var_list%get_lval (&
        var_str ("?ewa_keep_energy"))
   select type (data)
   type is (ewa_data_t)
      call data%init &
           (model, pdg_in (i_beam(1)), ewa_x_min, &
           ewa_pt_max, sqrts, ewa_recoil, &
           ewa_keep_energy, ewa_mass)
      call data%set_id (ewa_id)
      call data%check ()
   end select
case ("circe1")
   allocate (circe1_data_t :: data)
   select type (data)
   type is (circe1_data_t)
      circe1_photon1 = &
           var_list%get_lval (var_str ("?circe1_photon1"))
      circe1_photon2 = &
           var_list%get_lval (var_str ("?circe1_photon2"))
      circe1_sqrts = &
           var_list%get_rval (var_str ("circe1_sqrts"))
      circe1_eps = &
           var_list%get_rval (var_str ("circe1_eps"))
      if (circe1_sqrts <= 0)  circe1_sqrts = sqrts
```

```
      circe1_generate = &
           var_list%get_lval (var_str ("?circe1_generate"))
      circe1_version = &
           var_list%get_ival (var_str ("circe1_ver"))
      circe1_revision = &
           var_list%get_ival (var_str ("circe1_rev"))
      circe1_accelerator = &
           char (var_list%get_sval (var_str ("$circe1_acc")))
      circe1_chattiness = &
           var_list%get_ival (var_str ("circe1_chat"))
      circe1_with_radiation = &
           var_list%get_lval (var_str ("?circe1_with_radiation"))
      call data%init (model, pdg_in, circe1_sqrts, circe1_eps, &
           [circe1_photon1, circe1_photon2], &
           circe1_version, circe1_revision, circe1_accelerator, &
           circe1_chattiness, circe1_with_radiation)
      if (circe1_generate) then
         call msg_message ("CIRCE1: activating generator mode")
         call dispatch_rng_factory (rng_factory, var_list_global)
         call data%set_generator_mode (rng_factory)
      end if
   end select
case ("circe2")
   allocate (circe2_data_t :: data)
   select type (data)
   type is (circe2_data_t)
      circe2_polarized = &
           var_list%get_lval (var_str ("?circe2_polarized"))
      circe2_file = &
           var_list%get_sval (var_str ("$circe2_file"))
      circe2_design = &
           var_list%get_sval (var_str ("$circe2_design"))
      call data%init (os_data, model, pdg_in, sqrts, &
           circe2_polarized, polarized, circe2_file, circe2_design)
      call msg_message ("CIRCE2: activating generator mode")
      call dispatch_rng_factory (rng_factory, var_list_global)
      call data%set_generator_mode (rng_factory)
   end select
case ("gaussian")
   allocate (gaussian_data_t :: data)
   select type (data)
   type is (gaussian_data_t)
      gaussian_spread = &
           [var_list%get_rval (var_str ("gaussian_spread1")), &
           var_list%get_rval (var_str ("gaussian_spread2"))]
      call dispatch_rng_factory (rng_factory, var_list_global)
      call data%init (model, pdg_in, gaussian_spread, rng_factory)
   end select
case ("beam_events")
   allocate (beam_events_data_t :: data)
   select type (data)
   type is (beam_events_data_t)
      beam_events_dir = os_data%whizard_beamsimpath
      beam_events_file = var_list%get_sval (&
```

```
            var_str ("$beam_events_file"))
         beam_events_warn_eof = var_list%get_lval (&
            var_str ("?beam_events_warn_eof"))
         call data%init (model, pdg_in, &
               beam_events_dir, beam_events_file, beam_events_warn_eof)
      end select
   case ("energy_scan")
      escan_normalize = &
         var_list%get_lval (var_str ("?energy_scan_normalize"))
      allocate (escan_data_t :: data)
      select type (data)
      type is (escan_data_t)
         if (escan_normalize) then
            call data%init (model, pdg_in)
         else
            call data%init (model, pdg_in, sqrts)
         end if
      end select
   case default
      if (associated (dispatch_sf_data_extra)) then
         call dispatch_sf_data_extra (data, sf_method, i_beam, &
            sf_prop, var_list, var_list_global, model, os_data, sqrts, pdg_in, &
            pdg_prc, polarized)
      end if
      if (.not. allocated (data)) then
         call msg_fatal ("Structure function '" &
            // char (sf_method) // "' not implemented")
      end if
   end select
   if (allocated (data)) then
      allocate (pdg_out (size (pdg_prc, 1)))
      call data%get_pdg_out (pdg_out)
      pdg_in(i_beam) = pdg_out
   end if
end subroutine dispatch_sf_data
```

This is a hook that allows us to inject further handlers for structure-function objects, in particular a test structure function.

⟨*Dispatch beams: public*⟩+≡

```
public :: dispatch_sf_data_extra
```

⟨*Dispatch beams: variables*⟩≡

```
procedure (dispatch_sf_data), pointer :: &
   dispatch_sf_data_extra => null ()
```

This is an auxiliary procedure, used by the beam-structure expansion: tell for a given structure function name, whether it corresponds to a pair spectrum ($n = 2$), a single-particle structure function ($n = 1$), or nothing ($n = 0$). Though `energy_scan` can in principle also be a pair spectrum, it always has only one parameter.

⟨*Dispatch beams: public*⟩+≡

```
public :: strfun_mode
```

⟨*Dispatch beams: procedures*⟩+≡

```
function strfun_mode (name) result (n)
  type(string_t), intent(in) :: name
  integer :: n
  select case (char (name))
  case ("none")
     n = 0
  case ("sf_test_0", "sf_test_1")
     n = 1
  case ("pdf_builtin","pdf_builtin_photon", &
        "lhapdf","lhapdf_photon")
     n = 1
  case ("isr","epa","ewa")
     n = 1
  case ("circe1", "circe2")
     n = 2
  case ("gaussian")
     n = 2
  case ("beam_events")
     n = 2
  case ("energy_scan")
     n = 2
  case default
     n = -1
     call msg_bug ("Structure function '" // char (name) &
          // "' not supported yet")
  end select
end function strfun_mode
```

Dispatch a whole structure-function chain, given beam data and beam structure data.

This could be done generically, but we should look at the specific combination of structure functions in order to select appropriate mappings.

The `beam_structure` argument gets copied because we want to expand it to canonical form (one valid structure-function entry per record) before proceeding further.

The `pdg_prc` argument is the array of incoming flavors. The first index is the beam index, the second one the process component index. Each element is itself a PDG array, notrivial if there is a flavor sum for the incoming state of this component.

The dispatcher is divided in two parts. The first part configures the structure function data themselves. After this, we can configure the phase space for the elementary process.

⟨*Dispatch beams: public*⟩+≡

```
public :: dispatch_sf_config
```

⟨*Dispatch beams: procedures*⟩+≡

```
subroutine dispatch_sf_config (sf_config, sf_prop, beam_structure, &
       var_list, var_list_global, model, os_data, sqrts, pdg_prc)
  type(sf_config_t), dimension(:), allocatable, intent(out) :: sf_config
  type(sf_prop_t), intent(out) :: sf_prop
  type(beam_structure_t), intent(inout) :: beam_structure
  type(var_list_t), intent(in) :: var_list
  type(var_list_t), intent(inout) :: var_list_global
```

```
    class(model_data_t), target, intent(in) :: model
    type(os_data_t), intent(in) :: os_data
    real(default), intent(in) :: sqrts
    class(sf_data_t), allocatable :: sf_data
    type(beam_structure_t) :: beam_structure_tmp
    type(pdg_array_t), dimension(:,:), intent(in) :: pdg_prc
    type(string_t), dimension(:), allocatable :: prt_in
    type(pdg_array_t), dimension(:), allocatable :: pdg_in
    type(flavor_t) :: flv_in
    integer :: n_beam, n_record, i
    beam_structure_tmp = beam_structure
    call beam_structure_tmp%expand (strfun_mode)
    n_record = beam_structure_tmp%get_n_record ()
    allocate (sf_config (n_record))
    n_beam = beam_structure_tmp%get_n_beam ()
    if (n_beam > 0) then
       allocate (prt_in (n_beam), pdg_in (n_beam))
       prt_in = beam_structure_tmp%get_prt ()
       do i = 1, n_beam
          call flv_in%init (prt_in(i), model)
          pdg_in(i) = flv_in%get_pdg ()
       end do
    else
       n_beam = size (pdg_prc, 1)
       allocate (pdg_in (n_beam))
       pdg_in = pdg_prc(:,1)
    end if
    do i = 1, n_record
       call dispatch_sf_data (sf_data, &
            beam_structure_tmp%get_name (i), &
            beam_structure_tmp%get_i_entry (i), &
            sf_prop, var_list, var_list_global, model, os_data, sqrts, &
            pdg_in, pdg_prc, &
            beam_structure_tmp%polarized ())
       call sf_config(i)%init (beam_structure_tmp%get_i_entry (i), sf_data)
       deallocate (sf_data)
    end do
  end subroutine dispatch_sf_config
```

15.18.1 QCD coupling

Allocate the `alpha` (running coupling) component of the `qcd` block with a concrete implementation, depending on the variable settings in the `global` record.

If a fixed α_s is requested, we do not allocate the `qcd%alpha` object. In this case, the matrix element code will just take the model parameter as-is, which implies fixed α_s. If the object is allocated, the α_s value is computed and updated for each matrix-element call.

⟨*Dispatch beams: public*⟩+≡

```
  public :: dispatch_qcd
```

⟨*Dispatch beams: procedures*⟩+≡

```
  subroutine dispatch_qcd (qcd, var_list, os_data)
```

```
    type(qcd_t), intent(inout) :: qcd
    type(var_list_t), intent(in) :: var_list
    type(os_data_t), intent(in) :: os_data
    logical :: fixed, from_mz, from_pdf_builtin, from_lhapdf, from_lambda_qcd
    real(default) :: mz, alpha_val, lambda
    integer :: nf, order, lhapdf_member
    type(string_t) :: pdfset, lhapdf_dir, lhapdf_file
    call unpack_variables ()
    if (allocated (qcd%alpha))  deallocate (qcd%alpha)
    if (from_lhapdf .and. from_pdf_builtin) then
        call msg_fatal (" Mixing alphas evolution",  &
             [var_str (" from LHAPDF and builtin PDF is not permitted")])
    end if
    select case (count ([from_mz, from_pdf_builtin, from_lhapdf, from_lambda_qcd]))
    case (0)
       if (fixed) then
          allocate (alpha_qcd_fixed_t :: qcd%alpha)
       else
          call msg_fatal ("QCD alpha: no calculation mode set")
       end if
    case (2:)
       call msg_fatal ("QCD alpha: calculation mode is ambiguous")
    case (1)
       if (fixed) then
          call msg_fatal ("QCD alpha: use '?alpha_s_is_fixed = false' for " // &
               "running alphas")
       else if (from_mz) then
          allocate (alpha_qcd_from_scale_t :: qcd%alpha)
       else if (from_pdf_builtin) then
          allocate (alpha_qcd_pdf_builtin_t :: qcd%alpha)
       else if (from_lhapdf) then
          allocate (alpha_qcd_lhapdf_t :: qcd%alpha)
       else if (from_lambda_qcd) then
          allocate (alpha_qcd_from_lambda_t :: qcd%alpha)
       end if
       call msg_message ("QCD alpha: using a running strong coupling")
    end select
    call init_alpha ()
  contains
  ⟨Dispatch qcd: dispatch qcd: procedures⟩
  end subroutine dispatch_qcd
```

⟨*Dispatch qcd: dispatch qcd: procedures*⟩≡

```
  subroutine unpack_variables ()
    fixed = var_list%get_lval (var_str ("?alpha_s_is_fixed"))
    from_mz = var_list%get_lval (var_str ("?alpha_s_from_mz"))
    from_pdf_builtin = &
         var_list%get_lval (var_str ("?alpha_s_from_pdf_builtin"))
    from_lhapdf = &
         var_list%get_lval (var_str ("?alpha_s_from_lhapdf"))
    from_lambda_qcd = &
         var_list%get_lval (var_str ("?alpha_s_from_lambda_qcd"))
    pdfset = var_list%get_sval (var_str ("$pdf_builtin_set"))
    lambda = var_list%get_rval (var_str ("lambda_qcd"))
```

```
    nf = var_list%get_ival (var_str ("alpha_s_nf"))
    order = var_list%get_ival (var_str ("alpha_s_order"))
    lhapdf_dir = var_list%get_sval (var_str ("$lhapdf_dir"))
    lhapdf_file = var_list%get_sval (var_str ("$lhapdf_file"))
    lhapdf_member = var_list%get_ival (var_str ("lhapdf_member"))
    if (var_list%contains (var_str ("mZ"))) then
       mz = var_list%get_rval (var_str ("mZ"))
    else
       mz = MZ_REF
    end if
    if (var_list%contains (var_str ("alphas"))) then
       alpha_val = var_list%get_rval (var_str ("alphas"))
    else
       alpha_val = ALPHA_QCD_MZ_REF
    end if
  end subroutine unpack_variables
```

⟨*Dispatch qcd: dispatch qcd: procedures*⟩+≡

```
  subroutine init_alpha ()
    select type (alpha => qcd%alpha)
    type is (alpha_qcd_fixed_t)
       alpha%val = alpha_val
    type is (alpha_qcd_from_scale_t)
       alpha%mu_ref = mz
       alpha%ref = alpha_val
       alpha%order = order
       alpha%nf = nf
    type is (alpha_qcd_from_lambda_t)
       alpha%lambda = lambda
       alpha%order = order
       alpha%nf = nf
    type is (alpha_qcd_pdf_builtin_t)
       call alpha%init (pdfset, &
            os_data%pdf_builtin_datapath)
    type is (alpha_qcd_lhapdf_t)
       call alpha%init (lhapdf_file, lhapdf_member, lhapdf_dir)
    end select
  end subroutine init_alpha
```

Chapter 16

Interface for Matrix Element Objects

These modules manage internal and, in particular, external matrix-element code.

prc_core We define the abstract `prc_core_t` type which handles all specific features of kinematics matrix-element evaluation that depend on a particular class of processes. This abstract type supplements the `prc_core_def_t` type and related types in another module. Together, they provide a complete set of matrix-element handlers that are implemented in the concrete types below.

These are the implementations:

prc_template_me Implements matrix-element code without actual content (trivial value), but full-fledged interface. This can be used for injecting user-defined matrix-element code.

prc_omega Matrix elements calculated by `O'MEGA` are the default for WHIZARD. Here, we provide all necessary support.

prc_gosam

prc_openloops

prc_user_defined_test

prc_threshold

16.1 Abstract process core

In this module we provide abstract data types for process classes. Each process class represents a set of processes which are handled by a common "method", e.g., by the `O'MEGA` matrix-element generator. The process class is also able to select a particular implementation for the phase-space and integration modules.

For a complete implementation of a process class, we have to provide extensions of the following abstract types:

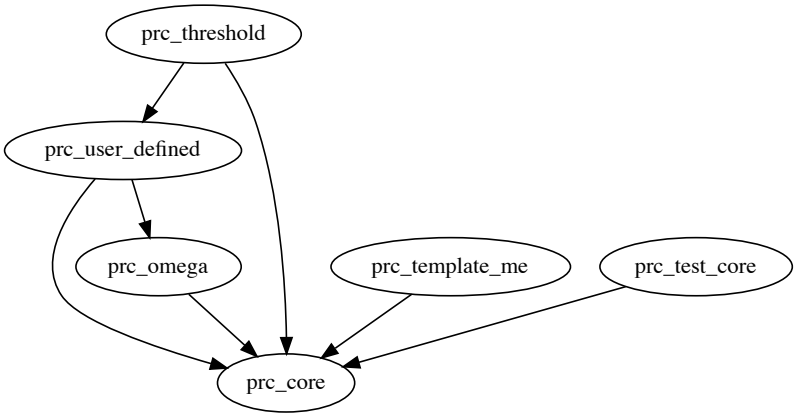

Figure 16.1: Module dependencies in `src/me_methods`.

prc_core_def_t process and matrix-element configuration

prc_writer_t (optional) writing external matrix-element code

prc_driver_t accessing the matrix element (internal or external)

prc_core_t evaluating kinematics and matrix element. The process core also selects phase-space and integrator implementations as appropriate for the process class and configuration.

In the actual process-handling data structures, each process component contains an instance of such a process class as its core. This allows us to keep the `processes` module below, which supervises matrix-element evaluation, integration, and event generation, free of any reference to concrete implementations (for the process class, phase space, and integrator).

There are no unit tests, these are deferred to the `processes` module.

⟨`prc_core.f90`⟩≡
```
  ⟨File header⟩
  module prc_core

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use io_units
    use diagnostics
    use lorentz
    use interactions

    use process_constants
    use prc_core_def
    use process_libraries
    use sf_base
```

⟨*Standard module head*⟩

⟨*Prc core: public*⟩

⟨*Prc core: types*⟩

⟨*Prc core: interfaces*⟩

```
contains
```

⟨*Prc core: procedures*⟩

```
end module prc_core
```

16.1.1 The process core

The process core is of abstract data type. Different types of matrix element will be represented by different implementations.

⟨*Prc core: public*⟩≡

```
public :: prc_core_t
```

⟨*Prc core: types*⟩≡

```
type, abstract :: prc_core_t
   class(prc_core_def_t), pointer :: def => null ()
   logical :: data_known = .false.
   type(process_constants_t) :: data
   class(prc_core_driver_t), allocatable :: driver
   logical :: use_color_factors = .false.
   integer :: nc = 3
 contains
 ⟨Prc core: process core: TBP⟩
end type prc_core_t
```

In any case there must be an output routine.

⟨*Prc core: process core: TBP*⟩≡

```
procedure(prc_core_write), deferred :: write
```

⟨*Prc core: interfaces*⟩≡

```
abstract interface
   subroutine prc_core_write (object, unit)
     import
     class(prc_core_t), intent(in) :: object
     integer, intent(in), optional :: unit
   end subroutine prc_core_write
end interface
```

Just type the name of the actual core method.

⟨*Prc core: process core: TBP*⟩+≡

```
procedure(prc_core_write_name), deferred :: write_name
```

⟨*Prc core: interfaces*⟩+≡

```
abstract interface
   subroutine prc_core_write_name (object, unit)
```

```
      import
      class(prc_core_t), intent(in) :: object
      integer, intent(in), optional :: unit
    end subroutine prc_core_write_name
end interface
```

For initialization, we assign a pointer to the process entry in the relevant library. This allows us to access all process functions via the implementation of `prc_core_t`.

We declare the `object` as `intent(inout)`, since just after allocation it may be useful to store some extra data in the object, which we can then use in the actual initialization. This applies to extensions of `prc_core` which override the `init` method.

⟨*Prc core: process core: TBP*⟩+≡

```
procedure :: init => prc_core_init
procedure :: base_init => prc_core_init
```

⟨*Prc core: procedures*⟩≡

```
subroutine prc_core_init (object, def, lib, id, i_component)
  class(prc_core_t), intent(inout) :: object
  class(prc_core_def_t), intent(in), target :: def
  type(process_library_t), intent(in), target :: lib
  type(string_t), intent(in) :: id
  integer, intent(in) :: i_component
  object%def => def
  call lib%connect_process (id, i_component, object%data, object%driver)
  object%data_known = .true.
end subroutine prc_core_init
```

Return true if the matrix element generation was successful. This can be tested by looking at the number of generated flavor states, which should be nonzero.

⟨*Prc core: process core: TBP*⟩+≡

```
procedure :: has_matrix_element => prc_core_has_matrix_element
```

⟨*Prc core: procedures*⟩+≡

```
function prc_core_has_matrix_element (object) result (flag)
  class(prc_core_t), intent(in) :: object
  logical :: flag
  flag = object%data%n_flv /= 0
end function prc_core_has_matrix_element
```

Tell whether a particular combination of flavor/helicity/color state is allowed for the matrix element.

⟨*Prc core: process core: TBP*⟩+≡

```
procedure(prc_core_is_allowed), deferred :: is_allowed
```

⟨*Prc core: interfaces*⟩+≡

```
abstract interface
   function prc_core_is_allowed (object, i_term, f, h, c) result (flag)
     import
     class(prc_core_t), intent(in) :: object
     integer, intent(in) :: i_term, f, h, c
     logical :: flag
```

```
    end function prc_core_is_allowed
  end interface
```

Set the constant process data for a specific term. By default, these are the constants stored inside the object, ignoring the term index. Type extensions may override this and provide term-specific data.

⟨*Prc core: process core: TBP*⟩+≡

```
  procedure :: get_constants => prc_core_get_constants
```

⟨*Prc core: procedures*⟩+≡

```
  subroutine prc_core_get_constants (object, data, i_term)
    class(prc_core_t), intent(in) :: object
    type(process_constants_t), intent(out) :: data
    integer, intent(in) :: i_term
    data = object%data
  end subroutine prc_core_get_constants
```

The strong coupling is not among the process constants. The default implementation is to return a negative number, which indicates that α_s is not available. This may be overridden by an implementation that provides an (event-specific) value. The value can be stored in the process-specific workspace.

⟨*Prc core: process core: TBP*⟩+≡

```
  procedure :: get_alpha_s => prc_core_get_alpha_s
```

⟨*Prc core: procedures*⟩+≡

```
  function prc_core_get_alpha_s (object, core_state) result (alpha)
    class(prc_core_t), intent(in) :: object
    class(prc_core_state_t), intent(in), allocatable :: core_state
    real(default) :: alpha
    alpha = -1
  end function prc_core_get_alpha_s
```

Allocate the workspace associated to a process component. The default is that there is no workspace, so we do nothing. A type extension may override this and allocate a workspace object of appropriate type, which can be used in further calculations.

In any case, the `intent(out)` attribute deletes any previously allocated workspace.

⟨*Prc core: process core: TBP*⟩+≡

```
  procedure :: allocate_workspace => prc_core_ignore_workspace
```

⟨*Prc core: procedures*⟩+≡

```
  subroutine prc_core_ignore_workspace (object, core_state)
    class(prc_core_t), intent(in) :: object
    class(prc_core_state_t), intent(inout), allocatable :: core_state
  end subroutine prc_core_ignore_workspace
```

Compute the momenta in the hard interaction, taking the seed kinematics as input. The `i_term` index tells us which term we want to compute. (The standard method is to just transfer the momenta to the hard interaction.)

⟨*Prc core: process core: TBP*⟩+≡

```
  procedure(prc_core_compute_hard_kinematics), deferred :: &
       compute_hard_kinematics
```

⟨*Prc core: interfaces*⟩+≡

```
abstract interface
   subroutine prc_core_compute_hard_kinematics &
        (object, p_seed, i_term, int_hard, core_state)
     import
     class(prc_core_t), intent(in) :: object
     type(vector4_t), dimension(:), intent(in) :: p_seed
     integer, intent(in) :: i_term
     type(interaction_t), intent(inout) :: int_hard
     class(prc_core_state_t), intent(inout), allocatable :: core_state
   end subroutine prc_core_compute_hard_kinematics
end interface
```

Compute the momenta in the effective interaction, taking the hard kinematics as input. (This is called only if parton recombination is to be applied for the process variant.)

⟨*Prc core: process core: TBP*⟩+≡

```
procedure(prc_core_compute_eff_kinematics), deferred :: &
     compute_eff_kinematics
```

⟨*Prc core: interfaces*⟩+≡

```
abstract interface
   subroutine prc_core_compute_eff_kinematics &
        (object, i_term, int_hard, int_eff, core_state)
     import
     class(prc_core_t), intent(in) :: object
     integer, intent(in) :: i_term
     type(interaction_t), intent(in) :: int_hard
     type(interaction_t), intent(inout) :: int_eff
     class(prc_core_state_t), intent(inout), allocatable :: core_state
   end subroutine prc_core_compute_eff_kinematics
end interface
```

The process core must implement this function. Here, `j` is the index of the particular term we want to compute. The amplitude may depend on the factorization and renormalization scales.

The `core_state` (workspace) argument may be used if it is provided by the caller. Otherwise, the routine should compute the result directly.

⟨*Prc core: process core: TBP*⟩+≡

```
procedure(prc_core_compute_amplitude), deferred :: compute_amplitude
```

⟨*Prc core: interfaces*⟩+≡

```
abstract interface
   function prc_core_compute_amplitude &
        (object, j, p, f, h, c, fac_scale, ren_scale, alpha_qcd_forced, &
        core_state) result (amp)
     import
     complex(default) :: amp
     class(prc_core_t), intent(in) :: object
     integer, intent(in) :: j
     type(vector4_t), dimension(:), intent(in) :: p
     integer, intent(in) :: f, h, c
     real(default), intent(in) :: fac_scale, ren_scale
```

```
      real(default), intent(in), allocatable :: alpha_qcd_forced
      class(prc_core_state_t), intent(inout), allocatable, optional :: &
           core_state
   end function prc_core_compute_amplitude
end interface
```

16.1.2 Storage for intermediate results

The abstract `prc_core_state_t` type allows process cores to set up temporary workspace. The object is an extra argument for each of the individual calculations between kinematics setup and matrix-element evaluation.

⟨*Prc core: public*⟩+≡

```
public :: prc_core_state_t
```

⟨*Prc core: types*⟩+≡

```
type, abstract :: prc_core_state_t
 contains
   procedure(workspace_write), deferred :: write
   procedure(workspace_reset_new_kinematics), deferred :: reset_new_kinematics
end type prc_core_state_t
```

For debugging, we should at least have an output routine.

⟨*Prc core: interfaces*⟩+≡

```
abstract interface
   subroutine workspace_write (object, unit)
     import
     class(prc_core_state_t), intent(in) :: object
     integer, intent(in), optional :: unit
   end subroutine workspace_write
end interface
```

This is used during the NLO calculation, see there for more information.

⟨*Prc core: interfaces*⟩+≡

```
abstract interface
  subroutine workspace_reset_new_kinematics (object)
    import
    class(prc_core_state_t), intent(inout) :: object
  end subroutine workspace_reset_new_kinematics
end interface
```

16.1.3 Helicity selection data

This is intended for use with `O'MEGA`, but may also be made available to other process methods. We set thresholds for counting the times a specific helicity amplitude is zero. When the threshold is reached, we skip this amplitude in subsequent calls.

For initializing the helicity counters, we need an object that holds the two parameters, the threshold (large real number) and the cutoff (integer).

A helicity value suppressed by more than `threshold` (a value which multiplies `epsilon`, to be compared with the average of the current amplitude, default is 10^{10}) is treated as zero. A matrix element is assumed to be zero and not called again if it has been zero `cutoff` times.

⟨*Prc core: public*⟩+≡
```
public :: helicity_selection_t
```

⟨*Prc core: types*⟩+≡
```
type :: helicity_selection_t
   logical :: active = .false.
   real(default) :: threshold = 0
   integer :: cutoff = 0
 contains
 ⟨Prc core: helicity selection: TBP⟩
end type helicity_selection_t
```

Output. If the selection is inactive, print nothing.

⟨*Prc core: helicity selection: TBP*⟩≡
```
procedure :: write => helicity_selection_write
```

⟨*Prc core: procedures*⟩+≡
```
subroutine helicity_selection_write (object, unit)
  class(helicity_selection_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  if (object%active) then
     write (u, "(3x,A)")  "Helicity selection data:"
     write (u, "(5x,A,ES17.10)") &
          "threshold =", object%threshold
     write (u, "(5x,A,I0)") &
          "cutoff    = ", object%cutoff
  end if
end subroutine helicity_selection_write
```

16.2 Test process type

For the following tests, we define a simple implementation of the abstract `prc_core_t`, designed such as to complement the `prc_test_t` process definition type.

Note that it is not given that the actual process is defined as `prc_test_t` type. We enforce this by calling `prc_test_create_library`. The driver component in the process core will then become of type `prc_test_t`.

⟨`prc_test_core.f90`⟩≡
```
⟨File header⟩

module prc_test_core

⟨Use kinds⟩
  use io_units
  use lorentz
```

```
  use interactions
  use prc_test
  use prc_core
```

⟨*Standard module head*⟩

⟨*Prc test core: public*⟩

⟨*Prc test core: types*⟩

```
contains
```

⟨*Prc test core: procedures*⟩

```
end module prc_test_core
```

⟨*Prc test core: public*⟩≡

```
  public :: test_t
```

⟨*Prc test core: types*⟩≡

```
  type, extends (prc_core_t) :: test_t
   contains
   ⟨Prc test core: test type: TBP⟩
  end type test_t
```

⟨*Prc test core: test type: TBP*⟩≡

```
  procedure :: write => test_write
```

⟨*Prc test core: procedures*⟩≡

```
  subroutine test_write (object, unit)
    class(test_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(3x,A)")  "test type implementing prc_test"
  end subroutine test_write
```

⟨*Prc test core: test type: TBP*⟩+≡

```
  procedure :: write_name => test_write_name
```

⟨*Prc test core: procedures*⟩+≡

```
  subroutine test_write_name (object, unit)
    class(test_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u,"(1x,A)")  "Core: prc_test"
  end subroutine test_write_name
```

This process type always needs a MC parameter set and a single term. This only state is always allowed.

⟨*Prc test core: test type: TBP*⟩+≡

```
  procedure :: needs_mcset => test_needs_mcset
  procedure :: get_n_terms => test_get_n_terms
  procedure :: is_allowed => test_is_allowed
```

⟨*Prc test core: procedures*⟩+≡

```
function test_needs_mcset (object) result (flag)
  class(test_t), intent(in) :: object
  logical :: flag
  flag = .true.
end function test_needs_mcset

function test_get_n_terms (object) result (n)
  class(test_t), intent(in) :: object
  integer :: n
  n = 1
end function test_get_n_terms

function test_is_allowed (object, i_term, f, h, c) result (flag)
  class(test_t), intent(in) :: object
  integer, intent(in) :: i_term, f, h, c
  logical :: flag
  flag = .true.
end function test_is_allowed
```

Transfer the generated momenta directly to the hard interaction in the (only) term. We assume that everything has been set up correctly, so the array fits.

⟨*Prc test core: test type: TBP*⟩+≡

```
procedure :: compute_hard_kinematics => test_compute_hard_kinematics
```

⟨*Prc test core: procedures*⟩+≡

```
subroutine test_compute_hard_kinematics &
     (object, p_seed, i_term, int_hard, core_state)
  class(test_t), intent(in) :: object
  type(vector4_t), dimension(:), intent(in) :: p_seed
  integer, intent(in) :: i_term
  type(interaction_t), intent(inout) :: int_hard
  class(prc_core_state_t), intent(inout), allocatable :: core_state
  call int_hard%set_momenta (p_seed)
end subroutine test_compute_hard_kinematics
```

This procedure is not called for `test_t`, just a placeholder.

⟨*Prc test core: test type: TBP*⟩+≡

```
procedure :: compute_eff_kinematics => test_compute_eff_kinematics
```

⟨*Prc test core: procedures*⟩+≡

```
subroutine test_compute_eff_kinematics &
     (object, i_term, int_hard, int_eff, core_state)
  class(test_t), intent(in) :: object
  integer, intent(in) :: i_term
  type(interaction_t), intent(in) :: int_hard
  type(interaction_t), intent(inout) :: int_eff
  class(prc_core_state_t), intent(inout), allocatable :: core_state
end subroutine test_compute_eff_kinematics
```

Transfer the incoming momenta of `p_seed` directly to the effective interaction, and vice versa for the outgoing momenta.

int_hard is left untouched since int_eff is an alias (via pointer) to it.

⟨*Prc test core: test type: TBP*⟩+≡

```
procedure :: recover_kinematics => test_recover_kinematics
```

⟨*Prc test core: procedures*⟩+≡

```
subroutine test_recover_kinematics &
     (object, p_seed, int_hard, int_eff, core_state)
  class(test_t), intent(in) :: object
  type(vector4_t), dimension(:), intent(inout) :: p_seed
  type(interaction_t), intent(inout) :: int_hard
  type(interaction_t), intent(inout) :: int_eff
  class(prc_core_state_t), intent(inout), allocatable :: core_state
  integer :: n_in
  n_in = int_eff%get_n_in ()
  call int_eff%set_momenta (p_seed(1:n_in), outgoing = .false.)
  p_seed(n_in+1:) = int_eff%get_momenta (outgoing = .true.)
end subroutine test_recover_kinematics
```

Compute the amplitude. The driver ignores all quantum numbers and, in fact, returns a constant. Nevertheless, we properly transfer the momentum vectors.

⟨*Prc test core: test type: TBP*⟩+≡

```
procedure :: compute_amplitude => test_compute_amplitude
```

⟨*Prc test core: procedures*⟩+≡

```
function test_compute_amplitude &
     (object, j, p, f, h, c, fac_scale, ren_scale, alpha_qcd_forced, core_state) &
     result (amp)
  class(test_t), intent(in) :: object
  integer, intent(in) :: j
  type(vector4_t), dimension(:), intent(in) :: p
  integer, intent(in) :: f, h, c
  real(default), intent(in) :: fac_scale, ren_scale
  real(default), intent(in), allocatable :: alpha_qcd_forced
  class(prc_core_state_t), intent(inout), allocatable, optional :: core_state
  complex(default) :: amp
  real(default), dimension(:,:), allocatable :: parray
  integer :: i, n_tot
  select type (driver => object%driver)
  type is (prc_test_t)
     if (driver%scattering) then
        n_tot = 4
     else
        n_tot = 3
     end if
     allocate (parray (0:3,n_tot))
     forall (i = 1:n_tot)  parray(:,i) = vector4_get_components (p(i))
     amp = driver%get_amplitude (parray)
  end select
end function test_compute_amplitude
```

16.3 Template matrix elements

Here, we provide template matrix elements that are in structure very similar to O'MEGA matrix elements, but do not need its infrastructure. Per default, the matrix elements are flat, i.e. they have the constant value one. Analogous to the O'MEGA implementation, this section implements the interface to the templates (via the makefile) and the driver.

⟨prc_template_me.f90⟩≡

```
⟨File header⟩

module prc_template_me

  use, intrinsic ::  iso_c_binding !NODEP!

  use kinds
⟨Use strings⟩
  use io_units
  use system_defs, only: TAB
  use diagnostics
  use os_interface
  use lorentz
  use flavors
  use interactions
  use model_data

  use particle_specifiers, only: new_prt_spec
  use process_constants
  use prclib_interfaces
  use prc_core_def
  use process_libraries
  use prc_core

⟨Standard module head⟩

⟨Template matrix elements: public⟩

⟨Template matrix elements: types⟩

⟨Template matrix elements: interfaces⟩

contains

⟨Template matrix elements: procedures⟩

end module prc_template_me
```

16.3.1 Process definition

For the process definition we implement an extension of the prc_core_def_t abstract type.

⟨*Template matrix elements: public*⟩≡

```
  public :: template_me_def_t
```

⟨*Template matrix elements: types*⟩≡
```
type, extends (prc_core_def_t) :: template_me_def_t
 contains
 ⟨Template matrix elements: template ME def: TBP⟩
end type template_me_def_t
```

⟨*Template matrix elements: template ME def: TBP*⟩≡
```
procedure, nopass :: type_string => template_me_def_type_string
```

⟨*Template matrix elements: procedures*⟩≡
```
function template_me_def_type_string () result (string)
  type(string_t) :: string
  string = "template"
end function template_me_def_type_string
```

Initialization: allocate the writer for the template matrix element. Also set any data for this process that the writer needs.

⟨*Template matrix elements: template ME def: TBP*⟩+≡
```
procedure :: init => template_me_def_init
```

⟨*Template matrix elements: procedures*⟩+≡
```
subroutine template_me_def_init &
     (object, model, prt_in, prt_out, unity)
  class(template_me_def_t), intent(out) :: object
  class(model_data_t), intent(in), target :: model
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
  logical, intent(in) :: unity
  allocate (template_me_writer_t :: object%writer)
  select type (writer => object%writer)
  type is (template_me_writer_t)
     call writer%init (model, prt_in, prt_out, unity)
  end select
end subroutine template_me_def_init
```

Write/read process- and method-specific data.

⟨*Template matrix elements: template ME def: TBP*⟩+≡
```
procedure :: write => template_me_def_write
```

⟨*Template matrix elements: procedures*⟩+≡
```
subroutine template_me_def_write (object, unit)
  class(template_me_def_t), intent(in) :: object
  integer, intent(in) :: unit
  select type (writer => object%writer)
  type is (template_me_writer_t)
     call writer%write (unit)
  end select
end subroutine template_me_def_write
```

⟨*Template matrix elements: template ME def: TBP*⟩+≡
```
procedure :: read => template_me_def_read
```

⟨*Template matrix elements: procedures*⟩+≡

```
subroutine template_me_def_read (object, unit)
  class(template_me_def_t), intent(out) :: object
  integer, intent(in) :: unit
  call msg_bug &
       ("WHIZARD template process definition: input not supported (yet)")
end subroutine template_me_def_read
```

Allocate the driver for template matrix elements.

⟨*Template matrix elements: template ME def: TBP*⟩+≡

```
procedure :: allocate_driver => template_me_def_allocate_driver
```

⟨*Template matrix elements: procedures*⟩+≡

```
subroutine template_me_def_allocate_driver (object, driver, basename)
  class(template_me_def_t), intent(in) :: object
  class(prc_core_driver_t), intent(out), allocatable :: driver
  type(string_t), intent(in) :: basename
  allocate (template_me_driver_t :: driver)
end subroutine template_me_def_allocate_driver
```

We need code:

⟨*Template matrix elements: template ME def: TBP*⟩+≡

```
procedure, nopass :: needs_code => template_me_def_needs_code
```

⟨*Template matrix elements: procedures*⟩+≡

```
function template_me_def_needs_code () result (flag)
  logical :: flag
  flag = .true.
end function template_me_def_needs_code
```

These are the features that a template matrix element provides.

⟨*Template matrix elements: template ME def: TBP*⟩+≡

```
procedure, nopass :: get_features => template_me_def_get_features
```

⟨*Template matrix elements: procedures*⟩+≡

```
subroutine template_me_def_get_features (features)
  type(string_t), dimension(:), allocatable, intent(out) :: features
  allocate (features (5))
  features = [ &
       var_str ("init"), &
       var_str ("update_alpha_s"), &
       var_str ("is_allowed"), &
       var_str ("new_event"), &
       var_str ("get_amplitude")]
end subroutine template_me_def_get_features
```

The interface of the specific features.

⟨*Template matrix elements: interfaces*⟩≡

```
abstract interface
   subroutine init_t (par, scheme) bind(C)
     import
     real(c_default_float), dimension(*), intent(in) :: par
     integer(c_int), intent(in) :: scheme
```

```
   end subroutine init_t
end interface

abstract interface
   subroutine update_alpha_s_t (alpha_s) bind(C)
     import
     real(c_default_float), intent(in) :: alpha_s
   end subroutine update_alpha_s_t
end interface

abstract interface
   subroutine is_allowed_t (flv, hel, col, flag) bind(C)
     import
     integer(c_int), intent(in) :: flv, hel, col
     logical(c_bool), intent(out) :: flag
   end subroutine is_allowed_t
end interface

abstract interface
   subroutine new_event_t (p) bind(C)
     import
     real(c_default_float), dimension(0:3,*), intent(in) :: p
   end subroutine new_event_t
end interface

abstract interface
   subroutine get_amplitude_t (flv, hel, col, amp) bind(C)
     import
     integer(c_int), intent(in) :: flv, hel, col
     complex(c_default_complex), intent(out):: amp
   end subroutine get_amplitude_t
end interface
```

Connect the template matrix element features with the process driver.

⟨*Template matrix elements: template ME def: TBP*⟩+≡

```
procedure :: connect => template_me_def_connect
```

⟨*Template matrix elements: procedures*⟩+≡

```
subroutine template_me_def_connect (def, lib_driver, i, proc_driver)
  class(template_me_def_t), intent(in) :: def
  class(prclib_driver_t), intent(in) :: lib_driver
  integer, intent(in) :: i
  class(prc_core_driver_t), intent(inout) :: proc_driver
  integer(c_int) :: pid, fid
  type(c_funptr) :: fptr
  select type (proc_driver)
  type is  (template_me_driver_t)
     pid = i
     fid = 1
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%init)
     fid = 2
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%update_alpha_s)
```

```
      fid = 3
      call lib_driver%get_fptr (pid, fid, fptr)
      call c_f_procpointer (fptr, proc_driver%is_allowed)
      fid = 4
      call lib_driver%get_fptr (pid, fid, fptr)
      call c_f_procpointer (fptr, proc_driver%new_event)
      fid = 5
      call lib_driver%get_fptr (pid, fid, fptr)
      call c_f_procpointer (fptr, proc_driver%get_amplitude)
    end select
  end subroutine template_me_def_connect
```

16.3.2 The Template Matrix element writer

Unlike O'MEGA, the template matrix element is directly written by the main WHIZARD program, so there will be no entry in the makefile for calling an external program. The template matrix element writer is responsible for writing interfaces and wrappers.

⟨*Template matrix elements: types*⟩+≡

```
  type, extends (prc_writer_f_module_t) :: template_me_writer_t
     class(model_data_t), pointer :: model => null ()
     type(string_t) :: model_name
     logical :: unity
     type(string_t), dimension(:), allocatable :: prt_in
     type(string_t), dimension(:), allocatable :: prt_out
     integer :: n_in
     integer :: n_out
     integer :: n_tot
   contains
   ⟨Template matrix elements: template ME writer: TBP⟩
  end type template_me_writer_t
```

The reported type is the same as for the template_me_def_t type.

⟨*Template matrix elements: template ME writer: TBP*⟩≡

```
  procedure, nopass :: type_name => template_me_writer_type_name
```

⟨*Template matrix elements: procedures*⟩+≡

```
  function template_me_writer_type_name () result (string)
    type(string_t) :: string
    string = "template"
  end function template_me_writer_type_name
```

Taking into account the prefix for template ME module names.

⟨*Template matrix elements: template ME writer: TBP*⟩+≡

```
  procedure, nopass :: get_module_name => template_me_writer_get_module_name
```

⟨*Template matrix elements: procedures*⟩+≡

```
  function template_me_writer_get_module_name (id) result (name)
    type(string_t) :: name
    type(string_t), intent(in) :: id
    name = "tpr_" // id
  end function template_me_writer_get_module_name
```

Output. This is called by template_me_def_write.

⟨*Template matrix elements: template ME writer: TBP*⟩+≡
```
procedure :: write => template_me_writer_write
```

⟨*Template matrix elements: procedures*⟩+≡
```
subroutine template_me_writer_write (object, unit)
  class(template_me_writer_t), intent(in) :: object
  integer, intent(in) :: unit
  integer :: i, j
  write (unit, "(5x,A,I0)") "# incoming part. = ", object%n_in
  write (unit, "(7x,A)", advance="no") &
                          "   Initial state: "
  do i = 1, object%n_in - 1
     write (unit, "(1x,A)", advance="no") char (object%prt_in(i))
  end do
  write (unit, "(1x,A)") char (object%prt_in(object%n_in))
  write (unit, "(5x,A,I0)") "# outgoing part. = ", object%n_out
  write (unit, "(7x,A)", advance="no") &
                          "   Final state:   "
  do j = 1, object%n_out - 1
     write (unit, "(1x,A)", advance="no") char (object%prt_out(j))
  end do
  write (unit, "(1x,A)") char (object%prt_out(object%n_out))
  write (unit, "(5x,A,I0)") "# part. (total) = ", object%n_tot
end subroutine template_me_writer_write
```

Initialize with process data.

⟨*Template matrix elements: template ME writer: TBP*⟩+≡
```
procedure :: init => template_me_writer_init
```

⟨*Template matrix elements: procedures*⟩+≡
```
subroutine template_me_writer_init (writer, model, &
     prt_in, prt_out, unity)
  class(template_me_writer_t), intent(out) :: writer
  class(model_data_t), intent(in), target :: model
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
  logical, intent(in) :: unity
  writer%model => model
  writer%model_name = model%get_name ()
  writer%n_in = size (prt_in)
  writer%n_out = size (prt_out)
  writer%n_tot = size (prt_in) + size (prt_out)
  allocate (writer%prt_in (size (prt_in)), source = prt_in)
  allocate (writer%prt_out (size (prt_out)), source = prt_out)
  writer%unity = unity
end subroutine template_me_writer_init
```

The makefile is the driver file for the test matrix elements.

⟨*Template matrix elements: template ME writer: TBP*⟩+≡
```
procedure :: write_makefile_code => template_me_write_makefile_code
```

⟨*Template matrix elements: procedures*⟩+≡
```
subroutine template_me_write_makefile_code (writer, unit, id, os_data, testflag)
```

```
    class(template_me_writer_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id
    type(os_data_t), intent(in) :: os_data
    logical, intent(in), optional :: testflag
    write (unit, "(5A)")  "SOURCES += ", char (id), ".f90"
    write (unit, "(5A)")  "OBJECTS += ", char (id), ".lo"
    write (unit, "(5A)")  "clean-", char (id), ":"
    write (unit, "(5A)")  TAB, "rm -f tpr_", char (id), ".mod"
    write (unit, "(5A)")  TAB, "rm -f ", char (id), ".lo"
    write (unit, "(5A)")  "CLEAN_SOURCES += ", char (id), ".f90"
    write (unit, "(5A)")  "CLEAN_OBJECTS += tpr_", char (id), ".mod"
    write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), ".lo"
    write (unit, "(5A)")  char (id), ".lo: ", char (id), ".f90"
    write (unit, "(5A)")  TAB, "$(LTFCOMPILE) $<"
  end subroutine template_me_write_makefile_code
```

The source is written by this routine.

⟨*Template matrix elements: template ME writer: TBP*⟩+≡

```
  procedure :: write_source_code => template_me_write_source_code
```

⟨*Template matrix elements: procedures*⟩+≡

```
  subroutine template_me_write_source_code (writer, id)
    class(template_me_writer_t), intent(in) :: writer
    type(string_t), intent(in) :: id
    integer, dimension(writer%n_in) :: prt_in, mult_in, col_in
    type(flavor_t), dimension(1:writer%n_in) :: flv_in
    integer, dimension(writer%n_out) :: prt_out, mult_out
    integer, dimension(writer%n_tot) :: prt, mult
    integer, dimension(:,:), allocatable :: sxxx
    integer :: dummy, status
    type(flavor_t), dimension(1:writer%n_out) :: flv_out
    type(string_t) :: proc_str, comment_str
    integer :: u, i, j
    integer :: hel, hel_in, hel_out, fac, factor, col_fac
    type(string_t) :: filename
    comment_str = ""
    do i = 1, writer%n_in
       comment_str = comment_str // writer%prt_in(i) // " "
    end do
    do j = 1, writer%n_out
       comment_str = comment_str // writer%prt_out(j) // " "
    end do
    do i = 1, writer%n_in
       prt_in(i) = writer%model%get_pdg (writer%prt_in(i))
       call flv_in(i)%init (prt_in(i), writer%model)
       mult_in(i) = flv_in(i)%get_multiplicity ()
       col_in(i) = abs (flv_in(i)%get_color_type ())
       mult(i) = mult_in(i)
       end do
    do j = 1, writer%n_out
       prt_out(j) = writer%model%get_pdg (writer%prt_out(j))
       call flv_out(j)%init (prt_out(j), writer%model)
       mult_out(j) = flv_out(j)%get_multiplicity ()
```

```
   mult(writer%n_in + j) = mult_out(j)
   end do
prt(1:writer%n_in) = prt_in(1:writer%n_in)
prt(writer%n_in+1:writer%n_tot) = prt_out(1:writer%n_out)
proc_str = converter (prt)
hel_in = product (mult_in)
hel_out = product (mult_out)
col_fac = product (col_in)
hel = hel_in * hel_out
fac = hel
dummy = 1
factor = 1
if (writer%n_out >= 3) then
   do i = 3, writer%n_out
      factor = factor * (i - 2) * (i - 1)
   end do
end if
factor = factor * col_fac
allocate (sxxx(1:hel,1:writer%n_tot))
call create_spin_table (dummy,hel,fac,mult,sxxx)
call msg_message ("Writing test matrix element for process '" &
     // char (id) // "'")
filename = id // ".f90"
u = free_unit ()
open (unit=u, file=char(filename), action="write")
write (u, "(A)") "! File generated automatically by WHIZARD"
write (u, "(A)") "!                                         "
write (u, "(A)") "! Note that irresp. of what you demanded WHIZARD"
write (u, "(A)") "! treats this as colorless process       "
write (u, "(A)") "!                                         "
write (u, "(A)") "module tpr_" // char(id)
write (u, "(A)") "                                          "
write (u, "(A)") "  use kinds"
write (u, "(A)") "  use omega_color, OCF => omega_color_factor"
write (u, "(A)") "                                          "
write (u, "(A)") "  implicit none"
write (u, "(A)") "  private"
write (u, "(A)") "                                          "
write (u, "(A)") "  public :: md5sum"
write (u, "(A)") "  public :: number_particles_in, number_particles_out"
write (u, "(A)") "  public :: number_spin_states, spin_states"
write (u, "(A)") "  public :: number_flavor_states, flavor_states"
write (u, "(A)") "  public :: number_color_flows, color_flows"
write (u, "(A)") "  public :: number_color_indices, number_color_factors, &"
write (u, "(A)") "     color_factors, color_sum, openmp_supported"
write (u, "(A)") "  public :: init, final, update_alpha_s"
write (u, "(A)") "                                          "
write (u, "(A)") "  public :: new_event, is_allowed, get_amplitude"
write (u, "(A)") "       "
write (u, "(A)") "  real(default), parameter :: &"
write (u, "(A)") "       & conv = 0.38937966e12_default"
write (u, "(A)") "       "
write (u, "(A)") "  real(default), parameter :: &"
write (u, "(A)") "       & pi = 3.14159265358979323846264338327950288419 72_default"
```

```
write (u, "(A)") "       "
write (u, "(A)") "  real(default), parameter :: &"
if (writer%unity) then
   write (u, "(A)") "                   & const = 1"
else
   write (u, "(A,1x,I0,A)") "       & const = (16 * pi / conv) * " &
      // "(16 * pi**2)**(", writer%n_out, "-2) "
end if
write (u, "(A)") "       "
write (u, "(A,1x,I0)") "  integer, parameter, private :: n_prt =  ", &
   writer%n_tot
write (u, "(A,1x,I0)") "  integer, parameter, private :: n_in = ", &
   writer%n_in
write (u, "(A,1x,I0)") "  integer, parameter, private :: n_out = ", &
   writer%n_out
write (u, "(A)") "  integer, parameter, private :: n_cflow = 1"
write (u, "(A)") "  integer, parameter, private :: n_cindex = 2"
write (u, "(A)") "  !!! We ignore tensor products and take only one flavor state."
write (u, "(A)") "  integer, parameter, private :: n_flv = 1"
write (u, "(A,1x,I0)") "  integer, parameter, private :: n_hel = ", hel
write (u, "(A)") "                                            "
write (u, "(A)") "  logical, parameter, private :: T = .true."
write (u, "(A)") "  logical, parameter, private :: F = .false."
write (u, "(A)") "                                            "
do i = 1, hel
   write (u, "(A)") "  integer, dimension(n_prt), parameter, private :: &"
   write (u, "(A)") "    " // s_conv(i) // " = [ " // &
        char(converter(sxxx(i,1:writer%n_tot))) // " ]"
end do
write (u, "(A)") "  integer, dimension(n_prt,n_hel), parameter, private :: table_spin_states =
write (u, "(A)") "    reshape ( [ & "
do i = 1, hel-1
   write (u, "(A)") "                 " // s_conv(i) // ", & "
end do
write (u, "(A)") "                 " // s_conv(hel) // " & "
write (u, "(A)") "              ], [ n_prt, n_hel ] )"
write (u, "(A)") "                                                    "
write (u, "(A)") "  integer, dimension(n_prt), parameter, private :: &"
write (u, "(A)") "    f0001 = [ " // char(proc_str) // " ]   !  " // char(comment_str)
write (u, "(A)") "  integer, dimension(n_prt,n_flv), parameter, private :: table_flavor_states
write (u, "(A)") "    reshape ( [ f0001 ], [ n_prt, n_flv ] )"
write (u, "(A)") "                                                    "
write (u, "(A)") "  integer, dimension(n_cindex, n_prt), parameter, private :: &"
write (u, "(A)") "    c0001 = reshape ( [ " // char (dummy_colorizer (flv_in)) // " " // &
  (repeat ("0,0, ", writer%n_out-1)) // "0,0 ], " // " [ n_cindex, n_prt ] )"
write (u, "(A)") "  integer, dimension(n_cindex, n_prt, n_cflow), parameter, private :: &"
write (u, "(A)") "  table_color_flows = reshape ( [ c0001 ], [ n_cindex, n_prt, n_cflow ] )"
write (u, "(A)") "                                            "
write (u, "(A)") "  logical, dimension(n_prt), parameter, private :: & "
write (u, "(A)") "    g0001 = [ "  // (repeat ("F, ", writer%n_tot-1)) // "F ] "
write (u, "(A)") "  logical, dimension(n_prt, n_cflow), parameter, private " &
     // ":: table_ghost_flags = &"
write (u, "(A)") "    reshape ( [ g0001 ], [ n_prt, n_cflow ] )"
write (u, "(A)") "                                            "
```

```
write (u, "(A)") "  integer, parameter, private :: n_cfactors = 1"
write (u, "(A)") "  type(OCF), dimension(n_cfactors), parameter, private :: &"
write (u, "(A)") "    table_color_factors = [  OCF(1,1,+1._default) ]"
write (u, "(A)") "                                                   "
write (u, "(A)") "  logical, dimension(n_flv), parameter, private :: a0001 = [ T ]"
write (u, "(A)") "  logical, dimension(n_flv, n_cflow), parameter, private :: &"
write (u, "(A)") "    flv_col_is_allowed = reshape ( [ a0001 ], [ n_flv, n_cflow ] )"
write (u, "(A)") "                                                   "
write (u, "(A)") "  complex(default), dimension (n_flv, n_hel, n_cflow), private, save :: amp"
write (u, "(A)") "                                                   "
write (u, "(A)") "  logical, dimension(n_hel), private, save :: hel_is_allowed = T"
write (u, "(A)") "                                                   "
write (u, "(A)") "contains"
write (u, "(A)") "                                                   "
write (u, "(A)") "  pure function md5sum ()"
write (u, "(A)") "    character(len=32) :: md5sum"
write (u, "(A)") "    ! DON'T EVEN THINK of modifying the following line!"
write (u, "(A)") "    md5sum = """ // writer%md5sum // """"
write (u, "(A)") "  end function md5sum"
write (u, "(A)") "                                                   "
write (u, "(A)") "  subroutine init (par, scheme)"
write (u, "(A)") "    real(default), dimension(*), intent(in) :: par"
write (u, "(A)") "    integer, intent(in) :: scheme"
write (u, "(A)") "  end subroutine init"
write (u, "(A)") "                                                   "
write (u, "(A)") "  subroutine final ()"
write (u, "(A)") "  end subroutine final"
write (u, "(A)") "                                                   "
write (u, "(A)") "  subroutine update_alpha_s (alpha_s)"
write (u, "(A)") "    real(default), intent(in) :: alpha_s"
write (u, "(A)") "  end subroutine update_alpha_s"
write (u, "(A)") "                                                   "
write (u, "(A)") "  pure function number_particles_in () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = n_in"
write (u, "(A)") "  end function number_particles_in"
write (u, "(A)") "                                                   "
write (u, "(A)") "  pure function number_particles_out () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = n_out"
write (u, "(A)") "  end function number_particles_out"
write (u, "(A)") "                                                   "
write (u, "(A)") "  pure function number_spin_states () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = size (table_spin_states, dim=2)"
write (u, "(A)") "  end function number_spin_states"
write (u, "(A)") "                                                   "
write (u, "(A)") "  pure subroutine spin_states (a)"
write (u, "(A)") "    integer, dimension(:,:), intent(out) :: a"
write (u, "(A)") "    a = table_spin_states"
write (u, "(A)") "  end subroutine spin_states"
write (u, "(A)") "                                                   "
write (u, "(A)") "  pure function number_flavor_states () result (n)"
write (u, "(A)") "    integer :: n"
```

```
write (u, "(A)") "    n = 1"
write (u, "(A)") "  end function number_flavor_states"
write (u, "(A)") "                                          "
write (u, "(A)") "  pure subroutine flavor_states (a)"
write (u, "(A)") "    integer, dimension(:,:), intent(out) :: a"
write (u, "(A)") "    a = table_flavor_states"
write (u, "(A)") "  end subroutine flavor_states"
write (u, "(A)") "                                          "
write (u, "(A)") "  pure function number_color_indices () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = size(table_color_flows, dim=1)"
write (u, "(A)") "  end function number_color_indices"
write (u, "(A)") "                                          "
write (u, "(A)") "  pure subroutine color_factors (cf)"
write (u, "(A)") "    type(OCF), dimension(:), intent(out) :: cf"
write (u, "(A)") "    cf = table_color_factors"
write (u, "(A)") "  end subroutine color_factors"
write (u, "(A)") "                                          "
!pure unless OpenMP
!write (u, "(A)") "  pure function color_sum (flv, hel) result (amp2)"
write (u, "(A)") "  function color_sum (flv, hel) result (amp2)"
write (u, "(A)") "    integer, intent(in) :: flv, hel"
write (u, "(A)") "    real(kind=default) :: amp2"
write (u, "(A)") "    amp2 = real (omega_color_sum (flv, hel, amp, table_color_factors))"
write (u, "(A)") "  end function color_sum"
write (u, "(A)") "                                          "
write (u, "(A)") "  pure function number_color_flows () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = size (table_color_flows, dim=3)"
write (u, "(A)") "  end function number_color_flows"
write (u, "(A)") "                                          "
write (u, "(A)") "  pure subroutine color_flows (a, g)"
write (u, "(A)") "    integer, dimension(:,:,:), intent(out) :: a"
write (u, "(A)") "    logical, dimension(:,:), intent(out) :: g"
write (u, "(A)") "    a = table_color_flows"
write (u, "(A)") "    g = table_ghost_flags"
write (u, "(A)") "  end subroutine color_flows"
write (u, "(A)") "                                          "
write (u, "(A)") "  pure function number_color_factors () result (n)"
write (u, "(A)") "    integer :: n"
write (u, "(A)") "    n = size (table_color_factors)"
write (u, "(A)") "  end function number_color_factors"
write (u, "(A)") "                                          "
write (u, "(A)") "  pure function openmp_supported () result (status)"
write (u, "(A)") "    logical :: status"
write (u, "(A)") "    status = .false."
write (u, "(A)") "  end function openmp_supported"
write (u, "(A)") "                                          "
write (u, "(A)") "  subroutine new_event (p)"
write (u, "(A)") "    real(default), dimension(0:3,*), intent(in) :: p"
write (u, "(A)") "    call calculate_amplitudes (amp, p)"
write (u, "(A)") "  end subroutine new_event"
write (u, "(A)") "                                          "
write (u, "(A)") "  pure function is_allowed (flv, hel, col) result (yorn)"
```

```
  write (u, "(A)") "    logical :: yorn"
  write (u, "(A)") "    integer, intent(in) :: flv, hel, col"
  write (u, "(A)") "    yorn = hel_is_allowed(hel) .and. flv_col_is_allowed(flv,col)"
  write (u, "(A)") "  end function is_allowed"
  write (u, "(A)") "                                          "
  write (u, "(A)") "  pure function get_amplitude (flv, hel, col) result (amp_result)"
  write (u, "(A)") "    complex(default) :: amp_result"
  write (u, "(A)") "    integer, intent(in) :: flv, hel, col"
  write (u, "(A)") "    amp_result = amp (flv, hel, col)"
  write (u, "(A)") "  end function get_amplitude"
  write (u, "(A)") "                                          "
  write (u, "(A)") "  pure subroutine calculate_amplitudes (amp, k)"
  write (u, "(A)") "    complex(default), dimension(:,:,:), intent(out) :: amp"
  write (u, "(A)") "    real(default), dimension(0:3,*), intent(in) :: k"
  write (u, "(A)") "    real(default) :: fac"
  write (u, "(A)") "    integer :: i"
  write (u, "(A)") "    ! We give all helicities the same weight!"
  if (writer%unity) then
     write (u, "(A,1x,I0,1x,A)") "    fac = ", col_fac
     write (u, "(A)") "    amp = const * sqrt(fac)"
  else
     write (u, "(A,1x,I0,1x,A)") "    fac = ", factor
     write (u, "(A)") "    amp = sqrt((2 * (k(0,1)*k(0,2) &"
     write (u, "(A,1x,I0,A)") "         - dot_product (k(1:,1), k(1:,2)))) ** (3-", &
                              writer%n_out, ")) * sqrt(const * fac)"
  end if
  write (u, "(A,1x,I0,A)") "    amp = amp / sqrt(", hel_out, "._default)"
  write (u, "(A)") "  end subroutine calculate_amplitudes"
  write (u, "(A)") "                                          "
  write (u, "(A)") "end module tpr_" // char(id)
  close (u, iostat=status)
  deallocate (sxxx)
contains
  function s_conv (num) result (chrt)
    integer, intent(in) :: num
    character(len=10) :: chrt
    write (chrt, "(I10)") num
    chrt = trim(adjustl(chrt))
    if (num < 10) then
       chrt = "s000" // chrt
    else if (num < 100) then
       chrt = "s00" // chrt
    else if (num < 1000) then
       chrt = "s0" // chrt
    else
       chrt = "s" // chrt
    end if
  end function s_conv
  function converter (flv) result (str)
    integer, dimension(:), intent(in) :: flv
    type(string_t) :: str
    character(len=150), dimension(size(flv)) :: chrt
    integer :: i
    str = ""
```

```
  do i = 1, size(flv) - 1
     write (chrt(i), "(I10)") flv(i)
     str = str // var_str(trim(adjustl(chrt(i)))) // ", "
  end do
  write (chrt(size(flv)), "(I10)") flv(size(flv))
  str = str // trim(adjustl(chrt(size(flv))))
end function converter
integer function sj (j,m)
  integer, intent(in) :: j, m
  if (((j == 1) .and. (m == 1)) .or. &
      ((j == 2) .and. (m == 2)) .or. &
      ((j == 3) .and. (m == 3)) .or. &
      ((j == 4) .and. (m == 3)) .or. &
      ((j == 5) .and. (m == 4))) then
     sj = 1
  else if (((j == 2) .and. (m == 1)) .or. &
      ((j == 3) .and. (m == 1)) .or. &
      ((j == 4) .and. (m == 2)) .or. &
      ((j == 5) .and. (m == 2))) then
     sj = -1
  else if (((j == 3) .and. (m == 2)) .or. &
      ((j == 5) .and. (m == 3))) then
     sj = 0
  else if (((j == 4) .and. (m == 1)) .or. &
      ((j == 5) .and. (m == 1))) then
     sj = -2
  else if (((j == 4) .and. (m == 4)) .or. &
      ((j == 5) .and. (m == 5))) then
     sj = 2
  else
     call msg_fatal ("template_me_write_source_code: Wrong spin type")
  end if
end function sj
recursive subroutine create_spin_table (index, nhel, fac, mult, inta)
  integer, intent(inout) :: index, fac
  integer, intent(in) :: nhel
  integer, dimension(:), intent(in) :: mult
  integer, dimension(nhel,size(mult)), intent(out) :: inta
  integer :: j
  if (index > size(mult)) return
  fac = fac / mult(index)
  do j = 1, nhel
     inta(j,index) = sj (mult(index),mod(((j-1)/fac),mult(index))+1)
  end do
  index = index + 1
  call create_spin_table (index, nhel, fac, mult, inta)
end subroutine create_spin_table
function dummy_colorizer (flv) result (str)
  type(flavor_t), dimension(:), intent(in) :: flv
  type(string_t) :: str
  integer :: i, k
  str = ""
  k = 0
  do i = 1, size(flv)
```

```
        k = k + 1
        select case (flv(i)%get_color_type ())
        case (1,-1)
           str = str // "0,0, "
        case (3)
           str = str // int2string(k) // ",0, "
        case (-3)
           str = str // "0," // int2string(-k) // ", "
        case (8)
           str = str // int2string(k) // "," // int2string(-k-1) // ", "
           k = k + 1
        case default
           call msg_error ("Color type not supported.")
        end select
     end do
     str = adjustl(trim(str))
   end function dummy_colorizer
end subroutine template_me_write_source_code
```

Return the name of a procedure that implements a given feature, as it is provided by the template matrix-element code. Template ME names are chosen completely in analogy to the O'MEGA matrix element conventions.

⟨*Template matrix elements: template ME writer: TBP*⟩+≡

```
procedure, nopass :: get_procname => template_me_writer_get_procname
```

⟨*Template matrix elements: procedures*⟩+≡

```
function template_me_writer_get_procname (feature) result (name)
  type(string_t) :: name
  type(string_t), intent(in) :: feature
  select case (char (feature))
  case ("n_in");   name = "number_particles_in"
  case ("n_out");  name = "number_particles_out"
  case ("n_flv");  name = "number_flavor_states"
  case ("n_hel");  name = "number_spin_states"
  case ("n_col");  name = "number_color_flows"
  case ("n_cin");  name = "number_color_indices"
  case ("n_cf");   name = "number_color_factors"
  case ("flv_state");  name = "flavor_states"
  case ("hel_state");  name = "spin_states"
  case ("col_state");  name = "color_flows"
  case default
     name = feature
  end select
end function template_me_writer_get_procname
```

The interfaces for the template-specific features.

⟨*Template matrix elements: template ME writer: TBP*⟩+≡

```
procedure :: write_interface => template_me_write_interface
```

⟨*Template matrix elements: procedures*⟩+≡

```
subroutine template_me_write_interface (writer, unit, id, feature)
  class(template_me_writer_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
```

```
    type(string_t), intent(in) :: feature
    type(string_t) :: name
    name = writer%get_c_procname (id, feature)
    write (unit, "(2x,9A)")  "interface"
    select case (char (feature))
    case ("init")
       write (unit, "(5x,9A)")  "subroutine ", char (name), &
            " (par, scheme) bind(C)"
       write (unit, "(7x,9A)")  "import"
       write (unit, "(7x,9A)")  "real(c_default_float), dimension(*), &
            &intent(in) :: par"
       write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: scheme"
       write (unit, "(5x,9A)")  "end subroutine ", char (name)
    case ("update_alpha_s")
       write (unit, "(5x,9A)")  "subroutine ", char (name), " (alpha_s) bind(C)"
       write (unit, "(7x,9A)")  "import"
       write (unit, "(7x,9A)")  "real(c_default_float), intent(in) :: alpha_s"
       write (unit, "(5x,9A)")  "end subroutine ", char (name)
    case ("is_allowed")
       write (unit, "(5x,9A)")  "subroutine ", char (name), " &
            &(flv, hel, col, flag) bind(C)"
       write (unit, "(7x,9A)")  "import"
       write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
       write (unit, "(7x,9A)")  "logical(c_bool), intent(out) :: flag"
       write (unit, "(5x,9A)")  "end subroutine ", char (name)
    case ("new_event")
       write (unit, "(5x,9A)")  "subroutine ", char (name), " (p) bind(C)"
       write (unit, "(7x,9A)")  "import"
       write (unit, "(7x,9A)")  "real(c_default_float), dimension(0:3,*), &
            &intent(in) :: p"
       write (unit, "(5x,9A)")  "end subroutine ", char (name)
    case ("get_amplitude")
       write (unit, "(5x,9A)")  "subroutine ", char (name), " &
            &(flv, hel, col, amp) bind(C)"
       write (unit, "(7x,9A)")  "import"
       write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
       write (unit, "(7x,9A)")  "complex(c_default_complex), intent(out) &
            &:: amp"
       write (unit, "(5x,9A)")  "end subroutine ", char (name)
    end select
    write (unit, "(2x,9A)")  "end interface"
  end subroutine template_me_write_interface
```

The wrappers have to take into account conversion between C and Fortran data types.

NOTE: The case `c_default_float` $\neq$ `default` is not yet covered.

⟨*Template matrix elements: template ME writer: TBP*⟩+≡

```
  procedure :: write_wrapper => template_me_write_wrapper
```

⟨*Template matrix elements: procedures*⟩+≡

```
  subroutine template_me_write_wrapper (writer, unit, id, feature)
    class(template_me_writer_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id, feature
```

```
type(string_t) :: name
name = writer%get_c_procname (id, feature)
write (unit, *)
select case (char (feature))
case ("init")
   write (unit, "(9A)")  "subroutine ", char (name), &
        " (par, scheme) bind(C)"
   write (unit, "(2x,9A)")  "use iso_c_binding"
   write (unit, "(2x,9A)")  "use kinds"
   write (unit, "(2x,9A)")  "use tpr_", char (id)
   write (unit, "(2x,9A)")  "real(c_default_float), dimension(*), &
        &intent(in) :: par"
   write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: scheme"
   if (c_default_float == default .and. c_int == kind(1)) then
      write (unit, "(2x,9A)")  "call ", char (feature), " (par, scheme)"
   end if
   write (unit, "(9A)")  "end subroutine ", char (name)
case ("update_alpha_s")
   write (unit, "(9A)")  "subroutine ", char (name), " (alpha_s) bind(C)"
   write (unit, "(2x,9A)")  "use iso_c_binding"
   write (unit, "(2x,9A)")  "use kinds"
   write (unit, "(2x,9A)")  "use tpr_", char (id)
   if (c_default_float == default) then
      write (unit, "(2x,9A)")  "real(c_default_float), intent(in) &
           &:: alpha_s"
      write (unit, "(2x,9A)")  "call ", char (feature), " (alpha_s)"
   end if
   write (unit, "(9A)")  "end subroutine ", char (name)
case ("is_allowed")
   write (unit, "(9A)")  "subroutine ", char (name), &
        " (flv, hel, col, flag) bind(C)"
   write (unit, "(2x,9A)")  "use iso_c_binding"
   write (unit, "(2x,9A)")  "use kinds"
   write (unit, "(2x,9A)")  "use tpr_", char (id)
   write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
   write (unit, "(2x,9A)")  "logical(c_bool), intent(out) :: flag"
   write (unit, "(2x,9A)")  "flag = ", char (feature), &
        " (int (flv), int (hel), int (col))"
   write (unit, "(9A)")  "end subroutine ", char (name)
case ("new_event")
   write (unit, "(9A)")  "subroutine ", char (name), " (p) bind(C)"
   write (unit, "(2x,9A)")  "use iso_c_binding"
   write (unit, "(2x,9A)")  "use kinds"
   write (unit, "(2x,9A)")  "use tpr_", char (id)
   if (c_default_float == default) then
      write (unit, "(2x,9A)")  "real(c_default_float), dimension(0:3,*), &
           &intent(in) :: p"
      write (unit, "(2x,9A)")  "call ", char (feature), " (p)"
   end if
   write (unit, "(9A)")  "end subroutine ", char (name)
case ("get_amplitude")
   write (unit, "(9A)")  "subroutine ", char (name), &
        " (flv, hel, col, amp) bind(C)"
   write (unit, "(2x,9A)")  "use iso_c_binding"
```

```
      write (unit, "(2x,9A)")  "use kinds"
      write (unit, "(2x,9A)")  "use tpr_", char (id)
      write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
      write (unit, "(2x,9A)")  "complex(c_default_complex), intent(out) &
           &:: amp"
      write (unit, "(2x,9A)")  "amp = ", char (feature), &
           " (int (flv), int (hel), int (col))"
      write (unit, "(9A)")  "end subroutine ", char (name)
    end select
  end subroutine template_me_write_wrapper
```

16.3.3 Driver

⟨*Template matrix elements: public*⟩+≡

```
  public :: template_me_driver_t
```

⟨*Template matrix elements: types*⟩+≡

```
  type, extends (prc_core_driver_t) :: template_me_driver_t
     procedure(init_t), nopass, pointer :: &
          init => null ()
     procedure(update_alpha_s_t), nopass, pointer :: &
          update_alpha_s => null ()
     procedure(is_allowed_t), nopass, pointer :: &
          is_allowed => null ()
     procedure(new_event_t), nopass, pointer :: &
          new_event => null ()
     procedure(get_amplitude_t), nopass, pointer :: &
          get_amplitude => null ()
   contains
   ⟨Template matrix elements: template ME driver: TBP⟩
  end type template_me_driver_t
```

The reported type is the same as for the `template_me_def_t` type.

⟨*Template matrix elements: template ME driver: TBP*⟩≡

```
  procedure, nopass :: type_name => template_me_driver_type_name
```

⟨*Template matrix elements: procedures*⟩+≡

```
  function template_me_driver_type_name () result (string)
    type(string_t) :: string
    string = "template"
  end function template_me_driver_type_name
```

16.3.4 High-level process definition

This procedure wraps the details filling a process-component definition entry as appropriate for an template matrix element.

NOTE: For calling the `import_component` method, we must explicitly address the `process_def_t` parent object. The natural way to call the method of the extended type triggers a bug in gfortran 4.6. The string array arguments `prt_in` and `prt_out` become corrupted and cause a segfault.

⟨*Template matrix elements: public*⟩+≡

```
public :: template_me_make_process_component
```

⟨*Template matrix elements: procedures*⟩+≡

```
subroutine template_me_make_process_component (entry, component_index, &
      model, model_name, prt_in, prt_out, unity)
  class(process_def_entry_t), intent(inout) :: entry
  integer, intent(in) :: component_index
  type(string_t), intent(in) :: model_name
  class(model_data_t), intent(in), target :: model
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
  logical, intent(in) :: unity
  class(prc_core_def_t), allocatable :: def
  allocate (template_me_def_t :: def)
  select type (def)
  type is (template_me_def_t)
     call def%init (model, prt_in, prt_out, unity)
  end select
  call entry%process_def_t%import_component (component_index, &
       n_out = size (prt_out), &
       prt_in  = new_prt_spec (prt_in), &
       prt_out = new_prt_spec (prt_out), &
       method = var_str ("template"), &
       variant = def)
end subroutine template_me_make_process_component
```

16.3.5 The prc_template_me_t wrapper

This is an instance of the generic prc_core_t object. It contains a pointer to the process definition (template_me_def_t), a data component (process_constants_t), and the matrix-element driver (template_me_driver_t).

⟨*Template matrix elements: public*⟩+≡

```
public :: prc_template_me_t
```

⟨*Template matrix elements: types*⟩+≡

```
type, extends (prc_core_t) :: prc_template_me_t
   real(default), dimension(:), allocatable :: par
   integer :: scheme = 0
 contains
 ⟨Template matrix elements: prc template ME: TBP⟩
end type prc_template_me_t
```

The workspace associated to a prc_template_me_t object contains a single flag. The flag is used to suppress re-evaluating the matrix element for each quantum-number combination, after the first amplitude belonging to a given kinematics has been computed.

We can also store the value of a running coupling once it has been calculated for an event. The default value is negative, which indicates an undefined value in this context.

⟨*Template matrix elements: types*⟩+≡

```
type, extends (prc_core_state_t) :: template_me_state_t
   logical :: new_kinematics = .true.
```

```
    real(default) :: alpha_qcd = -1
  contains
    procedure :: write => template_me_state_write
    procedure :: reset_new_kinematics => template_me_state_reset_new_kinematics
end type template_me_state_t
```

⟨*Template matrix elements: procedures*⟩+≡

```
subroutine template_me_state_write (object, unit)
  class(template_me_state_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(3x,A,L1)")  "Template ME state: new kinematics = ", &
       object%new_kinematics
end subroutine template_me_state_write
```

⟨*Template matrix elements: procedures*⟩+≡

```
subroutine template_me_state_reset_new_kinematics (object)
  class(template_me_state_t), intent(inout) :: object
end subroutine template_me_state_reset_new_kinematics
```

Allocate the workspace with the above specific type.

⟨*Template matrix elements: prc template ME: TBP*⟩≡

```
procedure :: allocate_workspace => prc_template_me_allocate_workspace
```

⟨*Template matrix elements: procedures*⟩+≡

```
subroutine prc_template_me_allocate_workspace (object, core_state)
  class(prc_template_me_t), intent(in) :: object
  class(prc_core_state_t), intent(inout), allocatable :: core_state
  allocate (template_me_state_t :: core_state)
end subroutine prc_template_me_allocate_workspace
```

The following procedures are inherited from the base type as deferred, thus must be implemented. The corresponding unit tests are skipped here; the procedures are tested when called from the `processes` module.

Output: print just the ID of the associated matrix element. Then display any stored parameters.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡

```
procedure :: write => prc_template_me_write
```

⟨*Template matrix elements: procedures*⟩+≡

```
subroutine prc_template_me_write (object, unit)
  class(prc_template_me_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(3x,A)", advance="no")  "Template process core:"
  if (object%data_known) then
     write (u, "(1x,A)")  char (object%data%id)
  else
     write (u, "(1x,A)")  "[undefined]"
  end if
```

```
    if (allocated (object%par)) then
       write (u, "(3x,A)")  "Parameter array:"
       do i = 1, size (object%par)
          write (u, "(5x,I0,1x,ES17.10)")  i, object%par(i)
       end do
    end if
  end subroutine prc_template_me_write
```

⟨*Template matrix elements: prc template ME: TBP*⟩+≡

```
  procedure :: write_name => prc_template_me_write_name
```

⟨*Template matrix elements: procedures*⟩+≡

```
  subroutine prc_template_me_write_name (object, unit)
    class(prc_template_me_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u,"(1x,A)") "Core: template"
  end subroutine prc_template_me_write_name
```

Temporarily store the parameter array inside the `prc_template_me` object, so we can use it later during the actual initialization.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡

```
  procedure :: set_parameters => prc_template_me_set_parameters
```

⟨*Template matrix elements: procedures*⟩+≡

```
  subroutine prc_template_me_set_parameters (prc_template_me, model)
    class(prc_template_me_t), intent(inout) :: prc_template_me
    class(model_data_t), intent(in), target, optional :: model
    if (present (model)) then
       if (.not. allocated (prc_template_me%par)) &
            allocate (prc_template_me%par (model%get_n_real ()))
       call model%real_parameters_to_array (prc_template_me%par)
       prc_template_me%scheme = model%get_scheme_num ()
    end if
  end subroutine prc_template_me_set_parameters
```

To fully initialize the process core, we perform base initialization, then initialize the external matrix element code.

This procedure overrides the `init` method of the base type, which we nevertheless can access via its binding `base_init`. When done, we have an allocated driver. The driver will call the `init` procedure for the external matrix element, and thus transfer the parameter set to where it finally belongs.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡

```
  procedure :: init => prc_template_me_init
```

⟨*Template matrix elements: procedures*⟩+≡

```
  subroutine prc_template_me_init (object, def, lib, id, i_component)
    class(prc_template_me_t), intent(inout) :: object
    class(prc_core_def_t), intent(in), target :: def
    type(process_library_t), intent(in), target :: lib
    type(string_t), intent(in) :: id
    integer, intent(in) :: i_component
```

```
    call object%base_init (def, lib, id, i_component)
    call object%activate_parameters ()
  end subroutine prc_template_me_init
```

Activate the stored parameters by transferring them to the external matrix element.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡

```
  procedure :: activate_parameters => prc_template_me_activate_parameters
```

⟨*Template matrix elements: procedures*⟩+≡

```
  subroutine prc_template_me_activate_parameters (object)
    class (prc_template_me_t), intent(inout) :: object
    if (allocated (object%driver)) then
       if (allocated (object%par)) then
          select type (driver => object%driver)
          type is (template_me_driver_t)
             if (associated (driver%init)) then
                call driver%init (object%par, object%scheme)
             end if
          end select
       else
          call msg_bug ("prc_template_me_activate: parameter set is not allocated")
       end if
    else
       call msg_bug ("prc_template_me_activate: driver is not allocated")
    end if
  end subroutine prc_template_me_activate_parameters
```

Tell whether a particular combination of flavor, helicity, color is allowed. Here we have to consult the matrix-element driver.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡

```
  procedure :: is_allowed => prc_template_me_is_allowed
```

⟨*Template matrix elements: procedures*⟩+≡

```
  function prc_template_me_is_allowed (object, i_term, f, h, c) result (flag)
    class(prc_template_me_t), intent(in) :: object
    integer, intent(in) :: i_term, f, h, c
    logical :: flag
    logical(c_bool) :: cflag
    select type (driver => object%driver)
    type is (template_me_driver_t)
       call driver%is_allowed (f, h, c, cflag)
       flag = cflag
    end select
  end function prc_template_me_is_allowed
```

Transfer the generated momenta directly to the hard interaction in the (only) term. We assume that everything has been set up correctly, so the array fits.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡

```
  procedure :: compute_hard_kinematics => &
       prc_template_me_compute_hard_kinematics
```

⟨*Template matrix elements: procedures*⟩+≡
```
subroutine prc_template_me_compute_hard_kinematics &
     (object, p_seed, i_term, int_hard, core_state)
  class(prc_template_me_t), intent(in) :: object
  type(vector4_t), dimension(:), intent(in) :: p_seed
  integer, intent(in) :: i_term
  type(interaction_t), intent(inout) :: int_hard
  class(prc_core_state_t), intent(inout), allocatable :: core_state
  call int_hard%set_momenta (p_seed)
end subroutine prc_template_me_compute_hard_kinematics
```

This procedure is not called for `prc_template_me_t`, just a placeholder.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡
```
procedure :: compute_eff_kinematics => &
     prc_template_me_compute_eff_kinematics
```

⟨*Template matrix elements: procedures*⟩+≡
```
subroutine prc_template_me_compute_eff_kinematics &
     (object, i_term, int_hard, int_eff, core_state)
  class(prc_template_me_t), intent(in) :: object
  integer, intent(in) :: i_term
  type(interaction_t), intent(in) :: int_hard
  type(interaction_t), intent(inout) :: int_eff
  class(prc_core_state_t), intent(inout), allocatable :: core_state
end subroutine prc_template_me_compute_eff_kinematics
```

Compute the amplitude. For the tree-level process, we can ignore the scale settings. The term index `j` is also irrelevant.

We first call `new_event` for the given momenta (which we must unpack), then retrieve the amplitude value for the given quantum numbers.

If the `core_state` status flag is present, we can make sure that we call `new_event` only once for a given kinematics. After the first call, we unset the `new_kinematics` flag.

⟨*Template matrix elements: prc template ME: TBP*⟩+≡
```
procedure :: compute_amplitude => prc_template_me_compute_amplitude
```

⟨*Template matrix elements: procedures*⟩+≡
```
function prc_template_me_compute_amplitude &
     (object, j, p, f, h, c, fac_scale, ren_scale, alpha_qcd_forced, &
     core_state)  result (amp)
  class(prc_template_me_t), intent(in) :: object
  integer, intent(in) :: j
  type(vector4_t), dimension(:), intent(in) :: p
  integer, intent(in) :: f, h, c
  real(default), intent(in) :: fac_scale, ren_scale
  real(default), intent(in), allocatable :: alpha_qcd_forced
  class(prc_core_state_t), intent(inout), allocatable, optional :: core_state
  complex(default) :: amp
  integer :: n_tot, i
  real(c_default_float), dimension(:,:), allocatable :: parray
  complex(c_default_complex) :: camp
  logical :: new_event
  select type (driver => object%driver)
```

```
    type is (template_me_driver_t)
       new_event = .true.
       if (present (core_state)) then
          if (allocated (core_state)) then
             select type (core_state)
             type is (template_me_state_t)
                new_event = core_state%new_kinematics
                core_state%new_kinematics = .false.
             end select
          end if
       end if
       if (new_event) then
          n_tot = object%data%n_in + object%data%n_out
          allocate (parray (0:3, n_tot))
          forall (i = 1:n_tot)  parray(:,i) = vector4_get_components (p(i))
          call driver%new_event (parray)
       end if
       if (object%is_allowed (1, f, h, c)) then
          call driver%get_amplitude &
               (int (f, c_int), int (h, c_int), int (c, c_int), camp)
          amp = camp
       else
          amp = 0
       end if
    end select
  end function prc_template_me_compute_amplitude
```

We do not overwrite the `prc_core_t` routine for α_s.

16.3.6 Unit Test

Test module, followed by the corresponding implementation module.

⟨`prc_template_me_ut.f90`⟩≡
  ⟨*File header*⟩

```
module prc_template_me_ut
  use unit_tests
  use prc_template_me_uti
```

⟨*Standard module head*⟩

⟨*Template matrix elements: public test*⟩

```
contains
```

⟨*Template matrix elements: test driver*⟩

```
end module prc_template_me_ut
```

⟨`prc_template_me_uti.f90`⟩≡
  ⟨*File header*⟩

```
module prc_template_me_uti
```

```
  use, intrinsic :: iso_c_binding !NODEP!

  use kinds
⟨Use strings⟩
  use os_interface
  use particle_specifiers, only: new_prt_spec
  use model_data
  use prc_core_def
  use process_constants
  use process_libraries
  use model_testbed, only: prepare_model, cleanup_model

  use prc_template_me

⟨Standard module head⟩

⟨Template matrix elements: test declarations⟩

contains

⟨Template matrix elements: tests⟩

end module prc_template_me_uti
```

API: driver for the unit tests below.

⟨*Template matrix elements: public test*⟩≡

```
  public :: prc_template_me_test
```

⟨*Template matrix elements: test driver*⟩≡

```
  subroutine prc_template_me_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨Template matrix elements: execute tests⟩
end subroutine prc_template_me_test
```

Generate, compile and load a simple process matrix element

The process is $e^+e^- \to \mu^+\mu^-$ for vanishing masses and $e = 0.3$. We initialize the process, build the library, and compute a particular matrix element for momenta of unit energy and right-angle scattering. The matrix element, as it happens, is equal to e^2. (Note that are no conversion factors applied, so this result is exact.)

For `GNU make`, `makeflags` is set to `-j1`. This eliminates a potential clash with a `-j<n>` flag if this test is called from a parallel make.

⟨*Template matrix elements: execute tests*⟩≡

```
  call test (prc_template_me_1, "prc_template_me_1", &
       "build and load simple template process", &
       u, results)
```

⟨*Template matrix elements: test declarations*⟩≡

```
  public :: prc_template_me_1
```

⟨*Template matrix elements: tests*⟩≡

```
subroutine prc_template_me_1 (u)
  integer, intent(in) :: u
  type(process_library_t) :: lib
  class(prc_core_def_t), allocatable :: def
  type(process_def_entry_t), pointer :: entry
  type(os_data_t) :: os_data
  class(model_data_t), pointer :: model
  type(string_t) :: model_name
  type(string_t), dimension(:), allocatable :: prt_in, prt_out
  type(process_constants_t) :: data
  class(prc_core_driver_t), allocatable :: driver
  integer, parameter :: cdf = c_default_float
  integer, parameter :: ci = c_int
  real(cdf), dimension(4) :: par
  real(cdf), dimension(0:3,4) :: p
  logical(c_bool) :: flag
  complex(c_default_complex) :: amp
  integer :: i

  write (u, "(A)")  "* Test output: prc_template_me_1"
  write (u, "(A)")  "*   Purpose: create a template matrix element,"
  write (u, "(A)")  "*            normalized to give unit integral,"
  write (u, "(A)")  "*            build a library, link, load, and &
       &access the matrix element"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize a process library with one entry"
  write (u, "(A)")
  call lib%init (var_str ("template_me1"))
  call os_data_init (os_data)

  model_name = "QED"
  call prepare_model (model, model_name)

  allocate (prt_in (2), prt_out (2))
  prt_in = [var_str ("e+"), var_str ("e-")]
  prt_out = [var_str ("m+"), var_str ("m-")]

  allocate (template_me_def_t :: def)
  select type (def)
  type is (template_me_def_t)
     call def%init (model, prt_in, prt_out, unity = .false.)
  end select
  allocate (entry)
  call entry%init (var_str ("template_me1_a"), model_name = model_name, &
       n_in = 2, n_components = 1)
  call entry%import_component (1, n_out = size (prt_out), &
       prt_in  = new_prt_spec (prt_in), &
       prt_out = new_prt_spec (prt_out), &
       method  = var_str ("template"), &
       variant = def)
  call lib%append (entry)
```

```
write (u, "(A)")  "* Configure library"
write (u, "(A)")
call lib%configure (os_data)

write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Clean any left-over files"
write (u, "(A)")
call lib%clean (os_data, distclean = .false.)

write (u, "(A)")  "* Write driver"
write (u, "(A)")
call lib%write_driver (force = .true.)

write (u, "(A)")  "* Write process source code, compile, link, load"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u, libpath = .false.)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                 = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes               = ", &
     lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Constants of template_me1_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("template_me1_a"), 1, data, driver)

write (u, "(1x,A,A)")  "component ID     = ", char (data%id)
write (u, "(1x,A,A)")  "model name       = ", char (data%model_name)
write (u, "(1x,A,A,A)")  "md5sum           = '", data%md5sum, "'"
write (u, "(1x,A,I0)") "n_in  = ", data%n_in
write (u, "(1x,A,I0)") "n_out = ", data%n_out
write (u, "(1x,A,I0)") "n_flv = ", data%n_flv
write (u, "(1x,A,I0)") "n_hel = ", data%n_hel
write (u, "(1x,A,I0)") "n_col = ", data%n_col
write (u, "(1x,A,I0)") "n_cin = ", data%n_cin
write (u, "(1x,A,I0)") "n_cf  = ", data%n_cf
write (u, "(1x,A,10(1x,I0))") "flv state =", data%flv_state
write (u, "(1x,A,10(1x,I2))") "hel state =", data%hel_state(:,1)
do i = 2, 16
   write (u, "(12x,4(1x,I2))")  data%hel_state(:,i)
end do
write (u, "(1x,A,10(1x,I0))") "col state =", data%col_state
write (u, "(1x,A,10(1x,L1))") "ghost flag =", data%ghost_flag
```

```
    write (u, "(1x,A,10(1x,F5.3))") "color factors =", data%color_factors
    write (u, "(1x,A,10(1x,I0))") "cf index =", data%cf_index

    write (u, "(A)")
    write (u, "(A)")  "* Set parameters for template_me1_a and initialize:"
    write (u, "(A)")

    par = [0.3_cdf, 0.0_cdf, 0.0_cdf, 0.0_cdf]
    write (u, "(2x,A,F6.4)")  "ee   = ", par(1)
    write (u, "(2x,A,F6.4)")  "me   = ", par(2)
    write (u, "(2x,A,F6.4)")  "mmu  = ", par(3)
    write (u, "(2x,A,F6.4)")  "mtau = ", par(4)

    write (u, "(A)")
    write (u, "(A)")  "* Set kinematics:"
    write (u, "(A)")

    p = reshape ([ &
         1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
         1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
         1.0_cdf, 1.0_cdf, 0.0_cdf, 0.0_cdf, &
         1.0_cdf,-1.0_cdf, 0.0_cdf, 0.0_cdf &
         ], [4,4])
    do i = 1, 4
       write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
    end do

    select type (driver)
    type is (template_me_driver_t)
       call driver%init (par, 0)

       call driver%new_event (p)

       write (u, "(A)")
       write (u, "(A)")  "* Compute matrix element:"
       write (u, "(A)")

       call driver%is_allowed (1_ci, 6_ci, 1_ci, flag)
       write (u, "(1x,A,L1)") "is_allowed (1, 6, 1) = ", flag

       call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
       write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)
    end select

    call lib%final ()
    call cleanup_model (model)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prc_template_me_1"

  end subroutine prc_template_me_1
```

⟨*Template matrix elements: execute tests*⟩+≡

```
  call test (prc_template_me_2, "prc_template_me_2", &
```

```
        "build and load simple template_unity process", &
        u, results)
```

⟨*Template matrix elements: test declarations*⟩+≡

```
    public :: prc_template_me_2
```

⟨*Template matrix elements: tests*⟩+≡

```
    subroutine prc_template_me_2 (u)
      integer, intent(in) :: u
      type(process_library_t) :: lib
      class(prc_core_def_t), allocatable :: def
      type(process_def_entry_t), pointer :: entry
      type(os_data_t) :: os_data
      class(model_data_t), pointer :: model
      type(string_t) :: model_name
      type(string_t), dimension(:), allocatable :: prt_in, prt_out
      type(process_constants_t) :: data
      class(prc_core_driver_t), allocatable :: driver
      integer, parameter :: cdf = c_default_float
      integer, parameter :: ci = c_int
      real(cdf), dimension(4) :: par
      real(cdf), dimension(0:3,4) :: p
      logical(c_bool) :: flag
      complex(c_default_complex) :: amp
      integer :: i

      write (u, "(A)")  "* Test output: prc_template_me_1"
      write (u, "(A)")  "*   Purpose: create a template matrix element,"
      write (u, "(A)")  "*            being identical to unity,"
      write (u, "(A)")  "*            build a library, link, load, and &
           &access the matrix element"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize a process library with one entry"
      write (u, "(A)")
      call lib%init (var_str ("template_me2"))
      call os_data_init (os_data)

      model_name = "QED"
      call prepare_model (model, model_name)

      allocate (prt_in (2), prt_out (2))
      prt_in = [var_str ("e+"), var_str ("e-")]
      prt_out = [var_str ("m+"), var_str ("m-")]

      allocate (template_me_def_t :: def)
      select type (def)
      type is (template_me_def_t)
         call def%init (model, prt_in, prt_out, unity = .true.)
      end select
      allocate (entry)
      call entry%init (var_str ("template_me2_a"), model_name = model_name, &
           n_in = 2, n_components = 1)
      call entry%import_component (1, n_out = size (prt_out), &
           prt_in  = new_prt_spec (prt_in), &
```

```
     prt_out = new_prt_spec (prt_out), &
     method  = var_str ("template_unity"), &
     variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")
call lib%configure (os_data)

write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Clean any left-over files"
write (u, "(A)")
call lib%clean (os_data, distclean = .false.)

write (u, "(A)")  "* Write driver"
write (u, "(A)")
call lib%write_driver (force = .true.)

write (u, "(A)")  "* Write process source code, compile, link, load"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u, libpath = .false.)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                 = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes               = ", &
     lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Constants of template_me2_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("template_me2_a"), 1, data, driver)

write (u, "(1x,A,A)")  "component ID     = ", char (data%id)
write (u, "(1x,A,A)")  "model name       = ", char (data%model_name)
write (u, "(1x,A,A,A)")  "md5sum           = '", data%md5sum, "'"
write (u, "(1x,A,I0)") "n_in  = ", data%n_in
write (u, "(1x,A,I0)") "n_out = ", data%n_out
write (u, "(1x,A,I0)") "n_flv = ", data%n_flv
write (u, "(1x,A,I0)") "n_hel = ", data%n_hel
write (u, "(1x,A,I0)") "n_col = ", data%n_col
write (u, "(1x,A,I0)") "n_cin = ", data%n_cin
write (u, "(1x,A,I0)") "n_cf  = ", data%n_cf
write (u, "(1x,A,10(1x,I0))") "flv state =", data%flv_state
write (u, "(1x,A,10(1x,I2))") "hel state =", data%hel_state(:,1)
```

```
do i = 2, 16
   write (u, "(12x,4(1x,I2))")  data%hel_state(:,i)
end do
write (u, "(1x,A,10(1x,I0))") "col state =", data%col_state
write (u, "(1x,A,10(1x,L1))") "ghost flag =", data%ghost_flag
write (u, "(1x,A,10(1x,F5.3))") "color factors =", data%color_factors
write (u, "(1x,A,10(1x,I0))") "cf index =", data%cf_index

write (u, "(A)")
write (u, "(A)")  "* Set parameters for template_me2_a and initialize:"
write (u, "(A)")

par = [0.3_cdf, 0.0_cdf, 0.0_cdf, 0.0_cdf]
write (u, "(2x,A,F6.4)")  "ee   = ", par(1)
write (u, "(2x,A,F6.4)")  "me   = ", par(2)
write (u, "(2x,A,F6.4)")  "mmu  = ", par(3)
write (u, "(2x,A,F6.4)")  "mtau = ", par(4)

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
     1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
     1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
     1.0_cdf, 1.0_cdf, 0.0_cdf, 0.0_cdf, &
     1.0_cdf,-1.0_cdf, 0.0_cdf, 0.0_cdf &
     ], [4,4])
do i = 1, 4
   write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
end do

select type (driver)
type is (template_me_driver_t)
   call driver%init (par, 0)

   call driver%new_event (p)

   write (u, "(A)")
   write (u, "(A)")  "* Compute matrix element:"
   write (u, "(A)")

   call driver%is_allowed (1_ci, 6_ci, 1_ci, flag)
   write (u, "(1x,A,L1)") "is_allowed (1, 6, 1) = ", flag

   call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
   write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)
end select

call lib%final ()
call cleanup_model (model)

write (u, "(A)")
write (u, "(A)")  "* Test output end: prc_template_me_2"
```

```
  end subroutine prc_template_me_2
```

16.4 O'MEGA Interface

The standard method for process computation with WHIZARD is the O'MEGA matrix element generator.

This section implements the interface to the code generator (via the makefile) and the driver for the features provided by the O'MEGA matrix element.

There are actually two different methods steered by this interface, the traditional one which delivers compiled Fortran code, while the O'MEGA virtual machine (OVM) produces bytecode with look-up tables.

⟨prc_omega.f90⟩≡

```
⟨File header⟩

module prc_omega

  use, intrinsic :: iso_c_binding !NODEP!

  use kinds
⟨Use strings⟩
  use io_units
  use system_defs, only: TAB
  use diagnostics
  use os_interface
  use lorentz
  use sm_qcd
  use interactions
  use model_data

  use particle_specifiers, only: new_prt_spec
  use process_constants
  use prclib_interfaces
  use prc_core_def
  use process_libraries
  use prc_core

⟨Standard module head⟩

⟨Omega interface: public⟩

⟨Omega interface: types⟩

⟨Omega interface: interfaces⟩

contains

⟨Omega interface: procedures⟩

end module prc_omega
```

16.4.1 Process definition

For the process definition we implement an extension of the `prc_core_def_t` abstract type.

⟨*Omega interface: public*⟩≡
```
public :: omega_def_t
```

⟨*Omega interface: types*⟩≡
```
type, abstract, extends (prc_core_def_t) :: omega_def_t
 contains
 ⟨Omega interface: omega def: TBP⟩
end type omega_def_t
```

⟨*Omega interface: public*⟩+≡
```
public :: omega_omega_def_t
```

⟨*Omega interface: types*⟩+≡
```
type, extends (omega_def_t) :: omega_omega_def_t
 contains
 ⟨Omega interface: omega omega def: TBP⟩
end type omega_omega_def_t
```

⟨*Omega interface: public*⟩+≡
```
public :: omega_ovm_def_t
```

⟨*Omega interface: types*⟩+≡
```
type, extends (omega_def_t) :: omega_ovm_def_t
 contains
 ⟨Omega interface: omega ovm def: TBP⟩
end type omega_ovm_def_t
```

⟨*Omega interface: omega omega def: TBP*⟩≡
```
procedure, nopass :: type_string => omega_omega_def_type_string
```

⟨*Omega interface: procedures*⟩≡
```
function omega_omega_def_type_string () result (string)
  type(string_t) :: string
  string = "omega"
end function omega_omega_def_type_string
```

⟨*Omega interface: omega ovm def: TBP*⟩≡
```
procedure, nopass :: type_string => omega_ovm_def_type_string
```

⟨*Omega interface: procedures*⟩+≡
```
function omega_ovm_def_type_string () result (string)
  type(string_t) :: string
  string = "ovm"
end function omega_ovm_def_type_string
```

Initialization: allocate the writer for the `O'MEGA` matrix element. Also set any data for this process that the writer needs.

⟨*Omega interface: omega def: TBP*⟩≡
```
procedure :: init => omega_def_init
```

⟨*Omega interface: procedures*⟩+≡

```
subroutine omega_def_init (object, model_name, prt_in, prt_out, &
     restrictions, cms_scheme, openmp_support, report_progress, &
     extra_options, diags, diags_color)
  class(omega_def_t), intent(out) :: object
  type(string_t), intent(in) :: model_name
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
  logical, intent(in), optional :: cms_scheme
  type(string_t), intent(in), optional :: restrictions
  logical, intent(in), optional :: openmp_support
  logical, intent(in), optional :: report_progress
  logical, intent(in), optional :: diags, diags_color
  type(string_t), intent(in), optional :: extra_options
  select type (object)
  type is (omega_omega_def_t)
     allocate (omega_omega_writer_t :: object%writer)
     select type (writer => object%writer)
     type is (omega_omega_writer_t)
        call writer%init (model_name, prt_in, prt_out, &
             restrictions, cms_scheme, openmp_support, &
             report_progress, extra_options, diags, diags_color)
     end select
  type is (omega_ovm_def_t)
     allocate (omega_ovm_writer_t :: object%writer)
     select type (writer => object%writer)
     type is (omega_ovm_writer_t)
        call writer%init (model_name, prt_in, prt_out, &
             restrictions, cms_scheme, openmp_support, &
             report_progress, extra_options, diags, diags_color)
     end select
  end select
end subroutine omega_def_init
```

Write/read process- and method-specific data.

⟨*Omega interface: omega def: TBP*⟩+≡

```
procedure :: write => omega_def_write
```

⟨*Omega interface: procedures*⟩+≡

```
subroutine omega_def_write (object, unit)
  class(omega_def_t), intent(in) :: object
  integer, intent(in) :: unit
  select type (writer => object%writer)
  type is (omega_omega_writer_t)
     call writer%write (unit)
  type is (omega_ovm_writer_t)
     call writer%write (unit)
  end select
end subroutine omega_def_write
```

⟨*Omega interface: omega def: TBP*⟩+≡

```
procedure :: read => omega_def_read
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine omega_def_read (object, unit)
  class(omega_def_t), intent(out) :: object
  integer, intent(in) :: unit
  call msg_bug ("O'Mega process definition: input not supported yet")
end subroutine omega_def_read
```

Allocate the driver for `O'MEGA`matrix elements.

⟨*Omega interface: omega def: TBP*⟩+≡
```
procedure :: allocate_driver => omega_def_allocate_driver
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine omega_def_allocate_driver (object, driver, basename)
  class(omega_def_t), intent(in) :: object
  class(prc_core_driver_t), intent(out), allocatable :: driver
  type(string_t), intent(in) :: basename
  allocate (omega_driver_t :: driver)
end subroutine omega_def_allocate_driver
```

We need code:

⟨*Omega interface: omega def: TBP*⟩+≡
```
procedure, nopass :: needs_code => omega_def_needs_code
```

⟨*Omega interface: procedures*⟩+≡
```
function omega_def_needs_code () result (flag)
  logical :: flag
  flag = .true.
end function omega_def_needs_code
```

These are the features that an `O'MEGA` matrix element provides.

⟨*Omega interface: omega def: TBP*⟩+≡
```
procedure, nopass :: get_features => omega_def_get_features
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine omega_def_get_features (features)
  type(string_t), dimension(:), allocatable, intent(out) :: features
  allocate (features (6))
  features = [ &
       var_str ("init"), &
       var_str ("update_alpha_s"), &
       var_str ("reset_helicity_selection"), &
       var_str ("is_allowed"), &
       var_str ("new_event"), &
       var_str ("get_amplitude")]
end subroutine omega_def_get_features
```

The interface of the specific features.

⟨*Omega interface: interfaces*⟩≡
```
abstract interface
   subroutine init_t (par, scheme) bind(C)
     import
     real(c_default_float), dimension(*), intent(in) :: par
     integer(c_int), intent(in) :: scheme
```

```
   end subroutine init_t
end interface

abstract interface
   subroutine update_alpha_s_t (alpha_s) bind(C)
     import
     real(c_default_float), intent(in) :: alpha_s
   end subroutine update_alpha_s_t
end interface

abstract interface
   subroutine reset_helicity_selection_t (threshold, cutoff) bind(C)
     import
     real(c_default_float), intent(in) :: threshold
     integer(c_int), intent(in) :: cutoff
   end subroutine reset_helicity_selection_t
end interface

abstract interface
   subroutine is_allowed_t (flv, hel, col, flag) bind(C)
     import
     integer(c_int), intent(in) :: flv, hel, col
     logical(c_bool), intent(out) :: flag
   end subroutine is_allowed_t
end interface

abstract interface
   subroutine new_event_t (p) bind(C)
     import
     real(c_default_float), dimension(0:3,*), intent(in) :: p
   end subroutine new_event_t
end interface

abstract interface
   subroutine get_amplitude_t (flv, hel, col, amp) bind(C)
     import
     integer(c_int), intent(in) :: flv, hel, col
     complex(c_default_complex), intent(out):: amp
   end subroutine get_amplitude_t
end interface
```

Connect the O'MEGA features with the process driver.

⟨*Omega interface: omega def: TBP*⟩+≡

```
procedure :: connect => omega_def_connect
```

⟨*Omega interface: procedures*⟩+≡

```
subroutine omega_def_connect (def, lib_driver, i, proc_driver)
  class(omega_def_t), intent(in) :: def
  class(prclib_driver_t), intent(in) :: lib_driver
  integer, intent(in) :: i
  class(prc_core_driver_t), intent(inout) :: proc_driver
  integer(c_int) :: pid, fid
  type(c_funptr) :: fptr
  select type (proc_driver)
```

```
  type is  (omega_driver_t)
     pid = i
     fid = 1
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%init)
     fid = 2
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%update_alpha_s)
     fid = 3
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%reset_helicity_selection)
     fid = 4
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%is_allowed)
     fid = 5
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%new_event)
     fid = 6
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%get_amplitude)
  end select
end subroutine omega_def_connect
```

16.4.2 The O'MEGA writer

The O'MEGA writer is responsible for inserting the appropriate lines in the makefile that call O'MEGA, and for writing interfaces and wrappers.

⟨*Omega interface: types*⟩+≡

```
type, extends (prc_writer_f_module_t), abstract :: omega_writer_t
   type(string_t) :: model_name
   type(string_t) :: process_mode
   type(string_t) :: process_string
   type(string_t) :: restrictions
   logical :: openmp_support = .false.
   logical :: report_progress = .false.
   logical :: diags = .false.
   logical :: diags_color = .false.
   logical :: complex_mass_scheme = .false.
   type(string_t) :: extra_options
 contains
 ⟨Omega interface: omega writer: TBP⟩
end type omega_writer_t
```

⟨*Omega interface: types*⟩+≡

```
type, extends (omega_writer_t) :: omega_omega_writer_t
 contains
 ⟨Omega interface: omega omega writer: TBP⟩
end type omega_omega_writer_t
```

⟨*Omega interface: types*⟩+≡

```
type, extends (omega_writer_t) :: omega_ovm_writer_t
```

```
 contains
 ⟨Omega interface: omega ovm writer: TBP⟩
end type omega_ovm_writer_t
```

⟨*Omega interface: omega omega writer: TBP*⟩≡

```
procedure, nopass :: type_name => omega_omega_writer_type_name
```

⟨*Omega interface: procedures*⟩+≡

```
function omega_omega_writer_type_name () result (string)
  type(string_t) :: string
  string = "omega"
end function omega_omega_writer_type_name
```

⟨*Omega interface: omega ovm writer: TBP*⟩≡

```
procedure, nopass :: type_name => omega_ovm_writer_type_name
```

⟨*Omega interface: procedures*⟩+≡

```
function omega_ovm_writer_type_name () result (string)
  type(string_t) :: string
  string = "ovm"
end function omega_ovm_writer_type_name
```

Taking into account the prefix for `O'MEGA` module names.

⟨*Omega interface: omega writer: TBP*⟩≡

```
procedure, nopass :: get_module_name => omega_writer_get_module_name
```

⟨*Omega interface: procedures*⟩+≡

```
function omega_writer_get_module_name (id) result (name)
  type(string_t) :: name
  type(string_t), intent(in) :: id
  name = "opr_" // id
end function omega_writer_get_module_name
```

Output. This is called by `omega_def_write`.

⟨*Omega interface: omega writer: TBP*⟩+≡

```
procedure :: write => omega_writer_write
```

⟨*Omega interface: procedures*⟩+≡

```
subroutine omega_writer_write (object, unit)
  class(omega_writer_t), intent(in) :: object
  integer, intent(in) :: unit
  write (unit, "(5x,A,A)")  "Model name        = ", &
       '"' // char (object%model_name) // '"'
  write (unit, "(5x,A,A)")  "Mode string       = ", &
       '"' // char (object%process_mode) // '"'
  write (unit, "(5x,A,A)")  "Process string    = ", &
       '"' // char (object%process_string) // '"'
  write (unit, "(5x,A,A)")  "Restrictions      = ", &
       '"' // char (object%restrictions) // '"'
  write (unit, "(5x,A,L1)")  "OpenMP support    = ", object%openmp_support
  write (unit, "(5x,A,L1)")  "Report progress   = ", object%report_progress
  write (unit, "(5x,A,A)")  "Extra options     = ", &
       '"' // char (object%extra_options) // '"'
  write (unit, "(5x,A,L1)")  "Write diagrams    = ", object%diags
```

```
    write (unit, "(5x,A,L1)")  "Write color diag. = ", object%diags_color
    write (unit, "(5x,A,L1)")  "Complex Mass S.   = ", &
         object%complex_mass_scheme
  end subroutine omega_writer_write
```

Initialize with process data.

⟨*Omega interface: omega writer: TBP*⟩+≡

```
  procedure :: init => omega_writer_init
```

⟨*Omega interface: procedures*⟩+≡

```
  subroutine omega_writer_init (writer, model_name, prt_in, prt_out, &
      restrictions, cms_scheme, openmp_support, report_progress, &
      extra_options, diags, diags_color)
    class(omega_writer_t), intent(out) :: writer
    type(string_t), intent(in) :: model_name
    type(string_t), dimension(:), intent(in) :: prt_in
    type(string_t), dimension(:), intent(in) :: prt_out
    logical, intent(in), optional :: cms_scheme
    type(string_t), intent(in), optional :: restrictions
    logical, intent(in), optional :: openmp_support
    logical, intent(in), optional :: report_progress
    logical, intent(in), optional :: diags, diags_color
    type(string_t), intent(in), optional :: extra_options
    integer :: i
    writer%model_name = model_name
    if (present (restrictions)) then
       writer%restrictions = restrictions
    else
       writer%restrictions = ""
    end if
    if (present (openmp_support))  writer%openmp_support = openmp_support
    if (present (report_progress))  writer%report_progress = report_progress
    if (present (cms_scheme))  writer%complex_mass_scheme = cms_scheme
    if (present (diags))  writer%diags = diags
    if (present (diags_color))  writer%diags_color = diags_color
    if (present (extra_options)) then
       writer%extra_options = " " // extra_options
    else
       writer%extra_options = ""
    end if
    select case (size (prt_in))
    case (1);  writer%process_mode = " -decay"
    case (2);  writer%process_mode = " -scatter"
    end select
    associate (s => writer%process_string)
      s = " '"
      do i = 1, size (prt_in)
         if (i > 1)  s = s // " "
         s = s // prt_in(i)
      end do
      s = s // " ->"
      do i = 1, size (prt_out)
         s = s // " " // prt_out(i)
      end do
```

```
    s = s // "'"
  end associate
end subroutine omega_writer_init
```

The makefile implements the actual O'MEGA call. For old LaTeX distributions, we filter out the hyperref options for O'MEGA diagrams, at least in the testsuite.

⟨*Omega interface: omega writer: TBP*⟩+≡

```
procedure :: write_makefile_code => omega_write_makefile_code
```

⟨*Omega interface: procedures*⟩+≡

```
subroutine omega_write_makefile_code (writer, unit, id, os_data, testflag)
  class(omega_writer_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  type(os_data_t), intent(in) :: os_data
  logical, intent(in), optional :: testflag
  type(string_t) :: omega_binary, omega_path
  type(string_t) :: restrictions_string
  type(string_t) :: openmp_string
  type(string_t) :: kmatrix_string
  type(string_t) :: progress_string
  type(string_t) :: diagrams_string
  type(string_t) :: cms_string
  logical :: escape_hyperref
  escape_hyperref = .false.
  if (present (testflag))  escape_hyperref = testflag
  select type (writer)
  type is (omega_omega_writer_t)
     omega_binary = "omega_" // writer%model_name // ".opt"
  type is (omega_ovm_writer_t)
     select case (char (writer%model_name))
     case ("SM", "SM_CKM", "SM_Higgs", "2HDM", "2HDM_CKM", &
           "HSExt", "QED", "QCD", "Zprime")
     case default
        call msg_fatal ("The model " // char (writer%model_name) &
              // " is not available for the O'Mega VM.")
     end select
     omega_binary = "omega_" // writer%model_name // "_VM.opt"
  end select
  omega_path = os_data%whizard_omega_binpath // "/" // omega_binary
  if (writer%restrictions /= "") then
     restrictions_string = " -cascade '" // writer%restrictions // "'"
  else
     restrictions_string = ""
  end if
  if (writer%openmp_support) then
     openmp_string = " -target:openmp"
  else
     openmp_string = ""
  end if
  if (writer%report_progress) then
     progress_string = " -fusion:progress"
  else
     progress_string = ""
```

```
end if
if (writer%diags) then
   if (writer%diags_color) then
      diagrams_string = " -diagrams:C " // char(id) // &
           "_diags -diagrams_LaTeX"
   else
      diagrams_string = " -diagrams " // char(id) // &
           "_diags -diagrams_LaTeX"
   end if
else
   if (writer%diags_color) then
      diagrams_string = " -diagrams:c " // char(id) // &
           "_diags -diagrams_LaTeX"
   else
      diagrams_string = ""
   end if
end if
if (writer%complex_mass_scheme) then
   cms_string = " -model:cms_width"
else
   cms_string = ""
end if
select case (char (writer%model_name))
case ("SM_rx", "SSC", "NoH_rx", "AltH")
   kmatrix_string = " -target:kmatrix_2_write"
case ("SSC_2", "SSC_AltT", "SM_ul")
   kmatrix_string = " -target:kmatrix_write"
case default
   kmatrix_string = ""
end select
write (unit, "(5A)")  "SOURCES += ", char (id), ".f90"
select type (writer)
type is (omega_ovm_writer_t)
   write (unit, "(5A)")  "SOURCES += ", char (id), ".hbc"
end select
if (writer%diags .or. writer%diags_color) then
   write (unit, "(5A)")  "TEX_SOURCES += ", char (id), "_diags.tex"
   if (os_data%event_analysis_pdf) then
      write (unit, "(5A)")  "TEX_OBJECTS += ", char (id), "_diags.pdf"
   else
      write (unit, "(5A)")  "TEX_OBJECTS += ", char (id), "_diags.ps"
   end if
end if
write (unit, "(5A)")  "OBJECTS += ", char (id), ".lo"
select type (writer)
type is (omega_omega_writer_t)
   write (unit, "(5A)")  char (id), ".f90:"
   write (unit, "(99A)")  TAB, char (omega_path), &
        " -o ", char (id), ".f90", &
        " -target:whizard", &
        " -target:parameter_module parameters_", char (writer%model_name), &
        " -target:module opr_", char (id), &
        " -target:md5sum '", writer%md5sum, "'", &
        char (cms_string), &
```

```
        char (openmp_string), &
        char (progress_string), &
        char (kmatrix_string), &
        char (writer%process_mode), char (writer%process_string), &
        char (restrictions_string), char (diagrams_string), &
        char (writer%extra_options)
type is (omega_ovm_writer_t)
   write (unit, "(5A)")  char (id), ".hbc:"
   write (unit, "(99A)")  TAB, char (omega_path), &
        " -o ", char (id), ".hbc", &
        char (progress_string), &
        char (cms_string), &
        char (writer%process_mode), char (writer%process_string), &
        char (restrictions_string), char (diagrams_string), &
        char (writer%extra_options)
   write (unit, "(5A)")  char (id), ".f90:"
   write (unit, "(99A)")  TAB, char (omega_path), &
        " -o ", char (id), ".f90 -params", &
        " -target:whizard ", &
        " -target:bytecode_file ", char (id), ".hbc", &
        " -target:wrapper_module opr_", char (id), &
        " -target:parameter_module_external parameters_", &
        char (writer%model_name), &
        " -target:md5sum '", writer%md5sum, "'", &
        char (openmp_string)
end select
if (writer%diags .or. writer%diags_color) &
   write (unit, "(5A)")  char (id), "_diags.tex: ", char (id), ".f90"
write (unit, "(5A)")  "clean-", char (id), ":"
write (unit, "(5A)")  TAB, "rm -f ", char (id), ".f90"
write (unit, "(5A)")  TAB, "rm -f opr_", char (id), ".mod"
write (unit, "(5A)")  TAB, "rm -f ", char (id), ".lo"
write (unit, "(5A)")  "CLEAN_SOURCES += ", char (id), ".f90"
select type (writer)
type is (omega_ovm_writer_t)
   write (unit, "(5A)")  "CLEAN_SOURCES += ", char (id), ".hbc"
end select
if (writer%diags .or. writer%diags_color) then
   write (unit, "(5A)")  "CLEAN_SOURCES += ", char (id), "_diags.tex"
end if
write (unit, "(5A)")  "CLEAN_OBJECTS += opr_", char (id), ".mod"
write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), ".lo"
if (writer%diags .or. writer%diags_color) then
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.aux"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.log"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.dvi"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.toc"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.out"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.[1-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.[1-9][0-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.[1-9][0-9][0-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.t[1-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.t[1-9][0-9]"
   write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.t[1-9][0-9][0-9]"
```

```
      write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.mp"
      write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags-fmf.log"
      write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.dvi"
      write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.ps"
      if (os_data%event_analysis_pdf) &
           write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_diags.pdf"
   end if
   write (unit, "(5A)")  char (id), ".lo: ", char (id), ".f90"
   write (unit, "(5A)")  TAB, "$(LTFCOMPILE) $<"
   if (writer%diags .or. writer%diags_color) then
      if (os_data%event_analysis_ps) then
         if (os_data%event_analysis_pdf) then
            write (unit, "(5A)")  char (id), "_diags.pdf: ", char (id), "_diags.tex"
         else
            write (unit, "(5A)")  char (id), "_diags.ps: ", char (id), "_diags.tex"
         end if
         if (escape_hyperref) then
            write (unit, "(5A)")  TAB, "-cat ", char (id), "_diags.tex | \"
            write (unit, "(5A)")  TAB, "   sed -e" // &
               "'s/\\usepackage\[colorlinks\]{hyperref}.*/%\\usepackage" // &
               "\[colorlinks\]{hyperref}/' > \"
            write (unit, "(5A)")  TAB, "   ", char (id), "_diags.tex.tmp"
            write (unit, "(5A)")  TAB, "mv -f ", char (id), "_diags.tex.tmp \"
            write (unit, "(5A)")  TAB, "   ", char (id), "_diags.tex"
         end if
         write (unit, "(5A)")  TAB, "-TEXINPUTS=$(TEX_FLAGS) $(LATEX) " // &
              char (id) // "_diags.tex"
         write (unit, "(5A)")  TAB, "MPINPUTS=$(MP_FLAGS) $(MPOST) " // &
              char (id) // "_diags-fmf.mp"
         write (unit, "(5A)")  TAB, "TEXINPUTS=$(TEX_FLAGS) $(LATEX) " // &
              char (id) // "_diags.tex"
         write (unit, "(5A)")  TAB, "$(DVIPS) -o " // char (id) // "_diags.ps " // &
              char (id) // "_diags.dvi"
         if (os_data%event_analysis_pdf) then
            write (unit, "(5A)")  TAB, "$(PS2PDF) " // char (id) // "_diags.ps"
         end if
      end if
   end if
end subroutine omega_write_makefile_code
```

The source is written by the makefile, so nothing to do here.

⟨*Omega interface: omega writer: TBP*⟩+≡

```
procedure :: write_source_code => omega_write_source_code
```

⟨*Omega interface: procedures*⟩+≡

```
subroutine omega_write_source_code (writer, id)
  class(omega_writer_t), intent(in) :: writer
  type(string_t), intent(in) :: id
end subroutine omega_write_source_code
```

Return the name of a procedure that implements a given feature, as it is provided by the external matrix-element code. `O'MEGA` names some procedures differently, therefore we translate here and override the binding of the base type.

⟨*Omega interface: omega writer: TBP*⟩+≡

```
procedure, nopass :: get_procname => omega_writer_get_procname
```

⟨*Omega interface: procedures*⟩+≡

```
function omega_writer_get_procname (feature) result (name)
  type(string_t) :: name
  type(string_t), intent(in) :: feature
  select case (char (feature))
  case ("n_in");   name = "number_particles_in"
  case ("n_out");  name = "number_particles_out"
  case ("n_flv");  name = "number_flavor_states"
  case ("n_hel");  name = "number_spin_states"
  case ("n_col");  name = "number_color_flows"
  case ("n_cin");  name = "number_color_indices"
  case ("n_cf");   name = "number_color_factors"
  case ("flv_state");  name = "flavor_states"
  case ("hel_state");  name = "spin_states"
  case ("col_state");  name = "color_flows"
  case default
     name = feature
  end select
end function omega_writer_get_procname
```

The interfaces for the O'MEGA-specific features.

⟨*Omega interface: omega writer: TBP*⟩+≡

```
procedure :: write_interface => omega_write_interface
```

⟨*Omega interface: procedures*⟩+≡

```
subroutine omega_write_interface (writer, unit, id, feature)
  class(omega_writer_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  type(string_t), intent(in) :: feature
  type(string_t) :: name
  name = writer%get_c_procname (id, feature)
  write (unit, "(2x,9A)")  "interface"
  select case (char (feature))
  case ("init")
     write (unit, "(5x,9A)")  "subroutine ", char (name), &
          " (par, scheme) bind(C)"
     write (unit, "(7x,9A)")  "import"
     write (unit, "(7x,9A)")  "real(c_default_float), dimension(*), &
          &intent(in) :: par"
     write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: scheme"
     write (unit, "(5x,9A)")  "end subroutine ", char (name)
  case ("update_alpha_s")
     write (unit, "(5x,9A)")  "subroutine ", char (name), " (alpha_s) bind(C)"
     write (unit, "(7x,9A)")  "import"
     write (unit, "(7x,9A)")  "real(c_default_float), intent(in) :: alpha_s"
     write (unit, "(5x,9A)")  "end subroutine ", char (name)
  case ("reset_helicity_selection")
     write (unit, "(5x,9A)")  "subroutine ", char (name), " &
          &(threshold, cutoff) bind(C)"
     write (unit, "(7x,9A)")  "import"
     write (unit, "(7x,9A)")  "real(c_default_float), intent(in) :: threshold"
```

```
      write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: cutoff"
      write (unit, "(5x,9A)")  "end subroutine ", char (name)
   case ("is_allowed")
      write (unit, "(5x,9A)")  "subroutine ", char (name), " &
           &(flv, hel, col, flag) bind(C)"
      write (unit, "(7x,9A)")  "import"
      write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
      write (unit, "(7x,9A)")  "logical(c_bool), intent(out) :: flag"
      write (unit, "(5x,9A)")  "end subroutine ", char (name)
   case ("new_event")
      write (unit, "(5x,9A)")  "subroutine ", char (name), " (p) bind(C)"
      write (unit, "(7x,9A)")  "import"
      write (unit, "(7x,9A)")  "real(c_default_float), dimension(0:3,*), &
           &intent(in) :: p"
      write (unit, "(5x,9A)")  "end subroutine ", char (name)
   case ("get_amplitude")
      write (unit, "(5x,9A)")  "subroutine ", char (name), " &
           &(flv, hel, col, amp) bind(C)"
      write (unit, "(7x,9A)")  "import"
      write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
      write (unit, "(7x,9A)")  "complex(c_default_complex), intent(out) &
           &:: amp"
      write (unit, "(5x,9A)")  "end subroutine ", char (name)
   end select
   write (unit, "(2x,9A)")  "end interface"
end subroutine omega_write_interface
```

The wrappers have to take into account conversion between C and Fortran data types.

NOTE: The case `c_default_float` $\neq$ `default` is not yet covered.

⟨*Omega interface: omega writer: TBP*⟩+≡

```
procedure :: write_wrapper => omega_write_wrapper
```

⟨*Omega interface: procedures*⟩+≡

```
subroutine omega_write_wrapper (writer, unit, id, feature)
  class(omega_writer_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  type(string_t) :: name
  name = writer%get_c_procname (id, feature)
  write (unit, *)
  select case (char (feature))
  case ("init")
     write (unit, "(9A)")  "subroutine ", char (name), &
          " (par, scheme) bind(C)"
     write (unit, "(2x,9A)")  "use iso_c_binding"
     write (unit, "(2x,9A)")  "use kinds"
     write (unit, "(2x,9A)")  "use opr_", char (id)
     write (unit, "(2x,9A)")  "real(c_default_float), dimension(*), &
          &intent(in) :: par"
     write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: scheme"
     if (c_default_float == default .and. c_int == kind(1)) then
        write (unit, "(2x,9A)")  "call ", char (feature), " (par, scheme)"
     end if
```

```
   write (unit, "(9A)")  "end subroutine ", char (name)
case ("update_alpha_s")
   write (unit, "(9A)")  "subroutine ", char (name), " (alpha_s) bind(C)"
   write (unit, "(2x,9A)")  "use iso_c_binding"
   write (unit, "(2x,9A)")  "use kinds"
   write (unit, "(2x,9A)")  "use opr_", char (id)
   if (c_default_float == default) then
      write (unit, "(2x,9A)")  "real(c_default_float), intent(in) &
           &:: alpha_s"
      write (unit, "(2x,9A)")  "call ", char (feature), " (alpha_s)"
   end if
   write (unit, "(9A)")  "end subroutine ", char (name)
case ("reset_helicity_selection")
   write (unit, "(9A)")  "subroutine ", char (name), &
        " (threshold, cutoff) bind(C)"
   write (unit, "(2x,9A)")  "use iso_c_binding"
   write (unit, "(2x,9A)")  "use kinds"
   write (unit, "(2x,9A)")  "use opr_", char (id)
   if (c_default_float == default) then
      write (unit, "(2x,9A)")  "real(c_default_float), intent(in) &
           &:: threshold"
      write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: cutoff"
      write (unit, "(2x,9A)")  "call ", char (feature), &
           " (threshold, int (cutoff))"
   end if
   write (unit, "(9A)")  "end subroutine ", char (name)
case ("is_allowed")
   write (unit, "(9A)")  "subroutine ", char (name), &
        " (flv, hel, col, flag) bind(C)"
   write (unit, "(2x,9A)")  "use iso_c_binding"
   write (unit, "(2x,9A)")  "use kinds"
   write (unit, "(2x,9A)")  "use opr_", char (id)
   write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
   write (unit, "(2x,9A)")  "logical(c_bool), intent(out) :: flag"
   write (unit, "(2x,9A)")  "flag = ", char (feature), &
        " (int (flv), int (hel), int (col))"
   write (unit, "(9A)")  "end subroutine ", char (name)
case ("new_event")
   write (unit, "(9A)")  "subroutine ", char (name), " (p) bind(C)"
   write (unit, "(2x,9A)")  "use iso_c_binding"
   write (unit, "(2x,9A)")  "use kinds"
   write (unit, "(2x,9A)")  "use opr_", char (id)
   if (c_default_float == default) then
      write (unit, "(2x,9A)")  "real(c_default_float), dimension(0:3,*), &
           &intent(in) :: p"
      write (unit, "(2x,9A)")  "call ", char (feature), " (p)"
   end if
   write (unit, "(9A)")  "end subroutine ", char (name)
case ("get_amplitude")
   write (unit, "(9A)")  "subroutine ", char (name), &
        " (flv, hel, col, amp) bind(C)"
   write (unit, "(2x,9A)")  "use iso_c_binding"
   write (unit, "(2x,9A)")  "use kinds"
   write (unit, "(2x,9A)")  "use opr_", char (id)
```

```
      write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
      write (unit, "(2x,9A)")  "complex(c_default_complex), intent(out) &
           &:: amp"
      write (unit, "(2x,9A)")  "amp = ", char (feature), &
           " (int (flv), int (hel), int (col))"
      write (unit, "(9A)")  "end subroutine ", char (name)
    end select
  end subroutine omega_write_wrapper
```

16.4.3 Driver

⟨*Omega interface: public*⟩+≡

```
  public :: omega_driver_t
```

⟨*Omega interface: types*⟩+≡

```
  type, extends (prc_core_driver_t) :: omega_driver_t
     procedure(init_t), nopass, pointer :: &
          init => null ()
     procedure(update_alpha_s_t), nopass, pointer :: &
          update_alpha_s => null ()
     procedure(reset_helicity_selection_t), nopass, pointer :: &
          reset_helicity_selection => null ()
     procedure(is_allowed_t), nopass, pointer :: &
          is_allowed => null ()
     procedure(new_event_t), nopass, pointer :: &
          new_event => null ()
     procedure(get_amplitude_t), nopass, pointer :: &
          get_amplitude => null ()
   contains
   ⟨Omega interface: omega driver: TBP⟩
  end type omega_driver_t
```

The reported type is the same as for the `omega_def_t` type.

⟨*Omega interface: omega driver: TBP*⟩≡

```
  procedure, nopass :: type_name => omega_driver_type_name
```

⟨*Omega interface: procedures*⟩+≡

```
  function omega_driver_type_name () result (string)
    type(string_t) :: string
    string = "omega"
  end function omega_driver_type_name
```

16.4.4 High-level process definition

This procedure wraps the details filling a process-component definition entry as appropriate for an `O'MEGA` matrix element.

NOTE: For calling the `import_component` method, we must explicitly address the `process_def_t` parent object. The natural way to call the method of the extended type triggers a bug in gfortran 4.6. The string array arguments `prt_in` and `prt_out` become corrupted and cause a segfault.

⟨*Omega interface: public*⟩+≡

```
public :: omega_make_process_component
```

⟨*Omega interface: procedures*⟩+≡

```
subroutine omega_make_process_component (entry, component_index, &
       model_name, prt_in, prt_out, restrictions, openmp_support, &
       cms_scheme, report_progress, extra_options, diags, diags_color)
  class(process_def_entry_t), intent(inout) :: entry
  integer, intent(in) :: component_index
  type(string_t), intent(in) :: model_name
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
  logical, intent(in), optional :: cms_scheme
  type(string_t), intent(in), optional :: restrictions
  logical, intent(in), optional :: openmp_support
  logical, intent(in), optional :: report_progress
  logical, intent(in), optional :: diags, diags_color
  type(string_t), intent(in), optional :: extra_options
  class(prc_core_def_t), allocatable :: def
  allocate (omega_omega_def_t :: def)
  select type (def)
  type is (omega_omega_def_t)
     call def%init (model_name, prt_in, prt_out, &
          restrictions, cms_scheme, openmp_support, &
          report_progress, extra_options, diags, diags_color)
  end select
  call entry%process_def_t%import_component (component_index, &
       n_out = size (prt_out), &
       prt_in  = new_prt_spec (prt_in), &
       prt_out = new_prt_spec (prt_out), &
       method = var_str ("omega"), &
       variant = def)
end subroutine omega_make_process_component
```

16.4.5 The prc_omega_t wrapper

This is an instance of the generic prc_core_t object. It contains a pointer to the process definition (omega_def_t), a data component (process_constants_t), and the matrix-element driver (omega_driver_t).

⟨*Omega interface: public*⟩+≡

```
public :: prc_omega_t
```

⟨*Omega interface: types*⟩+≡

```
type, extends (prc_core_t) :: prc_omega_t
   real(default), dimension(:), allocatable :: par
   integer :: scheme = 0
   type(helicity_selection_t) :: helicity_selection
   type(qcd_t) :: qcd
 contains
 ⟨Omega interface: prc omega: TBP⟩
end type prc_omega_t
```

The workspace associated to a prc_omega_t object contains a single flag. The flag is used to suppress re-evaluating the matrix element for each quantum-number combination, after the first amplitude belonging to a given kinematics has been computed.

We can also store the value of a running coupling once it has been calculated for an event. The default value is negative, which indicates an undefined value in this context.

⟨*Omega interface: public*⟩+≡
```
public :: omega_state_t
```

⟨*Omega interface: types*⟩+≡
```
type, extends (prc_core_state_t) :: omega_state_t
   logical :: new_kinematics = .true.
   real(default) :: alpha_qcd = -1
 contains
⟨Omega interface: omega state: TBP⟩
end type omega_state_t
```

⟨*Omega interface: omega state: TBP*⟩≡
```
procedure :: write => omega_state_write
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine omega_state_write (object, unit)
  class(omega_state_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(3x,A,L1)")  "O'Mega state: new kinematics = ", &
       object%new_kinematics
end subroutine omega_state_write
```

⟨*Omega interface: omega state: TBP*⟩+≡
```
procedure :: reset_new_kinematics => omega_state_reset_new_kinematics
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine omega_state_reset_new_kinematics (object)
  class(omega_state_t), intent(inout) :: object
  object%new_kinematics = .true.
end subroutine omega_state_reset_new_kinematics
```

Allocate the workspace with the above specific type.

⟨*Omega interface: prc omega: TBP*⟩≡
```
procedure :: allocate_workspace => prc_omega_allocate_workspace
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine prc_omega_allocate_workspace (object, core_state)
  class(prc_omega_t), intent(in) :: object
  class(prc_core_state_t), intent(inout), allocatable :: core_state
  allocate (omega_state_t :: core_state)
end subroutine prc_omega_allocate_workspace
```

The following procedures are inherited from the base type as deferred, thus must be implemented. The corresponding unit tests are skipped here; the procedures are tested when called from the `processes` module.

Output: print just the ID of the associated matrix element. Then display any stored parameters and the helicity selection data. (The latter are printed only if active.)

⟨*Omega interface: prc omega: TBP*⟩+≡
```
procedure :: write => prc_omega_write
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine prc_omega_write (object, unit)
  class(prc_omega_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(3x,A)", advance="no")  "O'Mega process core:"
  if (object%data_known) then
     write (u, "(1x,A)")  char (object%data%id)
  else
     write (u, "(1x,A)")  "[undefined]"
  end if
  if (allocated (object%par)) then
     write (u, "(3x,A)")  "Parameter array:"
     do i = 1, size (object%par)
        write (u, "(5x,I0,1x,ES17.10)")  i, object%par(i)
     end do
  end if
  call object%helicity_selection%write (u)
  call object%qcd%write (u)
end subroutine prc_omega_write
```

⟨*Omega interface: prc omega: TBP*⟩+≡
```
procedure :: write_name => prc_omega_write_name
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine prc_omega_write_name (object, unit)
  class(prc_omega_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u,"(1x,A)") "Core: O'Mega"
end subroutine prc_omega_write_name
```

Temporarily store the parameter array inside the `prc_omega` object, so we can use it later during the actual initialization. Also store threshold and cutoff for helicity selection.

⟨*Omega interface: prc omega: TBP*⟩+≡
```
procedure :: set_parameters => prc_omega_set_parameters
```

⟨*Omega interface: procedures*⟩+≡
```
subroutine prc_omega_set_parameters (prc_omega, model, &
     helicity_selection, qcd, use_color_factors)
  class(prc_omega_t), intent(inout) :: prc_omega
  class(model_data_t), intent(in), target, optional :: model
```

```
    type(helicity_selection_t), intent(in), optional :: helicity_selection
    type(qcd_t), intent(in), optional :: qcd
    logical, intent(in), optional :: use_color_factors
    if (present (model)) then
       if (.not. allocated (prc_omega%par)) &
            allocate (prc_omega%par (model%get_n_real ()))
       call model%real_parameters_to_array (prc_omega%par)
       prc_omega%scheme = model%get_scheme_num ()
    end if
    if (present (helicity_selection)) then
       prc_omega%helicity_selection = helicity_selection
    end if
    if (present (qcd)) then
       prc_omega%qcd = qcd
    end if
    if (present (use_color_factors)) then
       prc_omega%use_color_factors = use_color_factors
    end if
  end subroutine prc_omega_set_parameters
```

To fully initialize the process core, we perform base initialization, then initialize the external matrix element code.

This procedure overrides the `init` method of the base type, which we nevertheless can access via its binding `base_init`. When done, we have an allocated driver. The driver will call the `init` procedure for the external matrix element, and thus transfer the parameter set to where it finally belongs.

If requested, we initialize the helicity selction counter.

⟨*Omega interface: prc omega: TBP*⟩+≡

```
  procedure :: init => prc_omega_init
```

⟨*Omega interface: procedures*⟩+≡

```
  subroutine prc_omega_init (object, def, lib, id, i_component)
    class(prc_omega_t), intent(inout) :: object
    class(prc_core_def_t), intent(in), target :: def
    type(process_library_t), intent(in), target :: lib
    type(string_t), intent(in) :: id
    integer, intent(in) :: i_component
    call object%base_init (def, lib, id, i_component)
    call object%activate_parameters ()
  end subroutine prc_omega_init
```

Activate the stored parameters by transferring them to the external matrix element. Also reset the helicity selection, if requested.

⟨*Omega interface: prc omega: TBP*⟩+≡

```
  procedure :: activate_parameters => prc_omega_activate_parameters
```

⟨*Omega interface: procedures*⟩+≡

```
  subroutine prc_omega_activate_parameters (object)
    class (prc_omega_t), intent(inout) :: object
    if (allocated (object%driver)) then
       if (allocated (object%par)) then
          select type (driver => object%driver)
          type is (omega_driver_t)
```

```
            if (associated (driver%init)) then
               call driver%init (object%par, object%scheme)
            end if
         end select
      else
         call msg_bug ("prc_omega_activate: parameter set is not allocated")
      end if
      call object%reset_helicity_selection ()
   else
      call msg_bug ("prc_omega_activate: driver is not allocated")
   end if
end subroutine prc_omega_activate_parameters
```

Tell whether a particular combination of flavor, helicity, color is allowed. Here we have to consult the matrix-element driver.

⟨*Omega interface: prc omega: TBP*⟩+≡

```
procedure :: is_allowed => prc_omega_is_allowed
```

⟨*Omega interface: procedures*⟩+≡

```
function prc_omega_is_allowed (object, i_term, f, h, c) result (flag)
   class(prc_omega_t), intent(in) :: object
   integer, intent(in) :: i_term, f, h, c
   logical :: flag
   logical(c_bool) :: cflag
   select type (driver => object%driver)
   type is (omega_driver_t)
      call driver%is_allowed (f, h, c, cflag)
      flag = cflag
   end select
end function prc_omega_is_allowed
```

Transfer the generated momenta directly to the hard interaction in the (only) term. We assume that everything has been set up correctly, so the array fits.

We don't reset the `new_kinematics` flag here. This has to be done explicitly by the caller (`reset_new_kinematics`) when a new kinematics configuration is to be considered.

⟨*Omega interface: prc omega: TBP*⟩+≡

```
procedure :: compute_hard_kinematics => prc_omega_compute_hard_kinematics
```

⟨*Omega interface: procedures*⟩+≡

```
subroutine prc_omega_compute_hard_kinematics &
     (object, p_seed, i_term, int_hard, core_state)
   class(prc_omega_t), intent(in) :: object
   type(vector4_t), dimension(:), intent(in) :: p_seed
   integer, intent(in) :: i_term
   type(interaction_t), intent(inout) :: int_hard
   class(prc_core_state_t), intent(inout), allocatable :: core_state
   call int_hard%set_momenta (p_seed)
end subroutine prc_omega_compute_hard_kinematics
```

This procedure is not called for `prc_omega_t`, just a placeholder.

⟨*Omega interface: prc omega: TBP*⟩+≡

```
procedure :: compute_eff_kinematics => prc_omega_compute_eff_kinematics
```

⟨*Omega interface: procedures*⟩+≡

```
subroutine prc_omega_compute_eff_kinematics &
     (object, i_term, int_hard, int_eff, core_state)
  class(prc_omega_t), intent(in) :: object
  integer, intent(in) :: i_term
  type(interaction_t), intent(in) :: int_hard
  type(interaction_t), intent(inout) :: int_eff
  class(prc_core_state_t), intent(inout), allocatable :: core_state
end subroutine prc_omega_compute_eff_kinematics
```

Reset the helicity selection counters and start counting zero helicities. We assume that the `helicity_selection` object is allocated. Otherwise, reset and switch off helicity counting.

In the test routine, the driver is allocated but the driver methods are not. Therefore, guard against a disassociated method.

⟨*Omega interface: prc omega: TBP*⟩+≡

```
procedure :: reset_helicity_selection => prc_omega_reset_helicity_selection
```

⟨*Omega interface: procedures*⟩+≡

```
subroutine prc_omega_reset_helicity_selection (object)
  class(prc_omega_t), intent(inout) :: object
  select type (driver => object%driver)
  type is (omega_driver_t)
     if (associated (driver%reset_helicity_selection)) then
        if (object%helicity_selection%active) then
           call driver%reset_helicity_selection &
                (real (object%helicity_selection%threshold, &
                c_default_float), &
                int (object%helicity_selection%cutoff, c_int))
        else
           call driver%reset_helicity_selection &
                (0._c_default_float, 0_c_int)
        end if
     end if
  end select
end subroutine prc_omega_reset_helicity_selection
```

Compute the amplitude. For the tree-level process, we can ignore the scale settings. The term index `j` is also irrelevant.

We first call `new_event` for the given momenta (which we must unpack), then retrieve the amplitude value for the given quantum numbers.

If the `core_state` status flag is present, we can make sure that we call `new_event` only once for a given kinematics. After the first call, we unset the `new_kinematics` flag.

The core objects computes the appropriate α_s value via the `qcd` subobject, taking into account the provided `fac_scale` value. However, if the extra parameter `alpha_qcd_forced` is allocated, it overrides this setting.

The `is_allowed` query is not redundant, since the status may change during the run if helicities are switched off.

⟨*Omega interface: prc omega: TBP*⟩+≡

```
procedure :: compute_amplitude => prc_omega_compute_amplitude
```

⟨*Omega interface: procedures*⟩+≡

```
function prc_omega_compute_amplitude &
     (object, j, p, f, h, c, fac_scale, ren_scale, alpha_qcd_forced, &
     core_state)  result (amp)
  class(prc_omega_t), intent(in) :: object
  integer, intent(in) :: j
  type(vector4_t), dimension(:), intent(in) :: p
  integer, intent(in) :: f, h, c
  real(default), intent(in) :: fac_scale, ren_scale
  real(default), intent(in), allocatable :: alpha_qcd_forced
  class(prc_core_state_t), intent(inout), allocatable, optional :: core_state
  real(default) :: alpha_qcd
  complex(default) :: amp
  integer :: n_tot, i
  real(c_default_float), dimension(:,:), allocatable :: parray
  complex(c_default_complex) :: camp
  logical :: new_event
  select type (driver => object%driver)
  type is (omega_driver_t)
     new_event = .true.
     if (present (core_state)) then
        if (allocated (core_state)) then
           select type (core_state)
           type is (omega_state_t)
              new_event = core_state%new_kinematics
              core_state%new_kinematics = .false.
           end select
        end if
     end if
     if (new_event) then
        if (allocated (object%qcd%alpha)) then
           if (allocated (alpha_qcd_forced)) then
              alpha_qcd = alpha_qcd_forced
           else
              alpha_qcd = object%qcd%alpha%get (fac_scale)
           end if
           call driver%update_alpha_s (alpha_qcd)
           if (present (core_state)) then
              if (allocated (core_state)) then
                 select type (core_state)
                 type is (omega_state_t)
                    core_state%alpha_qcd = alpha_qcd
                 end select
              end if
           end if
        end if
        n_tot = object%data%get_n_tot ()
        allocate (parray (0:3, n_tot))
        do i = 1, n_tot
           parray(:,i) = vector4_get_components (p(i))
        end do
        call driver%new_event (parray)
     end if
     if (object%is_allowed (1, f, h, c)) then
```

```
            call driver%get_amplitude &
                 (int (f, c_int), int (h, c_int), int (c, c_int), camp)
            amp = camp
         else
            amp = 0
         end if
      end select
   end function prc_omega_compute_amplitude
```

After the amplitude has been computed, we may read off the current value of α_s. This works only if α_s varies, and if the workspace `core_state` is present which stores this value.

⟨*Omega interface: prc omega: TBP*⟩+≡

```
   procedure :: get_alpha_s => prc_omega_get_alpha_s
```

⟨*Omega interface: procedures*⟩+≡

```
   function prc_omega_get_alpha_s (object, core_state) result (alpha)
     class(prc_omega_t), intent(in) :: object
     class(prc_core_state_t), intent(in), allocatable :: core_state
     real(default) :: alpha
     alpha = -1
     if (allocated (object%qcd%alpha) .and. allocated (core_state)) then
        select type (core_state)
        type is (omega_state_t)
           alpha = core_state%alpha_qcd
        end select
     end if
   end function prc_omega_get_alpha_s
```

16.4.6 Unit Test

Test module, followed by the corresponding implementation module. There is a separate test for testing `O'MEGA` diagram generation as this depends on a working analysis setup.

⟨`prc_omega_ut.f90`⟩≡

```
  ⟨File header⟩

  module prc_omega_ut
    use unit_tests
    use prc_omega_uti

  ⟨Standard module head⟩

  ⟨Omega interface: public test⟩

  contains

  ⟨Omega interface: test driver⟩

  end module prc_omega_ut
```

⟨prc_omega_uti.f90⟩≡

```
  ⟨File header⟩

  module prc_omega_uti

    use, intrinsic :: iso_c_binding !NODEP!

    use kinds
  ⟨Use strings⟩
    use io_units
    use file_utils, only: delete_file
    use os_interface
    use sm_qcd
    use lorentz
    use model_data
    use var_base
    use particle_specifiers, only: new_prt_spec
    use prc_core_def
    use process_constants
    use process_libraries
    use prc_core
    use model_testbed, only: prepare_model, cleanup_model

    use prc_omega

  ⟨Standard module head⟩

  ⟨Omega interface: test declarations⟩

  contains

  ⟨Omega interface: tests⟩

  end module prc_omega_uti
```

API: driver for the unit tests below.

⟨*Omega interface: public test*⟩≡

```
    public :: prc_omega_test
```

⟨*Omega interface: test driver*⟩≡

```
    subroutine prc_omega_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Omega interface: execute tests⟩
  end subroutine prc_omega_test
```

⟨*Omega interface: public test*⟩+≡

```
    public :: prc_omega_diags_test
```

⟨*Omega interface: test driver*⟩+≡

```
    subroutine prc_omega_diags_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Omega interface: execute diags tests⟩
  end subroutine prc_omega_diags_test
```

Generate, compile and load a simple process matrix element

The process is $e^+e^- \to \mu^+\mu^-$ for vanishing masses and $e = 0.3$. We initialize the process, build the library, and compute a particular matrix element for momenta of unit energy and right-angle scattering. The matrix element, as it happens, is equal to e^2. (Note that are no conversion factors applied, so this result is exact.)

For `GNU make`, `makeflags` is set to `-j1`. This eliminates a potential clash with a `-j<n>` flag if this test is called from a parallel make.

⟨*Omega interface: execute tests*⟩≡
```
call test (prc_omega_1, "prc_omega_1", &
     "build and load simple OMega process", &
     u, results)
```

⟨*Omega interface: test declarations*⟩≡
```
public :: prc_omega_1
```

⟨*Omega interface: tests*⟩≡
```
subroutine prc_omega_1 (u)
  integer, intent(in) :: u
  type(process_library_t) :: lib
  class(prc_core_def_t), allocatable :: def
  type(process_def_entry_t), pointer :: entry
  type(os_data_t) :: os_data
  type(string_t) :: model_name
  type(string_t), dimension(:), allocatable :: prt_in, prt_out
  type(process_constants_t) :: data
  class(prc_core_driver_t), allocatable :: driver
  integer, parameter :: cdf = c_default_float
  integer, parameter :: ci = c_int
  real(cdf), dimension(4) :: par
  real(cdf), dimension(0:3,4) :: p
  logical(c_bool) :: flag
  complex(c_default_complex) :: amp
  integer :: i

  write (u, "(A)")  "* Test output: prc_omega_1"
  write (u, "(A)")  "*   Purpose: create a simple process with OMega"
  write (u, "(A)")  "*            build a library, link, load, and &
       &access the matrix element"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize a process library with one entry"
  write (u, "(A)")
  call lib%init (var_str ("omega1"))
  call os_data_init (os_data)

  model_name = "QED"
  allocate (prt_in (2), prt_out (2))
  prt_in = [var_str ("e+"), var_str ("e-")]
  prt_out = [var_str ("m+"), var_str ("m-")]

  allocate (omega_omega_def_t :: def)
  select type (def)
  type is (omega_omega_def_t)
```

```
   call def%init (model_name, prt_in, prt_out)
end select
allocate (entry)
call entry%init (var_str ("omega1_a"), model_name = model_name, &
     n_in = 2, n_components = 1)
call entry%import_component (1, n_out = size (prt_out), &
     prt_in  = new_prt_spec (prt_in), &
     prt_out = new_prt_spec (prt_out), &
     method  = var_str ("omega"), &
     variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure library"
write (u, "(A)")
call lib%configure (os_data)

write (u, "(A)")  "* Write makefile"
write (u, "(A)")
call lib%write_makefile (os_data, force = .true.)

write (u, "(A)")  "* Clean any left-over files"
write (u, "(A)")
call lib%clean (os_data, distclean = .false.)

write (u, "(A)")  "* Write driver"
write (u, "(A)")
call lib%write_driver (force = .true.)

write (u, "(A)")  "* Write process source code, compile, link, load"
write (u, "(A)")
call lib%load (os_data)

call lib%write (u, libpath = .false.)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                 = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes               = ", &
     lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Constants of omega1_a_i1:"
write (u, "(A)")

call lib%connect_process (var_str ("omega1_a"), 1, data, driver)

write (u, "(1x,A,A)")  "component ID     = ", char (data%id)
write (u, "(1x,A,A)")  "model name       = ", char (data%model_name)
write (u, "(1x,A,A,A)")  "md5sum           = '", data%md5sum, "'"
write (u, "(1x,A,L1)") "openmp supported = ", data%openmp_supported
write (u, "(1x,A,I0)") "n_in  = ", data%n_in
```

```
write (u, "(1x,A,I0)") "n_out = ", data%n_out
write (u, "(1x,A,I0)") "n_flv = ", data%n_flv
write (u, "(1x,A,I0)") "n_hel = ", data%n_hel
write (u, "(1x,A,I0)") "n_col = ", data%n_col
write (u, "(1x,A,I0)") "n_cin = ", data%n_cin
write (u, "(1x,A,I0)") "n_cf  = ", data%n_cf
write (u, "(1x,A,10(1x,I0))") "flv state =", data%flv_state
write (u, "(1x,A,10(1x,I2))") "hel state =", data%hel_state(:,1)
do i = 2, 16
   write (u, "(12x,4(1x,I2))")  data%hel_state(:,i)
end do
write (u, "(1x,A,10(1x,I0))") "col state =", data%col_state
write (u, "(1x,A,10(1x,L1))") "ghost flag =", data%ghost_flag
write (u, "(1x,A,10(1x,F5.3))") "color factors =", data%color_factors
write (u, "(1x,A,10(1x,I0))") "cf index =", data%cf_index

write (u, "(A)")
write (u, "(A)")  "* Set parameters for omega1_a and initialize:"
write (u, "(A)")

par = [0.3_cdf, 0.0_cdf, 0.0_cdf, 0.0_cdf]
write (u, "(2x,A,F6.4)")  "ee   = ", par(1)
write (u, "(2x,A,F6.4)")  "me   = ", par(2)
write (u, "(2x,A,F6.4)")  "mmu  = ", par(3)
write (u, "(2x,A,F6.4)")  "mtau = ", par(4)

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
     1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
     1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
     1.0_cdf, 1.0_cdf, 0.0_cdf, 0.0_cdf, &
     1.0_cdf,-1.0_cdf, 0.0_cdf, 0.0_cdf &
     ], [4,4])
do i = 1, 4
   write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
end do

select type (driver)
type is (omega_driver_t)
   call driver%init (par, 0)

   call driver%new_event (p)

   write (u, "(A)")
   write (u, "(A)")  "* Compute matrix element:"
   write (u, "(A)")

   call driver%is_allowed (1_ci, 6_ci, 1_ci, flag)
   write (u, "(1x,A,L1)") "is_allowed (1, 6, 1) = ", flag

   call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
```

```
      write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)
    end select

    call lib%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prc_omega_1"

  end subroutine prc_omega_1
```

Check prc_omega_t wrapper and options

The process is $e^-e^+ \to e^-e^+$ for vanishing masses and $e = 0.3$. We build the library using the high-level procedure omega_make_process_component and the "black box" prc_omega_t object. Two variants with different settings for restrictions and OpenMP.

For GNU make, makeflags is set to -j1. This eliminates a potential clash with a -j<n> flag if this test is called from a parallel make.

⟨*Omega interface: execute tests*⟩+≡

```
  call test (prc_omega_2, "prc_omega_2", &
       "OMega option passing", &
       u, results)
```

⟨*Omega interface: test declarations*⟩+≡

```
  public :: prc_omega_2
```

⟨*Omega interface: tests*⟩+≡

```
  subroutine prc_omega_2 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(process_def_entry_t), pointer :: entry
    type(os_data_t) :: os_data
    type(string_t) :: model_name
    class(model_data_t), pointer :: model
    class(vars_t), pointer :: vars
    type(string_t), dimension(:), allocatable :: prt_in, prt_out
    type(string_t) :: restrictions
    type(process_component_def_t), pointer :: config
    type(prc_omega_t) :: prc1, prc2
    type(process_constants_t) :: data
    integer, parameter :: cdf = c_default_float
    integer, parameter :: ci = c_int
    real(cdf), dimension(:), allocatable :: par
    real(cdf), dimension(0:3,4) :: p
    complex(c_default_complex) :: amp
    integer :: i
    logical :: exist

    write (u, "(A)")  "* Test output: prc_omega_2"
    write (u, "(A)")  "*   Purpose: create simple processes with OMega"
    write (u, "(A)")  "*            use the prc_omega wrapper for this"
    write (u, "(A)")  "*            and check OMega options"
    write (u, "(A)")
```

```
write (u, "(A)")  "* Initialize a process library with two entries, &
     &different options."
write (u, "(A)")  "* (1) e- e+ -> e- e+   &
     &(all diagrams, no OpenMP, report progress)"
write (u, "(A)")  "* (2) e- e+ -> e- e+   &
     &(s-channel only, with OpenMP, report progress to file)"

call lib%init (var_str ("omega2"))
call os_data_init (os_data)

model_name = "QED"
call prepare_model (model, model_name, vars)

allocate (prt_in (2), prt_out (2))
prt_in = [var_str ("e-"), var_str ("e+")]
prt_out = prt_in
restrictions = "3+4~A"

allocate (entry)
call entry%init (var_str ("omega2_a"), &
     model, n_in = 2, n_components = 2)

call omega_make_process_component (entry, 1, &
     model_name, prt_in, prt_out, &
     report_progress=.true.)
call omega_make_process_component (entry, 2, &
     model_name, prt_in, prt_out, &
     restrictions=restrictions, openmp_support=.true., &
     extra_options=var_str ("-fusion:progress_file omega2.log"))

call lib%append (entry)

write (u, "(A)")
write (u, "(A)")  "* Remove left-over file"
write (u, "(A)")

call delete_file ("omega2.log")
inquire (file="omega2.log", exist=exist)
write (u, "(1x,A,L1)")  "omega2.log exists = ", exist

write (u, "(A)")
write (u, "(A)")  "* Build and load library"

call lib%configure (os_data)
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

write (u, "(A)")
write (u, "(A)")  "* Check extra output of OMega"
write (u, "(A)")
```

```
inquire (file="omega2.log", exist=exist)
write (u, "(1x,A,L1)")  "omega2.log exists = ", exist

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                 = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes               = ", &
     lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Set parameters for omega2_a and initialize:"
write (u, "(A)")

call vars%set_rval (var_str ("ee"), 0.3_default)
call vars%set_rval (var_str ("me"), 0._default)
call vars%set_rval (var_str ("mmu"), 0._default)
call vars%set_rval (var_str ("mtau"), 0._default)
allocate (par (model%get_n_real ()))
call model%real_parameters_to_c_array (par)

write (u, "(2x,A,F6.4)")  "ee   = ", par(1)
write (u, "(2x,A,F6.4)")  "me   = ", par(2)
write (u, "(2x,A,F6.4)")  "mmu  = ", par(3)
write (u, "(2x,A,F6.4)")  "mtau = ", par(4)

call prc1%set_parameters (model)
call prc2%set_parameters (model)

write (u, "(A)")
write (u, "(A)")  "* Constants of omega2_a_i1:"
write (u, "(A)")

config => lib%get_component_def_ptr (var_str ("omega2_a"), 1)
call prc1%init (config%get_core_def_ptr (), &
     lib, var_str ("omega2_a"), 1)
call prc1%get_constants (data, 1)

write (u, "(1x,A,A)")  "component ID     = ", &
     char (data%id)
write (u, "(1x,A,L1)") "openmp supported = ", &
     data%openmp_supported
write (u, "(1x,A,A,A)") "model name       = '", &
     char (data%model_name), "'"

write (u, "(A)")
write (u, "(A)")  "* Constants of omega2_a_i2:"
write (u, "(A)")

config => lib%get_component_def_ptr (var_str ("omega2_a"), 2)
call prc2%init (config%get_core_def_ptr (), &
     lib, var_str ("omega2_a"), 2)
```

```
call prc2%get_constants (data, 1)

write (u, "(1x,A,A)")  "component ID     = ", &
     char (data%id)
write (u, "(1x,A,L1)") "openmp supported = ", &
     data%openmp_supported
write (u, "(1x,A,A,A)") "model name       = '", &
     char (data%model_name), "'"

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
     1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
     1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
     1.0_cdf, 1.0_cdf, 0.0_cdf, 0.0_cdf, &
     1.0_cdf,-1.0_cdf, 0.0_cdf, 0.0_cdf &
     ], [4,4])
do i = 1, 4
   write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
end do

write (u, "(A)")
write (u, "(A)")  "* Compute matrix element:"
write (u, "(A)")

select type (driver => prc1%driver)
type is (omega_driver_t)
   call driver%new_event (p)
   call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
   write (u, "(2x,A,1x,E11.4)") "(1) |amp (1, 6, 1)| =", abs (amp)
end select

select type (driver => prc2%driver)
type is (omega_driver_t)
   call driver%new_event (p)
   call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
   write (u, "(2x,A,1x,E11.4)") "(2) |amp (1, 6, 1)| =", abs (amp)
end select

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
     1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
     1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
     1.0_cdf, sqrt(0.5_cdf), 0.0_cdf, sqrt(0.5_cdf), &
     1.0_cdf,-sqrt(0.5_cdf), 0.0_cdf,-sqrt(0.5_cdf) &
     ], [4,4])
do i = 1, 4
   write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
end do
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Compute matrix element:"
    write (u, "(A)")

    select type (driver => prc1%driver)
    type is (omega_driver_t)
       call driver%new_event (p)
       call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
       write (u, "(2x,A,1x,E11.4)") "(1) |amp (1, 6, 1)| =", abs (amp)
    end select

    select type (driver => prc2%driver)
    type is (omega_driver_t)
       call driver%new_event (p)
       call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
       write (u, "(2x,A,1x,E11.4)") "(2) |amp (1, 6, 1)| =", abs (amp)
    end select

    call lib%final ()
    call cleanup_model (model)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prc_omega_2"

  end subroutine prc_omega_2
```

Check helicity selection

The process is $e^-e^+ \to e^-e^+$ for vanishing masses. We call the matrix element several times to verify the switching off of irrelevant helicities.

⟨*Omega interface: execute tests*⟩+≡

```
  call test (prc_omega_3, "prc_omega_3", &
       "helicity selection", &
       u, results)
```

⟨*Omega interface: test declarations*⟩+≡

```
  public :: prc_omega_3
```

⟨*Omega interface: tests*⟩+≡

```
  subroutine prc_omega_3 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(process_def_entry_t), pointer :: entry
    type(os_data_t) :: os_data
    type(string_t) :: model_name
    class(model_data_t), pointer :: model
    class(vars_t), pointer :: vars => null ()
    type(string_t), dimension(:), allocatable :: prt_in, prt_out
    type(process_component_def_t), pointer :: config
    type(prc_omega_t) :: prc1
    type(process_constants_t) :: data
    integer, parameter :: cdf = c_default_float
```

```
real(cdf), dimension(:), allocatable :: par
real(cdf), dimension(0:3,4) :: p
type(helicity_selection_t) :: helicity_selection
integer :: i, h

write (u, "(A)")  "* Test output: prc_omega_3"
write (u, "(A)")  "*   Purpose: create simple process with OMega"
write (u, "(A)")  "*            and check helicity selection"
write (u, "(A)")

write (u, "(A)")  "* Initialize a process library."
write (u, "(A)")  "* (1) e- e+ -> e- e+   (all diagrams, no OpenMP)"

call lib%init (var_str ("omega3"))
call os_data_init (os_data)

model_name = "QED"
call prepare_model (model, model_name, vars)

allocate (prt_in (2), prt_out (2))
prt_in = [var_str ("e-"), var_str ("e+")]
prt_out = prt_in

allocate (entry)
call entry%init (var_str ("omega3_a"), &
     model, n_in = 2, n_components = 1)

call omega_make_process_component (entry, 1, &
     model_name, prt_in, prt_out)
call lib%append (entry)

write (u, "(A)")
write (u, "(A)")  "* Build and load library"

call lib%configure (os_data)
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

write (u, "(A)")
write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active                 = ", &
     lib%is_active ()
write (u, "(1x,A,I0)")  "n_processes               = ", &
     lib%get_n_processes ()

write (u, "(A)")
write (u, "(A)")  "* Set parameters for omega3_a and initialize:"
write (u, "(A)")

call vars%set_rval (var_str ("ee"), 0.3_default)
```

```
call vars%set_rval (var_str ("me"), 0._default)
call vars%set_rval (var_str ("mmu"), 0._default)
call vars%set_rval (var_str ("mtau"), 0._default)
allocate (par (model%get_n_real ()))
call model%real_parameters_to_c_array (par)

write (u, "(2x,A,F6.4)")  "ee   = ", par(1)
write (u, "(2x,A,F6.4)")  "me   = ", par(2)
write (u, "(2x,A,F6.4)")  "mmu  = ", par(3)
write (u, "(2x,A,F6.4)")  "mtau = ", par(4)

call prc1%set_parameters (model, helicity_selection=helicity_selection)

write (u, "(A)")
write (u, "(A)")  "* Helicity states of omega3_a_i1:"
write (u, "(A)")

config => lib%get_component_def_ptr (var_str ("omega3_a"), 1)
call prc1%init (config%get_core_def_ptr (), &
     lib, var_str ("omega3_a"), 1)
call prc1%get_constants (data, 1)

do i = 1, data%n_hel
   write (u, "(3x,I2,':',4(1x,I2))") i, data%hel_state(:,i)
end do

write (u, "(A)")
write (u, "(A)")  "* Initially allowed helicities:"
write (u, "(A)")

write (u, "(4x,16(1x,I2))")  [(h, h = 1, data%n_hel)]
write (u, "(4x)", advance = "no")
do h = 1, data%n_hel
   write (u, "(2x,L1)", advance = "no")  prc1%is_allowed (1, 1, h, 1)
end do
write (u, "(A)")

write (u, "(A)")
write (u, "(A)")  "* Reset helicity selection (cutoff = 4)"
write (u, "(A)")

helicity_selection%active = .true.
helicity_selection%threshold = 1e10_default
helicity_selection%cutoff = 4
call helicity_selection%write (u)

call prc1%set_parameters (model, helicity_selection=helicity_selection)
call prc1%reset_helicity_selection ()

write (u, "(A)")
write (u, "(A)")  "* Allowed helicities:"
write (u, "(A)")

write (u, "(4x,16(1x,I2))")  [(h, h = 1, data%n_hel)]
```

```
write (u, "(4x)", advance = "no")
do h = 1, data%n_hel
   write (u, "(2x,L1)", advance = "no")  prc1%is_allowed (1, 1, h, 1)
end do
write (u, "(A)")

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
     1.0_cdf, 0.0_cdf, 0.0_cdf, 1.0_cdf, &
     1.0_cdf, 0.0_cdf, 0.0_cdf,-1.0_cdf, &
     1.0_cdf, 1.0_cdf, 0.0_cdf, 0.0_cdf, &
     1.0_cdf,-1.0_cdf, 0.0_cdf, 0.0_cdf &
     ], [4,4])
do i = 1, 4
   write (u, "(2x,A,I0,A,4(1x,F7.4))")  "p", i, " =", p(:,i)
end do

write (u, "(A)")
write (u, "(A)")  "* Compute scattering matrix 5 times"
write (u, "(A)")

write (u, "(4x,16(1x,I2))")  [(h, h = 1, data%n_hel)]

select type (driver => prc1%driver)
type is (omega_driver_t)
   do i = 1, 5
      call driver%new_event (p)
      write (u, "(2x,I2)", advance = "no")  i
      do h = 1, data%n_hel
         write (u, "(2x,L1)", advance = "no")  prc1%is_allowed (1, 1, h, 1)
      end do
      write (u, "(A)")
   end do
end select

write (u, "(A)")
write (u, "(A)")  "* Reset helicity selection again"
write (u, "(A)")

call prc1%activate_parameters ()

write (u, "(A)")  "* Allowed helicities:"
write (u, "(A)")

write (u, "(4x,16(1x,I2))")  [(h, h = 1, data%n_hel)]
write (u, "(4x)", advance = "no")
do h = 1, data%n_hel
   write (u, "(2x,L1)", advance = "no")  prc1%is_allowed (1, 1, h, 1)
end do
write (u, "(A)")
```

```
    call lib%final ()
    call cleanup_model (model)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prc_omega_3"

  end subroutine prc_omega_3
```

QCD coupling

The process is $u\bar{u} \to d\bar{d}$ for vanishing masses. We compute the amplitude for a fixed configuration once, then reset α_s, then compute again.

For GNU make, makeflags is set to -j1. This eliminates a potential clash with a -j<n> flag if this test is called from a parallel make.

⟨*Omega interface: execute tests*⟩+≡

```
  call test (prc_omega_4, "prc_omega_4", &
       "update QCD alpha", &
       u, results)
```

⟨*Omega interface: test declarations*⟩+≡

```
  public :: prc_omega_4
```

⟨*Omega interface: tests*⟩+≡

```
  subroutine prc_omega_4 (u)
    integer, intent(in) :: u
    type(process_library_t) :: lib
    class(prc_core_def_t), allocatable :: def
    type(process_def_entry_t), pointer :: entry
    type(os_data_t) :: os_data
    type(string_t) :: model_name
    type(string_t), dimension(:), allocatable :: prt_in, prt_out
    type(process_constants_t) :: data
    class(prc_core_driver_t), allocatable :: driver
    integer, parameter :: cdf = c_default_float
    integer, parameter :: ci = c_int
    real(cdf), dimension(8) :: par
    real(cdf), dimension(0:3,4) :: p
    logical(c_bool) :: flag
    complex(c_default_complex) :: amp
    integer :: i
    real(cdf) :: alpha_s

    write (u, "(A)")  "* Test output: prc_omega_4"
    write (u, "(A)")  "*   Purpose: create a QCD process with OMega"
    write (u, "(A)")  "*            and check alpha_s dependence"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize a process library with one entry"
    write (u, "(A)")
    call lib%init (var_str ("prc_omega_4_lib"))
    call os_data_init (os_data)

    model_name = "QCD"
```

```
allocate (prt_in (2), prt_out (2))
prt_in = [var_str ("u"), var_str ("ubar")]
prt_out = [var_str ("d"), var_str ("dbar")]

allocate (omega_omega_def_t :: def)
select type (def)
type is (omega_omega_def_t)
   call def%init (model_name, prt_in, prt_out)
end select
allocate (entry)
call entry%init (var_str ("prc_omega_4_p"), model_name = model_name, &
     n_in = 2, n_components = 1)
call entry%import_component (1, n_out = size (prt_out), &
     prt_in  = new_prt_spec (prt_in), &
     prt_out = new_prt_spec (prt_out), &
     method  = var_str ("omega"), &
     variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure and compile process"
write (u, "(A)")
call lib%configure (os_data)
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

write (u, "(A)")  "* Probe library API:"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active = ", lib%is_active ()

write (u, "(A)")
write (u, "(A)")  "* Set parameters:"
write (u, "(A)")

alpha_s = 0.1178_cdf

par = [alpha_s, &
     0._cdf, 0._cdf, 0._cdf, 0._cdf, 0._cdf, 173.1_cdf, 1.523_cdf]
write (u, "(2x,A,F8.4)")  "alpha_s = ", par(1)
write (u, "(2x,A,F8.4)")  "md      = ", par(2)
write (u, "(2x,A,F8.4)")  "mu      = ", par(3)
write (u, "(2x,A,F8.4)")  "ms      = ", par(4)
write (u, "(2x,A,F8.4)")  "mc      = ", par(5)
write (u, "(2x,A,F8.4)")  "mb      = ", par(6)
write (u, "(2x,A,F8.4)")  "mtop    = ", par(7)
write (u, "(2x,A,F8.4)")  "wtop    = ", par(8)

write (u, "(A)")
write (u, "(A)")  "* Set kinematics:"
write (u, "(A)")

p = reshape ([ &
```

```
      100.0_cdf, 0.0_cdf, 0.0_cdf, 100.0_cdf, &
      100.0_cdf, 0.0_cdf, 0.0_cdf,-100.0_cdf, &
      100.0_cdf, 100.0_cdf, 0.0_cdf, 0.0_cdf, &
      100.0_cdf,-100.0_cdf, 0.0_cdf, 0.0_cdf &
      ], [4,4])
   do i = 1, 4
      write (u, "(2x,A,I0,A,4(1x,F7.1))")  "p", i, " =", p(:,i)
   end do

   call lib%connect_process (var_str ("prc_omega_4_p"), 1, data, driver)

   select type (driver)
   type is (omega_driver_t)
      call driver%init (par, 0)

      write (u, "(A)")
      write (u, "(A)")  "* Compute matrix element:"
      write (u, "(A)")

      call driver%new_event (p)

      call driver%is_allowed (1_ci, 6_ci, 1_ci, flag)
      write (u, "(1x,A,L1)") "is_allowed (1, 6, 1) = ", flag

      call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
      write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)

      write (u, "(A)")
      write (u, "(A)")  "* Double alpha_s and compute matrix element again:"
      write (u, "(A)")

      call driver%update_alpha_s (2 * alpha_s)
      call driver%new_event (p)

      call driver%is_allowed (1_ci, 6_ci, 1_ci, flag)
      write (u, "(1x,A,L1)") "is_allowed (1, 6, 1) = ", flag

      call driver%get_amplitude (1_ci, 6_ci, 1_ci, amp)
      write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)
   end select

   call lib%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: prc_omega_4"

end subroutine prc_omega_4
```

Amplitude and QCD coupling

The same process as before. Here, we initialize with a running α_s coupling and compute twice with different scales. We use the high-level method `compute_amplitude`.

⟨*Omega interface: execute tests*⟩+≡

```
call test (prc_omega_5, "prc_omega_5", &
     "running QCD alpha", &
     u, results)
```

⟨*Omega interface: test declarations*⟩+≡

```
public :: prc_omega_5
```

⟨*Omega interface: tests*⟩+≡

```
subroutine prc_omega_5 (u)
  integer, intent(in) :: u
  type(process_library_t) :: lib
  class(prc_core_def_t), allocatable :: def
  type(process_component_def_t), pointer :: cdef_ptr
  class(prc_core_def_t), pointer :: def_ptr
  type(process_def_entry_t), pointer :: entry
  type(os_data_t) :: os_data
  class(model_data_t), pointer :: model
  type(string_t) :: model_name
  type(string_t), dimension(:), allocatable :: prt_in, prt_out
  type(qcd_t) :: qcd
  class(prc_core_t), allocatable :: core
  class(prc_core_state_t), allocatable :: core_state
  type(vector4_t), dimension(4) :: p
  complex(default) :: amp
  real(default) :: fac_scale
  real(default), allocatable :: alpha_qcd_forced
  integer :: i

  write (u, "(A)")  "* Test output: prc_omega_5"
  write (u, "(A)")  "*   Purpose: create a QCD process with OMega"
  write (u, "(A)")  "*            and check alpha_s dependence"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize a process library with one entry"
  write (u, "(A)")
  call lib%init (var_str ("prc_omega_5_lib"))
  call os_data_init (os_data)

  model_name = "QCD"
  call prepare_model (model, model_name)

  allocate (prt_in (2), prt_out (2))
  prt_in = [var_str ("u"), var_str ("ubar")]
  prt_out = [var_str ("d"), var_str ("dbar")]

  allocate (omega_omega_def_t :: def)
  select type (def)
  type is (omega_omega_def_t)
     call def%init (model_name, prt_in, prt_out)
  end select
  allocate (entry)
  call entry%init (var_str ("prc_omega_5_p"), model_name = model_name, &
       n_in = 2, n_components = 1)
  call entry%import_component (1, n_out = size (prt_out), &
```

```
     prt_in  = new_prt_spec (prt_in), &
     prt_out = new_prt_spec (prt_out), &
     method  = var_str ("omega"), &
     variant = def)
call lib%append (entry)

write (u, "(A)")  "* Configure and compile process"
write (u, "(A)")
call lib%configure (os_data)
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

write (u, "(A)")  "* Probe library API"
write (u, "(A)")

write (u, "(1x,A,L1)")  "is active = ", lib%is_active ()

write (u, "(A)")
write (u, "(A)")  "* Set kinematics"
write (u, "(A)")

p(1) = vector4_moving (100._default, 100._default, 3)
p(2) = vector4_moving (100._default,-100._default, 3)
p(3) = vector4_moving (100._default, 100._default, 1)
p(4) = vector4_moving (100._default,-100._default, 1)
do i = 1, 4
   call vector4_write (p(i), u)
end do

write (u, "(A)")
write (u, "(A)")  "* Setup QCD data"
write (u, "(A)")

allocate (alpha_qcd_from_scale_t :: qcd%alpha)

write (u, "(A)")  "* Setup process core"
write (u, "(A)")

allocate (prc_omega_t :: core)
cdef_ptr => lib%get_component_def_ptr (var_str ("prc_omega_5_p"), 1)
def_ptr => cdef_ptr%get_core_def_ptr ()

select type (core)
type is (prc_omega_t)
   call core%allocate_workspace (core_state)
   call core%set_parameters (model, qcd = qcd)
   call core%init (def_ptr, lib, var_str ("prc_omega_5_p"), 1)
   call core%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Compute matrix element"
   write (u, "(A)")
```

```
      fac_scale = 100
      write (u, "(1x,A,F4.0)")  "factorization scale = ", fac_scale

      amp = core%compute_amplitude &
           (1, p, 1, 6, 1, fac_scale, 100._default, alpha_qcd_forced)

      write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)

      write (u, "(A)")
      write (u, "(A)")  "* Modify factorization scale and &
           &compute matrix element again"
      write (u, "(A)")

      fac_scale = 200
      write (u, "(1x,A,F4.0)")  "factorization scale = ", fac_scale

      amp = core%compute_amplitude &
           (1, p, 1, 6, 1, fac_scale, 100._default, alpha_qcd_forced)

      write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)

      write (u, "(A)")
      write (u, "(A)")  "* Set alpha(QCD) directly and &
           &compute matrix element again"
      write (u, "(A)")

      allocate (alpha_qcd_forced, source = 0.1_default)
      write (u, "(1x,A,F6.4)")  "alpha_qcd = ", alpha_qcd_forced

      amp = core%compute_amplitude &
           (1, p, 1, 6, 1, fac_scale, 100._default, alpha_qcd_forced)

      write (u, "(1x,A,1x,E11.4)") "|amp (1, 6, 1)| =", abs (amp)

    end select

    call lib%final ()
    call cleanup_model (model)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: prc_omega_5"

  end subroutine prc_omega_5
```

Generate matrix element diagrams

The same process as before. No amplitude is computed here, instead we just generate Feynman (and color flow) diagrams, and check whether PS and PDF files have been generated. This test is only run if event analysis is possible.

⟨*Omega interface: execute diags tests*⟩≡

```
  call test (prc_omega_diags_1, "prc_omega_diags_1", &
```

```
        "generate Feynman diagrams", &
        u, results)
```

⟨*Omega interface: test declarations*⟩+≡

```
    public :: prc_omega_diags_1
```

⟨*Omega interface: tests*⟩+≡

```
    subroutine prc_omega_diags_1 (u)
      integer, intent(in) :: u
      type(process_library_t) :: lib
      class(prc_core_def_t), allocatable :: def
      type(process_def_entry_t), pointer :: entry
      type(os_data_t) :: os_data
      type(string_t) :: model_name
      type(string_t), dimension(:), allocatable :: prt_in, prt_out
      type(string_t) :: diags_file, pdf_file, ps_file
      logical :: exist, exist_pdf, exist_ps
      integer :: iostat, u_diags
      character(128) :: buffer

      write (u, "(A)")  "* Test output: prc_omega_diags_1"
      write (u, "(A)")  "*   Purpose: generate Feynman diagrams"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize a process library with one entry"
      write (u, "(A)")
      call lib%init (var_str ("prc_omega_diags_1_lib"))
      call os_data_init (os_data)

      model_name = "SM"

      allocate (prt_in (2), prt_out (2))
      prt_in = [var_str ("u"), var_str ("ubar")]
      prt_out = [var_str ("d"), var_str ("dbar")]

      allocate (omega_omega_def_t :: def)
      select type (def)
      type is (omega_omega_def_t)
         call def%init (model_name, prt_in, prt_out, &
              diags = .true., diags_color = .true.)
      end select
      allocate (entry)
      call entry%init (var_str ("prc_omega_diags_1_p"), model_name = model_name, &
           n_in = 2, n_components = 1)
      call entry%import_component (1, n_out = size (prt_out), &
           prt_in  = new_prt_spec (prt_in), &
           prt_out = new_prt_spec (prt_out), &
           method  = var_str ("omega"), &
           variant = def)
      call lib%append (entry)

      write (u, "(A)")  "* Configure and compile process"
      write (u, "(A)")  "    and generate diagrams"
      write (u, "(A)")
      call lib%configure (os_data)
```

```
  call lib%write_makefile (os_data, force = .true., testflag = .true.)
  call lib%clean (os_data, distclean = .false.)
  call lib%write_driver (force = .true.)
  call lib%load (os_data)

  write (u, "(A)")  "* Probe library API"
  write (u, "(A)")

  write (u, "(1x,A,L1)")  "is active = ", lib%is_active ()

  write (u, "(A)")  "* Check produced diagram files"
  write (u, "(A)")

  diags_file = "prc_omega_diags_1_p_i1_diags.tex"
  ps_file  = "prc_omega_diags_1_p_i1_diags.ps"
  pdf_file = "prc_omega_diags_1_p_i1_diags.pdf"
  inquire (file = char (diags_file), exist = exist)
  if (exist) then
     u_diags = free_unit ()
     open (u_diags, file = char (diags_file), action = "read", status = "old")
     iostat = 0
     do while (iostat == 0)
        read (u_diags, "(A)", iostat = iostat)  buffer
        if (iostat == 0)  write (u, "(A)")  trim (buffer)
     end do
     close (u_diags)
  else
     write (u, "(A)")  "[Feynman diagrams LaTeX file is missing]"
  end if
  inquire (file = char (ps_file), exist = exist_ps)
  if (exist_ps) then
     write (u, "(A)")  "[Feynman diagrams Postscript file exists and is nonempty]"
  else
     write (u, "(A)")  "[Feynman diagrams Postscript file is missing/non-regular]"
  end if
  inquire (file = char (pdf_file), exist = exist_pdf)
  if (exist_pdf) then
     write (u, "(A)")  "[Feynman diagrams PDF file exists and is nonempty]"
  else
     write (u, "(A)")  "[Feynman diagrams PDF file is missing/non-regular]"
  end if

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"
  write (u, "(A)")

  call lib%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: prc_omega_diags_1"

end subroutine prc_omega_diags_1
```

16.5 User-defined matrix elements squared

For certain applications, only the squared matrix elements instead of the pure amplitude can be obtained, e.g. for BLHA One-Loop providers or for the QCD-resummed top-threshold calculation. This is enabled by the `prc_user_defined_base_t`-type.

⟨`prc_user_defined.f90`⟩≡

```
⟨File header⟩
module prc_user_defined

  use, intrinsic :: iso_c_binding !NODEP!

  use kinds
  use constants
  use io_units
⟨Use strings⟩
  use system_defs, only: TAB
  use physics_defs, only: CF
  use diagnostics
  use os_interface
  use lorentz
  use interactions
  use sm_qcd
  use variables, only: var_list_t

  use model_data
  use prclib_interfaces
  use prc_core_def
  use prc_core
  use prc_omega, only: omega_state_t

  use sf_base
  use sf_pdf_builtin, only: pdf_builtin_t
  use sf_lhapdf, only: lhapdf_t
  use pdg_arrays, only: is_gluon, is_quark

⟨Standard module head⟩

⟨Prc User: public⟩

⟨Prc User: parameters⟩

⟨Prc User: types⟩

⟨Prc User: interfaces⟩

contains

⟨Prc User: procedures⟩

end module prc_user_defined
```

16.5.1 Handling of structure functions

User-defined matrix elements do not have access to the structure functions stored in the evaluators. The current solution to this problem is to just apply them explicitly after the computation of the matrix element.

⟨*Prc User: parameters*⟩≡
```
integer, parameter :: LEPTONS = 1
integer, parameter :: HADRONS = 2
```

⟨*Prc User: types*⟩≡
```
type :: sf_handler_t
   integer :: initial_state_type = 0
   integer :: n_sf = -1
   real(default) :: val = one
contains
⟨Prc User: sf handler: TBP⟩
end type sf_handler_t
```

⟨*Prc User: sf handler: TBP*⟩≡
```
procedure :: init => sf_handler_init
```

⟨*Prc User: procedures*⟩≡
```
subroutine sf_handler_init (sf_handler, sf_chain)
  class(sf_handler_t), intent(out) :: sf_handler
  type(sf_chain_instance_t), intent(in) :: sf_chain
  integer :: i
  sf_handler%n_sf = size (sf_chain%sf)
  if (sf_handler%n_sf == 0) then
     sf_handler%initial_state_type = LEPTONS
  else
     do i = 1, sf_handler%n_sf
        select type (int => sf_chain%sf(i)%int)
        type is (pdf_builtin_t)
           sf_handler%initial_state_type = HADRONS
        type is (lhapdf_t)
           sf_handler%initial_state_type = HADRONS
        class default
           sf_handler%initial_state_type = LEPTONS
        end select
     end do
   end if
end subroutine sf_handler_init
```

⟨*Prc User: sf handler: TBP*⟩+≡
```
procedure :: init_dummy => sf_handler_init_dummy
```

⟨*Prc User: procedures*⟩+≡
```
subroutine sf_handler_init_dummy (sf_handler)
  class(sf_handler_t), intent(out) :: sf_handler
  sf_handler%n_sf = 0
  sf_handler%initial_state_type = LEPTONS
end subroutine sf_handler_init_dummy
```

⟨*Prc User: sf handler: TBP*⟩+≡

```
procedure :: apply_structure_functions => sf_handler_apply_structure_functions
```

⟨*Prc User: procedures*⟩+≡

```
subroutine sf_handler_apply_structure_functions (sf_handler, sf_chain, flavors)
   class(sf_handler_t), intent(inout) :: sf_handler
   type(sf_chain_instance_t), intent(in) :: sf_chain
   integer, intent(in), dimension(2) :: flavors
   integer :: i
   real(default), dimension(:), allocatable :: f
   if (sf_handler%n_sf < 0) call msg_fatal ("sf_handler not initialized")
   sf_handler%val = one
   do i = 1, sf_handler%n_sf
      select case (sf_handler%initial_state_type)
      case (HADRONS)
         sf_handler%val = sf_handler%val * sf_handler%get_pdf (sf_chain, i, flavors(i))
      case (LEPTONS)
         call sf_chain%get_matrix_elements (i, f)
         sf_handler%val = sf_handler%val * f(1)
      case default
         call msg_fatal ("sf_handler not initialized")
      end select
   end do
end subroutine sf_handler_apply_structure_functions
```

⟨*Prc User: sf handler: TBP*⟩+≡

```
procedure :: get_pdf => sf_handler_get_pdf
```

⟨*Prc User: procedures*⟩+≡

```
function sf_handler_get_pdf (sf_handler, sf_chain, i, flavor) result (f)
   real(default) :: f
   class(sf_handler_t), intent(in) :: sf_handler
   type(sf_chain_instance_t), intent(in) :: sf_chain
   integer, intent(in) :: i, flavor
   integer :: k
   real(default), dimension(:), allocatable :: ff
   integer, parameter :: n_flv_light = 6

   call sf_chain%get_matrix_elements (i, ff)

   if (is_gluon (flavor)) then
      k = n_flv_light + 1
   else if (is_quark (abs(flavor))) then
      k = n_flv_light + 1 + flavor
   else
      call msg_fatal ("Not a colored particle")
   end if

   f = ff(k)
 end function sf_handler_get_pdf
```

16.5.2 Abstract user defined processes

This process class allows us to factor out common necessities of user defined processes.

Workspace

This is the workspace that is available for user defined processes.

⟨*Prc User: public*⟩≡
```
public :: user_defined_state_t
```

⟨*Prc User: types*⟩+≡
```
type, abstract, extends (prc_core_state_t) :: user_defined_state_t
  logical :: new_kinematics = .true.
  real(default) :: alpha_qcd = -1
contains
⟨Prc User: user defined state: TBP⟩
end type user_defined_state_t
```

⟨*Prc User: user defined state: TBP*⟩≡
```
procedure :: reset_new_kinematics => user_defined_state_reset_new_kinematics
```

⟨*Prc User: procedures*⟩+≡
```
subroutine user_defined_state_reset_new_kinematics (object)
  class(user_defined_state_t), intent(inout) :: object
  object%new_kinematics = .true.
end subroutine user_defined_state_reset_new_kinematics
```

⟨*Prc User: user defined state: TBP*⟩+≡
```
procedure :: get_alpha_qcd => user_defined_state_get_alpha_qcd
```

⟨*Prc User: procedures*⟩+≡
```
function user_defined_state_get_alpha_qcd (object) result (alpha_qcd)
  real(default) :: alpha_qcd
  class(user_defined_state_t), intent(in) :: object
  alpha_qcd = object%alpha_qcd
end function user_defined_state_get_alpha_qcd
```

Driver

We have to add two O'Mega-routines to the user-defined driver to ensure proper process setup. The problem is that during the setup of the real component, the particle and flavor data are taken from the Born component to set up the subtraction terms. However, the Born component expects this data to be obtained from the Omega code, accessed by the driver.

⟨*Prc User: public*⟩+≡
```
public :: user_defined_driver_t
```

⟨*Prc User: types*⟩+≡
```
type, abstract, extends (prc_core_driver_t) :: user_defined_driver_t
   procedure(omega_update_alpha_s), nopass, pointer :: &
            update_alpha_s => null ()
   procedure(omega_is_allowed), nopass, pointer :: &
            is_allowed => null ()
!contains
!<Prc User: user defined driver: TBP>>
end type user_defined_driver_t
```

Core

⟨*Prc User: public*⟩+≡
```
public :: prc_user_defined_base_t
```

⟨*Prc User: types*⟩+≡
```
type, abstract, extends (prc_core_t) :: prc_user_defined_base_t
  type(qcd_t) :: qcd
  integer :: n_flv = 1
  real(default), dimension(:), allocatable :: par
  integer :: scheme = 0
  type(sf_handler_t) :: sf_handler
  real(default) :: maximum_accuracy = 10000.0
contains
⟨Prc User: prc user defined base: TBP⟩
end type prc_user_defined_base_t
```

⟨*Prc User: prc user defined base: TBP*⟩≡
```
procedure :: get_n_flvs => prc_user_defined_base_get_n_flvs
```

⟨*Prc User: procedures*⟩+≡
```
pure function prc_user_defined_base_get_n_flvs (object, i_flv) result (n)
  integer :: n
  class(prc_user_defined_base_t), intent(in) :: object
  integer, intent(in) :: i_flv
  n = size (object%data%flv_state (:,i_flv))
end function prc_user_defined_base_get_n_flvs
```

⟨*Prc User: prc user defined base: TBP*⟩+≡
```
procedure :: get_flv_state => prc_user_defined_base_get_flv_state
```

⟨*Prc User: procedures*⟩+≡
```
function prc_user_defined_base_get_flv_state (object, i_flv) result (flv)
  integer, dimension(:), allocatable :: flv
  class(prc_user_defined_base_t), intent(in) :: object
  integer, intent(in) :: i_flv
  allocate (flv (size (object%data%flv_state (:,i_flv))))
  flv = object%data%flv_state (:,i_flv)
end function prc_user_defined_base_get_flv_state
```

Return one single squared test matrix element. It is fixed to 1, therefore the integration output will be the phase space volume.

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
procedure :: compute_sqme => prc_user_defined_base_compute_sqme
```

⟨*Prc User: procedures*⟩+≡

```
subroutine prc_user_defined_base_compute_sqme (object, i_flv, p, &
      ren_scale, sqme, bad_point)
   class(prc_user_defined_base_t), intent(in) :: object
   integer, intent(in) :: i_flv
   type(vector4_t), dimension(:), intent(in) :: p
   real(default), intent(in) :: ren_scale
   real(default), intent(out) :: sqme
   logical, intent(out) :: bad_point
   sqme = one
   bad_point = .false.
end subroutine prc_user_defined_base_compute_sqme
```

Return an array of 4 numbers corresponding to the BLHA output convention. Used for testing.

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
procedure :: compute_sqme_virt => prc_user_defined_base_compute_sqme_virt
```

⟨*Prc User: procedures*⟩+≡

```
subroutine prc_user_defined_base_compute_sqme_virt (object, i_flv, &
   p, ren_scale, sqme, bad_point)
  class(prc_user_defined_base_t), intent(in) :: object
  integer, intent(in) :: i_flv
  type(vector4_t), dimension(:), intent(in) :: p
  real(default), intent(in) :: ren_scale
  logical, intent(out) :: bad_point
  real(default), dimension(4), intent(out) :: sqme
  call msg_debug2 (D_ME_METHODS, "prc_user_defined_base_compute_sqme_virt")
  sqme(1) = 0.001_default
  sqme(2) = 0.001_default
  sqme(3) = 0.001_default
  sqme(4) = 0.0015_default
  bad_point = .false.
end subroutine prc_user_defined_base_compute_sqme_virt
```

Also return test output for color-correlated matrix elements. We only give a sensible result for the processes used in the functional tests, which have 2 -¿ 2 topology. All other processes will obtain a vanishing dummy color-correlation. This effectively switches off the subtraction contributions, reproducing the real phase-space volume.

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
procedure :: compute_sqme_cc => prc_user_defined_base_compute_sqme_cc
```

⟨*Prc User: procedures*⟩+≡

```
subroutine prc_user_defined_base_compute_sqme_cc (object, i_flv, p, &
   ren_scale, born_cc, bad_point, born_out)
  class(prc_user_defined_base_t), intent(inout) :: object
  integer, intent(in) :: i_flv
  type(vector4_t), intent(in), dimension(:) :: p
```

```
    real(default), intent(in) :: ren_scale
    real(default), intent(inout), dimension(:,:) :: born_cc
    logical, intent(out) :: bad_point
    real(default), intent(out), optional :: born_out
    call msg_debug2 (D_ME_METHODS, "prc_user_defined_base_compute_sqme_cc")
    if (size (p) == 4) then
       if (present (born_out)) then
          born_out = 0.0015_default
          born_cc = zero
          born_cc(3,3) = - CF * born_out
          born_cc(4,4) = - CF * born_out
          born_cc(3,4) = CF * born_out
          born_cc(4,3) = born_cc(3,4)
          bad_point = .false.
       end if
    else
       if (present (born_out)) born_out = zero
       born_cc = zero
    end if
  end subroutine prc_user_defined_base_compute_sqme_cc
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
  procedure :: compute_alpha_s => prc_user_defined_base_compute_alpha_s
```

⟨*Prc User: procedures*⟩+≡

```
  subroutine prc_user_defined_base_compute_alpha_s (object, core_state, fac_scale)
    class(prc_user_defined_base_t), intent(in) :: object
    class(user_defined_state_t), intent(inout) :: core_state
    real(default), intent(in) :: fac_scale
    core_state%alpha_qcd = object%qcd%alpha%get (fac_scale)
  end subroutine prc_user_defined_base_compute_alpha_s
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
  procedure :: get_alpha_s => prc_user_defined_base_get_alpha_s
```

⟨*Prc User: procedures*⟩+≡

```
  function prc_user_defined_base_get_alpha_s (object, core_state) result (alpha)
    class(prc_user_defined_base_t), intent(in) :: object
    class(prc_core_state_t), intent(in), allocatable :: core_state
    real(default) :: alpha
    if (allocated (core_state)) then
      select type (core_state)
      class is (user_defined_state_t)
         alpha = core_state%alpha_qcd
      type is (omega_state_t)
         alpha = core_state%alpha_qcd
      class default
         alpha = zero
      end select
    else
       alpha = zero
    end if
  end function prc_user_defined_base_get_alpha_s
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
procedure :: is_allowed => prc_user_defined_base_is_allowed
```

⟨*Prc User: procedures*⟩+≡

```
function prc_user_defined_base_is_allowed (object, i_term, f, h, c) result (flag)
  class(prc_user_defined_base_t), intent(in) :: object
  integer, intent(in) :: i_term, f, h, c
  logical :: flag
  logical(c_bool) :: cflag
  select type (driver => object%driver)
  class is (user_defined_driver_t)
     call driver%is_allowed (f, h, c, cflag)
     flag = cflag
  class default
     call msg_fatal &
        ("Driver does not fit to user_defined_base_t")
  end select
end function prc_user_defined_base_is_allowed
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
procedure :: get_nflv => prc_user_defined_base_get_nflv
```

⟨*Prc User: procedures*⟩+≡

```
function prc_user_defined_base_get_nflv (object) result (n_flv)
  class(prc_user_defined_base_t), intent(in) :: object
  integer :: n_flv
  n_flv = object%n_flv
end function prc_user_defined_base_get_nflv
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
procedure :: compute_hard_kinematics => prc_user_defined_base_compute_hard_kinematics
```

⟨*Prc User: procedures*⟩+≡

```
subroutine prc_user_defined_base_compute_hard_kinematics &
     (object, p_seed, i_term, int_hard, core_state)
  class(prc_user_defined_base_t), intent(in) :: object
  type(vector4_t), dimension(:), intent(in) :: p_seed
  integer, intent(in) :: i_term
  type(interaction_t), intent(inout) :: int_hard
  class(prc_core_state_t), intent(inout), allocatable :: core_state
  call int_hard%set_momenta (p_seed)
  if (allocated (core_state)) then
    select type (core_state)
    class is (user_defined_state_t); core_state%new_kinematics = .true.
    end select
  end if
end subroutine prc_user_defined_base_compute_hard_kinematics
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
procedure :: compute_eff_kinematics => prc_user_defined_base_compute_eff_kinematics
```

⟨*Prc User: procedures*⟩+≡

```
subroutine prc_user_defined_base_compute_eff_kinematics &
     (object, i_term, int_hard, int_eff, core_state)
```

```
    class(prc_user_defined_base_t), intent(in) :: object
    integer, intent(in) :: i_term
    type(interaction_t), intent(in) :: int_hard
    type(interaction_t), intent(inout) :: int_eff
    class(prc_core_state_t), intent(inout), allocatable :: core_state
  end subroutine prc_user_defined_base_compute_eff_kinematics
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
  procedure :: set_parameters => prc_user_defined_base_set_parameters
```

⟨*Prc User: procedures*⟩+≡

```
  subroutine prc_user_defined_base_set_parameters (object, qcd, model)
    class(prc_user_defined_base_t), intent(inout) :: object
    type(qcd_t), intent(in) :: qcd
    class(model_data_t), intent(in), target, optional :: model
    object%qcd = qcd
    if (present (model)) then
       if (.not. allocated (object%par)) &
            allocate (object%par (model%get_n_real ()))
       call model%real_parameters_to_array (object%par)
       object%scheme = model%get_scheme_num ()
    end if
  end subroutine prc_user_defined_base_set_parameters
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
  procedure :: update_alpha_s => prc_user_defined_base_update_alpha_s
```

⟨*Prc User: procedures*⟩+≡

```
  subroutine prc_user_defined_base_update_alpha_s (object, core_state, fac_scale)
    class(prc_user_defined_base_t), intent(in) :: object
    class(prc_core_state_t), intent(inout), allocatable :: core_state
    real(default), intent(in) :: fac_scale
    real(default) :: alpha_qcd
    if (allocated (object%qcd%alpha)) then
       alpha_qcd = object%qcd%alpha%get (fac_scale)
       select type (driver => object%driver)
       class is (user_defined_driver_t)
          call driver%update_alpha_s (alpha_qcd)
       end select
       select type (core_state)
       class is (user_defined_state_t)
          core_state%alpha_qcd = alpha_qcd
       type is (omega_state_t)
          core_state%alpha_qcd = alpha_qcd
       end select
    end if
  end subroutine prc_user_defined_base_update_alpha_s
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
  procedure :: init_sf_handler => prc_user_defined_base_init_sf_handler
```

⟨*Prc User: procedures*⟩+≡

```
  subroutine prc_user_defined_base_init_sf_handler (core, sf_chain)
     class(prc_user_defined_base_t), intent(inout) :: core
```

```
      type(sf_chain_instance_t), intent(in) :: sf_chain
      if (allocated (sf_chain%sf)) then
         call core%sf_handler%init (sf_chain)
      else
         call core%sf_handler%init_dummy ()
      end if
   end subroutine prc_user_defined_base_init_sf_handler
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
   procedure :: init_sf_handler_dummy => prc_user_defined_base_init_sf_handler_dummy
```

⟨*Prc User: procedures*⟩+≡

```
   subroutine prc_user_defined_base_init_sf_handler_dummy (core)
      class(prc_user_defined_base_t), intent(inout) :: core
      call core%sf_handler%init_dummy ()
   end subroutine prc_user_defined_base_init_sf_handler_dummy
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
   procedure :: apply_structure_functions => prc_user_defined_base_apply_structure_functions
```

⟨*Prc User: procedures*⟩+≡

```
   subroutine prc_user_defined_base_apply_structure_functions (core, sf_chain, flavors)
     class(prc_user_defined_base_t), intent(inout) :: core
     type(sf_chain_instance_t), intent(in) :: sf_chain
     integer, dimension(2), intent(in) :: flavors
     call core%sf_handler%apply_structure_functions (sf_chain, flavors)
   end subroutine prc_user_defined_base_apply_structure_functions
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
   procedure :: get_sf_value => prc_user_defined_base_get_sf_value
```

⟨*Prc User: procedures*⟩+≡

```
   function prc_user_defined_base_get_sf_value (core) result (val)
     real(default) :: val
     class(prc_user_defined_base_t), intent(in) :: core
     val = core%sf_handler%val
   end function prc_user_defined_base_get_sf_value
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
   procedure :: get_helicity_list_base => prc_user_defined_base_get_helicity_list
```

⟨*Prc User: procedures*⟩+≡

```
   function prc_user_defined_base_get_helicity_list (object, i) result (i_out)
     integer :: i_out
     class(prc_user_defined_base_t), intent(in) :: object
     integer, intent(in) :: i
     i_out = i
   end function prc_user_defined_base_get_helicity_list
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
   procedure(prc_user_defined_base_includes_polarization), deferred :: &
     includes_polarization
```

⟨*Prc User: interfaces*⟩≡

```
abstract interface
  function prc_user_defined_base_includes_polarization (object) result (polarized)
    import
    logical :: polarized
    class(prc_user_defined_base_t), intent(in) :: object
  end function prc_user_defined_base_includes_polarization
end interface
```

⟨*Prc User: prc user defined base: TBP*⟩+≡

```
procedure(prc_user_defined_base_create_and_load_extra_libraries), deferred :: &
  create_and_load_extra_libraries
```

⟨*Prc User: interfaces*⟩+≡

```
abstract interface
  subroutine prc_user_defined_base_create_and_load_extra_libraries ( &
      core, flv_states, var_list, os_data, libname, model, i_core)
    import
    class(prc_user_defined_base_t), intent(inout) :: core
    integer, intent(in), dimension(:,:), allocatable :: flv_states
    type(var_list_t), intent(in) :: var_list
    type(os_data_t), intent(in) :: os_data
    type(string_t), intent(in) :: libname
    type(model_data_t), intent(in), target :: model
    integer, intent(in) :: i_core
  end subroutine prc_user_defined_base_create_and_load_extra_libraries
end interface
```

Configuration

This is the abstract user defined process definition

⟨*Prc User: public*⟩+≡

```
public :: user_defined_def_t
```

⟨*Prc User: types*⟩+≡

```
type, abstract, extends (prc_core_def_t) :: user_defined_def_t
  type(string_t) :: basename
contains
⟨Prc User: user def: TBP⟩
end type user_defined_def_t
```

⟨*Prc User: user def: TBP*⟩≡

```
procedure :: set_active_writer => user_defined_def_set_active_writer
```

⟨*Prc User: procedures*⟩+≡

```
subroutine user_defined_def_set_active_writer (def, active)
  class(user_defined_def_t), intent(inout) :: def
  logical, intent(in) :: active
  select type (writer => def%writer)
  class is (prc_user_defined_writer_t)
     writer%active = active
  end select
end subroutine user_defined_def_set_active_writer
```

⟨*Prc User: user def: TBP*⟩+≡

```
procedure, nopass :: get_features => user_defined_def_get_features
```

⟨*Prc User: procedures*⟩+≡

```
subroutine user_defined_def_get_features (features)
  type(string_t), dimension(:), allocatable, intent(out) :: features
  allocate (features (6))
  features = [ &
       var_str ("init"), &
       var_str ("update_alpha_s"), &
       var_str ("reset_helicity_selection"), &
       var_str ("is_allowed"), &
       var_str ("new_event"), &
       var_str ("get_amplitude")]
end subroutine user_defined_def_get_features
```

⟨*Prc User: user def: TBP*⟩+≡

```
procedure :: connect => user_defined_def_connect
procedure :: omega_connect => user_defined_def_connect
```

⟨*Prc User: procedures*⟩+≡

```
subroutine user_defined_def_connect (def, lib_driver, i, proc_driver)
  class(user_defined_def_t), intent(in) :: def
  class(prclib_driver_t), intent(in) :: lib_driver
  integer, intent(in) :: i
  class(prc_core_driver_t), intent(inout) :: proc_driver
  integer :: pid, fid
  type(c_funptr) :: fptr
  select type (proc_driver)
  class is (user_defined_driver_t)
     pid = i
     fid = 2
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%update_alpha_s)
     fid = 4
     call lib_driver%get_fptr (pid, fid, fptr)
     call c_f_procpointer (fptr, proc_driver%is_allowed)
  end select
end subroutine user_defined_def_connect
```

⟨*Prc User: user def: TBP*⟩+≡

```
procedure, nopass :: needs_code => user_def_needs_code
```

⟨*Prc User: procedures*⟩+≡

```
function user_def_needs_code () result (flag)
  logical :: flag
  flag = .true.
end function user_def_needs_code
```

⟨*Prc User: interfaces*⟩+≡

```
abstract interface
   subroutine omega_update_alpha_s (alpha_s) bind(C)
     import
     real(c_default_float), intent(in) :: alpha_s
```

```
   end subroutine omega_update_alpha_s
end interface

abstract interface
   subroutine omega_is_allowed (flv, hel, col, flag) bind(C)
     import
     integer(c_int), intent(in) :: flv, hel, col
     logical(c_bool), intent(out) :: flag
   end subroutine omega_is_allowed
end interface
```

Writer

⟨*Prc User: public*⟩+≡

```
public :: prc_user_defined_writer_t
```

⟨*Prc User: types*⟩+≡

```
type, abstract, extends (prc_writer_f_module_t) :: prc_user_defined_writer_t
  type(string_t) :: model_name
  type(string_t) :: process_mode
  type(string_t) :: process_string
  type(string_t) :: restrictions
  logical :: active = .true.
contains
⟨Prc User: user defined writer: TBP⟩
end type prc_user_defined_writer_t
```

⟨*Prc User: user defined writer: TBP*⟩≡

```
procedure :: init => user_defined_writer_init
procedure :: base_init => user_defined_writer_init
```

⟨*Prc User: procedures*⟩+≡

```
pure subroutine user_defined_writer_init &
     (writer, model_name, prt_in, prt_out, restrictions)
  class(prc_user_defined_writer_t), intent(inout) :: writer
  type(string_t), intent(in) :: model_name
  type(string_t), dimension(:), intent(in) :: prt_in, prt_out
  type(string_t), intent(in), optional :: restrictions
  integer :: i
  writer%model_name = model_name
  if (present (restrictions)) then
     writer%restrictions = restrictions
  else
     writer%restrictions = ""
  end if
  select case (size (prt_in))
     case(1); writer%process_mode = " -decay"
     case(2); writer%process_mode = " -scatter"
  end select
  associate (s => writer%process_string)
    s = " '"
    do i = 1, size (prt_in)
       if (i > 1) s = s // " "
```

```
      s = s // prt_in(i)
    end do
    s = s // " ->"
    do i = 1, size (prt_out)
      s = s // " " // prt_out(i)
    end do
    s = s // "'"
  end associate
end subroutine user_defined_writer_init
```

⟨*Prc User: user defined writer: TBP*⟩+≡

```
procedure, nopass :: get_module_name => prc_user_defined_writer_get_module_name
```

⟨*Prc User: procedures*⟩+≡

```
function prc_user_defined_writer_get_module_name (id) result (name)
  type(string_t) :: name
  type(string_t), intent(in) :: id
  name = "opr_" // id
end function prc_user_defined_writer_get_module_name
```

⟨*Prc User: user defined writer: TBP*⟩+≡

```
procedure :: write_wrapper => prc_user_defined_writer_write_wrapper
```

⟨*Prc User: procedures*⟩+≡

```
subroutine prc_user_defined_writer_write_wrapper (writer, unit, id, feature)
  class(prc_user_defined_writer_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id, feature
  type(string_t) :: name
  name = writer%get_c_procname (id, feature)
  write (unit, *)
  select case (char (feature))
  case ("init")
     write (unit, "(9A)")  "subroutine ", char (name), &
          " (par, scheme) bind(C)"
     write (unit, "(2x,9A)")  "use iso_c_binding"
     write (unit, "(2x,9A)")  "use kinds"
     write (unit, "(2x,9A)")  "use opr_", char (id)
     write (unit, "(2x,9A)")  "real(c_default_float), dimension(*), &
          &intent(in) :: par"
     write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: scheme"
     if (c_default_float == default .and. c_int == kind (1)) then
        write (unit, "(2x,9A)")  "call ", char (feature), " (par, scheme)"
     end if
     write (unit, "(9A)")  "end subroutine ", char (name)
  case ("update_alpha_s")
     write (unit, "(9A)")  "subroutine ", char (name), " (alpha_s) bind(C)"
     write (unit, "(2x,9A)")  "use iso_c_binding"
     write (unit, "(2x,9A)")  "use kinds"
     write (unit, "(2x,9A)")  "use opr_", char (id)
     if (c_default_float == default) then
        write (unit, "(2x,9A)")  "real(c_default_float), intent(in) &
             &:: alpha_s"
        write (unit, "(2x,9A)")  "call ", char (feature), " (alpha_s)"
```

```
      end if
      write (unit, "(9A)")  "end subroutine ", char (name)
   case ("reset_helicity_selection")
      write (unit, "(9A)")  "subroutine ", char (name), &
           " (threshold, cutoff) bind(C)"
      write (unit, "(2x,9A)")  "use iso_c_binding"
      write (unit, "(2x,9A)")  "use kinds"
      write (unit, "(2x,9A)")  "use opr_", char (id)
      if (c_default_float == default) then
         write (unit, "(2x,9A)")  "real(c_default_float), intent(in) &
              &:: threshold"
         write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: cutoff"
         write (unit, "(2x,9A)")  "call ", char (feature), &
              " (threshold, int (cutoff))"
      end if
      write (unit, "(9A)")  "end subroutine ", char (name)
   case ("is_allowed")
      write (unit, "(9A)")  "subroutine ", char (name), &
           " (flv, hel, col, flag) bind(C)"
      write (unit, "(2x,9A)")  "use iso_c_binding"
      write (unit, "(2x,9A)")  "use kinds"
      write (unit, "(2x,9A)")  "use opr_", char (id)
      write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
      write (unit, "(2x,9A)")  "logical(c_bool), intent(out) :: flag"
      write (unit, "(2x,9A)")  "flag = ", char (feature), &
           " (int (flv), int (hel), int (col))"
      write (unit, "(9A)")  "end subroutine ", char (name)
   case ("new_event")
      write (unit, "(9A)")  "subroutine ", char (name), " (p) bind(C)"
      write (unit, "(2x,9A)")  "use iso_c_binding"
      write (unit, "(2x,9A)")  "use kinds"
      write (unit, "(2x,9A)")  "use opr_", char (id)
      if (c_default_float == default) then
         write (unit, "(2x,9A)")  "real(c_default_float), dimension(0:3,*), &
              &intent(in) :: p"
         write (unit, "(2x,9A)")  "call ", char (feature), " (p)"
      end if
      write (unit, "(9A)")  "end subroutine ", char (name)
   case ("get_amplitude")
      write (unit, "(9A)")  "subroutine ", char (name), &
           " (flv, hel, col, amp) bind(C)"
      write (unit, "(2x,9A)")  "use iso_c_binding"
      write (unit, "(2x,9A)")  "use kinds"
      write (unit, "(2x,9A)")  "use opr_", char (id)
      write (unit, "(2x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
      write (unit, "(2x,9A)")  "complex(c_default_complex), intent(out) &
           &:: amp"
      write (unit, "(2x,9A)")  "amp = ", char (feature), &
           " (int (flv), int (hel), int (col))"
      write (unit, "(9A)")  "end subroutine ", char (name)
   end select

end subroutine prc_user_defined_writer_write_wrapper
```

⟨*Prc User: user defined writer: TBP*⟩+≡

```
procedure :: write_interface => prc_user_defined_writer_write_interface
```

⟨*Prc User: procedures*⟩+≡

```
subroutine prc_user_defined_writer_write_interface (writer, unit, id, feature)
  class(prc_user_defined_writer_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  type(string_t), intent(in) :: feature
  type(string_t) :: name
  name = writer%get_c_procname (id, feature)
  write (unit, "(2x,9A)")  "interface"
  select case (char (feature))
  case ("init")
     write (unit, "(5x,9A)")  "subroutine ", char (name), &
          " (par, scheme) bind(C)"
     write (unit, "(7x,9A)")  "import"
     write (unit, "(7x,9A)")  "real(c_default_float), dimension(*), &
          &intent(in) :: par"
     write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: scheme"
     write (unit, "(5x,9A)")  "end subroutine ", char (name)
  case ("update_alpha_s")
     write (unit, "(5x,9A)")  "subroutine ", char (name), " (alpha_s) bind(C)"
     write (unit, "(7x,9A)")  "import"
     write (unit, "(7x,9A)")  "real(c_default_float), intent(in) :: alpha_s"
     write (unit, "(5x,9A)")  "end subroutine ", char (name)
  case ("reset_helicity_selection")
     write (unit, "(5x,9A)")  "subroutine ", char (name), " &
          &(threshold, cutoff) bind(C)"
     write (unit, "(7x,9A)")  "import"
     write (unit, "(7x,9A)")  "real(c_default_float), intent(in) :: threshold"
     write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: cutoff"
     write (unit, "(5x,9A)")  "end subroutine ", char (name)
  case ("is_allowed")
     write (unit, "(5x,9A)")  "subroutine ", char (name), " &
          &(flv, hel, col, flag) bind(C)"
     write (unit, "(7x,9A)")  "import"
     write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
     write (unit, "(7x,9A)")  "logical(c_bool), intent(out) :: flag"
     write (unit, "(5x,9A)")  "end subroutine ", char (name)
  case ("new_event")
     write (unit, "(5x,9A)")  "subroutine ", char (name), " (p) bind(C)"
     write (unit, "(7x,9A)")  "import"
     write (unit, "(7x,9A)")  "real(c_default_float), dimension(0:3,*), &
          &intent(in) :: p"
     write (unit, "(5x,9A)")  "end subroutine ", char (name)
  case ("get_amplitude")
     write (unit, "(5x,9A)")  "subroutine ", char (name), " &
          &(flv, hel, col, amp) bind(C)"
     write (unit, "(7x,9A)")  "import"
     write (unit, "(7x,9A)")  "integer(c_int), intent(in) :: flv, hel, col"
     write (unit, "(7x,9A)")  "complex(c_default_complex), intent(out) &
          &:: amp"
     write (unit, "(5x,9A)")  "end subroutine ", char (name)
  end select
```

```
    write (unit, "(2x,9A)")  "end interface"
  end subroutine prc_user_defined_writer_write_interface
```

⟨*Prc User: user defined writer: TBP*⟩+≡

```
  procedure :: write_source_code => prc_user_defined_writer_write_source_code
```

⟨*Prc User: procedures*⟩+≡

```
  subroutine prc_user_defined_writer_write_source_code (writer, id)
    class(prc_user_defined_writer_t), intent(in) :: writer
    type(string_t), intent(in) :: id
    !!! This is a dummy
  end subroutine prc_user_defined_writer_write_source_code
```

⟨*Prc User: user defined writer: TBP*⟩+≡

```
  procedure :: write_makefile_code => prc_user_defined_writer_write_makefile_code
  procedure :: base_write_makefile_code => prc_user_defined_writer_write_makefile_code
```

⟨*Prc User: procedures*⟩+≡

```
  subroutine prc_user_defined_writer_write_makefile_code (writer, unit, id, os_data, testflag)
    class(prc_user_defined_writer_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id
    type(os_data_t), intent(in) :: os_data
    logical, intent(in), optional :: testflag
    type(string_t) :: omega_binary, omega_path
    type(string_t) :: restrictions_string
    omega_binary = "omega_" // writer%model_name // ".opt"
    omega_path = os_data%whizard_omega_binpath // "/" // omega_binary
    if (writer%restrictions /= "") then
       restrictions_string = " -cascade '" // writer%restrictions // "'"
    else
       restrictions_string = ""
    end if
    write (unit, "(5A)")  "OBJECTS += ", char (id), ".lo"
    write (unit, "(5A)")  char (id), ".f90:"
    write (unit, "(99A)")  TAB, char (omega_path), &
         " -o ", char (id), ".f90", &
         " -target:whizard", &
         " -target:parameter_module parameters_", char (writer%model_name), &
         " -target:module opr_", char (id), &
         " -target:md5sum '", writer%md5sum, "'", &
         char (writer%process_mode), char (writer%process_string), &
         char (restrictions_string)
    write (unit, "(5A)")  "clean-", char (id), ":"
    write (unit, "(5A)")  TAB, "rm -f ", char (id), ".f90"
    write (unit, "(5A)")  TAB, "rm -f opr_", char (id), ".mod"
    write (unit, "(5A)")  TAB, "rm -f ", char (id), ".lo"
    write (unit, "(5A)")  "CLEAN_SOURCES += ", char (id), ".f90"
    write (unit, "(5A)")  "CLEAN_OBJECTS += opr_", char (id), ".mod"
    write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), ".lo"
    write (unit, "(5A)")  char (id), ".lo: ", char (id), ".f90"
    write (unit, "(5A)")  TAB, "$(LTFCOMPILE) $<"

  end subroutine prc_user_defined_writer_write_makefile_code
```

⟨*Prc User: user defined writer: TBP*⟩+≡
```
procedure, nopass:: get_procname => prc_user_defined_writer_writer_get_procname
```

⟨*Prc User: procedures*⟩+≡
```
function prc_user_defined_writer_writer_get_procname (feature) result (name)
  type(string_t) :: name
  type(string_t), intent(in) :: feature
  select case (char (feature))
  case ("n_in");   name = "number_particles_in"
  case ("n_out");  name = "number_particles_out"
  case ("n_flv");  name = "number_flavor_states"
  case ("n_hel");  name = "number_spin_states"
  case ("n_col");  name = "number_color_flows"
  case ("n_cin");  name = "number_color_indices"
  case ("n_cf");   name = "number_color_factors"
  case ("flv_state");  name = "flavor_states"
  case ("hel_state");  name = "spin_states"
  case ("col_state");  name = "color_flows"
  case default
     name = feature
  end select
end function prc_user_defined_writer_writer_get_procname
```

16.5.3 User defined test

Writer

⟨*Prc User: public*⟩+≡
```
public :: user_defined_test_writer_t
```

⟨*Prc User: types*⟩+≡
```
type, extends (prc_user_defined_writer_t) :: user_defined_test_writer_t
contains
⟨Prc User: user test writer: TBP⟩
end type user_defined_test_writer_t
```

⟨*Prc User: user test writer: TBP*⟩≡
```
procedure, nopass :: type_name => user_defined_test_writer_type_name
```

⟨*Prc User: procedures*⟩+≡
```
function user_defined_test_writer_type_name () result (string)
  type(string_t) :: string
  string = "User-defined dummy"
end function user_defined_test_writer_type_name
```

Workspace

This looks pretty useless. Why don't we make `user_defined_state_t` nonabstract and remove this?

⟨*Prc User: public*⟩+≡
```
public :: user_defined_test_state_t
```

⟨*Prc User: types*⟩+≡
```
type, extends (user_defined_state_t) :: user_defined_test_state_t
contains
⟨Prc User: user defined test state: TBP⟩
end type user_defined_test_state_t
```

⟨*Prc User: user defined test state: TBP*⟩≡
```
procedure :: write => user_defined_test_state_write
```

⟨*Prc User: procedures*⟩+≡
```
subroutine user_defined_test_state_write (object, unit)
  class(user_defined_test_state_t), intent(in) :: object
  integer, intent(in), optional :: unit
end subroutine user_defined_test_state_write
```

Driver

⟨*Prc User: public*⟩+≡
```
public :: user_defined_test_driver_t
```

⟨*Prc User: types*⟩+≡
```
type, extends (user_defined_driver_t) :: user_defined_test_driver_t
contains
⟨Prc User: user test driver: TBP⟩
end type user_defined_test_driver_t
```

⟨*Prc User: user test driver: TBP*⟩≡
```
procedure, nopass :: type_name => user_defined_test_driver_type_name
```

⟨*Prc User: procedures*⟩+≡
```
function user_defined_test_driver_type_name () result (type)
  type(string_t) :: type
  type = "User-defined dummy"
end function user_defined_test_driver_type_name
```

Configuration

A user defined test definition.

⟨*Prc User: public*⟩+≡
```
public :: user_defined_test_def_t
```

⟨*Prc User: types*⟩+≡
```
type, extends (user_defined_def_t) :: user_defined_test_def_t
contains
⟨Prc User: user test def: TBP⟩
end type user_defined_test_def_t
```

⟨*Prc User: user test def: TBP*⟩≡
```
procedure :: init => user_defined_test_def_init
```

⟨*Prc User: procedures*⟩+≡

```
subroutine user_defined_test_def_init (object, basename, model_name, &
    prt_in, prt_out)
  class(user_defined_test_def_t), intent(inout) :: object
  type(string_t), intent(in) :: basename, model_name
  type(string_t), dimension(:), intent(in) :: prt_in, prt_out
  object%basename = basename
  allocate (user_defined_test_writer_t :: object%writer)
  select type (writer => object%writer)
  type is (user_defined_test_writer_t)
     call writer%init (model_name, prt_in, prt_out)
  end select
end subroutine user_defined_test_def_init
```

⟨*Prc User: user test def: TBP*⟩+≡

```
procedure, nopass :: type_string => user_defined_test_def_type_string
```

⟨*Prc User: procedures*⟩+≡

```
function user_defined_test_def_type_string () result (string)
  type(string_t) :: string
  string = "user test dummy"
end function user_defined_test_def_type_string
```

⟨*Prc User: user test def: TBP*⟩+≡

```
procedure :: write => user_defined_test_def_write
```

⟨*Prc User: procedures*⟩+≡

```
subroutine user_defined_test_def_write (object, unit)
  class(user_defined_test_def_t), intent(in) :: object
  integer, intent(in) :: unit
end subroutine user_defined_test_def_write
```

⟨*Prc User: user test def: TBP*⟩+≡

```
procedure :: read => user_defined_test_def_read
```

⟨*Prc User: procedures*⟩+≡

```
subroutine user_defined_test_def_read (object, unit)
  class(user_defined_test_def_t), intent(out) :: object
  integer, intent(in) :: unit
end subroutine user_defined_test_def_read
```

⟨*Prc User: user test def: TBP*⟩+≡

```
procedure :: allocate_driver => user_defined_test_def_allocate_driver
```

⟨*Prc User: procedures*⟩+≡

```
subroutine user_defined_test_def_allocate_driver (object, driver, basename)
  class(user_defined_test_def_t), intent(in) :: object
  class(prc_core_driver_t), intent(out), allocatable :: driver
  type(string_t), intent(in) :: basename
  if (.not. allocated (driver)) allocate (user_defined_test_driver_t :: driver)
end subroutine user_defined_test_def_allocate_driver
```

Core

This user defined test just returns $|\mathcal{M}|^2 = 1$ and thus the result of the integration is the n-particle-phase-space volume.

⟨*Prc User: public*⟩+≡
```
public :: prc_user_defined_test_t
```

⟨*Prc User: types*⟩+≡
```
type, extends (prc_user_defined_base_t) :: prc_user_defined_test_t
contains
⟨Prc User: prc test: TBP⟩
end type prc_user_defined_test_t
```

⟨*Prc User: prc test: TBP*⟩≡
```
procedure :: write => prc_user_defined_test_write
```

⟨*Prc User: procedures*⟩+≡
```
subroutine prc_user_defined_test_write (object, unit)
  class(prc_user_defined_test_t), intent(in) :: object
  integer, intent(in), optional :: unit
  call msg_message ("Test user-defined matrix elements")
end subroutine prc_user_defined_test_write
```

⟨*Prc User: prc test: TBP*⟩+≡
```
procedure :: write_name => prc_user_defined_test_write_name
```

⟨*Prc User: procedures*⟩+≡
```
subroutine prc_user_defined_test_write_name (object, unit)
  class(prc_user_defined_test_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u,"(1x,A)") "Core: user defined test"
end subroutine prc_user_defined_test_write_name
```

⟨*Prc User: prc test: TBP*⟩+≡
```
procedure :: compute_amplitude => prc_user_defined_test_compute_amplitude
```

⟨*Prc User: procedures*⟩+≡
```
function prc_user_defined_test_compute_amplitude &
     (object, j, p, f, h, c, fac_scale, ren_scale, alpha_qcd_forced, &
     core_state)  result (amp)
  class(prc_user_defined_test_t), intent(in) :: object
  integer, intent(in) :: j
  type(vector4_t), dimension(:), intent(in) :: p
  integer, intent(in) :: f, h, c
  real(default), intent(in) :: fac_scale, ren_scale
  real(default), intent(in), allocatable :: alpha_qcd_forced
  class(prc_core_state_t), intent(inout), allocatable, optional :: core_state
  complex(default) :: amp
  select type (core_state)
  class is (user_defined_test_state_t)
     core_state%alpha_qcd = object%qcd%alpha%get (fac_scale)
  end select
```

```
    amp = 0.0
  end function prc_user_defined_test_compute_amplitude
```

⟨*Prc User: prc test: TBP*⟩+≡

```
  procedure :: allocate_workspace => prc_user_defined_test_allocate_workspace
```

⟨*Prc User: procedures*⟩+≡

```
  subroutine prc_user_defined_test_allocate_workspace (object, core_state)
    class(prc_user_defined_test_t), intent(in) :: object
    class(prc_core_state_t), intent(inout), allocatable :: core_state
    allocate (user_defined_test_state_t :: core_state)
  end subroutine prc_user_defined_test_allocate_workspace
```

⟨*Prc User: prc test: TBP*⟩+≡

```
  procedure :: includes_polarization => prc_user_defined_test_includes_polarization
```

⟨*Prc User: procedures*⟩+≡

```
  function prc_user_defined_test_includes_polarization (object) result (polarized)
    logical :: polarized
    class(prc_user_defined_test_t), intent(in) :: object
    polarized = .false.
  end function prc_user_defined_test_includes_polarization
```

⟨*Prc User: prc test: TBP*⟩+≡

```
  procedure :: create_and_load_extra_libraries => &
       prc_user_defined_test_create_and_load_extra_libraries
```

⟨*Prc User: procedures*⟩+≡

```
  subroutine prc_user_defined_test_create_and_load_extra_libraries ( &
         core, flv_states, var_list, os_data, libname, model, i_core)
    class(prc_user_defined_test_t), intent(inout) :: core
    integer, intent(in), dimension(:,:), allocatable :: flv_states
    type(var_list_t), intent(in) :: var_list
    type(os_data_t), intent(in) :: os_data
    type(string_t), intent(in) :: libname
    type(model_data_t), intent(in), target :: model
    integer, intent(in) :: i_core
  end subroutine prc_user_defined_test_create_and_load_extra_libraries
```

16.5.4 Threshold

⟨prc_threshold.f90⟩≡

```
  ⟨File header⟩
  module prc_threshold

    use, intrinsic :: iso_c_binding !NODEP!

    use kinds
    use constants
    use numeric_utils
    use string_utils, only: lower_case
    use io_units
```

⟨*Use strings*⟩
```
use physics_defs
use system_defs, only: TAB
use diagnostics
use os_interface
use lorentz
use interactions
use sm_qcd
use model_data
use variables, only: var_list_t

use prclib_interfaces
use process_libraries
use prc_core_def
use prc_core
use prc_user_defined
```

⟨*Standard module head*⟩

⟨*Prc Threshold: public*⟩

⟨*Prc Threshold: interfaces*⟩

⟨*Prc Threshold: types*⟩

```
contains
```

⟨*Prc Threshold: procedures*⟩

```
end module prc_threshold
```

Writer

⟨*Prc Threshold: public*⟩≡
```
public :: threshold_writer_t
```

⟨*Prc Threshold: types*⟩≡
```
type, extends (prc_user_defined_writer_t) :: threshold_writer_t
   integer :: nlo_type
contains
```
⟨*Prc Threshold: threshold writer: TBP*⟩
```
end type threshold_writer_t
```

⟨*Prc Threshold: threshold writer: TBP*⟩≡
```
procedure :: write_makefile_extra => threshold_writer_write_makefile_extra
```

⟨*Prc Threshold: procedures*⟩≡
```
subroutine threshold_writer_write_makefile_extra (writer, unit, id, os_data, nlo_type)
  class(threshold_writer_t), intent(in) :: writer
  integer, intent(in) :: unit
  type(string_t), intent(in) :: id
  type(os_data_t), intent(in) :: os_data
  integer, intent(in) :: nlo_type
  type(string_t) :: f90in, f90, lo, extra
```

```
    call msg_debug (D_ME_METHODS, "threshold_writer_write_makefile_extra")
    if (nlo_type /= BORN) then
       extra = "_" // component_status (nlo_type)
    else
       extra = var_str ("")
    end if
    f90 = id // "_threshold" // extra //".f90"
    f90in = f90 // ".in"
    lo = id // "_threshold" // extra // ".lo"
    write (unit, "(A)") "OBJECTS += " // char (lo)
    write (unit, "(A)") char (f90in) // ":"
    write (unit, "(A)") char (TAB // "if ! test -f " // f90in // &
         "; then cp " // os_data%whizard_sharepath // &
         "/SM_tt_threshold_data/threshold" // extra // ".f90 " // &
         f90in // "; fi")
    write (unit, "(A)") char(f90) // ": " // char (f90in)
    write (unit, "(A)") TAB // "sed 's/@ID@/" // char (id) // "/' " // &
         char (f90in) // " > " // char (f90)
    write (unit, "(5A)")  "CLEAN_SOURCES += ", char (f90)
    write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (f90in)
    write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (id), "_threshold.mod"
    write (unit, "(5A)")  "CLEAN_OBJECTS += ", char (lo)
    write (unit, "(A)") char(lo) // ": " // char (f90) // " " // &
         char(id) // ".f90"
    write (unit, "(5A)")  TAB, "$(LTFCOMPILE) $<"
  end subroutine threshold_writer_write_makefile_extra
```

⟨*Prc Threshold: threshold writer: TBP*⟩+≡

```
  procedure :: write_makefile_code => threshold_writer_write_makefile_code
```

⟨*Prc Threshold: procedures*⟩+≡

```
  subroutine threshold_writer_write_makefile_code (writer, unit, id, os_data, testflag)
    class(threshold_writer_t), intent(in) :: writer
    integer, intent(in) :: unit
    type(string_t), intent(in) :: id
    type(os_data_t), intent(in) :: os_data
    logical, intent(in), optional :: testflag
    call msg_debug (D_ME_METHODS, "threshold_writer_write_makefile_code")
    call writer%base_write_makefile_code (unit, id, os_data, testflag)
    call writer%write_makefile_extra (unit, id, os_data, BORN)
    if (writer%nlo_type == NLO_VIRTUAL .and. writer%active) &
         call writer%write_makefile_extra (unit, id, os_data, writer%nlo_type)
  end subroutine threshold_writer_write_makefile_code
```

⟨*Prc Threshold: threshold writer: TBP*⟩+≡

```
  procedure, nopass :: type_name => threshold_writer_type_name
```

⟨*Prc Threshold: procedures*⟩+≡

```
  function threshold_writer_type_name () result (string)
    type(string_t) :: string
    string = "Threshold"
  end function threshold_writer_type_name
```

Driver

⟨*Prc Threshold: public*⟩+≡
```
public :: threshold_set_process_mode
```
⟨*Prc Threshold: interfaces*⟩≡
```
interface
  subroutine threshold_set_process_mode (mode) bind(C)
    import
    integer(kind = c_int), intent(in) :: mode
  end subroutine threshold_set_process_mode
end interface
```

⟨*Prc Threshold: public*⟩+≡
```
public :: threshold_get_amp_squared
```
⟨*Prc Threshold: interfaces*⟩+≡
```
interface
  subroutine threshold_get_amp_squared (amp2, p_ofs, p_ons, leg, n_tot) bind(C)
    import
    real(c_default_float), intent(out) :: amp2
    real(c_default_float), dimension(0:3,*), intent(in) :: p_ofs
    real(c_default_float), dimension(0:3,*), intent(in) :: p_ons
    integer(kind = c_int) :: n_tot, leg
  end subroutine threshold_get_amp_squared
end interface
```

⟨*Prc Threshold: public*⟩+≡
```
public :: threshold_olp_eval2
```
⟨*Prc Threshold: interfaces*⟩+≡
```
interface
  subroutine threshold_olp_eval2 (i_flv, alpha_s_c, parray, mu_c, &
         sqme_c, acc_c) bind(C)
    import
    integer(c_int), intent(in) :: i_flv
    real(c_default_float), intent(in) :: alpha_s_c
    real(c_default_float), dimension(0:3,*), intent(in) :: parray
    real(c_default_float), intent(in) :: mu_c
    real(c_default_float), dimension(4), intent(out) :: sqme_c
    real(c_default_float), intent(out) :: acc_c
  end subroutine threshold_olp_eval2
end interface
```

⟨*Prc Threshold: public*⟩+≡
```
public :: threshold_init
```
⟨*Prc Threshold: interfaces*⟩+≡
```
interface
 subroutine threshold_init (par, scheme) bind(C)
    import
    real(c_default_float), dimension(*), intent(in) :: par
    integer(c_int), intent(in) :: scheme
  end subroutine threshold_init
end interface
```

⟨*Prc Threshold: public*⟩+≡

```
public :: threshold_start_openloops
```

⟨*Prc Threshold: interfaces*⟩+≡

```
interface
 subroutine threshold_start_openloops () bind(C)
    import
  end subroutine threshold_start_openloops
end interface
```

⟨*Prc Threshold: public*⟩+≡

```
public :: threshold_driver_t
```

⟨*Prc Threshold: types*⟩+≡

```
type, extends (user_defined_driver_t) :: threshold_driver_t
  procedure(threshold_olp_eval2), nopass, pointer :: &
       olp_eval2 => null ()
  procedure(threshold_set_process_mode), nopass, pointer :: &
       set_process_mode => null ()
  procedure(threshold_get_amp_squared), nopass, pointer :: &
       get_amp_squared => null ()
  procedure(threshold_start_openloops), nopass, pointer :: &
       start_openloops => null ()
  procedure(threshold_init), nopass, pointer :: &
       init => null ()
  type(string_t) :: id
  integer :: nlo_type = BORN
contains
⟨Prc Threshold: threshold driver: TBP⟩
end type threshold_driver_t
```

⟨*Prc Threshold: threshold driver: TBP*⟩≡

```
procedure, nopass :: type_name => threshold_driver_type_name
```

⟨*Prc Threshold: procedures*⟩+≡

```
function threshold_driver_type_name () result (type)
  type(string_t) :: type
  type = "Threshold"
end function threshold_driver_type_name
```

⟨*Prc Threshold: threshold driver: TBP*⟩+≡

```
procedure :: load => threshold_driver_load
```

⟨*Prc Threshold: procedures*⟩+≡

```
subroutine threshold_driver_load (threshold_driver, dlaccess)
  class(threshold_driver_t), intent(inout) :: threshold_driver
  type(dlaccess_t), intent(inout) :: dlaccess
  type(c_funptr) :: c_fptr
  type(string_t) :: lower_case_id
  call msg_debug (D_ME_METHODS, "threshold_driver_load")
  lower_case_id = lower_case (threshold_driver%id)
  c_fptr = dlaccess_get_c_funptr (dlaccess, lower_case_id // "_set_process_mode")
  call c_f_procpointer (c_fptr, threshold_driver%set_process_mode)
  call check_for_error (lower_case_id // "_set_process_mode")
```

```
    c_fptr = dlaccess_get_c_funptr (dlaccess, lower_case_id // "_get_amp_squared")
    call c_f_procpointer (c_fptr, threshold_driver%get_amp_squared)
    call check_for_error (lower_case_id // "_get_amp_squared")
    c_fptr = dlaccess_get_c_funptr (dlaccess, lower_case_id // "_threshold_init")
    call c_f_procpointer (c_fptr, threshold_driver%init)
    call check_for_error (lower_case_id // "_threshold_init")
    select type (threshold_driver)
    type is (threshold_driver_t)
       if (threshold_driver%nlo_type == NLO_VIRTUAL) then
          c_fptr = dlaccess_get_c_funptr (dlaccess, lower_case_id // "_start_openloops")
          call c_f_procpointer (c_fptr, threshold_driver%start_openloops)
          call check_for_error (lower_case_id // "_start_openloops")
          c_fptr = dlaccess_get_c_funptr (dlaccess, lower_case_id // "_olp_eval2")
          call c_f_procpointer (c_fptr, threshold_driver%olp_eval2)
          call check_for_error (lower_case_id // "_olp_eval2")
       end if
    end select
    call msg_message ("Loaded extra threshold functions")
    contains
      subroutine check_for_error (function_name)
        type(string_t), intent(in) :: function_name
        if (dlaccess_has_error (dlaccess)) &
             call msg_fatal (char ("Loading of " // function_name // " failed!"))
     end subroutine check_for_error
  end subroutine threshold_driver_load
```

Configuration

⟨*Prc Threshold: public*⟩+≡

```
  public :: threshold_def_t
```

⟨*Prc Threshold: types*⟩+≡

```
  type, extends (user_defined_def_t) :: threshold_def_t
     integer :: nlo_type
  contains
  ⟨Prc Threshold: threshold def: TBP⟩
  end type threshold_def_t
```

⟨*Prc Threshold: threshold def: TBP*⟩≡

```
  procedure :: init => threshold_def_init
```

⟨*Prc Threshold: procedures*⟩+≡

```
  subroutine threshold_def_init (object, basename, model_name, &
       prt_in, prt_out, nlo_type, restrictions)
    class(threshold_def_t), intent(inout) :: object
    type(string_t), intent(in) :: basename, model_name
    type(string_t), dimension(:), intent(in) :: prt_in, prt_out
    integer, intent(in) :: nlo_type
    type(string_t), intent(in), optional :: restrictions
    call msg_debug (D_ME_METHODS, "threshold_def_init")
    object%basename = basename
    object%nlo_type = nlo_type
    allocate (threshold_writer_t :: object%writer)
```

```
    select type (writer => object%writer)
    type is (threshold_writer_t)
       call writer%init (model_name, prt_in, prt_out, restrictions)
       writer%nlo_type = nlo_type
    end select
  end subroutine threshold_def_init
```

⟨*Prc Threshold: threshold def: TBP*⟩+≡

```
  procedure, nopass :: type_string => threshold_def_type_string
```

⟨*Prc Threshold: procedures*⟩+≡

```
  function threshold_def_type_string () result (string)
    type(string_t) :: string
    string = "threshold computation"
  end function threshold_def_type_string
```

`write` and `read` could be put in the abstract version

⟨*Prc Threshold: threshold def: TBP*⟩+≡

```
  procedure :: write => threshold_def_write
```

⟨*Prc Threshold: procedures*⟩+≡

```
  subroutine threshold_def_write (object, unit)
    class(threshold_def_t), intent(in) :: object
    integer, intent(in) :: unit
  end subroutine threshold_def_write
```

⟨*Prc Threshold: threshold def: TBP*⟩+≡

```
  procedure :: read => threshold_def_read
```

⟨*Prc Threshold: procedures*⟩+≡

```
  subroutine threshold_def_read (object, unit)
    class(threshold_def_t), intent(out) :: object
    integer, intent(in) :: unit
  end subroutine threshold_def_read
```

⟨*Prc Threshold: threshold def: TBP*⟩+≡

```
  procedure :: allocate_driver => threshold_def_allocate_driver
```

⟨*Prc Threshold: procedures*⟩+≡

```
  subroutine threshold_def_allocate_driver (object, driver, basename)
    class(threshold_def_t), intent(in) :: object
    class(prc_core_driver_t), intent(out), allocatable :: driver
    type(string_t), intent(in) :: basename
    call msg_debug (D_ME_METHODS, "threshold_def_allocate_driver")
    if (.not. allocated (driver)) allocate (threshold_driver_t :: driver)
    select type (driver)
    type is (threshold_driver_t)
       driver%id = basename
       driver%nlo_type = object%nlo_type
    end select
  end subroutine threshold_def_allocate_driver
```

⟨*Prc Threshold: threshold def: TBP*⟩+≡

```
  procedure :: connect => threshold_def_connect
```

⟨*Prc Threshold: procedures*⟩+≡

```
subroutine threshold_def_connect (def, lib_driver, i, proc_driver)
  class(threshold_def_t), intent(in) :: def
  class(prclib_driver_t), intent(in) :: lib_driver
  integer, intent(in) :: i
  class(prc_core_driver_t), intent(inout) :: proc_driver
  type(dlaccess_t) :: dlaccess
  logical :: skip
  call msg_debug (D_ME_METHODS, "threshold_def_connect")
  call def%omega_connect (lib_driver, i, proc_driver)
  select type (lib_driver)
  class is (prclib_driver_dynamic_t)
     dlaccess = lib_driver%dlaccess
  end select
  select type (proc_driver)
  class is (threshold_driver_t)
     select type (writer => def%writer)
     type is (threshold_writer_t)
        skip = writer%nlo_type == NLO_VIRTUAL .and. .not. writer%active
        if (.not. skip) call proc_driver%load (dlaccess)
     end select
  end select
end subroutine threshold_def_connect
```

Core state

⟨*Prc Threshold: public*⟩+≡

```
public :: threshold_state_t
```

⟨*Prc Threshold: types*⟩+≡

```
type, extends (user_defined_state_t) :: threshold_state_t
contains
⟨Prc Threshold: threshold state: TBP⟩
end type threshold_state_t
```

⟨*Prc Threshold: threshold state: TBP*⟩≡

```
procedure :: write => threshold_state_write
```

⟨*Prc Threshold: procedures*⟩+≡

```
subroutine threshold_state_write (object, unit)
  class(threshold_state_t), intent(in) :: object
  integer, intent(in), optional :: unit
end subroutine threshold_state_write
```

Core

⟨*Prc Threshold: public*⟩+≡

```
public :: prc_threshold_t
```

⟨*Prc Threshold: types*⟩+≡

```
type, extends (prc_user_defined_base_t) :: prc_threshold_t
   real(default), dimension(:,:), allocatable :: parray_ofs
```

```
   real(default), dimension(:,:), allocatable :: parray_ons
   integer :: leg
contains
⟨Prc Threshold: prc threshold: TBP⟩
end type prc_threshold_t
```

⟨*Prc Threshold: prc threshold: TBP*⟩≡

```
procedure :: write => prc_threshold_write
```

⟨*Prc Threshold: procedures*⟩+≡

```
subroutine prc_threshold_write (object, unit)
  class(prc_threshold_t), intent(in) :: object
  integer, intent(in), optional :: unit
  call msg_message ("Supply amplitudes squared for threshold computation")
end subroutine prc_threshold_write
```

⟨*Prc Threshold: prc threshold: TBP*⟩+≡

```
procedure :: write_name => prc_threshold_write_name
```

⟨*Prc Threshold: procedures*⟩+≡

```
subroutine prc_threshold_write_name (object, unit)
  class(prc_threshold_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u,"(1x,A)") "Core: Threshold"
end subroutine prc_threshold_write_name
```

⟨*Prc Threshold: prc threshold: TBP*⟩+≡

```
procedure :: compute_amplitude => prc_threshold_compute_amplitude
```

⟨*Prc Threshold: procedures*⟩+≡

```
function prc_threshold_compute_amplitude &
     (object, j, p, f, h, c, fac_scale, ren_scale, alpha_qcd_forced, &
     core_state)  result (amp)
  class(prc_threshold_t), intent(in) :: object
  integer, intent(in) :: j
  type(vector4_t), dimension(:), intent(in) :: p
  integer, intent(in) :: f, h, c
  real(default), intent(in) :: fac_scale, ren_scale
  real(default), intent(in), allocatable :: alpha_qcd_forced
  class(prc_core_state_t), intent(inout), allocatable, optional :: core_state
  complex(default) :: amp
  select type (core_state)
  class is (user_defined_test_state_t)
     core_state%alpha_qcd = object%qcd%alpha%get (fac_scale)
  end select
  amp = 0
end function prc_threshold_compute_amplitude
```

⟨*Prc Threshold: prc threshold: TBP*⟩+≡

```
procedure :: allocate_workspace => prc_threshold_allocate_workspace
```

⟨*Prc Threshold: procedures*⟩+≡

```
subroutine prc_threshold_allocate_workspace (object, core_state)
  class(prc_threshold_t), intent(in) :: object
  class(prc_core_state_t), intent(inout), allocatable :: core_state
  allocate (threshold_state_t :: core_state)
end subroutine prc_threshold_allocate_workspace
```

⟨*Prc Threshold: prc threshold: TBP*⟩+≡

```
procedure :: set_offshell_momenta => prc_threshold_set_offshell_momenta
```

⟨*Prc Threshold: procedures*⟩+≡

```
subroutine prc_threshold_set_offshell_momenta (object, p)
  class(prc_threshold_t), intent(inout) :: object
  type(vector4_t), intent(in), dimension(:) :: p
  integer :: i
  do i = 1, size(p)
     object%parray_ofs(:,i) = p(i)%p
  end do
end subroutine prc_threshold_set_offshell_momenta
```

⟨*Prc Threshold: prc threshold: TBP*⟩+≡

```
procedure :: set_onshell_momenta => prc_threshold_set_onshell_momenta
```

⟨*Prc Threshold: procedures*⟩+≡

```
subroutine prc_threshold_set_onshell_momenta (object, p)
  class(prc_threshold_t), intent(inout) :: object
  type(vector4_t), intent(in), dimension(:) :: p
  integer :: i
  do  i = 1, size(p)
     object%parray_ons(:,i) = p(i)%p
  end do
end subroutine prc_threshold_set_onshell_momenta
```

⟨*Prc Threshold: prc threshold: TBP*⟩+≡

```
procedure :: set_leg => prc_threshold_set_leg
```

⟨*Prc Threshold: procedures*⟩+≡

```
subroutine prc_threshold_set_leg (object, leg)
  class(prc_threshold_t), intent(inout) :: object
  integer, intent(in) :: leg
  object%leg = leg
end subroutine prc_threshold_set_leg
```

⟨*Prc Threshold: prc threshold: TBP*⟩+≡

```
procedure :: set_process_mode => prc_threshold_set_process_mode
```

⟨*Prc Threshold: procedures*⟩+≡

```
subroutine prc_threshold_set_process_mode (object, mode)
  class(prc_threshold_t), intent(in) :: object
  integer(kind = c_int), intent(in) :: mode
  select type (driver => object%driver)
  class is (threshold_driver_t)
     if (associated (driver%set_process_mode)) &
          call driver%set_process_mode (mode)
```

```
    end select
  end subroutine prc_threshold_set_process_mode
```

⟨*Prc Threshold: prc threshold: TBP*⟩+≡

```
  procedure :: compute_sqme => prc_threshold_compute_sqme
```

⟨*Prc Threshold: procedures*⟩+≡

```
  subroutine prc_threshold_compute_sqme (object, i_flv, p, &
        ren_scale, sqme, bad_point)
    class(prc_threshold_t), intent(in) :: object
    integer, intent(in) :: i_flv
    type(vector4_t), intent(in), dimension(:) :: p
    real(default), intent(in) :: ren_scale
    real(default), intent(out) :: sqme
    logical, intent(out) :: bad_point
    integer :: n_tot
    call msg_debug2 (D_ME_METHODS, "prc_threshold_compute_sqme")
    n_tot = size (p)
    select type (driver => object%driver)
    class is (threshold_driver_t)
       call driver%get_amp_squared (sqme, object%parray_ofs, object%parray_ons, object%leg, n_tot)
    end select
    bad_point = .false.
  end subroutine prc_threshold_compute_sqme
```

⟨*Prc Threshold: prc threshold: TBP*⟩+≡

```
  procedure :: compute_sqme_virt => prc_threshold_compute_sqme_virt
```

⟨*Prc Threshold: procedures*⟩+≡

```
  subroutine prc_threshold_compute_sqme_virt (object, i_flv, &
        p, ren_scale, sqme, bad_point)
    class(prc_threshold_t), intent(in) :: object
    integer, intent(in) :: i_flv
    type(vector4_t), dimension(:), intent(in) :: p
    real(default), intent(in) :: ren_scale
    real(default), dimension(4), intent(out) :: sqme
    real(c_default_float), dimension(:,:), allocatable, save :: parray
    logical, intent(out) :: bad_point
    integer :: n_tot, i
    real(default) :: mu
    real(c_default_float), dimension(4) :: sqme_c
    real(c_default_float) :: mu_c, acc_c, alpha_s_c
    integer(c_int) :: i_flv_c
    call msg_debug2 (D_ME_METHODS, "prc_threshold_compute_sqme_virt")
    n_tot = size (p)
    if (allocated (parray)) then
       if (size(parray) /= n_tot) deallocate (parray)
    end if
    if (.not. allocated (parray))  allocate (parray (0:3, n_tot))
    forall (i = 1:n_tot)  parray(:,i) = p(i)%p

    if (vanishes (ren_scale)) then
      mu = sqrt (2* (p(1)*p(2)))
    else
```

```
      mu = ren_scale
    end if
    mu_c = mu
    alpha_s_c = object%qcd%alpha%get (mu)
    i_flv_c = i_flv
    select type (driver => object%driver)
    class is (threshold_driver_t)
       if (associated (driver%olp_eval2)) then
          call driver%olp_eval2 (i_flv_c, alpha_s_c, &
             parray, mu_c, sqme_c, acc_c)
          bad_point = real(acc_c, kind=default) > object%maximum_accuracy
          sqme = sqme_c
       else
          sqme = 0._default
          bad_point = .true.
       end if
    end select
  end subroutine prc_threshold_compute_sqme_virt
```

⟨*Prc Threshold: prc threshold: TBP*⟩+≡

```
  procedure :: init => prc_threshold_init
```

⟨*Prc Threshold: procedures*⟩+≡

```
  subroutine prc_threshold_init (object, def, lib, id, i_component)
    class(prc_threshold_t), intent(inout) :: object
    class(prc_core_def_t), intent(in), target :: def
    type(process_library_t), intent(in), target :: lib
    type(string_t), intent(in) :: id
    integer, intent(in) :: i_component
    integer :: n_tot
    call object%base_init (def, lib, id, i_component)
    n_tot = object%data%n_in + object%data%n_out
    allocate (object%parray_ofs (0:3,n_tot), object%parray_ons (0:3,n_tot))
    if (n_tot == 4) then
       call object%set_process_mode (PROC_MODE_TT)
    else
       call object%set_process_mode (PROC_MODE_WBWB)
    end if
    call object%activate_parameters ()
  end subroutine prc_threshold_init
```

Activate the stored parameters by transferring them to the external matrix element.

⟨*Prc Threshold: prc threshold: TBP*⟩+≡

```
  procedure :: activate_parameters => prc_threshold_activate_parameters
```

⟨*Prc Threshold: procedures*⟩+≡

```
  subroutine prc_threshold_activate_parameters (object)
    class (prc_threshold_t), intent(inout) :: object
    call msg_debug (D_ME_METHODS, "prc_threshold_activate_parameters")
    if (allocated (object%driver)) then
       if (allocated (object%par)) then
          select type (driver => object%driver)
          type is (threshold_driver_t)
```

```
            if (associated (driver%init)) then
               call driver%init (object%par, object%scheme)
            end if
         end select
      else
         call msg_bug ("prc_threshold_activate: parameter set is not allocated")
      end if
   else
      call msg_bug ("prc_threshold_activate: driver is not allocated")
   end if
end subroutine prc_threshold_activate_parameters
```

⟨*Prc Threshold: prc threshold: TBP*⟩+≡

```
procedure :: create_and_load_extra_libraries => prc_threshold_create_and_load_extra_libraries
```

⟨*Prc Threshold: procedures*⟩+≡

```
subroutine prc_threshold_create_and_load_extra_libraries ( &
       core, flv_states, var_list, os_data, libname, model, i_core)
  class(prc_threshold_t), intent(inout) :: core
  integer, intent(in), dimension(:,:), allocatable :: flv_states
  type(var_list_t), intent(in) :: var_list
  type(os_data_t), intent(in) :: os_data
  type(string_t), intent(in) :: libname
  type(model_data_t), intent(in), target :: model
  integer, intent(in) :: i_core
  call msg_debug (D_ME_METHODS, "prc_threshold_create_and_load_extra_libraries")
  if (allocated (core%driver)) then
     select type (driver => core%driver)
     type is (threshold_driver_t)
        if (driver%nlo_type == NLO_VIRTUAL) call driver%start_openloops ()
     end select
  else
     call msg_bug ("prc_threshold_create_and_load_extra_libraries: driver is not allocated")
  end if
end subroutine prc_threshold_create_and_load_extra_libraries
```

⟨*Prc Threshold: prc threshold: TBP*⟩+≡

```
procedure :: includes_polarization => prc_threshold_includes_polarization
```

⟨*Prc Threshold: procedures*⟩+≡

```
function prc_threshold_includes_polarization (object) result (polarized)
  logical :: polarized
  class(prc_threshold_t), intent(in) :: object
  polarized = .false.
end function prc_threshold_includes_polarization
```

Chapter 17

Generic Event Handling

Event records allow the MC to communicate with the outside world. The event record should exhibit the observable contents of a physical event. We should be able to read and write events. The actual implementation of the event need not be defined yet, for that purpose.

We have the following basic modules:

event_base Abstract base type for event records. The base type contains a reference to a `particle_set_t` object as the event core, and it holds some data that we should always expect, such as the squared matrix element and event weight.

eio_data Transparent container for the metadata of an event sample.

eio_base Abstract base type for event-record input and output. The implementations of this base type represent specific event I/O formats.

These are the implementation modules:

eio_checkpoints Auxiliary output format. The only purpose is to provide screen diagnostics during event output.

eio_callback Auxiliary output format. The only purpose is to execute a callback procedure, so we have a hook for external access during event output.

eio_weights Print some event summary data, no details. The main use if for testing purposes.

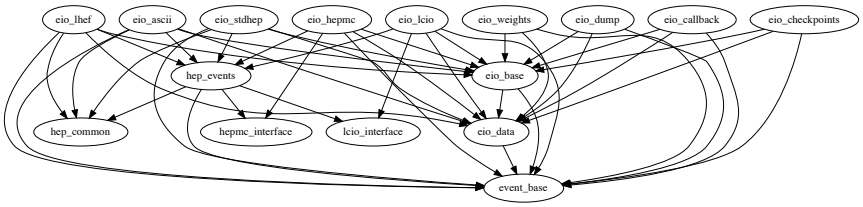

Figure 17.1: Module dependencies in `src/events`.

eio_dump Dump the contents of WHIZARD's `particle_set` internal record, using the `write` method of that record as-is. The main use if for testing purposes.

hep_common Implements traditional HEP common blocks that are (still) used by some of the event I/O formats below.

hepmc_interface Access particle objects of the HepMC package. Functional only if this package is linked.

lcio_interface Access objects of the LCIO package. Functional only if this package is linked.

hep_events Interface between the event record and the common blocks.

eio_ascii Collection of event output formats that write ASCII files.

eio_lhef LHEF for input and output.

eio_stdhep Support for the StdHEP format (binary, machine-independent).

eio_hepmc Support for the HepMC format (C++).

eio_lcio Support for the LCIO format (C++).

17.1 Generic Event Handling

We introduce events first in form of an abstract type, together with some utilities. Abstract events can be used by other modules, in particular event I/O, without introducing an explicit dependency on the event implementation.

⟨`event_base.f90`⟩≡

```
  ⟨File header⟩

  module event_base

    use system_defs, only: LF
  ⟨Use kinds⟩
    use kinds, only: i64
  ⟨Use strings⟩
    use io_units
    use string_utils, only: lower_case
    use diagnostics
    use model_data
    use particles

  ⟨Standard module head⟩

  ⟨Event base: public⟩

  ⟨Event base: parameters⟩

  ⟨Event base: types⟩

  ⟨Event base: interfaces⟩
```

```
contains

⟨Event base: procedures⟩

end module event_base
```

17.1.1 generic event type

⟨*Event base: public*⟩≡

```
public :: generic_event_t
```

⟨*Event base: types*⟩≡

```
type, abstract :: generic_event_t
   !private
   logical :: particle_set_is_valid = .false.
   type(particle_set_t), pointer :: particle_set => null ()
   logical :: sqme_ref_known = .false.
   real(default) :: sqme_ref = 0
   logical :: sqme_prc_known = .false.
   real(default) :: sqme_prc = 0
   logical :: weight_ref_known = .false.
   real(default) :: weight_ref = 0
   logical :: weight_prc_known = .false.
   real(default) :: weight_prc = 0
   logical :: excess_prc_known = .false.
   real(default) :: excess_prc = 0
   integer :: n_alt = 0
   logical :: sqme_alt_known = .false.
   real(default), dimension(:), allocatable :: sqme_alt
   logical :: weight_alt_known = .false.
   real(default), dimension(:), allocatable :: weight_alt
 contains
 ⟨Event base: generic event: TBP⟩
end type generic_event_t
```

17.1.2 Initialization

This determines the number of alternate weights and sqme values.

⟨*Event base: generic event: TBP*⟩≡

```
procedure :: base_init => generic_event_init
```

⟨*Event base: procedures*⟩≡

```
subroutine generic_event_init (event, n_alt)
  class(generic_event_t), intent(out) :: event
  integer, intent(in) :: n_alt
  event%n_alt = n_alt
  allocate (event%sqme_alt (n_alt))
  allocate (event%weight_alt (n_alt))
end subroutine generic_event_init
```

17.1.3 Access particle set

The particle set is the core of the event. We allow access to it via a pointer, and we maintain the information whether the particle set is valid, i.e., has been filled with meaningful data.

⟨*Event base: generic event: TBP*⟩+≡

```
procedure :: has_valid_particle_set => generic_event_has_valid_particle_set
procedure :: accept_particle_set => generic_event_accept_particle_set
procedure :: discard_particle_set => generic_event_discard_particle_set
```

⟨*Event base: procedures*⟩+≡

```
function generic_event_has_valid_particle_set (event) result (flag)
  class(generic_event_t), intent(in) :: event
  logical :: flag
  flag = event%particle_set_is_valid
end function generic_event_has_valid_particle_set

subroutine generic_event_accept_particle_set (event)
  class(generic_event_t), intent(inout) :: event
  event%particle_set_is_valid = .true.
end subroutine generic_event_accept_particle_set

subroutine generic_event_discard_particle_set (event)
  class(generic_event_t), intent(inout) :: event
  event%particle_set_is_valid = .false.
end subroutine generic_event_discard_particle_set
```

These procedures deal with the particle set directly. Return the pointer:

⟨*Event base: generic event: TBP*⟩+≡

```
procedure :: get_particle_set_ptr => generic_event_get_particle_set_ptr
```

⟨*Event base: procedures*⟩+≡

```
function generic_event_get_particle_set_ptr (event) result (ptr)
  class(generic_event_t), intent(in) :: event
  type(particle_set_t), pointer :: ptr
  ptr => event%particle_set
end function generic_event_get_particle_set_ptr
```

Let it point to some existing particle set:

⟨*Event base: generic event: TBP*⟩+≡

```
procedure :: link_particle_set => generic_event_link_particle_set
```

⟨*Event base: procedures*⟩+≡

```
subroutine generic_event_link_particle_set (event, particle_set)
  class(generic_event_t), intent(inout) :: event
  type(particle_set_t), intent(in), target :: particle_set
  event%particle_set => particle_set
  call event%accept_particle_set ()
end subroutine generic_event_link_particle_set
```

17.1.4 Access sqme and weight

There are several incarnations: the current value, a reference value, alternate values.

⟨*Event base: generic event: TBP*⟩+≡

```
procedure :: sqme_prc_is_known => generic_event_sqme_prc_is_known
procedure :: sqme_ref_is_known => generic_event_sqme_ref_is_known
procedure :: sqme_alt_is_known => generic_event_sqme_alt_is_known
procedure :: weight_prc_is_known => generic_event_weight_prc_is_known
procedure :: weight_ref_is_known => generic_event_weight_ref_is_known
procedure :: weight_alt_is_known => generic_event_weight_alt_is_known
procedure :: excess_prc_is_known => generic_event_excess_prc_is_known
```

⟨*Event base: procedures*⟩+≡

```
function generic_event_sqme_prc_is_known (event) result (flag)
  class(generic_event_t), intent(in) :: event
  logical :: flag
  flag = event%sqme_prc_known
end function generic_event_sqme_prc_is_known

function generic_event_sqme_ref_is_known (event) result (flag)
  class(generic_event_t), intent(in) :: event
  logical :: flag
  flag = event%sqme_ref_known
end function generic_event_sqme_ref_is_known

function generic_event_sqme_alt_is_known (event) result (flag)
  class(generic_event_t), intent(in) :: event
  logical :: flag
  flag = event%sqme_alt_known
end function generic_event_sqme_alt_is_known

function generic_event_weight_prc_is_known (event) result (flag)
  class(generic_event_t), intent(in) :: event
  logical :: flag
  flag = event%weight_prc_known
end function generic_event_weight_prc_is_known

function generic_event_weight_ref_is_known (event) result (flag)
  class(generic_event_t), intent(in) :: event
  logical :: flag
  flag = event%weight_ref_known
end function generic_event_weight_ref_is_known

function generic_event_weight_alt_is_known (event) result (flag)
  class(generic_event_t), intent(in) :: event
  logical :: flag
  flag = event%weight_alt_known
end function generic_event_weight_alt_is_known

function generic_event_excess_prc_is_known (event) result (flag)
  class(generic_event_t), intent(in) :: event
  logical :: flag
  flag = event%excess_prc_known
end function generic_event_excess_prc_is_known
```

⟨*Event base: generic event: TBP*⟩+≡

```
procedure :: get_n_alt => generic_event_get_n_alt
```

⟨*Event base: procedures*⟩+≡

```
function generic_event_get_n_alt (event) result (n)
  class(generic_event_t), intent(in) :: event
  integer :: n
  n = event%n_alt
end function generic_event_get_n_alt
```

⟨*Event base: generic event: TBP*⟩+≡

```
procedure :: get_sqme_prc => generic_event_get_sqme_prc
procedure :: get_sqme_ref => generic_event_get_sqme_ref
generic :: get_sqme_alt => &
     generic_event_get_sqme_alt_0, generic_event_get_sqme_alt_1
procedure :: generic_event_get_sqme_alt_0
procedure :: generic_event_get_sqme_alt_1
procedure :: get_weight_prc => generic_event_get_weight_prc
procedure :: get_weight_ref => generic_event_get_weight_ref
generic :: get_weight_alt => &
     generic_event_get_weight_alt_0, generic_event_get_weight_alt_1
procedure :: generic_event_get_weight_alt_0
procedure :: generic_event_get_weight_alt_1
procedure :: get_excess_prc => generic_event_get_excess_prc
```

⟨*Event base: procedures*⟩+≡

```
function generic_event_get_sqme_prc (event) result (sqme)
  class(generic_event_t), intent(in) :: event
  real(default) :: sqme
  if (event%sqme_prc_known) then
     sqme = event%sqme_prc
  else
     sqme = 0
  end if
end function generic_event_get_sqme_prc

function generic_event_get_sqme_ref (event) result (sqme)
  class(generic_event_t), intent(in) :: event
  real(default) :: sqme
  if (event%sqme_ref_known) then
     sqme = event%sqme_ref
  else
     sqme = 0
  end if
end function generic_event_get_sqme_ref

function generic_event_get_sqme_alt_0 (event, i) result (sqme)
  class(generic_event_t), intent(in) :: event
  integer, intent(in) :: i
  real(default) :: sqme
  if (event%sqme_alt_known) then
     sqme = event%sqme_alt(i)
  else
```

```
      sqme = 0
   end if
end function generic_event_get_sqme_alt_0

function generic_event_get_sqme_alt_1 (event) result (sqme)
   class(generic_event_t), intent(in) :: event
   real(default), dimension(event%n_alt) :: sqme
   sqme = event%sqme_alt
end function generic_event_get_sqme_alt_1

function generic_event_get_weight_prc (event) result (weight)
   class(generic_event_t), intent(in) :: event
   real(default) :: weight
   if (event%weight_prc_known) then
      weight = event%weight_prc
   else
      weight = 0
   end if
end function generic_event_get_weight_prc

function generic_event_get_weight_ref (event) result (weight)
   class(generic_event_t), intent(in) :: event
   real(default) :: weight
   if (event%weight_ref_known) then
      weight = event%weight_ref
   else
      weight = 0
   end if
end function generic_event_get_weight_ref

function generic_event_get_weight_alt_0 (event, i) result (weight)
   class(generic_event_t), intent(in) :: event
   integer, intent(in) :: i
   real(default) :: weight
   if (event%weight_alt_known) then
      weight = event%weight_alt(i)
   else
      weight = 0
   end if
end function generic_event_get_weight_alt_0

function generic_event_get_weight_alt_1 (event) result (weight)
   class(generic_event_t), intent(in) :: event
   real(default), dimension(event%n_alt) :: weight
   weight = event%weight_alt
end function generic_event_get_weight_alt_1

function generic_event_get_excess_prc (event) result (excess)
   class(generic_event_t), intent(in) :: event
   real(default) :: excess
   if (event%excess_prc_known) then
      excess = event%excess_prc
   else
      excess = 0
```

```
    end if
  end function generic_event_get_excess_prc
```

⟨*Event base: generic event: TBP*⟩+≡

```
  procedure :: set_sqme_prc => generic_event_set_sqme_prc
  procedure :: set_sqme_ref => generic_event_set_sqme_ref
  procedure :: set_sqme_alt => generic_event_set_sqme_alt
  procedure :: set_weight_prc => generic_event_set_weight_prc
  procedure :: set_weight_ref => generic_event_set_weight_ref
  procedure :: set_weight_alt => generic_event_set_weight_alt
  procedure :: set_excess_prc => generic_event_set_excess_prc
```

⟨*Event base: procedures*⟩+≡

```
  subroutine generic_event_set_sqme_prc (event, sqme)
    class(generic_event_t), intent(inout) :: event
    real(default), intent(in) :: sqme
    event%sqme_prc = sqme
    event%sqme_prc_known = .true.
  end subroutine generic_event_set_sqme_prc

  subroutine generic_event_set_sqme_ref (event, sqme)
    class(generic_event_t), intent(inout) :: event
    real(default), intent(in) :: sqme
    event%sqme_ref = sqme
    event%sqme_ref_known = .true.
  end subroutine generic_event_set_sqme_ref

  subroutine generic_event_set_sqme_alt (event, sqme)
    class(generic_event_t), intent(inout) :: event
    real(default), dimension(:), intent(in) :: sqme
    event%sqme_alt = sqme
    event%sqme_alt_known = .true.
  end subroutine generic_event_set_sqme_alt

  subroutine generic_event_set_weight_prc (event, weight)
    class(generic_event_t), intent(inout) :: event
    real(default), intent(in) :: weight
    event%weight_prc = weight
    event%weight_prc_known = .true.
  end subroutine generic_event_set_weight_prc

  subroutine generic_event_set_weight_ref (event, weight)
    class(generic_event_t), intent(inout) :: event
    real(default), intent(in) :: weight
    event%weight_ref = weight
    event%weight_ref_known = .true.
  end subroutine generic_event_set_weight_ref

  subroutine generic_event_set_weight_alt (event, weight)
    class(generic_event_t), intent(inout) :: event
    real(default), dimension(:), intent(in) :: weight
    event%weight_alt = weight
    event%weight_alt_known = .true.
  end subroutine generic_event_set_weight_alt
```

```
subroutine generic_event_set_excess_prc (event, excess)
  class(generic_event_t), intent(inout) :: event
  real(default), intent(in) :: excess
  event%excess_prc = excess
  event%excess_prc_known = .true.
end subroutine generic_event_set_excess_prc
```

Set the appropriate entry directly.

⟨*Event base: generic event: TBP*⟩+≡

```
procedure :: set => generic_event_set
```

⟨*Event base: procedures*⟩+≡

```
subroutine generic_event_set (event, &
     weight_ref, weight_prc, weight_alt, &
     excess_prc, &
     sqme_ref, sqme_prc, sqme_alt)
  class(generic_event_t), intent(inout) :: event
  real(default), intent(in), optional :: weight_ref, weight_prc
  real(default), intent(in), optional :: sqme_ref, sqme_prc
  real(default), dimension(:), intent(in), optional :: sqme_alt, weight_alt
  real(default), intent(in), optional :: excess_prc
  if (present (sqme_prc)) then
     call event%set_sqme_prc (sqme_prc)
  end if
  if (present (sqme_ref)) then
     call event%set_sqme_ref (sqme_ref)
  end if
  if (present (sqme_alt)) then
     call event%set_sqme_alt (sqme_alt)
  end if
  if (present (weight_prc)) then
     call event%set_weight_prc (weight_prc)
  end if
  if (present (weight_ref)) then
     call event%set_weight_ref (weight_ref)
  end if
  if (present (weight_alt)) then
     call event%set_weight_alt (weight_alt)
  end if
  if (present (excess_prc)) then
     call event%set_excess_prc (excess_prc)
  end if
end subroutine generic_event_set
```

17.1.5 Pure Virtual Methods

These procedures can only implemented in the concrete implementation.

Output (verbose, depending on parameters).

⟨*Event base: generic event: TBP*⟩+≡

```
procedure (generic_event_write), deferred :: write
```

⟨*Event base: interfaces*⟩≡

```
abstract interface
   subroutine generic_event_write (object, unit, &
        show_process, show_transforms, &
        show_decay, verbose, testflag)
     import
     class(generic_event_t), intent(in) :: object
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: show_process
     logical, intent(in), optional :: show_transforms
     logical, intent(in), optional :: show_decay
     logical, intent(in), optional :: verbose
     logical, intent(in), optional :: testflag
   end subroutine generic_event_write
end interface
```

Generate an event, based on a selector index `i_mci`, and optionally on an extra set of random numbers `r`. For the main bunch of random numbers that the generator needs, the event object should contain its own generator.

⟨*Event base: generic event: TBP*⟩+≡

```
procedure (generic_event_generate), deferred :: generate
```

⟨*Event base: interfaces*⟩+≡

```
abstract interface
   subroutine generic_event_generate (event, i_mci, r, i_nlo)
     import
     class(generic_event_t), intent(inout) :: event
     integer, intent(in) :: i_mci
     real(default), dimension(:), intent(in), optional :: r
     integer, intent(in), optional :: i_nlo
   end subroutine generic_event_generate
end interface
```

Alternative : inject a particle set that is supposed to represent the hard process. How this determines the event, is dependent on the event structure, therefore this is a deferred method.

⟨*Event base: generic event: TBP*⟩+≡

```
procedure (generic_event_set_hard_particle_set), deferred :: &
     set_hard_particle_set
```

⟨*Event base: interfaces*⟩+≡

```
abstract interface
   subroutine generic_event_set_hard_particle_set (event, particle_set)
     import
     class(generic_event_t), intent(inout) :: event
     type(particle_set_t), intent(in) :: particle_set
   end subroutine generic_event_set_hard_particle_set
end interface
```

Evaluate any expressions associated with the event. No argument needed.

⟨*Event base: generic event: TBP*⟩+≡

```
procedure (generic_event_handler), deferred :: evaluate_expressions
```

⟨*Event base: interfaces*⟩+≡

```
abstract interface
   subroutine generic_event_handler (event)
     import
     class(generic_event_t), intent(inout) :: event
   end subroutine generic_event_handler
end interface
```

Select internal parameters

⟨*Event base: generic event: TBP*⟩+≡

```
procedure (generic_event_select), deferred :: select
```

⟨*Event base: interfaces*⟩+≡

```
abstract interface
   subroutine generic_event_select (event,  i_mci, i_term, channel)
     import
     class(generic_event_t), intent(inout) :: event
     integer, intent(in) :: i_mci, i_term, channel
   end subroutine generic_event_select
end interface
```

Return a pointer to the model for the currently active process.

⟨*Event base: generic event: TBP*⟩+≡

```
procedure (generic_event_get_model_ptr), deferred :: get_model_ptr
```

⟨*Event base: interfaces*⟩+≡

```
abstract interface
   function generic_event_get_model_ptr (event) result (model)
     import
     class(generic_event_t), intent(in) :: event
     class(model_data_t), pointer :: model
   end function generic_event_get_model_ptr
end interface
```

Return data used by external event formats.

⟨*Event base: generic event: TBP*⟩+≡

```
procedure (generic_event_get_index), deferred :: get_index
procedure (generic_event_get_fac_scale), deferred :: get_fac_scale
procedure (generic_event_get_alpha_s), deferred :: get_alpha_s
procedure (generic_event_get_sqrts), deferred :: get_sqrts
procedure (generic_event_get_polarization), deferred :: get_polarization
procedure (generic_event_get_beam_file), deferred :: get_beam_file
procedure (generic_event_get_process_name), deferred :: &
     get_process_name
```

⟨*Event base: interfaces*⟩+≡

```
abstract interface
   function generic_event_get_index (event) result (index)
     import
     class(generic_event_t), intent(in) :: event
     integer :: index
   end function generic_event_get_index
end interface
```

```
abstract interface
   function generic_event_get_fac_scale (event) result (fac_scale)
     import
     class(generic_event_t), intent(in) :: event
     real(default) :: fac_scale
   end function generic_event_get_fac_scale
end interface

abstract interface
   function generic_event_get_alpha_s (event) result (alpha_s)
     import
     class(generic_event_t), intent(in) :: event
     real(default) :: alpha_s
   end function generic_event_get_alpha_s
end interface

abstract interface
   function generic_event_get_sqrts (event) result (sqrts)
     import
     class(generic_event_t), intent(in) :: event
     real(default) :: sqrts
   end function generic_event_get_sqrts
end interface

abstract interface
   function generic_event_get_polarization (event) result (pol)
     import
     class(generic_event_t), intent(in) :: event
     real(default), dimension(2) :: pol
   end function generic_event_get_polarization
end interface

abstract interface
   function generic_event_get_beam_file (event) result (file)
     import
     class(generic_event_t), intent(in) :: event
     type(string_t) :: file
   end function generic_event_get_beam_file
end interface

abstract interface
   function generic_event_get_process_name (event) result (name)
     import
     class(generic_event_t), intent(in) :: event
     type(string_t) :: name
   end function generic_event_get_process_name
end interface
```

Set data used by external event formats.

⟨*Event base: generic event: TBP*⟩+≡

```
procedure (generic_event_set_alpha_qcd_forced), deferred :: &
     set_alpha_qcd_forced
procedure (generic_event_set_scale_forced), deferred :: &
```

```
      set_scale_forced
```

⟨*Event base: interfaces*⟩+≡

```
abstract interface
   subroutine generic_event_set_alpha_qcd_forced (event, alpha_qcd)
     import
     class(generic_event_t), intent(inout) :: event
     real(default), intent(in) :: alpha_qcd
   end subroutine generic_event_set_alpha_qcd_forced
end interface

abstract interface
   subroutine generic_event_set_scale_forced (event, scale)
     import
     class(generic_event_t), intent(inout) :: event
     real(default), intent(in) :: scale
   end subroutine generic_event_set_scale_forced
end interface
```

17.1.6 Utilities

Applying this, current event contents are marked as incomplete but are not deleted. In particular, the initialization is kept.

⟨*Event base: generic event: TBP*⟩+≡

```
procedure :: reset => generic_event_reset
procedure :: base_reset => generic_event_reset
```

⟨*Event base: procedures*⟩+≡

```
subroutine generic_event_reset (event)
  class(generic_event_t), intent(inout) :: event
  call event%discard_particle_set ()
  event%sqme_ref_known = .false.
  event%sqme_prc_known = .false.
  event%sqme_alt_known = .false.
  event%weight_ref_known = .false.
  event%weight_prc_known = .false.
  event%weight_alt_known = .false.
  event%excess_prc_known = .false.
end subroutine generic_event_reset
```

Pacify particle set.

⟨*Event base: generic event: TBP*⟩+≡

```
procedure :: pacify_particle_set => generic_event_pacify_particle_set
```

⟨*Event base: procedures*⟩+≡

```
subroutine generic_event_pacify_particle_set (event)
  class(generic_event_t), intent(inout) :: event
  if (event%has_valid_particle_set ())  call pacify (event%particle_set)
end subroutine generic_event_pacify_particle_set
```

17.1.7 Event normalization

The parameters for event normalization. For unweighted events, NORM_UNIT is intended as default, while for weighted events, it is NORM_SIGMA.

Note: the unit test for this is in eio_data_2 below.

⟨*Event base: parameters*⟩≡

```
integer, parameter, public :: NORM_UNDEFINED = 0
integer, parameter, public :: NORM_UNIT = 1
integer, parameter, public :: NORM_N_EVT = 2
integer, parameter, public :: NORM_SIGMA = 3
integer, parameter, public :: NORM_S_N = 4
```

These functions translate between the user representation and the internal one.

⟨*Event base: public*⟩+≡

```
public :: event_normalization_mode
public :: event_normalization_string
```

⟨*Event base: procedures*⟩+≡

```
function event_normalization_mode (string, unweighted) result (mode)
  integer :: mode
  type(string_t), intent(in) :: string
  logical, intent(in) :: unweighted
  select case (lower_case (char (string)))
  case ("auto")
     if (unweighted) then
        mode = NORM_UNIT
     else
        mode = NORM_SIGMA
     end if
  case ("1")
     mode = NORM_UNIT
  case ("1/n")
     mode = NORM_N_EVT
  case ("sigma")
     mode = NORM_SIGMA
  case ("sigma/n")
     mode = NORM_S_N
  case default
     call msg_fatal ("Event normalization: unknown value '" &
          // char (string) // "'")
  end select
end function event_normalization_mode

function event_normalization_string (norm_mode) result (string)
  integer, intent(in) :: norm_mode
  type(string_t) :: string
  select case (norm_mode)
  case (NORM_UNDEFINED); string = "[undefined]"
  case (NORM_UNIT);      string = "'1'"
  case (NORM_N_EVT);     string = "'1/n'"
  case (NORM_SIGMA);     string = "'sigma'"
  case (NORM_S_N);       string = "'sigma/n'"
  case default;          string = "???"
  end select
```

```
end function event_normalization_string
```

We place this here as a generic helper, so we can update event weights whenever we need, not just in connection with an event sample data object.

⟨*Event base: public*⟩+≡

```
public :: event_normalization_update
```

⟨*Event base: procedures*⟩+≡

```
subroutine event_normalization_update (weight, sigma, n, mode_new, mode_old)
  real(default), intent(inout) :: weight
  real(default), intent(in) :: sigma
  integer, intent(in) :: n
  integer, intent(in) :: mode_new, mode_old
  if (mode_new /= mode_old) then
     if (sigma > 0 .and. n > 0) then
        weight = weight / factor (mode_old) * factor (mode_new)
     else
        call msg_fatal ("Event normalization update: null sample")
     end if
  end if
contains
  function factor (mode)
    real(default) :: factor
    integer, intent(in) :: mode
    select case (mode)
    case (NORM_UNIT);   factor = 1._default
    case (NORM_N_EVT);  factor = 1._default / n
    case (NORM_SIGMA);  factor = sigma
    case (NORM_S_N);    factor = sigma / n
    case default
       call msg_fatal ("Event normalization update: undefined mode")
    end select
  end function factor
end subroutine event_normalization_update
```

17.1.8 Callback container

This derived type contains a callback procedure that can be executed during event I/O. The callback procedure is given the event object (with class `generic_event`) and an event index.

This is a simple wrapper. The object is abstract, so the the actual procedure is introduced by overriding the deferred one. We use the PASS attribute, so we may supplement runtime data in the callback object if desired.

⟨*Event base: public*⟩+≡

```
public :: event_callback_t
```

⟨*Event base: types*⟩+≡

```
type, abstract :: event_callback_t
   private
 contains
   procedure(event_callback_write), deferred :: write
   procedure(event_callback_proc), deferred :: proc
```

```
end type event_callback_t
```

Identify the callback procedure in output

⟨*Event base: interfaces*⟩+≡

```
abstract interface
   subroutine event_callback_write (event_callback, unit)
     import
     class(event_callback_t), intent(in) :: event_callback
     integer, intent(in), optional :: unit
   end subroutine event_callback_write
end interface
```

This is the procedure interface.

⟨*Event base: interfaces*⟩+≡

```
abstract interface
   subroutine event_callback_proc (event_callback, i, event)
     import
     class(event_callback_t), intent(in) :: event_callback
     integer(i64), intent(in) :: i
     class(generic_event_t), intent(in) :: event
   end subroutine event_callback_proc
end interface
```

A dummy implementation for testing and fallback.

⟨*Event base: public*⟩+≡

```
public :: event_callback_nop_t
```

⟨*Event base: types*⟩+≡

```
type, extends (event_callback_t) :: event_callback_nop_t
   private
 contains
   procedure :: write => event_callback_nop_write
   procedure :: proc => event_callback_nop
end type event_callback_nop_t
```

⟨*Event base: procedures*⟩+≡

```
subroutine event_callback_nop_write (event_callback, unit)
  class(event_callback_nop_t), intent(in) :: event_callback
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "NOP"
end subroutine event_callback_nop_write

subroutine event_callback_nop (event_callback, i, event)
  class(event_callback_nop_t), intent(in) :: event_callback
  integer(i64), intent(in) :: i
  class(generic_event_t), intent(in) :: event
end subroutine event_callback_nop
```

17.2 Event Sample Data

We define a simple and transparent container for (meta)data that are associated with an event sample.

⟨eio_data.f90⟩≡
```
⟨File header⟩

module eio_data

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use numeric_utils
  use diagnostics

  use event_base

⟨Standard module head⟩

⟨EIO data: public⟩

⟨EIO data: types⟩

contains

⟨EIO data: procedures⟩

end module eio_data
```

17.2.1 Event Sample Data

These are data that apply to an event sample as a whole. They are given in an easily portable form (no fancy structure) and are used for initializing event formats.

There are two MD5 sums here. `md5sum_proc` depends only on the definition of the contributing processes. A sample with matching checksum can be rescanned with modified model parameters, beam structure etc, to recalculate observables. `md5sum_config` includes all relevant data. Rescanning a sample with matching checksum will produce identical observables. (A third checksum might be added which depends on the event sample itself. This is not needed, so far.)

If alternate weights are part of the event sample (`n_alt` nonzero), there is a configuration MD5 sum for each of them.

⟨EIO data: public⟩≡
```
  public :: event_sample_data_t
```

⟨EIO data: types⟩≡
```
  type :: event_sample_data_t
     character(32) :: md5sum_prc = ""
     character(32) :: md5sum_cfg = ""
     logical :: unweighted = .true.
     logical :: negative_weights = .false.
     integer :: norm_mode = NORM_UNDEFINED
```

```
    integer :: n_beam = 0
    integer, dimension(2) :: pdg_beam = 0
    real(default), dimension(2) :: energy_beam = 0
    integer :: n_proc = 0
    integer :: n_evt = 0
    integer :: nlo_multiplier = 1
    integer :: split_n_evt = 0
    integer :: split_n_kbytes = 0
    integer :: split_index = 0
    real(default) :: total_cross_section = 0
    integer, dimension(:), allocatable :: proc_num_id
    integer :: n_alt = 0
    character(32), dimension(:), allocatable :: md5sum_alt
    real(default), dimension(:), allocatable :: cross_section
    real(default), dimension(:), allocatable :: error
  contains
  ⟨EIO data: event sample data: TBP⟩
end type event_sample_data_t
```

Initialize: allocate for the number of processes

⟨*EIO data: event sample data: TBP*⟩≡

```
procedure :: init => event_sample_data_init
```

⟨*EIO data: procedures*⟩≡

```
subroutine event_sample_data_init (data, n_proc, n_alt)
  class(event_sample_data_t), intent(out) :: data
  integer, intent(in) :: n_proc
  integer, intent(in), optional :: n_alt
  data%n_proc = n_proc
  allocate (data%proc_num_id (n_proc), source = 0)
  allocate (data%cross_section (n_proc), source = 0._default)
  allocate (data%error (n_proc), source = 0._default)
  if (present (n_alt)) then
     data%n_alt = n_alt
     allocate (data%md5sum_alt (n_alt))
     data%md5sum_alt = ""
  end if
end subroutine event_sample_data_init
```

Output.

⟨*EIO data: event sample data: TBP*⟩+≡

```
procedure :: write => event_sample_data_write
```

⟨*EIO data: procedures*⟩+≡

```
subroutine event_sample_data_write (data, unit)
  class(event_sample_data_t), intent(in) :: data
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Event sample properties:"
  write (u, "(3x,A,A,A)")  "MD5 sum (proc)   = '", data%md5sum_prc, "'"
  write (u, "(3x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
  write (u, "(3x,A,L1)")  "unweighted       = ", data%unweighted
  write (u, "(3x,A,L1)")  "negative weights = ", data%negative_weights
```

```
    write (u, "(3x,A,A)")   "normalization    = ", &
         char (event_normalization_string (data%norm_mode))
    write (u, "(3x,A,I0)")  "number of beams  = ", data%n_beam
    write (u, "(5x,A,2(1x,I19))")  "PDG    = ", &
         data%pdg_beam(:data%n_beam)
    write (u, "(5x,A,2(1x,ES19.12))")  "Energy = ", &
         data%energy_beam(:data%n_beam)
    if (data%n_evt > 0) then
       write (u, "(3x,A,I0)")  "number of events = ", data%n_evt
    end if
    if (.not. vanishes (data%total_cross_section)) then
       write (u, "(3x,A,ES19.12)")  "total cross sec. = ", &
            data%total_cross_section
    end if
    write (u, "(3x,A,I0)")  "num of processes = ", data%n_proc
    do i = 1, data%n_proc
       write (u, "(3x,A,I0)")  "Process #", data%proc_num_id (i)
       select case (data%n_beam)
       case (1)
          write (u, "(5x,A,ES19.12)")  "Width = ", data%cross_section(i)
       case (2)
          write (u, "(5x,A,ES19.12)")  "CSec  = ", data%cross_section(i)
       end select
       write (u, "(5x,A,ES19.12)")  "Error = ", data%error(i)
    end do
    if (data%n_alt > 0) then
       write (u, "(3x,A,I0)")  "num of alt wgt   = ", data%n_alt
       do i = 1, data%n_alt
          write (u, "(5x,A,A,A,1x,I0)")  "MD5 sum (cfg)  = '", &
               data%md5sum_alt(i), "'", i
       end do
    end if
  end subroutine event_sample_data_write
```

17.2.2 Unit tests

Test module, followed by the corresponding implementation module.

⟨eio_data_ut.f90⟩≡

```
  ⟨File header⟩

  module eio_data_ut
    use unit_tests
    use eio_data_uti

  ⟨Standard module head⟩

  ⟨EIO data: public test⟩

  contains

  ⟨EIO data: test driver⟩
```

```
  end module eio_data_ut
```

⟨eio_data_uti.f90⟩≡

```
  ⟨File header⟩

  module eio_data_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use event_base

    use eio_data

  ⟨Standard module head⟩

  ⟨EIO data: test declarations⟩

  contains

  ⟨EIO data: tests⟩

  end module eio_data_uti
```

API: driver for the unit tests below.

⟨EIO data: public test⟩≡

```
    public :: eio_data_test
```

⟨EIO data: test driver⟩≡

```
    subroutine eio_data_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨EIO data: execute tests⟩
    end subroutine eio_data_test
```

Event Sample Data

Print the contents of a sample data block.

⟨EIO data: execute tests⟩≡

```
    call test (eio_data_1, "eio_data_1", &
         "event sample data", &
         u, results)
```

⟨EIO data: test declarations⟩≡

```
    public :: eio_data_1
```

⟨EIO data: tests⟩≡

```
    subroutine eio_data_1 (u)
      integer, intent(in) :: u
      type(event_sample_data_t) :: data

      write (u, "(A)")  "* Test output: eio_data_1"
      write (u, "(A)")  "*   Purpose:  display event sample data"
      write (u, "(A)")

      write (u, "(A)")  "* Decay process, one component"
```

```
    write (u, "(A)")

    call data%init (1, 1)
    data%n_beam = 1
    data%pdg_beam(1) = 25
    data%energy_beam(1) = 125

    data%norm_mode = NORM_UNIT

    data%proc_num_id = [42]
    data%cross_section = [1.23e-4_default]
    data%error = 5e-6_default

    data%md5sum_prc = "abcdefghijklmnopabcdefghijklmnop"
    data%md5sum_cfg = "12345678901234561234567890123456"
    data%md5sum_alt(1) = "uuuuuuuuuuuuuuuuuuuuuuuuuuuuuuuu"

    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Scattering process, two components"
    write (u, "(A)")

    call data%init (2)
    data%n_beam = 2
    data%pdg_beam = [2212, -2212]
    data%energy_beam = [8._default, 10._default]

    data%norm_mode = NORM_SIGMA

    data%proc_num_id = [12, 34]
    data%cross_section = [100._default, 88._default]
    data%error = [1._default, 0.1_default]

    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_data_1"

  end subroutine eio_data_1
```

Event Normalization

Check the functions for translating modes and updating weights.

⟨*EIO data: execute tests*⟩+≡

```
  call test (eio_data_2, "eio_data_2", &
       "event normalization", &
       u, results)
```

⟨*EIO data: test declarations*⟩+≡

```
  public :: eio_data_2
```

⟨*EIO data: tests*⟩+≡

```
  subroutine eio_data_2 (u)
```

```
integer, intent(in) :: u
type(string_t) :: s
logical :: unweighted
real(default) :: w, w0, sigma
integer :: n

write (u, "(A)")  "* Test output: eio_data_2"
write (u, "(A)")  "*   Purpose:  handle event normalization"
write (u, "(A)")

write (u, "(A)")  "* Normalization strings"
write (u, "(A)")

s = "auto"
unweighted = .true.
write (u, "(1x,A,1x,L1,1x,A)")  char (s), unweighted, &
     char (event_normalization_string &
     (event_normalization_mode (s, unweighted)))
s = "AUTO"
unweighted = .false.
write (u, "(1x,A,1x,L1,1x,A)")  char (s), unweighted, &
     char (event_normalization_string &
     (event_normalization_mode (s, unweighted)))

unweighted = .true.

s = "1"
write (u, "(2(1x,A))") char (s), char (event_normalization_string &
     (event_normalization_mode (s, unweighted)))
s = "1/n"
write (u, "(2(1x,A))") char (s), char (event_normalization_string &
     (event_normalization_mode (s, unweighted)))
s = "Sigma"
write (u, "(2(1x,A))") char (s), char (event_normalization_string &
     (event_normalization_mode (s, unweighted)))
s = "sigma/N"
write (u, "(2(1x,A))") char (s), char (event_normalization_string &
     (event_normalization_mode (s, unweighted)))

write (u, "(A)")
write (u, "(A)")  "* Normalization update"
write (u, "(A)")

sigma = 5
n = 2

w0 = 1

w = w0
call event_normalization_update (w, sigma, n, NORM_UNIT, NORM_UNIT)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_N_EVT, NORM_UNIT)
write (u, "(2(F6.3))")  w0, w
```

```
w = w0
call event_normalization_update (w, sigma, n, NORM_SIGMA, NORM_UNIT)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_S_N, NORM_UNIT)
write (u, "(2(F6.3))")  w0, w

write (u, *)

w0 = 0.5

w = w0
call event_normalization_update (w, sigma, n, NORM_UNIT, NORM_N_EVT)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_N_EVT, NORM_N_EVT)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_SIGMA, NORM_N_EVT)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_S_N, NORM_N_EVT)
write (u, "(2(F6.3))")  w0, w

write (u, *)

w0 = 5.0

w = w0
call event_normalization_update (w, sigma, n, NORM_UNIT, NORM_SIGMA)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_N_EVT, NORM_SIGMA)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_SIGMA, NORM_SIGMA)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_S_N, NORM_SIGMA)
write (u, "(2(F6.3))")  w0, w

write (u, *)

w0 = 2.5

w = w0
call event_normalization_update (w, sigma, n, NORM_UNIT, NORM_S_N)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_N_EVT, NORM_S_N)
write (u, "(2(F6.3))")  w0, w
w = w0
call event_normalization_update (w, sigma, n, NORM_SIGMA, NORM_S_N)
write (u, "(2(F6.3))")  w0, w
```

```
    w = w0
    call event_normalization_update (w, sigma, n, NORM_S_N, NORM_S_N)
    write (u, "(2(F6.3))")  w0, w

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_data_2"

  end subroutine eio_data_2
```

17.3 Abstract I/O Handler

This module defines an abstract object for event I/O and the associated methods.

There are `output` and `input` methods which write or read a single event from/to the I/O stream, respectively. The I/O stream itself may be a file, a common block, or an externally linked structure, depending on the concrete implementation.

A `write` method prints the current content of the implementation-dependent event record in human-readable form.

The `init_in`/`init_out` and `final` prepare and finalize the I/O stream, respectively. There is also a `switch_inout` method which turns an input stream into an output stream where events can be appended.

Optionally, output files can be split in chunks of well-defined size. The `split_out` method takes care of this.

⟨`eio_base.f90`⟩≡

```
⟨File header⟩

module eio_base

  use kinds, only: i64
⟨Use strings⟩
  use io_units
  use diagnostics
  use model_data
  use event_base
  use eio_data

⟨Standard module head⟩

⟨EIO base: public⟩

⟨EIO base: types⟩

⟨EIO base: interfaces⟩

contains

⟨EIO base: procedures⟩

end module eio_base
```

17.3.1 Type

We can assume that most implementations will need the file extension as a fixed string and, if they support file splitting, the current file index.

The fallback model is useful for implementations that are able to read unknown files which may contain hadrons etc., not in the current hard-interaction model.

⟨*EIO base: public*⟩≡
```
public :: eio_t
```

⟨*EIO base: types*⟩≡
```
type, abstract :: eio_t
   type(string_t) :: sample
   type(string_t) :: extension
   type(string_t) :: filename
   logical :: has_file = .false.
   logical :: split = .false.
   integer :: split_n_evt = 0
   integer :: split_n_kbytes = 0
   integer :: split_index = 0
   integer :: split_count = 0
   class(model_data_t), pointer :: fallback_model => null ()
 contains
 ⟨EIO base: eio: TBP⟩
end type eio_t
```

Write to screen. If possible, this should display the contents of the current event, i.e., the last one that was written or read.

⟨*EIO base: eio: TBP*⟩≡
```
procedure (eio_write), deferred :: write
```

⟨*EIO base: interfaces*⟩≡
```
abstract interface
   subroutine eio_write (object, unit)
     import
     class(eio_t), intent(in) :: object
     integer, intent(in), optional :: unit
   end subroutine eio_write
end interface
```

Finalize. This should write/read footer data and close input/output channels.

⟨*EIO base: eio: TBP*⟩+≡
```
procedure (eio_final), deferred :: final
```

⟨*EIO base: interfaces*⟩+≡
```
abstract interface
   subroutine eio_final (object)
     import
     class(eio_t), intent(inout) :: object
   end subroutine eio_final
end interface
```

Determine splitting parameters from the event sample data.

⟨*EIO base: eio: TBP*⟩+≡
```
procedure :: set_splitting => eio_set_splitting
```

⟨*EIO base: procedures*⟩≡
```
subroutine eio_set_splitting (eio, data)
  class(eio_t), intent(inout) :: eio
  type(event_sample_data_t), intent(in) :: data
  eio%split = data%split_n_evt > 0 .or. data%split_n_kbytes > 0
  if (eio%split) then
     eio%split_n_evt = data%split_n_evt
     eio%split_n_kbytes = data%split_n_kbytes
     eio%split_index = data%split_index
     eio%split_count = 0
  end if
end subroutine eio_set_splitting
```

Update the byte count and check if it has increased. We use integer division to determine the number of `n_kbytes` blocks that are in the event file.

⟨*EIO base: eio: TBP*⟩+≡
```
procedure :: update_split_count => eio_update_split_count
```

⟨*EIO base: procedures*⟩+≡
```
subroutine eio_update_split_count (eio, increased)
  class(eio_t), intent(inout) :: eio
  logical, intent(out) :: increased
  integer :: split_count_old
  if (eio%split_n_kbytes > 0) then
     split_count_old = eio%split_count
     eio%split_count = eio%file_size_kbytes () / eio%split_n_kbytes
     increased = eio%split_count > split_count_old
  end if
end subroutine eio_update_split_count
```

Generate a filename, taking a possible split index into account.

⟨*EIO base: eio: TBP*⟩+≡
```
procedure :: set_filename => eio_set_filename
```

⟨*EIO base: procedures*⟩+≡
```
subroutine eio_set_filename (eio)
  class(eio_t), intent(inout) :: eio
  character(32) :: buffer
  if (eio%split) then
     write (buffer, "(I0,'.')")  eio%split_index
     eio%filename = eio%sample // "." // trim (buffer) // eio%extension
     eio%has_file = .true.
  else
     eio%filename = eio%sample // "." // eio%extension
     eio%has_file = .true.
  end if
end subroutine eio_set_filename
```

Set the fallback model.

⟨*EIO base: eio: TBP*⟩+≡
```
procedure :: set_fallback_model => eio_set_fallback_model
```

⟨*EIO base: procedures*⟩+≡
```
subroutine eio_set_fallback_model (eio, model)
  class(eio_t), intent(inout) :: eio
  class(model_data_t), intent(in), target :: model
  eio%fallback_model => model
end subroutine eio_set_fallback_model
```

Initialize for output. We provide process names. This should open an event file if appropriate and write header data. Some methods may require event sample data.

⟨*EIO base: eio: TBP*⟩+≡
```
procedure (eio_init_out), deferred :: init_out
```

⟨*EIO base: interfaces*⟩+≡
```
abstract interface
   subroutine eio_init_out (eio, sample, data, success, extension)
     import
     class(eio_t), intent(inout) :: eio
     type(string_t), intent(in) :: sample
     type(event_sample_data_t), intent(in), optional :: data
     logical, intent(out), optional :: success
     type(string_t), intent(in), optional :: extension
   end subroutine eio_init_out
end interface
```

Initialize for input. We provide process names. This should open an event file if appropriate and read header data. The `md5sum` can be used to check the integrity of the configuration, it it provides a checksum to compare with. In case the extension has changed the extension is also given as an argument.

The `data` argument is `intent(inout)`: we may read part of it and keep other parts and/or check them against the data in the file.

⟨*EIO base: eio: TBP*⟩+≡
```
procedure (eio_init_in), deferred :: init_in
```

⟨*EIO base: interfaces*⟩+≡
```
abstract interface
   subroutine eio_init_in (eio, sample, data, success, extension)
     import
     class(eio_t), intent(inout) :: eio
     type(string_t), intent(in) :: sample
     type(event_sample_data_t), intent(inout), optional :: data
     logical, intent(out), optional :: success
     type(string_t), intent(in), optional :: extension
   end subroutine eio_init_in
end interface
```

Re-initialize for output. This should change the status of any event file from input to output and position it for appending new events.

⟨*EIO base: eio: TBP*⟩+≡
```
procedure (eio_switch_inout), deferred :: switch_inout
```

⟨*EIO base: interfaces*⟩+≡

```
abstract interface
   subroutine eio_switch_inout (eio, success)
     import
     class(eio_t), intent(inout) :: eio
     logical, intent(out), optional :: success
   end subroutine eio_switch_inout
end interface
```

This is similar: split the output, i.e., close the current file and open a new one. The default implementation does nothing. For the feature to work, an implementation must override this.

⟨*EIO base: eio: TBP*⟩+≡

```
procedure :: split_out => eio_split_out
```

⟨*EIO base: procedures*⟩+≡

```
subroutine eio_split_out (eio)
  class(eio_t), intent(inout) :: eio
end subroutine eio_split_out
```

Determine the file size in kilobytes. More exactly, determine the size in units of 1024 storage units, as returned by the INQUIRE statement.

The implementation returns zero if there is no file. The `has_file` flag is set by the `set_filename` method, so we can be confident that the `inquire` call is meaningful. If this algorithm doesn't apply for a particular format, we still can override the procedure.

⟨*EIO base: eio: TBP*⟩+≡

```
procedure :: file_size_kbytes => eio_file_size_kbytes
```

⟨*EIO base: procedures*⟩+≡

```
function eio_file_size_kbytes (eio) result (kbytes)
  class(eio_t), intent(in) :: eio
  integer :: kbytes
  integer(i64) :: bytes
  if (eio%has_file) then
     inquire (file = char (eio%filename), size = bytes)
     if (bytes > 0) then
        kbytes = bytes / 1024
     else
        kbytes = 0
     end if
  else
     kbytes = 0
  end if
end function eio_file_size_kbytes
```

Output an event. All data can be taken from the `event` record. The index `i_prc` identifies the process among the processes that are contained in the current sample. The `reading` flag, if present, indicates that the event was read from file, not generated.

The `passed` flag tells us that this event has passed the selection criteria. Depending on the event format, we may choose to skip events that have not

passed.

⟨*EIO base: eio: TBP*⟩+≡
```
procedure (eio_output), deferred :: output
```

⟨*EIO base: interfaces*⟩+≡
```
abstract interface
   subroutine eio_output (eio, event, i_prc, reading, passed, pacify)
     import
     class(eio_t), intent(inout) :: eio
     class(generic_event_t), intent(in), target :: event
     integer, intent(in) :: i_prc
     logical, intent(in), optional :: reading, passed, pacify
   end subroutine eio_output
end interface
```

Input an event. This should fill all event data that cannot be inferred from the associated process.

The input is broken down into two parts. First we read the `i_prc` index. So we know which process to expect in the subsequent event. If we have reached end of file, we also will know. Then, we read the event itself.

The parameter `iostat` is supposed to be set as the Fortran standard requires, negative for EOF and positive for error.

⟨*EIO base: eio: TBP*⟩+≡
```
procedure (eio_input_i_prc), deferred :: input_i_prc
procedure (eio_input_event), deferred :: input_event
```

⟨*EIO base: interfaces*⟩+≡
```
abstract interface
   subroutine eio_input_i_prc (eio, i_prc, iostat)
     import
     class(eio_t), intent(inout) :: eio
     integer, intent(out) :: i_prc
     integer, intent(out) :: iostat
   end subroutine eio_input_i_prc
end interface

abstract interface
   subroutine eio_input_event (eio, event, iostat)
     import
     class(eio_t), intent(inout) :: eio
     class(generic_event_t), intent(inout), target :: event
     integer, intent(out) :: iostat
   end subroutine eio_input_event
end interface
```

⟨*EIO base: eio: TBP*⟩+≡
```
procedure (eio_skip), deferred :: skip
```

⟨*EIO base: interfaces*⟩+≡
```
abstract interface
   subroutine eio_skip (eio, iostat)
     import
     class(eio_t), intent(inout) :: eio
     integer, intent(out) :: iostat
```

```
      end subroutine eio_skip
    end interface
```

17.3.2 Unit tests

Test module, followed by the corresponding implementation module.

⟨eio_base_ut.f90⟩≡

```
  ⟨File header⟩

  module eio_base_ut
    use unit_tests
    use eio_base_uti

  ⟨Standard module head⟩

  ⟨EIO base: public test⟩

  ⟨EIO base: public test auxiliary⟩

  contains

  ⟨EIO base: test driver⟩

  end module eio_base_ut
```

⟨eio_base_uti.f90⟩≡

```
  ⟨File header⟩

  module eio_base_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use io_units
    use lorentz
    use model_data
    use particles
    use event_base
    use eio_data

    use eio_base

  ⟨Standard module head⟩

  ⟨EIO base: public test auxiliary⟩

  ⟨EIO base: test declarations⟩

  ⟨EIO base: test types⟩

  ⟨EIO base: test variables⟩

  contains
```

⟨*EIO base: tests*⟩

⟨*EIO base: test auxiliary*⟩

```
end module eio_base_uti
```

API: driver for the unit tests below.

⟨*EIO base: public test*⟩≡

```
public :: eio_base_test
```

⟨*EIO base: test driver*⟩≡

```
subroutine eio_base_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨EIO base: execute tests⟩
end subroutine eio_base_test
```

The caller has to provide procedures that prepare and cleanup the test environment. They depend on modules that are not available here.

⟨*EIO base: test types*⟩≡

```
abstract interface
   subroutine eio_prepare_event (event, unweighted, n_alt)
     import
     class(generic_event_t), intent(inout), pointer :: event
     logical, intent(in), optional :: unweighted
     integer, intent(in), optional :: n_alt
   end subroutine eio_prepare_event
end interface

abstract interface
   subroutine eio_cleanup_event (event)
     import
     class(generic_event_t), intent(inout), pointer :: event
   end subroutine eio_cleanup_event
end interface
```

We store pointers to the test-environment handlers as module variables. This allows us to call them from the test routines themselves, which don't allow for extra arguments.

⟨*EIO base: public test auxiliary*⟩≡

```
public :: eio_prepare_test, eio_cleanup_test
```

⟨*EIO base: test types*⟩+≡

```
procedure(eio_prepare_event), pointer :: eio_prepare_test => null ()
procedure(eio_cleanup_event), pointer :: eio_cleanup_test => null ()
```

Similarly, for the fallback (hadron) model that some eio tests require:

⟨*EIO base: test types*⟩+≡

```
abstract interface
   subroutine eio_prepare_model (model)
     import
     class(model_data_t), intent(inout), pointer :: model
   end subroutine eio_prepare_model
```

```
end interface

abstract interface
   subroutine eio_cleanup_model (model)
     import
     class(model_data_t), intent(inout), pointer :: model
   end subroutine eio_cleanup_model
end interface
```

⟨*EIO base: public test auxiliary*⟩+≡

```
public :: eio_prepare_fallback_model, eio_cleanup_fallback_model
```

⟨*EIO base: test variables*⟩≡

```
procedure(eio_prepare_model), pointer :: eio_prepare_fallback_model => null ()
procedure(eio_cleanup_model), pointer :: eio_cleanup_fallback_model => null ()
```

Test type for event I/O

The contents simulate the contents of an external file. We have the `sample` string as the file name and the array of momenta `event_p` as the list of events. The second index is the event index. The `event_i` component is the pointer to the current event, `event_n` is the total number of stored events.

⟨*EIO base: test types*⟩+≡

```
type, extends (eio_t) :: eio_test_t
   integer :: event_n = 0
   integer :: event_i = 0
   integer :: i_prc = 0
   type(vector4_t), dimension(:,:), allocatable :: event_p
 contains
 ⟨EIO base: eio test: TBP⟩
end type eio_test_t
```

Write to screen. Pretend that this is an actual event format.

⟨*EIO base: eio test: TBP*⟩≡

```
procedure :: write => eio_test_write
```

⟨*EIO base: test auxiliary*⟩≡

```
subroutine eio_test_write (object, unit)
  class(eio_test_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Test event stream"
  if (object%event_i /= 0) then
     write (u, "(1x,A,I0,A)")  "Event #", object%event_i, ":"
     do i = 1, size (object%event_p, 1)
        call vector4_write (object%event_p(i, object%event_i), u)
     end do
  end if
end subroutine eio_test_write
```

Finalizer. For the test case, we just reset the event count, but keep the stored "events". For the real implementations, the events would be stored on an external medium, so we would delete the object contents.

⟨*EIO base: eio test: TBP*⟩+≡
```
procedure :: final => eio_test_final
```

⟨*EIO base: test auxiliary*⟩+≡
```
subroutine eio_test_final (object)
  class(eio_test_t), intent(inout) :: object
  object%event_i = 0
end subroutine eio_test_final
```

Initialization: We store the process IDs and the energy from the beam-data object. We also allocate the momenta (i.e., the simulated event record) for a fixed maximum size of 10 events, 2 momenta each. There is only a single process.

⟨*EIO base: eio test: TBP*⟩+≡
```
procedure :: init_out => eio_test_init_out
```

⟨*EIO base: test auxiliary*⟩+≡
```
subroutine eio_test_init_out (eio, sample, data, success, extension)
  class(eio_test_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(event_sample_data_t), intent(in), optional :: data
  logical, intent(out), optional :: success
  type(string_t), intent(in), optional :: extension
  eio%sample = sample
  eio%event_n = 0
  eio%event_i = 0
  allocate (eio%event_p (2, 10))
  if (present (success))  success = .true.
end subroutine eio_test_init_out
```

Initialization for input. Nothing to do for the test type.

⟨*EIO base: eio test: TBP*⟩+≡
```
procedure :: init_in => eio_test_init_in
```

⟨*EIO base: test auxiliary*⟩+≡
```
subroutine eio_test_init_in (eio, sample, data, success, extension)
  class(eio_test_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(event_sample_data_t), intent(inout), optional :: data
  logical, intent(out), optional :: success
  type(string_t), intent(in), optional :: extension
  if (present (success))  success = .true.
end subroutine eio_test_init_in
```

Switch from output to input. Again, nothing to do for the test type.

⟨*EIO base: eio test: TBP*⟩+≡
```
procedure :: switch_inout => eio_test_switch_inout
```

⟨*EIO base: test auxiliary*⟩+≡

```
subroutine eio_test_switch_inout (eio, success)
  class(eio_test_t), intent(inout) :: eio
  logical, intent(out), optional :: success
  if (present (success))  success = .true.
end subroutine eio_test_switch_inout
```

Output. Increment the event counter and store the momenta of the current event.

⟨*EIO base: eio test: TBP*⟩+≡

```
procedure :: output => eio_test_output
```

⟨*EIO base: test auxiliary*⟩+≡

```
subroutine eio_test_output (eio, event, i_prc, reading, passed, pacify)
  class(eio_test_t), intent(inout) :: eio
  class(generic_event_t), intent(in), target :: event
  logical, intent(in), optional :: reading, passed, pacify
  integer, intent(in) :: i_prc
  type(particle_set_t), pointer :: pset
  type(particle_t) :: prt
  eio%event_n = eio%event_n + 1
  eio%event_i = eio%event_n
  eio%i_prc = i_prc
  pset => event%get_particle_set_ptr ()
  prt = pset%get_particle (3)
  eio%event_p(1, eio%event_i) = prt%get_momentum ()
  prt = pset%get_particle (4)
  eio%event_p(2, eio%event_i) = prt%get_momentum ()
end subroutine eio_test_output
```

Input. Increment the event counter and retrieve the momenta of the current event. For the test case, we do not actually modify the current event.

⟨*EIO base: eio test: TBP*⟩+≡

```
procedure :: input_i_prc => eio_test_input_i_prc
procedure :: input_event => eio_test_input_event
```

⟨*EIO base: test auxiliary*⟩+≡

```
subroutine eio_test_input_i_prc (eio, i_prc, iostat)
  class(eio_test_t), intent(inout) :: eio
  integer, intent(out) :: i_prc
  integer, intent(out) :: iostat
  i_prc = eio%i_prc
  iostat = 0
end subroutine eio_test_input_i_prc

subroutine eio_test_input_event (eio, event, iostat)
  class(eio_test_t), intent(inout) :: eio
  class(generic_event_t), intent(inout), target :: event
  integer, intent(out) :: iostat
  eio%event_i = eio%event_i + 1
  iostat = 0
end subroutine eio_test_input_event
```

⟨*EIO base: eio test: TBP*⟩+≡

```
procedure :: skip => eio_test_skip
```

⟨*EIO base: test auxiliary*⟩+≡

```
subroutine eio_test_skip (eio, iostat)
  class(eio_test_t), intent(inout) :: eio
  integer, intent(out) :: iostat
  iostat = 0
end subroutine eio_test_skip
```

Test I/O methods

⟨*EIO base: execute tests*⟩≡

```
call test (eio_base_1, "eio_base_1", &
     "read and write event contents", &
     u, results)
```

⟨*EIO base: test declarations*⟩≡

```
public :: eio_base_1
```

⟨*EIO base: tests*⟩≡

```
subroutine eio_base_1 (u)
  integer, intent(in) :: u
  class(generic_event_t), pointer :: event
  class(eio_t), allocatable :: eio
  integer :: i_prc,  iostat
  type(string_t) :: sample

  write (u, "(A)")  "* Test output: eio_base_1"
  write (u, "(A)")  "*   Purpose: generate and read/write an event"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize test process"

  call eio_prepare_test (event, unweighted = .false.)

  write (u, "(A)")
  write (u, "(A)")  "* Generate and write an event"
  write (u, "(A)")

  sample = "eio_test1"

  allocate (eio_test_t :: eio)

  call eio%init_out (sample)
  call event%generate (1, [0._default, 0._default])
  call eio%output (event, 42)
  call eio%write (u)
  call eio%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Re-read the event"
  write (u, "(A)")

  call eio%init_in (sample)
```

```
    call eio%input_i_prc (i_prc, iostat)
    call eio%input_event (event, iostat)
    call eio%write (u)
    write (u, "(A)")
    write (u, "(1x,A,I0)")  "i = ", i_prc

    write (u, "(A)")
    write (u, "(A)")  "* Generate and append another event"
    write (u, "(A)")

    call eio%switch_inout ()
    call event%generate (1, [0._default, 0._default])
    call eio%output (event, 5)
    call eio%write (u)
    call eio%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Re-read both events"
    write (u, "(A)")

    call eio%init_in (sample)
    call eio%input_i_prc (i_prc, iostat)
    call eio%input_event (event, iostat)
    call eio%input_i_prc (i_prc, iostat)
    call eio%input_event (event, iostat)
    call eio%write (u)
    write (u, "(A)")
    write (u, "(1x,A,I0)")  "i = ", i_prc

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio%final ()
    deallocate (eio)

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_base_1"

  end subroutine eio_base_1
```

17.4 Event Generation Checkpoints

This is an output-only format. Its only use is to write screen messages every n events, to inform the user about progress.

⟨eio_checkpoints.f90⟩≡
  ⟨*File header*⟩

```
  module eio_checkpoints
```

```
⟨Use strings⟩
  use io_units
  use diagnostics
  use cputime
  use event_base
  use eio_data
  use eio_base

⟨Standard module head⟩

⟨EIO checkpoints: public⟩

⟨EIO checkpoints: parameters⟩

⟨EIO checkpoints: types⟩

contains

⟨EIO checkpoints: procedures⟩

end module eio_checkpoints
```

17.4.1 Type

⟨*EIO checkpoints: public*⟩≡

```
public :: eio_checkpoints_t
```

⟨*EIO checkpoints: types*⟩≡

```
type, extends (eio_t) :: eio_checkpoints_t
   logical :: active = .false.
   logical :: running = .false.
   integer :: val = 0
   integer :: n_events = 0
   integer :: n_read = 0
   integer :: i_evt = 0
   logical :: blank = .false.
   type(timer_t) :: timer
 contains
 ⟨EIO checkpoints: eio checkpoints: TBP⟩
end type eio_checkpoints_t
```

17.4.2 Specific Methods

Set parameters that are specifically used for checkpointing.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩≡

```
procedure :: set_parameters => eio_checkpoints_set_parameters
```

⟨*EIO checkpoints: procedures*⟩≡

```
subroutine eio_checkpoints_set_parameters (eio, checkpoint, blank)
  class(eio_checkpoints_t), intent(inout) :: eio
  integer, intent(in) :: checkpoint
  logical, intent(in), optional :: blank
  eio%val = checkpoint
```

```
    if (present (blank))  eio%blank = blank
  end subroutine eio_checkpoints_set_parameters
```

17.4.3 Common Methods

Output. This is not the actual event format, but a readable account of the current status.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡

```
  procedure :: write => eio_checkpoints_write
```

⟨*EIO checkpoints: procedures*⟩+≡

```
  subroutine eio_checkpoints_write (object, unit)
    class(eio_checkpoints_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    if (object%active) then
       write (u, "(1x,A)")  "Event-sample checkpoints:  active"
       write (u, "(3x,A,I0)")  "interval  = ", object%val
       write (u, "(3x,A,I0)")  "n_events  = ", object%n_events
       write (u, "(3x,A,I0)")  "n_read    = ", object%n_read
       write (u, "(3x,A,I0)")  "n_current = ", object%i_evt
       write (u, "(3x,A,L1)")  "blanking  = ", object%blank
       call object%timer%write (u)
    else
       write (u, "(1x,A)")  "Event-sample checkpoints:  off"
    end if
  end subroutine eio_checkpoints_write
```

Finalizer: trivial.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡

```
  procedure :: final => eio_checkpoints_final
```

⟨*EIO checkpoints: procedures*⟩+≡

```
  subroutine eio_checkpoints_final (object)
    class(eio_checkpoints_t), intent(inout) :: object
    object%active = .false.
  end subroutine eio_checkpoints_final
```

Activate checkpointing for event generation or writing.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡

```
  procedure :: init_out => eio_checkpoints_init_out
```

⟨*EIO checkpoints: procedures*⟩+≡

```
  subroutine eio_checkpoints_init_out (eio, sample, data, success, extension)
    class(eio_checkpoints_t), intent(inout) :: eio
    type(string_t), intent(in) :: sample
    type(string_t), intent(in), optional :: extension
    type(event_sample_data_t), intent(in), optional :: data
    logical, intent(out), optional :: success
    if (present (data)) then
       if (eio%val > 0) then
```

```
            eio%active = .true.
            eio%i_evt = 0
            eio%n_read = 0
            eio%n_events = data%n_evt * data%nlo_multiplier
         end if
      end if
      if (present (success))  success = .true.
   end subroutine eio_checkpoints_init_out
```

No checkpointing for event reading.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡

```
   procedure :: init_in => eio_checkpoints_init_in
```

⟨*EIO checkpoints: procedures*⟩+≡

```
   subroutine eio_checkpoints_init_in (eio, sample, data, success, extension)
      class(eio_checkpoints_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(string_t), intent(in), optional :: extension
      type(event_sample_data_t), intent(inout), optional :: data
      logical, intent(out), optional :: success
      call msg_bug ("Event checkpoints: event input not supported")
      if (present (success))  success = .false.
   end subroutine eio_checkpoints_init_in
```

Switch from input to output: also not supported.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡

```
   procedure :: switch_inout => eio_checkpoints_switch_inout
```

⟨*EIO checkpoints: procedures*⟩+≡

```
   subroutine eio_checkpoints_switch_inout (eio, success)
      class(eio_checkpoints_t), intent(inout) :: eio
      logical, intent(out), optional :: success
      call msg_bug ("Event checkpoints: in-out switch not supported")
      if (present (success))  success = .false.
   end subroutine eio_checkpoints_switch_inout
```

Checkpoints: display progress for the current event, if applicable.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡

```
   procedure :: output => eio_checkpoints_output
```

⟨*EIO checkpoints: procedures*⟩+≡

```
   subroutine eio_checkpoints_output (eio, event, i_prc, reading, passed, pacify)
      class(eio_checkpoints_t), intent(inout) :: eio
      class(generic_event_t), intent(in), target :: event
      integer, intent(in) :: i_prc
      logical, intent(in), optional :: reading, passed, pacify
      logical :: rd
      rd = .false.;  if (present (reading))  rd = reading
      if (eio%active) then
         if (.not. eio%running)  call eio%startup ()
         if (eio%running) then
            eio%i_evt = eio%i_evt + 1
            if (rd) then
               eio%n_read = eio%n_read + 1
```

```
        else if (mod (eio%i_evt, eio%val) == 0) then
           call eio%message (eio%blank)
        end if
        if (eio%i_evt == eio%n_events)  call eio%shutdown ()
     end if
  end if
end subroutine eio_checkpoints_output
```

When the first event is called, we have to initialize the screen output.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡

```
procedure :: startup => eio_checkpoints_startup
```

⟨*EIO checkpoints: procedures*⟩+≡

```
subroutine eio_checkpoints_startup (eio)
  class(eio_checkpoints_t), intent(inout) :: eio
  if (eio%active .and. eio%i_evt < eio%n_events) then
     call msg_message ("")
     call msg_message (checkpoint_bar)
     call msg_message (checkpoint_head)
     call msg_message (checkpoint_bar)
     write (msg_buffer, checkpoint_fmt) 0., 0, eio%n_events - eio%i_evt, "???"
     call msg_message ()
     eio%running = .true.
     call eio%timer%start ()
  end if
end subroutine eio_checkpoints_startup
```

This message is printed at every checkpoint.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡

```
procedure :: message => eio_checkpoints_message
```

⟨*EIO checkpoints: procedures*⟩+≡

```
subroutine eio_checkpoints_message (eio, testflag)
  class(eio_checkpoints_t), intent(inout) :: eio
  logical, intent(in), optional :: testflag
  real :: t
  type(time_t) :: time_remaining
  type(string_t) :: time_string
  call eio%timer%stop ()
  t = eio%timer
  call eio%timer%restart ()
  time_remaining = &
       nint (t / (eio%i_evt - eio%n_read) * (eio%n_events - eio%i_evt))
  time_string = time_remaining%to_string_ms (blank = testflag)
  write (msg_buffer, checkpoint_fmt) &
       100 * (eio%i_evt - eio%n_read) / real (eio%n_events - eio%n_read), &
       eio%i_evt - eio%n_read, &
       eio%n_events - eio%i_evt, &
       char (time_string)
  call msg_message ()
end subroutine eio_checkpoints_message
```

When the last event is called, wrap up.

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡
```
procedure :: shutdown => eio_checkpoints_shutdown
```

⟨*EIO checkpoints: procedures*⟩+≡
```
subroutine eio_checkpoints_shutdown (eio)
  class(eio_checkpoints_t), intent(inout) :: eio
  if (mod (eio%i_evt, eio%val) /= 0) then
     write (msg_buffer, checkpoint_fmt) &
          100., eio%i_evt - eio%n_read, 0, "0m:00s"
     call msg_message ()
  end if
  call msg_message (checkpoint_bar)
  call msg_message ("")
  eio%running = .false.
end subroutine eio_checkpoints_shutdown
```

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡
```
procedure :: input_i_prc => eio_checkpoints_input_i_prc
procedure :: input_event => eio_checkpoints_input_event
```

⟨*EIO checkpoints: procedures*⟩+≡
```
subroutine eio_checkpoints_input_i_prc (eio, i_prc, iostat)
  class(eio_checkpoints_t), intent(inout) :: eio
  integer, intent(out) :: i_prc
  integer, intent(out) :: iostat
  call msg_bug ("Event checkpoints: event input not supported")
  i_prc = 0
  iostat = 1
end subroutine eio_checkpoints_input_i_prc

subroutine eio_checkpoints_input_event (eio, event, iostat)
  class(eio_checkpoints_t), intent(inout) :: eio
  class(generic_event_t), intent(inout), target :: event
  integer, intent(out) :: iostat
  call msg_bug ("Event checkpoints: event input not supported")
  iostat = 1
end subroutine eio_checkpoints_input_event
```

⟨*EIO checkpoints: eio checkpoints: TBP*⟩+≡
```
procedure :: skip => eio_checkpoints_skip
```

⟨*EIO checkpoints: procedures*⟩+≡
```
subroutine eio_checkpoints_skip (eio, iostat)
  class(eio_checkpoints_t), intent(inout) :: eio
  integer, intent(out) :: iostat
  iostat = 0
end subroutine eio_checkpoints_skip
```

17.4.4 Message header

⟨*EIO checkpoints: parameters*⟩≡
```
character(*), parameter :: &
```

```
      checkpoint_head = "| % complete | events generated | events remaining &
      &| time remaining"
   character(*), parameter :: &
      checkpoint_bar  = "|==================================================&
      &=================|"
   character(*), parameter :: &
      checkpoint_fmt  = "('   ',F5.1,T16,I9,T35,I9,T58,A)"
```

17.4.5 Unit tests

Test module, followed by the corresponding implementation module.

⟨eio_checkpoints_ut.f90⟩≡

```
⟨File header⟩

module eio_checkpoints_ut
  use unit_tests
  use eio_checkpoints_uti

⟨Standard module head⟩

⟨EIO checkpoints: public test⟩

contains

⟨EIO checkpoints: test driver⟩

end module eio_checkpoints_ut
```

⟨eio_checkpoints_uti.f90⟩≡

```
⟨File header⟩

module eio_checkpoints_uti

⟨Use kinds⟩
⟨Use strings⟩
  use event_base
  use eio_data
  use eio_base

  use eio_checkpoints

  use eio_base_ut, only: eio_prepare_test, eio_cleanup_test

⟨Standard module head⟩

⟨EIO checkpoints: test declarations⟩

contains

⟨EIO checkpoints: tests⟩

end module eio_checkpoints_uti
```

API: driver for the unit tests below.

⟨*EIO checkpoints: public test*⟩≡
```
public :: eio_checkpoints_test
```

⟨*EIO checkpoints: test driver*⟩≡
```
subroutine eio_checkpoints_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨EIO checkpoints: execute tests⟩
end subroutine eio_checkpoints_test
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO checkpoints: execute tests*⟩≡
```
call test (eio_checkpoints_1, "eio_checkpoints_1", &
     "read and write event contents", &
     u, results)
```

⟨*EIO checkpoints: test declarations*⟩≡
```
public :: eio_checkpoints_1
```

⟨*EIO checkpoints: tests*⟩≡
```
subroutine eio_checkpoints_1 (u)
  integer, intent(in) :: u
  class(generic_event_t), pointer :: event
  class(eio_t), allocatable :: eio
  type(event_sample_data_t) :: data
  type(string_t) :: sample
  integer :: i, n_events

  write (u, "(A)")  "* Test output: eio_checkpoints_1"
  write (u, "(A)")  "*   Purpose: generate a number of events &
       &with screen output"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize test process"

  call eio_prepare_test (event)

  write (u, "(A)")
  write (u, "(A)")  "* Generate events"
  write (u, "(A)")

  sample = "eio_checkpoints_1"

  allocate (eio_checkpoints_t :: eio)

  n_events = 10
  call data%init (1, 0)
  data%n_evt = n_events

  select type (eio)
  type is (eio_checkpoints_t)
```

```
      call eio%set_parameters (checkpoint = 4)
   end select

   call eio%init_out (sample, data)

   do i = 1, n_events
      call event%generate (1, [0._default, 0._default])
      call eio%output (event, i_prc = 0)
   end do

   write (u, "(A)")  "* Checkpointing status"
   write (u, "(A)")

   call eio%write (u)
   call eio%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call eio_cleanup_test (event)

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: eio_checkpoints_1"

end subroutine eio_checkpoints_1
```

17.5 Event Generation Callback

This is an output-only format. Its only use is to write screen messages every n events, to inform the user about progress.

⟨eio_callback.f90⟩≡

```
⟨File header⟩

module eio_callback

  use kinds, only: i64
⟨Use strings⟩
  use io_units
  use diagnostics
  use cputime
  use event_base
  use eio_data
  use eio_base

⟨Standard module head⟩

⟨EIO callback: public⟩

⟨EIO callback: types⟩

contains
```

⟨*EIO callback: procedures*⟩

```
end module eio_callback
```

17.5.1 Type

⟨*EIO callback: public*⟩≡

```
public :: eio_callback_t
```

⟨*EIO callback: types*⟩≡

```
type, extends (eio_t) :: eio_callback_t
   class(event_callback_t), allocatable :: callback
   integer(i64) :: i_evt = 0
   integer :: i_interval = 0
   integer :: n_interval = 0
!    type(timer_t) :: timer
 contains
 ⟨EIO callback: eio callback: TBP⟩
end type eio_callback_t
```

17.5.2 Specific Methods

Set parameters that are specifically used for callback: the procedure and the number of events to wait until the procedure is called (again).

⟨*EIO callback: eio callback: TBP*⟩≡

```
procedure :: set_parameters => eio_callback_set_parameters
```

⟨*EIO callback: procedures*⟩≡

```
subroutine eio_callback_set_parameters (eio, callback, count_interval)
  class(eio_callback_t), intent(inout) :: eio
  class(event_callback_t), intent(in) :: callback
  integer, intent(in) :: count_interval
  allocate (eio%callback, source = callback)
  eio%n_interval = count_interval
end subroutine eio_callback_set_parameters
```

17.5.3 Common Methods

Output. This is not the actual event format, but a readable account of the current status.

⟨*EIO callback: eio callback: TBP*⟩+≡

```
procedure :: write => eio_callback_write
```

⟨*EIO callback: procedures*⟩+≡

```
subroutine eio_callback_write (object, unit)
  class(eio_callback_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Event-sample callback:"
```

```
      write (u, "(3x,A,I0)")  "interval  = ", object%n_interval
      write (u, "(3x,A,I0)")  "evt count = ", object%i_evt
!        call object%timer%write (u)
    end subroutine eio_callback_write
```

Finalizer: trivial.

⟨*EIO callback: eio callback: TBP*⟩+≡

```
    procedure :: final => eio_callback_final
```

⟨*EIO callback: procedures*⟩+≡

```
    subroutine eio_callback_final (object)
      class(eio_callback_t), intent(inout) :: object
    end subroutine eio_callback_final
```

Activate checkpointing for event generation or writing.

⟨*EIO callback: eio callback: TBP*⟩+≡

```
    procedure :: init_out => eio_callback_init_out
```

⟨*EIO callback: procedures*⟩+≡

```
    subroutine eio_callback_init_out (eio, sample, data, success, extension)
      class(eio_callback_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(string_t), intent(in), optional :: extension
      type(event_sample_data_t), intent(in), optional :: data
      logical, intent(out), optional :: success
      eio%i_evt = 0
      eiO%i_interval = 0
      if (present (success))  success = .true.
    end subroutine eio_callback_init_out
```

No callback for event reading.

⟨*EIO callback: eio callback: TBP*⟩+≡

```
    procedure :: init_in => eio_callback_init_in
```

⟨*EIO callback: procedures*⟩+≡

```
    subroutine eio_callback_init_in (eio, sample, data, success, extension)
      class(eio_callback_t), intent(inout) :: eio
      type(string_t), intent(in) :: sample
      type(string_t), intent(in), optional :: extension
      type(event_sample_data_t), intent(inout), optional :: data
      logical, intent(out), optional :: success
      call msg_bug ("Event callback: event input not supported")
      if (present (success))  success = .false.
    end subroutine eio_callback_init_in
```

Switch from input to output: also not supported.

⟨*EIO callback: eio callback: TBP*⟩+≡

```
    procedure :: switch_inout => eio_callback_switch_inout
```

⟨*EIO callback: procedures*⟩+≡

```
    subroutine eio_callback_switch_inout (eio, success)
      class(eio_callback_t), intent(inout) :: eio
      logical, intent(out), optional :: success
```

```
    call msg_bug ("Event callback: in-out switch not supported")
    if (present (success))  success = .false.
  end subroutine eio_callback_switch_inout
```

The actual callback. First increment counters, then call the procedure if the counter hits the interval.

⟨*EIO callback: eio callback: TBP*⟩+≡

```
  procedure :: output => eio_callback_output
```

⟨*EIO callback: procedures*⟩+≡

```
  subroutine eio_callback_output (eio, event, i_prc, reading, passed, pacify)
    class(eio_callback_t), intent(inout) :: eio
    class(generic_event_t), intent(in), target :: event
    integer, intent(in) :: i_prc
    logical, intent(in), optional :: reading, passed, pacify
    eio%i_evt = eio%i_evt + 1
    if (eio%n_interval > 0) then
       eio%i_interval = eio%i_interval + 1
       if (eio%i_interval >= eio%n_interval) then
          call eio%callback%proc (eio%i_evt, event)
          eio%i_interval = 0
       end if
    end if
  end subroutine eio_callback_output
```

No input.

⟨*EIO callback: eio callback: TBP*⟩+≡

```
  procedure :: input_i_prc => eio_callback_input_i_prc
  procedure :: input_event => eio_callback_input_event
```

⟨*EIO callback: procedures*⟩+≡

```
  subroutine eio_callback_input_i_prc (eio, i_prc, iostat)
    class(eio_callback_t), intent(inout) :: eio
    integer, intent(out) :: i_prc
    integer, intent(out) :: iostat
    call msg_bug ("Event callback: event input not supported")
    i_prc = 0
    iostat = 1
  end subroutine eio_callback_input_i_prc

  subroutine eio_callback_input_event (eio, event, iostat)
    class(eio_callback_t), intent(inout) :: eio
    class(generic_event_t), intent(inout), target :: event
    integer, intent(out) :: iostat
    call msg_bug ("Event callback: event input not supported")
    iostat = 1
  end subroutine eio_callback_input_event
```

⟨*EIO callback: eio callback: TBP*⟩+≡

```
  procedure :: skip => eio_callback_skip
```

⟨*EIO callback: procedures*⟩+≡

```
  subroutine eio_callback_skip (eio, iostat)
    class(eio_callback_t), intent(inout) :: eio
```

```
    integer, intent(out) :: iostat
    iostat = 0
  end subroutine eio_callback_skip
```

17.6 Event Weight Output

This is an output-only format. For each event, we print the indices that identify process, process part (MCI group), and term. As numerical information we print the squared matrix element (trace) and the event weight.

⟨eio_weights.f90⟩≡

```
⟨File header⟩

module eio_weights

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use diagnostics
  use event_base
  use eio_data
  use eio_base

⟨Standard module head⟩

⟨EIO weights: public⟩

⟨EIO weights: types⟩

contains

⟨EIO weights: procedures⟩

end module eio_weights
```

17.6.1 Type

⟨*EIO weights: public*⟩≡

```
  public :: eio_weights_t
```

⟨*EIO weights: types*⟩≡

```
  type, extends (eio_t) :: eio_weights_t
     logical :: writing = .false.
     integer :: unit = 0
     logical :: pacify = .false.
   contains
   ⟨EIO weights: eio weights: TBP⟩
  end type eio_weights_t
```

17.6.2 Specific Methods

Set pacify flags.

⟨*EIO weights: eio weights: TBP*⟩≡

```
procedure :: set_parameters => eio_weights_set_parameters
```

⟨*EIO weights: procedures*⟩≡

```
subroutine eio_weights_set_parameters (eio, pacify)
  class(eio_weights_t), intent(inout) :: eio
  logical, intent(in), optional :: pacify
  if (present (pacify))  eio%pacify = pacify
  eio%extension = "weights.dat"
end subroutine eio_weights_set_parameters
```

17.6.3 Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO weights: eio weights: TBP*⟩+≡

```
procedure :: write => eio_weights_write
```

⟨*EIO weights: procedures*⟩+≡

```
subroutine eio_weights_write (object, unit)
  class(eio_weights_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Weight stream:"
  if (object%writing) then
     write (u, "(3x,A,A)")  "Writing to file   = ", char (object%filename)
     write (u, "(3x,A,L1)") "Reduced I/O prec. = ", object%pacify
  else
     write (u, "(3x,A)")  "[closed]"
  end if
end subroutine eio_weights_write
```

Finalizer: close any open file.

⟨*EIO weights: eio weights: TBP*⟩+≡

```
procedure :: final => eio_weights_final
```

⟨*EIO weights: procedures*⟩+≡

```
subroutine eio_weights_final (object)
  class(eio_weights_t), intent(inout) :: object
  if (object%writing) then
     write (msg_buffer, "(A,A,A)")  "Events: closing weight stream file '", &
          char (object%filename), "'"
     call msg_message ()
     close (object%unit)
     object%writing = .false.
  end if
end subroutine eio_weights_final
```

Initialize event writing.

⟨*EIO weights: eio weights: TBP*⟩+≡

```
procedure :: init_out => eio_weights_init_out
```

⟨*EIO weights: procedures*⟩+≡

```
subroutine eio_weights_init_out (eio, sample, data, success, extension)
  class(eio_weights_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(string_t), intent(in), optional :: extension
  type(event_sample_data_t), intent(in), optional :: data
  logical, intent(out), optional :: success
  if (present(extension)) then
     eio%extension = extension
  else
     eio%extension = "weights.dat"
  end if
  eio%filename = sample // "." // eio%extension
  eio%unit = free_unit ()
  write (msg_buffer, "(A,A,A)")  "Events: writing to weight stream file '", &
       char (eio%filename), "'"
  call msg_message ()
  eio%writing = .true.
  open (eio%unit, file = char (eio%filename), &
       action = "write", status = "replace")
  if (present (success))  success = .true.
end subroutine eio_weights_init_out
```

Initialize event reading.

⟨*EIO weights: eio weights: TBP*⟩+≡

```
procedure :: init_in => eio_weights_init_in
```

⟨*EIO weights: procedures*⟩+≡

```
subroutine eio_weights_init_in (eio, sample, data, success, extension)
  class(eio_weights_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(string_t), intent(in), optional :: extension
  type(event_sample_data_t), intent(inout), optional :: data
  logical, intent(out), optional :: success
  call msg_bug ("Weight stream: event input not supported")
  if (present (success))  success = .false.
end subroutine eio_weights_init_in
```

Switch from input to output: reopen the file for reading.

⟨*EIO weights: eio weights: TBP*⟩+≡

```
procedure :: switch_inout => eio_weights_switch_inout
```

⟨*EIO weights: procedures*⟩+≡

```
subroutine eio_weights_switch_inout (eio, success)
  class(eio_weights_t), intent(inout) :: eio
  logical, intent(out), optional :: success
  call msg_bug ("Weight stream: in-out switch not supported")
  if (present (success))  success = .false.
end subroutine eio_weights_switch_inout
```

Output an event. Write first the event indices, then weight and two values of the squared matrix element: sqme_ref is the value stored in the event record, and sqme_prc is the one stored in the process instance. (They can differ: when recalculating, the former is read from file and the latter is the result of the new calculation.)

For the alternative entries, the sqme value is always obtained by a new calculation, and thus qualifies as sqme_prc.

⟨*EIO weights: eio weights: TBP*⟩+≡
```
procedure :: output => eio_weights_output
```

⟨*EIO weights: procedures*⟩+≡
```
subroutine eio_weights_output (eio, event, i_prc, reading, passed, pacify)
  class(eio_weights_t), intent(inout) :: eio
  class(generic_event_t), intent(in), target :: event
  integer, intent(in) :: i_prc
  logical, intent(in), optional :: reading, passed, pacify
  integer :: n_alt, i
  real(default) :: weight, sqme_ref, sqme_prc
  if (eio%writing) then
     weight = event%get_weight_prc ()
     sqme_ref = event%get_sqme_ref ()
     sqme_prc = event%get_sqme_prc ()
     n_alt = event%get_n_alt ()
1    format (I0,3(1x,ES17.10),3(1x,I0))
2    format (I0,3(1x,ES15.8),3(1x,I0))
     if (eio%pacify) then
        write (eio%unit, 2)  0, weight, sqme_prc, sqme_ref, &
             i_prc
     else
        write (eio%unit, 1)  0, weight, sqme_prc, sqme_ref, &
             i_prc
     end if
     do i = 1, n_alt
        weight = event%get_weight_alt(i)
        sqme_prc = event%get_sqme_alt(i)
        if (eio%pacify) then
           write (eio%unit, 2)  i, weight, sqme_prc
        else
           write (eio%unit, 1)  i, weight, sqme_prc
        end if
     end do
  else
     call eio%write ()
     call msg_fatal ("Weight stream file is not open for writing")
  end if
end subroutine eio_weights_output
```

Input an event.

⟨*EIO weights: eio weights: TBP*⟩+≡
```
procedure :: input_i_prc => eio_weights_input_i_prc
procedure :: input_event => eio_weights_input_event
```

⟨*EIO weights: procedures*⟩+≡
```
subroutine eio_weights_input_i_prc (eio, i_prc, iostat)
```

```
    class(eio_weights_t), intent(inout) :: eio
    integer, intent(out) :: i_prc
    integer, intent(out) :: iostat
    call msg_bug ("Weight stream: event input not supported")
    i_prc = 0
    iostat = 1
  end subroutine eio_weights_input_i_prc

  subroutine eio_weights_input_event (eio, event, iostat)
    class(eio_weights_t), intent(inout) :: eio
    class(generic_event_t), intent(inout), target :: event
    integer, intent(out) :: iostat
    call msg_bug ("Weight stream: event input not supported")
    iostat = 1
  end subroutine eio_weights_input_event
```

⟨*EIO weights: eio weights: TBP*⟩+≡

```
  procedure :: skip => eio_weights_skip
```

⟨*EIO weights: procedures*⟩+≡

```
  subroutine eio_weights_skip (eio, iostat)
    class(eio_weights_t), intent(inout) :: eio
    integer, intent(out) :: iostat
    iostat = 0
  end subroutine eio_weights_skip
```

17.6.4 Unit tests

Test module, followed by the corresponding implementation module.

⟨eio_weights_ut.f90⟩≡

```
⟨File header⟩

module eio_weights_ut
  use unit_tests
  use eio_weights_uti

⟨Standard module head⟩

⟨EIO weights: public test⟩

contains

⟨EIO weights: test driver⟩

end module eio_weights_ut
```

⟨eio_weights_uti.f90⟩≡

```
⟨File header⟩

module eio_weights_uti

⟨Use kinds⟩
⟨Use strings⟩
```

```
  use io_units
  use event_base
  use eio_data
  use eio_base

  use eio_weights

  use eio_base_ut, only: eio_prepare_test, eio_cleanup_test
```

⟨*Standard module head*⟩

⟨*EIO weights: test declarations*⟩

```
contains
```

⟨*EIO weights: tests*⟩

```
end module eio_weights_uti
```

API: driver for the unit tests below.

⟨*EIO weights: public test*⟩≡

```
  public :: eio_weights_test
```

⟨*EIO weights: test driver*⟩≡

```
  subroutine eio_weights_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```

⟨*EIO weights: execute tests*⟩

```
  end subroutine eio_weights_test
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO weights: execute tests*⟩≡

```
  call test (eio_weights_1, "eio_weights_1", &
       "read and write event contents", &
       u, results)
```

⟨*EIO weights: test declarations*⟩≡

```
  public :: eio_weights_1
```

⟨*EIO weights: tests*⟩≡

```
  subroutine eio_weights_1 (u)
    integer, intent(in) :: u
    class(generic_event_t), pointer :: event
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file
    character(80) :: buffer

    write (u, "(A)")  "* Test output: eio_weights_1"
    write (u, "(A)")  "*   Purpose: generate an event and write weight to file"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize test process"
```

```
  call eio_prepare_test (event, unweighted = .false.)

  write (u, "(A)")
  write (u, "(A)")  "* Generate and write an event"
  write (u, "(A)")

  sample = "eio_weights_1"

  allocate (eio_weights_t :: eio)

  call eio%init_out (sample)
  call event%generate (1, [0._default, 0._default])

  call eio%output (event, i_prc = 42)
  call eio%write (u)
  call eio%final ()

  write (u, "(A)")
  write (u, "(A)")  "* File contents: &
       &(weight, sqme(evt), sqme(prc), i_prc, i_mci, i_term)"
  write (u, "(A)")

  u_file = free_unit ()
  open (u_file, file = "eio_weights_1.weights.dat", &
       action = "read", status = "old")
  read (u_file, "(A)")  buffer
  write (u, "(A)") trim (buffer)
  close (u_file)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call eio_cleanup_test (event)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: eio_weights_1"
end subroutine eio_weights_1
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO weights: execute tests*⟩+≡

```
call test (eio_weights_2, "eio_weights_2", &
     "multiple weights", &
     u, results)
```

⟨*EIO weights: test declarations*⟩+≡

```
public :: eio_weights_2
```

⟨*EIO weights: tests*⟩+≡

```
subroutine eio_weights_2 (u)
  integer, intent(in) :: u
  class(generic_event_t), pointer :: event
```

```
class(eio_t), allocatable :: eio
type(string_t) :: sample
integer :: u_file, i
character(80) :: buffer

write (u, "(A)")  "* Test output: eio_weights_2"
write (u, "(A)")  "*   Purpose: generate an event and write weight to file"
write (u, "(A)")

write (u, "(A)")  "* Initialize test process"

call eio_prepare_test (event, unweighted = .false., n_alt = 2)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_weights_2"

allocate (eio_weights_t :: eio)

call eio%init_out (sample)
select type (eio)
type is (eio_weights_t)
   call eio%set_parameters (pacify = .true.)
end select
call event%generate (1, [0._default, 0._default])
call event%set (sqme_alt = [2._default, 3._default])
call event%set (weight_alt = &
     [2 * event%get_weight_prc (), 3 * event%get_weight_prc ()])

call eio%output (event, i_prc = 42)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents: &
     &(weight, sqme(evt), sqme(prc), i_prc, i_mci, i_term)"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = "eio_weights_2.weights.dat", &
     action = "read", status = "old")
do i = 1, 3
   read (u_file, "(A)")  buffer
   write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call eio_cleanup_test (event)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_weights_2"

  end subroutine eio_weights_2
```

17.7 Event Dump Output

This is an output-only format. We simply dump the contents of the `particle_set`, using the `write` method of that type. The event-format options are the options of that procedure.

⟨`eio_dump.f90`⟩≡

```
⟨File header⟩

module eio_dump

  use, intrinsic :: iso_fortran_env, only: output_unit

  use kinds, only: i64
⟨Use strings⟩
  use format_utils, only: write_separator
  use format_utils, only: pac_fmt
  use format_defs, only: FMT_16, FMT_19
  use io_units
  use diagnostics
  use event_base
  use eio_data
  use eio_base

⟨Standard module head⟩

⟨EIO dump: public⟩

⟨EIO dump: types⟩

contains

⟨EIO dump: procedures⟩

end module eio_dump
```

17.7.1 Type

⟨*EIO dump: public*⟩≡

```
  public :: eio_dump_t
```

⟨*EIO dump: types*⟩≡

```
  type, extends (eio_t) :: eio_dump_t
     integer(i64) :: count = 0
     integer :: unit = 0
     logical :: writing = .false.
     logical :: screen = .false.
```

```
    logical :: pacify = .false.
    logical :: weights = .false.
    logical :: compressed = .false.
    logical :: summary = .false.
  contains
  ⟨EIO dump: eio dump: TBP⟩
end type eio_dump_t
```

17.7.2 Specific Methods

Set pacify flags.

⟨*EIO dump: eio dump: TBP*⟩≡

```
procedure :: set_parameters => eio_dump_set_parameters
```

⟨*EIO dump: procedures*⟩≡

```
subroutine eio_dump_set_parameters (eio, extension, &
     pacify, weights, compressed, summary, screen)
  class(eio_dump_t), intent(inout) :: eio
  type(string_t), intent(in), optional :: extension
  logical, intent(in), optional :: pacify
  logical, intent(in), optional :: weights
  logical, intent(in), optional :: compressed
  logical, intent(in), optional :: summary
  logical, intent(in), optional :: screen
  if (present (pacify))  eio%pacify = pacify
  if (present (weights))  eio%weights = weights
  if (present (compressed))  eio%compressed = compressed
  if (present (summary))  eio%summary = summary
  if (present (screen))  eio%screen = screen
  eio%extension = "pset.dat"
  if (present (extension))  eio%extension = extension
end subroutine eio_dump_set_parameters
```

17.7.3 Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO dump: eio dump: TBP*⟩+≡

```
procedure :: write => eio_dump_write
```

⟨*EIO dump: procedures*⟩+≡

```
subroutine eio_dump_write (object, unit)
  class(eio_dump_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Dump event stream:"
  if (object%writing) then
     write (u, "(3x,A,L1)") "Screen output     = ", object%screen
     write (u, "(3x,A,A)")  "Writing to file   = ", char (object%filename)
     write (u, "(3x,A,L1)") "Reduced I/O prec. = ", object%pacify
```

```
      write (u, "(3x,A,L1)") "Show weights/sqme = ", object%weights
      write (u, "(3x,A,L1)") "Compressed        = ", object%compressed
      write (u, "(3x,A,L1)") "Summary           = ", object%summary
   else
      write (u, "(3x,A)")  "[closed]"
   end if
end subroutine eio_dump_write
```

Finalizer: close any open file.

⟨*EIO dump: eio dump: TBP*⟩+≡

```
procedure :: final => eio_dump_final
```

⟨*EIO dump: procedures*⟩+≡

```
subroutine eio_dump_final (object)
   class(eio_dump_t), intent(inout) :: object
   if (object%screen) then
      write (msg_buffer, "(A,A,A)")  "Events: display complete"
      call msg_message ()
      object%screen = .false.
   end if
   if (object%writing) then
      write (msg_buffer, "(A,A,A)")  "Events: closing event dump file '", &
           char (object%filename), "'"
      call msg_message ()
      close (object%unit)
      object%writing = .false.
   end if
end subroutine eio_dump_final
```

Initialize event writing.

⟨*EIO dump: eio dump: TBP*⟩+≡

```
procedure :: init_out => eio_dump_init_out
```

⟨*EIO dump: procedures*⟩+≡

```
subroutine eio_dump_init_out (eio, sample, data, success, extension)
   class(eio_dump_t), intent(inout) :: eio
   type(string_t), intent(in) :: sample
   type(string_t), intent(in), optional :: extension
   type(event_sample_data_t), intent(in), optional :: data
   logical, intent(out), optional :: success
   if (present(extension)) then
      eio%extension = extension
   else
      eio%extension = "pset.dat"
   end if
   if (sample /= "") then
      eio%filename = sample // "." // eio%extension
      eio%unit = free_unit ()
      write (msg_buffer, "(A,A,A)")  "Events: writing to event dump file '", &
           char (eio%filename), "'"
      call msg_message ()
      eio%writing = .true.
      open (eio%unit, file = char (eio%filename), &
           action = "write", status = "replace")
```

```
    end if
    if (eio%screen) then
       write (msg_buffer, "(A,A,A)")  "Events: display on standard output"
       call msg_message ()
    end if
    eio%count = 0
    if (present (success))  success = .true.
  end subroutine eio_dump_init_out
```

Initialize event reading.

⟨*EIO dump: eio dump: TBP*⟩+≡

```
  procedure :: init_in => eio_dump_init_in
```

⟨*EIO dump: procedures*⟩+≡

```
  subroutine eio_dump_init_in (eio, sample, data, success, extension)
    class(eio_dump_t), intent(inout) :: eio
    type(string_t), intent(in) :: sample
    type(string_t), intent(in), optional :: extension
    type(event_sample_data_t), intent(inout), optional :: data
    logical, intent(out), optional :: success
    call msg_bug ("Event dump: event input not supported")
    if (present (success))  success = .false.
  end subroutine eio_dump_init_in
```

Switch from input to output: reopen the file for reading.

⟨*EIO dump: eio dump: TBP*⟩+≡

```
  procedure :: switch_inout => eio_dump_switch_inout
```

⟨*EIO dump: procedures*⟩+≡

```
  subroutine eio_dump_switch_inout (eio, success)
    class(eio_dump_t), intent(inout) :: eio
    logical, intent(out), optional :: success
    call msg_bug ("Event dump: in-out switch not supported")
    if (present (success))  success = .false.
  end subroutine eio_dump_switch_inout
```

Output an event. Delegate the output call to the `write` method of the current particle set, if valid. Output both to file (if defined) and to screen (if requested).

⟨*EIO dump: eio dump: TBP*⟩+≡

```
  procedure :: output => eio_dump_output
```

⟨*EIO dump: procedures*⟩+≡

```
  subroutine eio_dump_output (eio, event, i_prc, reading, passed, pacify)
    class(eio_dump_t), intent(inout) :: eio
    class(generic_event_t), intent(in), target :: event
    integer, intent(in) :: i_prc
    logical, intent(in), optional :: reading, passed, pacify
    character(len=7) :: fmt
    integer :: i
    eio%count = eio%count + 1
    call pac_fmt (fmt, FMT_19, FMT_16, pacify)
    if (eio%writing)  call dump (eio%unit)
    if (eio%screen) then
```

```
     call dump (output_unit)
     if (logfile_unit () > 0)  call dump (logfile_unit ())
  end if
contains
  subroutine dump (u)
    integer, intent(in) :: u
    call write_separator (u, 2)
    write (u, "(1x,A,I0)")  "Event #", eio%count
    if (eio%weights) then
       call write_separator (u)
       if (event%sqme_ref_known) then
          write (u, "(1x,A," // fmt // ")")  "sqme (ref)   = ", &
               event%sqme_ref
       else
          write (u, "(1x,A)")  "sqme (ref)    = [undefined]"
       end if
       if (event%sqme_ref_known) then
          write (u, "(1x,A," // fmt // ")")  "sqme (prc)   = ", &
               event%sqme_prc
       else
          write (u, "(1x,A)")  "sqme (prc)    = [undefined]"
       end if
       if (event%weight_ref_known) then
          write (u, "(1x,A," // fmt // ")")  "weight (ref) = ", &
               event%weight_ref
       else
          write (u, "(1x,A)")  "weight (ref) = [undefined]"
       end if
       if (event%weight_prc_known) then
          write (u, "(1x,A," // fmt // ")")  "weight (prc) = ", &
               event%weight_prc
       else
          write (u, "(1x,A)")  "weight (prc) = [undefined]"
       end if
       if (event%excess_prc_known) then
          write (u, "(1x,A," // fmt // ")")  "excess (prc) = ", &
               event%excess_prc
       else
          write (u, "(1x,A)")  "excess (prc) = [undefined]"
       end if
       do i = 1, event%n_alt
          if (event%sqme_ref_known) then
             write (u, "(1x,A,I0,A," // fmt // ")")  "sqme (", i, ")     = ",&
                  event%sqme_prc
          else
             write (u, "(1x,A,I0,A)")  "sqme (", i, ")    = [undefined]"
          end if
          if (event%weight_prc_known) then
             write (u, "(1x,A,I0,A," // fmt // ")")  "weight (", i, ")   = ",&
                  event%weight_prc
          else
             write (u, "(1x,A,I0,A)")  "weight (", i, ")   = [undefined]"
          end if
       end do
```

```
      end if
      call write_separator (u)
      if (event%particle_set_is_valid) then
         call event%particle_set%write (unit = u, &
              summary = eio%summary, compressed = eio%compressed, &
              testflag = eio%pacify)
      else
         write (u, "(1x,A)")  "Particle set: [invalid]"
      end if
    end subroutine dump
  end subroutine eio_dump_output
```

Input an event.

⟨*EIO dump: eio dump: TBP*⟩+≡

```
  procedure :: input_i_prc => eio_dump_input_i_prc
  procedure :: input_event => eio_dump_input_event
```

⟨*EIO dump: procedures*⟩+≡

```
  subroutine eio_dump_input_i_prc (eio, i_prc, iostat)
    class(eio_dump_t), intent(inout) :: eio
    integer, intent(out) :: i_prc
    integer, intent(out) :: iostat
    call msg_bug ("Dump stream: event input not supported")
    i_prc = 0
    iostat = 1
  end subroutine eio_dump_input_i_prc

  subroutine eio_dump_input_event (eio, event, iostat)
    class(eio_dump_t), intent(inout) :: eio
    class(generic_event_t), intent(inout), target :: event
    integer, intent(out) :: iostat
    call msg_bug ("Dump stream: event input not supported")
    iostat = 1
  end subroutine eio_dump_input_event
```

⟨*EIO dump: eio dump: TBP*⟩+≡

```
  procedure :: skip => eio_dump_skip
```

⟨*EIO dump: procedures*⟩+≡

```
  subroutine eio_dump_skip (eio, iostat)
    class(eio_dump_t), intent(inout) :: eio
    integer, intent(out) :: iostat
    iostat = 0
  end subroutine eio_dump_skip
```

17.7.4 Unit tests

Test module, followed by the corresponding implementation module.

⟨`eio_dump_ut.f90`⟩≡

⟨*File header*⟩

```
module eio_dump_ut
```

```
  use unit_tests
  use eio_dump_uti

⟨Standard module head⟩

⟨EIO dump: public test⟩

contains

⟨EIO dump: test driver⟩

end module eio_dump_ut
```

⟨eio_dump_uti.f90⟩≡

```
⟨File header⟩

module eio_dump_uti

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use event_base
  use eio_data
  use eio_base

  use eio_dump

  use eio_base_ut, only: eio_prepare_test, eio_cleanup_test

⟨Standard module head⟩

⟨EIO dump: test declarations⟩

contains

⟨EIO dump: tests⟩

end module eio_dump_uti
```

API: driver for the unit tests below.

⟨*EIO dump: public test*⟩≡

```
  public :: eio_dump_test
```

⟨*EIO dump: test driver*⟩≡

```
  subroutine eio_dump_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨EIO dump: execute tests⟩
  end subroutine eio_dump_test
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO dump: execute tests*⟩≡

```
    call test (eio_dump_1, "eio_dump_1", &
         "read and write event contents", &
         u, results)
```

⟨*EIO dump: test declarations*⟩≡

```
    public :: eio_dump_1
```

⟨*EIO dump: tests*⟩≡

```
    subroutine eio_dump_1 (u)
      integer, intent(in) :: u
      class(generic_event_t), pointer :: event
      class(eio_t), allocatable :: eio
      integer :: u_file

      write (u, "(A)")  "* Test output: eio_dump_1"
      write (u, "(A)")  "*   Purpose: generate an event and dump to output"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize test process"

      call eio_prepare_test (event, unweighted = .false.)

      write (u, "(A)")
      write (u, "(A)")  "* Generate and write an event"
      write (u, "(A)")

      allocate (eio_dump_t :: eio)
      select type (eio)
      type is (eio_dump_t)
         eio%unit = u
         eio%writing = .true.
         eio%weights = .true.
      end select

      call eio%init_out (var_str (""))
      call event%generate (1, [0._default, 0._default])

      call eio%output (event, i_prc = 42)

      write (u, "(A)")
      write (u, "(A)")  "* Contents of eio_dump object"
      write (u, "(A)")

      call eio%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      select type (eio)
      type is (eio_dump_t)
         eio%writing = .false.
      end select
      call eio%final ()

      call eio_cleanup_test (event)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_dump_1"
  end subroutine eio_dump_1
```

17.8 ASCII File Formats

Here, we implement several ASCII file formats. It is possible to switch between them using flags.

⟨eio_ascii.f90⟩≡

```
⟨File header⟩

module eio_ascii

⟨Use strings⟩
  use io_units
  use diagnostics
  use event_base
  use eio_data
  use eio_base
  use hep_common
  use hep_events

⟨Standard module head⟩

⟨EIO ascii: public⟩

⟨EIO ascii: types⟩

contains

⟨EIO ascii: procedures⟩

end module eio_ascii
```

17.8.1 Type

⟨EIO ascii: public⟩≡

```
  public :: eio_ascii_t
```

⟨EIO ascii: types⟩≡

```
  type, abstract, extends (eio_t) :: eio_ascii_t
     logical :: writing = .false.
     integer :: unit = 0
     logical :: keep_beams = .false.
     logical :: keep_remnants = .true.
     logical :: ensure_order = .false.
   contains
   ⟨EIO ascii: eio ascii: TBP⟩
  end type eio_ascii_t
```

⟨*EIO ascii: public*⟩+≡
```
public :: eio_ascii_ascii_t
```
⟨*EIO ascii: types*⟩+≡
```
type, extends (eio_ascii_t) :: eio_ascii_ascii_t
end type eio_ascii_ascii_t
```

⟨*EIO ascii: public*⟩+≡
```
public :: eio_ascii_athena_t
```
⟨*EIO ascii: types*⟩+≡
```
type, extends (eio_ascii_t) :: eio_ascii_athena_t
end type eio_ascii_athena_t
```

The debug format has a few options that can be controlled by Sindarin variables.

⟨*EIO ascii: public*⟩+≡
```
public :: eio_ascii_debug_t
```
⟨*EIO ascii: types*⟩+≡
```
type, extends (eio_ascii_t) :: eio_ascii_debug_t
   logical :: show_process = .true.
   logical :: show_transforms = .true.
   logical :: show_decay = .true.
   logical :: verbose = .true.
end type eio_ascii_debug_t
```

⟨*EIO ascii: public*⟩+≡
```
public :: eio_ascii_hepevt_t
```
⟨*EIO ascii: types*⟩+≡
```
type, extends (eio_ascii_t) :: eio_ascii_hepevt_t
end type eio_ascii_hepevt_t
```

⟨*EIO ascii: public*⟩+≡
```
public :: eio_ascii_hepevt_verb_t
```
⟨*EIO ascii: types*⟩+≡
```
type, extends (eio_ascii_t) :: eio_ascii_hepevt_verb_t
end type eio_ascii_hepevt_verb_t
```

⟨*EIO ascii: public*⟩+≡
```
public :: eio_ascii_lha_t
```
⟨*EIO ascii: types*⟩+≡
```
type, extends (eio_ascii_t) :: eio_ascii_lha_t
end type eio_ascii_lha_t
```

⟨*EIO ascii: public*⟩+≡
```
public :: eio_ascii_lha_verb_t
```
⟨*EIO ascii: types*⟩+≡
```
type, extends (eio_ascii_t) :: eio_ascii_lha_verb_t
end type eio_ascii_lha_verb_t
```

⟨*EIO ascii: public*⟩+≡
```
public :: eio_ascii_long_t
```
⟨*EIO ascii: types*⟩+≡
```
type, extends (eio_ascii_t) :: eio_ascii_long_t
end type eio_ascii_long_t
```

⟨*EIO ascii: public*⟩+≡
```
public :: eio_ascii_mokka_t
```
⟨*EIO ascii: types*⟩+≡
```
type, extends (eio_ascii_t) :: eio_ascii_mokka_t
end type eio_ascii_mokka_t
```

⟨*EIO ascii: public*⟩+≡
```
public :: eio_ascii_short_t
```
⟨*EIO ascii: types*⟩+≡
```
type, extends (eio_ascii_t) :: eio_ascii_short_t
end type eio_ascii_short_t
```

17.8.2 Specific Methods

Set parameters that are specifically used with ASCII file formats. In particular, this is the file extension.

⟨*EIO ascii: eio ascii: TBP*⟩≡
```
procedure :: set_parameters => eio_ascii_set_parameters
```
⟨*EIO ascii: procedures*⟩≡
```
subroutine eio_ascii_set_parameters (eio, &
     keep_beams, keep_remnants, ensure_order, extension, &
     show_process, show_transforms, show_decay, verbose)
  class(eio_ascii_t), intent(inout) :: eio
  logical, intent(in), optional :: keep_beams
  logical, intent(in), optional :: keep_remnants
  logical, intent(in), optional :: ensure_order
  type(string_t), intent(in), optional :: extension
  logical, intent(in), optional :: show_process, show_transforms, show_decay
  logical, intent(in), optional :: verbose
  if (present (keep_beams))  eio%keep_beams = keep_beams
  if (present (keep_remnants))  eio%keep_remnants = keep_remnants
  if (present (ensure_order))  eio%ensure_order = ensure_order
  if (present (extension)) then
     eio%extension = extension
  else
     select type (eio)
     type is (eio_ascii_ascii_t)
        eio%extension = "evt"
     type is (eio_ascii_athena_t)
        eio%extension = "athena.evt"
     type is (eio_ascii_debug_t)
        eio%extension = "debug"
     type is (eio_ascii_hepevt_t)
```

```
         eio%extension = "hepevt"
      type is (eio_ascii_hepevt_verb_t)
         eio%extension = "hepevt.verb"
      type is (eio_ascii_lha_t)
         eio%extension = "lha"
      type is (eio_ascii_lha_verb_t)
         eio%extension = "lha.verb"
      type is (eio_ascii_long_t)
         eio%extension = "long.evt"
      type is (eio_ascii_mokka_t)
         eio%extension = "mokka.evt"
      type is (eio_ascii_short_t)
         eio%extension = "short.evt"
      end select
   end if
   select type (eio)
   type is (eio_ascii_debug_t)
      if (present (show_process))  eio%show_process = show_process
      if (present (show_transforms))  eio%show_transforms = show_transforms
      if (present (show_decay))  eio%show_decay = show_decay
      if (present (verbose))  eio%verbose = verbose
   end select
end subroutine eio_ascii_set_parameters
```

17.8.3 Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO ascii: eio ascii: TBP*⟩+≡

```
procedure :: write => eio_ascii_write
```

⟨*EIO ascii: procedures*⟩+≡

```
subroutine eio_ascii_write (object, unit)
  class(eio_ascii_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  select type (object)
  type is (eio_ascii_ascii_t)
     write (u, "(1x,A)")  "ASCII event stream (default format):"
  type is (eio_ascii_athena_t)
     write (u, "(1x,A)")  "ASCII event stream (ATHENA format):"
  type is (eio_ascii_debug_t)
     write (u, "(1x,A)")  "ASCII event stream (Debugging format):"
  type is (eio_ascii_hepevt_t)
     write (u, "(1x,A)")  "ASCII event stream (HEPEVT format):"
  type is (eio_ascii_hepevt_verb_t)
     write (u, "(1x,A)")  "ASCII event stream (verbose HEPEVT format):"
  type is (eio_ascii_lha_t)
     write (u, "(1x,A)")  "ASCII event stream (LHA format):"
  type is (eio_ascii_lha_verb_t)
     write (u, "(1x,A)")  "ASCII event stream (verbose LHA format):"
  type is (eio_ascii_long_t)
```

```
      write (u, "(1x,A)")  "ASCII event stream (long format):"
   type is (eio_ascii_mokka_t)
      write (u, "(1x,A)")  "ASCII event stream (MOKKA format):"
   type is (eio_ascii_short_t)
      write (u, "(1x,A)")  "ASCII event stream (short format):"
   end select
   if (object%writing) then
      write (u, "(3x,A,A)")  "Writing to file   = ", char (object%filename)
   else
      write (u, "(3x,A)")  "[closed]"
   end if
   write (u, "(3x,A,L1)")    "Keep beams        = ", object%keep_beams
   write (u, "(3x,A,L1)")    "Keep remnants     = ", object%keep_remnants
   select type (object)
   type is (eio_ascii_debug_t)
      write (u, "(3x,A,L1)")    "Show process      = ", object%show_process
      write (u, "(3x,A,L1)")    "Show transforms   = ", object%show_transforms
      write (u, "(3x,A,L1)")    "Show decay tree   = ", object%show_decay
      write (u, "(3x,A,L1)")    "Verbose output    = ", object%verbose
   end select
end subroutine eio_ascii_write
```

Finalizer: close any open file.

⟨*EIO ascii: eio ascii: TBP*⟩+≡

```
procedure :: final => eio_ascii_final
```

⟨*EIO ascii: procedures*⟩+≡

```
subroutine eio_ascii_final (object)
  class(eio_ascii_t), intent(inout) :: object
  if (object%writing) then
     write (msg_buffer, "(A,A,A)")  "Events: closing ASCII file '", &
          char (object%filename), "'"
     call msg_message ()
     close (object%unit)
     object%writing = .false.
  end if
end subroutine eio_ascii_final
```

Initialize event writing.

Check weight normalization. This applies to all ASCII-type files that use the HEPRUP common block. We can't allow normalization conventions that are not covered by the HEPRUP definition.

⟨*EIO ascii: eio ascii: TBP*⟩+≡

```
procedure :: init_out => eio_ascii_init_out
```

⟨*EIO ascii: procedures*⟩+≡

```
subroutine eio_ascii_init_out (eio, sample, data, success, extension)
  class(eio_ascii_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(string_t), intent(in), optional :: extension
  type(event_sample_data_t), intent(in), optional :: data
  logical, intent(out), optional :: success
  integer :: i
  if (.not. present (data)) &
```

```
      call msg_bug ("ASCII initialization: missing data")
   if (data%n_beam /= 2) &
      call msg_fatal ("ASCII: defined for scattering processes only")
   eio%sample = sample
   call eio%check_normalization (data)
   call eio%set_splitting (data)
   call eio%set_filename ()
   eio%unit = free_unit ()
   write (msg_buffer, "(A,A,A)")  "Events: writing to ASCII file '", &
      char (eio%filename), "'"
   call msg_message ()
   eio%writing = .true.
   open (eio%unit, file = char (eio%filename), &
      action = "write", status = "replace")
   select type (eio)
   type is (eio_ascii_lha_t)
      call heprup_init &
         (data%pdg_beam, &
         data%energy_beam, &
         n_processes = data%n_proc, &
         unweighted = data%unweighted, &
         negative_weights = data%negative_weights)
      do i = 1, data%n_proc
         call heprup_set_process_parameters (i = i, &
            process_id = data%proc_num_id(i), &
            cross_section = data%cross_section(i), &
            error = data%error(i))
      end do
      call heprup_write_ascii (eio%unit)
   type is (eio_ascii_lha_verb_t)
      call heprup_init &
         (data%pdg_beam, &
         data%energy_beam, &
         n_processes = data%n_proc, &
         unweighted = data%unweighted, &
         negative_weights = data%negative_weights)
      do i = 1, data%n_proc
         call heprup_set_process_parameters (i = i, &
            process_id = data%proc_num_id(i), &
            cross_section = data%cross_section(i), &
            error = data%error(i))
      end do
      call heprup_write_verbose (eio%unit)
   end select
   if (present (success))  success = .true.
end subroutine eio_ascii_init_out
```

Some event properties do not go well with some output formats. In particular, many formats require unweighted events.

⟨*EIO ascii: eio ascii: TBP*⟩+≡

```
procedure :: check_normalization => eio_ascii_check_normalization
```

⟨*EIO ascii: procedures*⟩+≡

```
subroutine eio_ascii_check_normalization (eio, data)
```

```
    class(eio_ascii_t), intent(in) :: eio
    type(event_sample_data_t), intent(in) :: data
    if (data%unweighted) then
    else
       select type (eio)
       type is (eio_ascii_athena_t);  call msg_fatal &
            ("Event output (Athena format): events must be unweighted.")
       type is (eio_ascii_hepevt_t);  call msg_fatal &
            ("Event output (HEPEVT format): events must be unweighted.")
       type is (eio_ascii_hepevt_verb_t);  call msg_fatal &
            ("Event output (HEPEVT format): events must be unweighted.")
       end select
       select case (data%norm_mode)
       case (NORM_SIGMA)
       case default
          select type (eio)
          type is (eio_ascii_lha_t)
             call msg_fatal &
                  ("Event output (LHA): normalization for weighted events &
                  &must be 'sigma'")
          type is (eio_ascii_lha_verb_t)
             call msg_fatal &
                  ("Event output (LHA): normalization for weighted events &
                  &must be 'sigma'")
          end select
       end select
    end if
  end subroutine eio_ascii_check_normalization
```

Initialize event reading.

⟨*EIO ascii: eio ascii: TBP*⟩+≡

```
  procedure :: init_in => eio_ascii_init_in
```

⟨*EIO ascii: procedures*⟩+≡

```
  subroutine eio_ascii_init_in (eio, sample, data, success, extension)
    class(eio_ascii_t), intent(inout) :: eio
    type(string_t), intent(in) :: sample
    type(string_t), intent(in), optional :: extension
    type(event_sample_data_t), intent(inout), optional :: data
    logical, intent(out), optional :: success
    call msg_bug ("ASCII: event input not supported")
    if (present (success))  success = .false.
  end subroutine eio_ascii_init_in
```

Switch from input to output: reopen the file for reading.

⟨*EIO ascii: eio ascii: TBP*⟩+≡

```
  procedure :: switch_inout => eio_ascii_switch_inout
```

⟨*EIO ascii: procedures*⟩+≡

```
  subroutine eio_ascii_switch_inout (eio, success)
    class(eio_ascii_t), intent(inout) :: eio
    logical, intent(out), optional :: success
    call msg_bug ("ASCII: in-out switch not supported")
    if (present (success))  success = .false.
```

```
end subroutine eio_ascii_switch_inout
```

Split event file: increment the counter, close the current file, open a new one. If the file needs a header, repeat it for the new file. (We assume that the common block contents are still intact.)

⟨*EIO ascii: eio ascii: TBP*⟩+≡

```
procedure :: split_out => eio_ascii_split_out
```

⟨*EIO ascii: procedures*⟩+≡

```
subroutine eio_ascii_split_out (eio)
  class(eio_ascii_t), intent(inout) :: eio
  if (eio%split) then
     eio%split_index = eio%split_index + 1
     call eio%set_filename ()
     write (msg_buffer, "(A,A,A)")  "Events: writing to ASCII file '", &
          char (eio%filename), "'"
     call msg_message ()
     close (eio%unit)
     open (eio%unit, file = char (eio%filename), &
          action = "write", status = "replace")
     select type (eio)
     type is (eio_ascii_lha_t)
        call heprup_write_ascii (eio%unit)
     type is (eio_ascii_lha_verb_t)
        call heprup_write_verbose (eio%unit)
     end select
  end if
end subroutine eio_ascii_split_out
```

Output an event. Write first the event indices, then weight and squared matrix element, then the particle set.

⟨*EIO ascii: eio ascii: TBP*⟩+≡

```
procedure :: output => eio_ascii_output
```

⟨*EIO ascii: procedures*⟩+≡

```
subroutine eio_ascii_output (eio, event, i_prc, reading, passed, pacify)
  class(eio_ascii_t), intent(inout) :: eio
  class(generic_event_t), intent(in), target :: event
  integer, intent(in) :: i_prc
  logical, intent(in), optional :: reading, passed, pacify
  if (present (passed)) then
     if (.not. passed)  return
  end if
  if (eio%writing) then
     select type (eio)
     type is (eio_ascii_lha_t)
        call hepeup_from_event (event, &
             process_index = i_prc, &
             keep_beams = eio%keep_beams, &
             keep_remnants = eio%keep_remnants)
        call hepeup_write_lha (eio%unit)
     type is (eio_ascii_lha_verb_t)
        call hepeup_from_event (event, &
             process_index = i_prc, &
```

```
        keep_beams = eio%keep_beams, &
        keep_remnants = eio%keep_remnants)
   call hepeup_write_verbose (eio%unit)
type is (eio_ascii_ascii_t)
   call event%write (eio%unit, &
        show_process = .false., &
        show_transforms = .false., &
        show_decay = .false., &
        verbose = .false., testflag = pacify)
type is (eio_ascii_athena_t)
   call hepevt_from_event (event, &
        i_evt = event%get_index (), &
        keep_beams = eio%keep_beams, &
        keep_remnants = eio%keep_remnants, &
        ensure_order = eio%ensure_order)
   call hepevt_write_athena (eio%unit)
type is (eio_ascii_debug_t)
   call event%write (eio%unit, &
        show_process = eio%show_process, &
        show_transforms = eio%show_transforms, &
        show_decay = eio%show_decay, &
        verbose = eio%verbose, &
        testflag = pacify)
type is (eio_ascii_hepevt_t)
   call hepevt_from_event (event, &
        i_evt = event%get_index (), &
        keep_beams = eio%keep_beams, &
        keep_remnants = eio%keep_remnants, &
        ensure_order = eio%ensure_order)
   call hepevt_write_hepevt (eio%unit)
type is (eio_ascii_hepevt_verb_t)
   call hepevt_from_event (event, &
        i_evt = event%get_index (), &
        keep_beams = eio%keep_beams, &
        keep_remnants = eio%keep_remnants, &
        ensure_order = eio%ensure_order)
   call hepevt_write_verbose (eio%unit)
type is (eio_ascii_long_t)
   call hepevt_from_event (event, &
        i_evt = event%get_index (), &
        keep_beams = eio%keep_beams, &
        keep_remnants = eio%keep_remnants, &
        ensure_order = eio%ensure_order)
   call hepevt_write_ascii (eio%unit, .true.)
type is (eio_ascii_mokka_t)
   call hepevt_from_event (event, &
        i_evt = event%get_index (), &
        keep_beams = eio%keep_beams, &
        keep_remnants = eio%keep_remnants, &
        ensure_order = eio%ensure_order)
   call hepevt_write_mokka (eio%unit)
type is (eio_ascii_short_t)
   call hepevt_from_event (event, &
        i_evt = event%get_index (), &
```

```
                keep_beams = eio%keep_beams, &
                keep_remnants = eio%keep_remnants, &
                ensure_order = eio%ensure_order)
            call hepevt_write_ascii (eio%unit, .false.)
         end select
      else
         call eio%write ()
         call msg_fatal ("ASCII file is not open for writing")
      end if
   end subroutine eio_ascii_output
```

Input an event.

⟨*EIO ascii: eio ascii: TBP*⟩+≡

```
   procedure :: input_i_prc => eio_ascii_input_i_prc
   procedure :: input_event => eio_ascii_input_event
```

⟨*EIO ascii: procedures*⟩+≡

```
   subroutine eio_ascii_input_i_prc (eio, i_prc, iostat)
     class(eio_ascii_t), intent(inout) :: eio
     integer, intent(out) :: i_prc
     integer, intent(out) :: iostat
     call msg_bug ("ASCII: event input not supported")
     i_prc = 0
     iostat = 1
   end subroutine eio_ascii_input_i_prc

   subroutine eio_ascii_input_event (eio, event, iostat)
     class(eio_ascii_t), intent(inout) :: eio
     class(generic_event_t), intent(inout), target :: event
     integer, intent(out) :: iostat
     call msg_bug ("ASCII: event input not supported")
     iostat = 1
   end subroutine eio_ascii_input_event
```

⟨*EIO ascii: eio ascii: TBP*⟩+≡

```
   procedure :: skip => eio_ascii_skip
```

⟨*EIO ascii: procedures*⟩+≡

```
   subroutine eio_ascii_skip (eio, iostat)
      class(eio_ascii_t), intent(inout) :: eio
      integer, intent(out) :: iostat
      iostat = 0
   end subroutine eio_ascii_skip
```

17.8.4 Unit tests

Test module, followed by the corresponding implementation module.

⟨eio_ascii_ut.f90⟩≡

```
  ⟨File header⟩

  module eio_ascii_ut
    use unit_tests
```

```
  use eio_ascii_uti

⟨Standard module head⟩

⟨EIO ascii: public test⟩

contains

⟨EIO ascii: test driver⟩

end module eio_ascii_ut
```

⟨eio_ascii_uti.f90⟩≡

```
⟨File header⟩

module eio_ascii_uti

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use model_data
  use event_base
  use eio_data
  use eio_base

  use eio_ascii

  use eio_base_ut, only: eio_prepare_test, eio_cleanup_test

⟨Standard module head⟩

⟨EIO ascii: test declarations⟩

contains

⟨EIO ascii: tests⟩

end module eio_ascii_uti
```

API: driver for the unit tests below.

⟨*EIO ascii: public test*⟩≡

```
  public :: eio_ascii_test
```

⟨*EIO ascii: test driver*⟩≡

```
  subroutine eio_ascii_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨EIO ascii: execute tests⟩
  end subroutine eio_ascii_test
```

Test I/O methods

We test the implementation of all I/O methods, method `ascii`:

⟨*EIO ascii: execute tests*⟩≡

```
    call test (eio_ascii_1, "eio_ascii_1", &
         "read and write event contents, format [ascii]", &
         u, results)
```

⟨*EIO ascii: test declarations*⟩≡

```
    public :: eio_ascii_1
```

⟨*EIO ascii: tests*⟩≡

```
    subroutine eio_ascii_1 (u)
      integer, intent(in) :: u
      class(generic_event_t), pointer :: event
      type(event_sample_data_t) :: data
      class(eio_t), allocatable :: eio
      type(string_t) :: sample
      integer :: u_file, iostat
      character(80) :: buffer

      write (u, "(A)")  "* Test output: eio_ascii_1"
      write (u, "(A)")  "*   Purpose: generate an event in ASCII ascii format"
      write (u, "(A)")  "*      and write weight to file"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize test process"

      call eio_prepare_test (event, unweighted = .false.)

      call data%init (1)
      data%n_evt = 1
      data%n_beam = 2
      data%pdg_beam = 25
      data%energy_beam = 500
      data%proc_num_id = [42]
      data%cross_section(1) = 100
      data%error(1) = 1
      data%total_cross_section = sum (data%cross_section)

      write (u, "(A)")
      write (u, "(A)")  "* Generate and write an event"
      write (u, "(A)")

      sample = "eio_ascii_1"

      allocate (eio_ascii_ascii_t :: eio)

      select type (eio)
      class is (eio_ascii_t);  call eio%set_parameters ()
      end select
      call eio%init_out (sample, data)
      call event%generate (1, [0._default, 0._default])
      call event%evaluate_expressions ()

      call eio%output (event, i_prc = 1)
      call eio%write (u)
      call eio%final ()
```

```
    write (u, "(A)")
    write (u, "(A)")  "* File contents:"
    write (u, "(A)")

    u_file = free_unit ()
    open (u_file, file = char (sample // ".evt"), &
         action = "read", status = "old")
    do
       read (u_file, "(A)", iostat = iostat)  buffer
       if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
       if (iostat /= 0)  exit
       write (u, "(A)") trim (buffer)
    end do
    close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Reset data"
    write (u, "(A)")

    deallocate (eio)
    allocate (eio_ascii_ascii_t :: eio)

    select type (eio)
    type is (eio_ascii_ascii_t)
       call eio%set_parameters (keep_beams = .true.)
    end select
    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_ascii_1"

  end subroutine eio_ascii_1
```

We test the implementation of all I/O methods, method `athena`:

⟨*EIO ascii: execute tests*⟩+≡

```
  call test (eio_ascii_2, "eio_ascii_2", &
       "read and write event contents, format [athena]", &
       u, results)
```

⟨*EIO ascii: test declarations*⟩+≡

```
  public :: eio_ascii_2
```

⟨*EIO ascii: tests*⟩+≡

```
  subroutine eio_ascii_2 (u)
    integer, intent(in) :: u
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat
```

```
character(80) :: buffer

write (u, "(A)")  "* Test output: eio_ascii_2"
write (u, "(A)")  "*   Purpose: generate an event in ASCII athena format"
write (u, "(A)")  "*      and write weight to file"
write (u, "(A)")

write (u, "(A)")  "* Initialize test process"

call eio_prepare_test (event, unweighted = .false.)

call data%init (1)
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_ascii_2"

allocate (eio_ascii_athena_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char(sample // ".athena.evt"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Reset data"
    write (u, "(A)")

    deallocate (eio)
    allocate (eio_ascii_athena_t :: eio)

    select type (eio)
    type is (eio_ascii_athena_t)
       call eio%set_parameters (keep_beams = .true.)
    end select
    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_ascii_2"

  end subroutine eio_ascii_2
```

We test the implementation of all I/O methods, method `debug`:

⟨*EIO ascii: execute tests*⟩+≡

```
  call test (eio_ascii_3, "eio_ascii_3", &
       "read and write event contents, format [debug]", &
       u, results)
```

⟨*EIO ascii: test declarations*⟩+≡

```
  public :: eio_ascii_3
```

⟨*EIO ascii: tests*⟩+≡

```
  subroutine eio_ascii_3 (u)
    integer, intent(in) :: u
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat
    character(80) :: buffer

    write (u, "(A)")  "* Test output: eio_ascii_3"
    write (u, "(A)")  "*   Purpose: generate an event in ASCII debug format"
    write (u, "(A)")  "*      and write weight to file"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize test process"

    call eio_prepare_test (event, unweighted = .false.)

    call data%init (1)
    data%n_evt = 1
    data%n_beam = 2
```

```
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_ascii_3"

allocate (eio_ascii_debug_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".debug"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Reset data"
write (u, "(A)")

deallocate (eio)
allocate (eio_ascii_debug_t :: eio)

select type (eio)
type is (eio_ascii_debug_t)
   call eio%set_parameters (keep_beams = .true.)
end select
call eio%write (u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_ascii_3"

  end subroutine eio_ascii_3
```

We test the implementation of all I/O methods, method `hepevt`:

⟨*EIO ascii: execute tests*⟩+≡

```
  call test (eio_ascii_4, "eio_ascii_4", &
       "read and write event contents, format [hepevt]", &
       u, results)
```

⟨*EIO ascii: test declarations*⟩+≡

```
  public :: eio_ascii_4
```

⟨*EIO ascii: tests*⟩+≡

```
  subroutine eio_ascii_4 (u)
    integer, intent(in) :: u
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat
    character(80) :: buffer

    write (u, "(A)")  "* Test output: eio_ascii_4"
    write (u, "(A)")  "*   Purpose: generate an event in ASCII hepevt format"
    write (u, "(A)")  "*      and write weight to file"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize test process"

    call eio_prepare_test (event, unweighted = .false.)

    call data%init (1)
    data%n_evt = 1
    data%n_beam = 2
    data%pdg_beam = 25
    data%energy_beam = 500
    data%proc_num_id = [42]
    data%cross_section(1) = 100
    data%error(1) = 1
    data%total_cross_section = sum (data%cross_section)

    write (u, "(A)")
    write (u, "(A)")  "* Generate and write an event"
    write (u, "(A)")

    sample = "eio_ascii_4"

    allocate (eio_ascii_hepevt_t :: eio)
```

```
    select type (eio)
    class is (eio_ascii_t);  call eio%set_parameters ()
    end select
    call eio%init_out (sample, data)
    call event%generate (1, [0._default, 0._default])
    call event%evaluate_expressions ()

    call eio%output (event, i_prc = 1)
    call eio%write (u)
    call eio%final ()

    write (u, "(A)")
    write (u, "(A)")  "* File contents:"
    write (u, "(A)")

    u_file = free_unit ()
    open (u_file, file = char (sample // ".hepevt"), &
         action = "read", status = "old")
    do
       read (u_file, "(A)", iostat = iostat)  buffer
       if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
       if (iostat /= 0)  exit
       write (u, "(A)") trim (buffer)
    end do
    close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Reset data"
    write (u, "(A)")

    deallocate (eio)
    allocate (eio_ascii_hepevt_t :: eio)

    select type (eio)
    type is (eio_ascii_hepevt_t)
       call eio%set_parameters (keep_beams = .true.)
    end select
    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_ascii_4"

  end subroutine eio_ascii_4
```

We test the implementation of all I/O methods, method `lha` (old LHA):

⟨*EIO ascii: execute tests*⟩+≡

```
  call test (eio_ascii_5, "eio_ascii_5", &
```

```
        "read and write event contents, format [lha]", &
        u, results)
```

⟨*EIO ascii: test declarations*⟩+≡

```
    public :: eio_ascii_5
```

⟨*EIO ascii: tests*⟩+≡

```
    subroutine eio_ascii_5 (u)
      integer, intent(in) :: u
      class(generic_event_t), pointer :: event
      type(event_sample_data_t) :: data
      class(eio_t), allocatable :: eio
      type(string_t) :: sample
      integer :: u_file, iostat
      character(80) :: buffer

      write (u, "(A)")  "* Test output: eio_ascii_5"
      write (u, "(A)")  "*   Purpose: generate an event in ASCII LHA format"
      write (u, "(A)")  "*      and write weight to file"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize test process"

      call eio_prepare_test (event, unweighted = .false.)

      call data%init (1)
      data%n_evt = 1
      data%n_beam = 2
      data%pdg_beam = 25
      data%energy_beam = 500
      data%proc_num_id = [42]
      data%cross_section(1) = 100
      data%error(1) = 1
      data%total_cross_section = sum (data%cross_section)

      write (u, "(A)")
      write (u, "(A)")  "* Generate and write an event"
      write (u, "(A)")

      sample = "eio_ascii_5"

      allocate (eio_ascii_lha_t :: eio)

      select type (eio)
      class is (eio_ascii_t);  call eio%set_parameters ()
      end select
      call eio%init_out (sample, data)
      call event%generate (1, [0._default, 0._default])
      call event%evaluate_expressions ()

      call eio%output (event, i_prc = 1)
      call eio%write (u)
      call eio%final ()

      write (u, "(A)")
```

```
    write (u, "(A)")  "* File contents:"
    write (u, "(A)")

    u_file = free_unit ()
    open (u_file, file = char (sample // ".lha"), &
         action = "read", status = "old")
    do
       read (u_file, "(A)", iostat = iostat)  buffer
       if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
       if (iostat /= 0)  exit
       write (u, "(A)") trim (buffer)
    end do
    close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Reset data"
    write (u, "(A)")

    deallocate (eio)
    allocate (eio_ascii_lha_t :: eio)

    select type (eio)
    type is (eio_ascii_lha_t)
       call eio%set_parameters (keep_beams = .true.)
    end select
    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_ascii_5"

  end subroutine eio_ascii_5
```

We test the implementation of all I/O methods, method `long`:

⟨*EIO ascii: execute tests*⟩+≡

```
  call test (eio_ascii_6, "eio_ascii_6", &
       "read and write event contents, format [long]", &
       u, results)
```

⟨*EIO ascii: test declarations*⟩+≡

```
  public :: eio_ascii_6
```

⟨*EIO ascii: tests*⟩+≡

```
  subroutine eio_ascii_6 (u)
    integer, intent(in) :: u
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat
    character(80) :: buffer
```

```
write (u, "(A)")  "* Test output: eio_ascii_6"
write (u, "(A)")  "*   Purpose: generate an event in ASCII long format"
write (u, "(A)")  "*      and write weight to file"
write (u, "(A)")

write (u, "(A)")  "* Initialize test process"

call eio_prepare_test (event, unweighted = .false.)

call data%init (1)
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_ascii_6"

allocate (eio_ascii_long_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".long.evt"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Reset data"
    write (u, "(A)")

    deallocate (eio)
    allocate (eio_ascii_long_t :: eio)

    select type (eio)
    type is (eio_ascii_long_t)
       call eio%set_parameters (keep_beams = .true.)
    end select
    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_ascii_6"

  end subroutine eio_ascii_6
```

We test the implementation of all I/O methods, method `mokka`:

⟨*EIO ascii: execute tests*⟩+≡

```
  call test (eio_ascii_7, "eio_ascii_7", &
       "read and write event contents, format [mokka]", &
       u, results)
```

⟨*EIO ascii: test declarations*⟩+≡

```
  public :: eio_ascii_7
```

⟨*EIO ascii: tests*⟩+≡

```
  subroutine eio_ascii_7 (u)
    integer, intent(in) :: u
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat
    character(80) :: buffer

    write (u, "(A)")  "* Test output: eio_ascii_7"
    write (u, "(A)")  "*   Purpose: generate an event in ASCII mokka format"
    write (u, "(A)")  "*      and write weight to file"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize test process"

    call eio_prepare_test (event, unweighted = .false.)

    call data%init (1)
    data%n_evt = 1
    data%n_beam = 2
    data%pdg_beam = 25
```

```
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_ascii_7"

allocate (eio_ascii_mokka_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".mokka.evt"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Reset data"
write (u, "(A)")

deallocate (eio)
allocate (eio_ascii_mokka_t :: eio)

select type (eio)
type is (eio_ascii_mokka_t)
   call eio%set_parameters (keep_beams = .true.)
end select
call eio%write (u)

write (u, "(A)")
```

```
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_ascii_7"

  end subroutine eio_ascii_7
```

We test the implementation of all I/O methods, method `short`:

⟨*EIO ascii: execute tests*⟩+≡

```
  call test (eio_ascii_8, "eio_ascii_8", &
       "read and write event contents, format [short]", &
       u, results)
```

⟨*EIO ascii: test declarations*⟩+≡

```
  public :: eio_ascii_8
```

⟨*EIO ascii: tests*⟩+≡

```
  subroutine eio_ascii_8 (u)
    integer, intent(in) :: u
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat
    character(80) :: buffer

    write (u, "(A)")  "* Test output: eio_ascii_8"
    write (u, "(A)")  "*   Purpose: generate an event in ASCII short format"
    write (u, "(A)")  "*      and write weight to file"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize test process"

    call eio_prepare_test (event, unweighted = .false.)

    call data%init (1)
    data%n_evt = 1
    data%n_beam = 2
    data%pdg_beam = 25
    data%energy_beam = 500
    data%proc_num_id = [42]
    data%cross_section(1) = 100
    data%error(1) = 1
    data%total_cross_section = sum (data%cross_section)

    write (u, "(A)")
    write (u, "(A)")  "* Generate and write an event"
    write (u, "(A)")

    sample = "eio_ascii_8"

    allocate (eio_ascii_short_t :: eio)
```

```
    select type (eio)
    class is (eio_ascii_t);  call eio%set_parameters ()
    end select
    call eio%init_out (sample, data)
    call event%generate (1, [0._default, 0._default])
    call event%evaluate_expressions ()

    call eio%output (event, i_prc = 1)
    call eio%write (u)
    call eio%final ()

    write (u, "(A)")
    write (u, "(A)")  "* File contents:"
    write (u, "(A)")

    u_file = free_unit ()
    open (u_file, file = char (sample // ".short.evt"), &
         action = "read", status = "old")
    do
       read (u_file, "(A)", iostat = iostat)  buffer
       if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
       if (iostat /= 0)  exit
       write (u, "(A)") trim (buffer)
    end do
    close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Reset data"
    write (u, "(A)")

    deallocate (eio)
    allocate (eio_ascii_short_t :: eio)

    select type (eio)
    type is (eio_ascii_short_t)
       call eio%set_parameters (keep_beams = .true.)
    end select
    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_ascii_8"

  end subroutine eio_ascii_8
```

We test the implementation of all I/O methods, method `lha` (old LHA) in verbose version:

⟨*EIO ascii: execute tests*⟩+≡

```
  call test (eio_ascii_9, "eio_ascii_9", &
```

```
        "read and write event contents, format [lha_verb]", &
        u, results)
```

⟨*EIO ascii: test declarations*⟩+≡

```
    public :: eio_ascii_9
```

⟨*EIO ascii: tests*⟩+≡

```
    subroutine eio_ascii_9 (u)
      integer, intent(in) :: u
      class(generic_event_t), pointer :: event
      type(event_sample_data_t) :: data
      class(eio_t), allocatable :: eio
      type(string_t) :: sample
      integer :: u_file, iostat
      character(80) :: buffer

      write (u, "(A)")  "* Test output: eio_ascii_9"
      write (u, "(A)")  "*   Purpose: generate an event in ASCII LHA verbose format"
      write (u, "(A)")  "*      and write weight to file"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize test process"

      call eio_prepare_test (event, unweighted = .false.)

      call data%init (1)
      data%n_evt = 1
      data%n_beam = 2
      data%pdg_beam = 25
      data%energy_beam = 500
      data%proc_num_id = [42]
      data%cross_section(1) = 100
      data%error(1) = 1
      data%total_cross_section = sum (data%cross_section)

      write (u, "(A)")
      write (u, "(A)")  "* Generate and write an event"
      write (u, "(A)")

      sample = "eio_ascii_9"

      allocate (eio_ascii_lha_verb_t :: eio)

      select type (eio)
      class is (eio_ascii_t);  call eio%set_parameters ()
      end select
      call eio%init_out (sample, data)
      call event%generate (1, [0._default, 0._default])
      call event%evaluate_expressions ()

      call eio%output (event, i_prc = 1)
      call eio%write (u)
      call eio%final ()

      write (u, "(A)")
```

```
    write (u, "(A)")  "* File contents:"
    write (u, "(A)")

    u_file = free_unit ()
    open (u_file, file = char (sample // ".lha.verb"), &
         action = "read", status = "old")
    do
       read (u_file, "(A)", iostat = iostat)  buffer
       if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
       if (iostat /= 0)  exit
       write (u, "(A)") trim (buffer)
    end do
    close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Reset data"
    write (u, "(A)")

    deallocate (eio)
    allocate (eio_ascii_lha_verb_t :: eio)

    select type (eio)
    type is (eio_ascii_lha_verb_t)
       call eio%set_parameters (keep_beams = .true.)
    end select
    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_ascii_9"

  end subroutine eio_ascii_9
```

We test the implementation of all I/O methods, method `hepevt_verb`:

⟨*EIO ascii: execute tests*⟩+≡

```
  call test (eio_ascii_10, "eio_ascii_10", &
       "read and write event contents, format [hepevt_verb]", &
       u, results)
```

⟨*EIO ascii: test declarations*⟩+≡

```
  public :: eio_ascii_10
```

⟨*EIO ascii: tests*⟩+≡

```
  subroutine eio_ascii_10 (u)
    integer, intent(in) :: u
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat
    character(80) :: buffer
```

```
write (u, "(A)")  "* Test output: eio_ascii_10"
write (u, "(A)")  "*   Purpose: generate an event in ASCII hepevt verbose format"
write (u, "(A)")  "*      and write weight to file"
write (u, "(A)")

write (u, "(A)")  "* Initialize test process"

call eio_prepare_test (event, unweighted = .false.)

call data%init (1)
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_ascii_10"

allocate (eio_ascii_hepevt_verb_t :: eio)

select type (eio)
class is (eio_ascii_t);  call eio%set_parameters ()
end select
call eio%init_out (sample, data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents:"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".hepevt.verb"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do
close (u_file)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Reset data"
    write (u, "(A)")

    deallocate (eio)
    allocate (eio_ascii_hepevt_verb_t :: eio)

    select type (eio)
    type is (eio_ascii_hepevt_verb_t)
       call eio%set_parameters (keep_beams = .true.)
    end select
    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_ascii_10"

  end subroutine eio_ascii_10
```

17.9 HEP Common Blocks

Long ago, to transfer data between programs one had to set up a common block and link both programs as libraries to the main executable. The HEP community standardizes several of those common blocks.

The modern way of data exchange uses data files with standard formats. However, the LHEF standard data format derives from a common block (actually, two).

WHIZARD used to support those common blocks, and LHEF was implemented via writing/reading blocks. We still keep this convention, but intend to eliminate common blocks (or any other static storage) from the workflow in the future. This will gain flexibility towards concurrent running of program images.

We encapsulate everything here in a module. The module holds the variables which are part of the common block. To access the common block variables, we just have to `use` this module. (They are nevertheless in the common block, since external software may access it in this way.)

Note: This code is taken essentially unchanged from WHIZARD 2.1 and does not (yet) provide unit tests.

⟨hep_common.f90⟩≡

```
  ⟨File header⟩

  module hep_common

  ⟨Use kinds⟩
    use kinds, only: double
    use constants
  ⟨Use strings⟩
```

```
  use io_units
  use diagnostics
  use numeric_utils
  use physics_defs, only: HADRON_REMNANT
  use physics_defs, only: HADRON_REMNANT_SINGLET
  use physics_defs, only: HADRON_REMNANT_TRIPLET
  use physics_defs, only: HADRON_REMNANT_OCTET
  use xml
  use lorentz
  use flavors
  use colors
  use polarizations
  use model_data
  use particles
  use subevents, only: PRT_BEAM, PRT_INCOMING, PRT_OUTGOING
  use subevents, only: PRT_UNDEFINED
  use subevents, only: PRT_VIRTUAL, PRT_RESONANT, PRT_BEAM_REMNANT
```

⟨*Standard module head*⟩

⟨*HEP common: public*⟩

⟨*HEP common: interfaces*⟩

⟨*HEP common: parameters*⟩

⟨*HEP common: variables*⟩

⟨*HEP common: common blocks*⟩

```
contains
```

⟨*HEP common: procedures*⟩

```
end module hep_common
```

17.9.1 Event characteristics

The maximal number of particles in an event record.

⟨*HEP common: parameters*⟩≡

```
integer, parameter, public :: MAXNUP = 500
```

The number of particles in this event.

⟨*HEP common: variables*⟩≡

```
integer, public :: NUP
```

The process ID for this event.

⟨*HEP common: variables*⟩+≡

```
integer, public :: IDPRUP
```

The weight of this event (±1 for unweighted events).

⟨*HEP common: variables*⟩+≡

```
double precision, public :: XWGTUP
```

The factorization scale that is used for PDF calculation (-1 if undefined).

⟨*HEP common: variables*⟩+≡
```
double precision, public :: SCALUP
```

The QED and QCD couplings α used for this event (-1 if undefined).

⟨*HEP common: variables*⟩+≡
```
double precision, public :: AQEDUP
double precision, public :: AQCDUP
```

17.9.2 Particle characteristics

The PDG code:

⟨*HEP common: variables*⟩+≡
```
integer, dimension(MAXNUP), public :: IDUP
```

The status code. Incoming: -1, outgoing: $+1$. Intermediate t-channel propagator: -2 (currently not used by WHIZARD). Intermediate resonance whose mass should be preserved: 2. Intermediate resonance for documentation: 3 (currently not used). Beam particles: -9.

⟨*HEP common: variables*⟩+≡
```
integer, dimension(MAXNUP), public :: ISTUP
```

Index of first and last mother.

⟨*HEP common: variables*⟩+≡
```
integer, dimension(2,MAXNUP), public :: MOTHUP
```

Color line index of the color and anticolor entry for the particle. The standard recommends using large numbers; we start from MAXNUP+1.

⟨*HEP common: variables*⟩+≡
```
integer, dimension(2,MAXNUP), public :: ICOLUP
```

Momentum, energy, and invariant mass: (p_x, p_y, p_z, E, M). For space-like particles, M is the negative square root of the absolute value of the invariant mass.

⟨*HEP common: variables*⟩+≡
```
double precision, dimension(5,MAXNUP), public :: PUP
```

Invariant lifetime (distance) from production to decay in mm.

⟨*HEP common: variables*⟩+≡
```
double precision, dimension(MAXNUP), public :: VTIMUP
```

Cosine of the angle between the spin-vector and a particle and the 3-momentum of its mother, given in the lab frame. If undefined/unpolarized: 9.

⟨*HEP common: variables*⟩+≡
```
double precision, dimension(MAXNUP), public :: SPINUP
```

17.9.3 The HEPRUP common block

This common block is filled once per run.

Run characteristics

The maximal number of different processes.

⟨*HEP common: parameters*⟩+≡
```
integer, parameter, public :: MAXPUP = 100
```

The beam PDG codes.

⟨*HEP common: variables*⟩+≡
```
integer, dimension(2), public :: IDBMUP
```

The beam energies in GeV.

⟨*HEP common: variables*⟩+≡
```
double precision, dimension(2), public :: EBMUP
```

The PDF group and set for the two beams. (Undefined: use -1; LHAPDF: use group $= 0$).

⟨*HEP common: variables*⟩+≡
```
integer, dimension(2), public :: PDFGUP
integer, dimension(2), public :: PDFSUP
```

The (re)weighting model. 1: events are weighted, the shower generator (SHG) selects processes according to the maximum weight (in pb) and unweights events. 2: events are weighted, the SHG selects processes according to their cross section (in pb) and unweights events. 3: events are unweighted and simply run through the SHG. 4: events are weighted, and the SHG keeps the weight. Negative numbers: negative weights are allowed (and are reweighted to ± 1 by the SHG, if allowed).

`WHIZARD` only supports modes 3 and 4, as the SHG is not given control over process selection. This is consistent with writing events to file, for offline showering.

⟨*HEP common: variables*⟩+≡
```
integer, public :: IDWTUP
```

The number of different processes.

⟨*HEP common: variables*⟩+≡
```
integer, public :: NPRUP
```

Process characteristics

Cross section and error in pb. (Cross section is needed only for `IDWTUP` $= 2$, so here both values are given for informational purposes only.)

⟨*HEP common: variables*⟩+≡
```
double precision, dimension(MAXPUP), public :: XSECUP
double precision, dimension(MAXPUP), public :: XERRUP
```

Maximum weight, i.e., the maximum value that `XWGTUP` can take. Also unused for the supported weighting models. It is ± 1 for unweighted events.

⟨*HEP common: variables*⟩+≡
```
double precision, dimension(MAXPUP), public :: XMAXUP
```

Internal ID of the selected process, matches `IDPRUP` below.

⟨*HEP common: variables*⟩+≡
```
integer, dimension(MAXPUP), public :: LPRUP
```

The common block

⟨*HEP common: common blocks*⟩≡
```
common /HEPRUP/ &
     IDBMUP, EBMUP, PDFGUP, PDFSUP, IDWTUP, NPRUP, &
     XSECUP, XERRUP, XMAXUP, LPRUP
save /HEPRUP/
```

Fill the run characteristics of the common block. The initialization sets the beam properties, number of processes, and weighting model.

⟨*HEP common: public*⟩≡
```
public :: heprup_init
```

⟨*HEP common: procedures*⟩≡
```
subroutine heprup_init &
     (beam_pdg, beam_energy, n_processes, unweighted, negative_weights)
  integer, dimension(2), intent(in) :: beam_pdg
  real(default), dimension(2), intent(in) :: beam_energy
  integer, intent(in) :: n_processes
  logical, intent(in) :: unweighted
  logical, intent(in) :: negative_weights
  IDBMUP = beam_pdg
  EBMUP = beam_energy
  PDFGUP = -1
  PDFSUP = -1
  if (unweighted) then
     IDWTUP = 3
  else
     IDWTUP = 4
  end if
  if (negative_weights)  IDWTUP = - IDWTUP
  NPRUP = n_processes
end subroutine heprup_init
```

The HEPRUP (event) common block is needed for the interface to the shower. Filling of it is triggered by some output file formats. If these are not present, the common block is filled with some dummy information. Be generous with the number of processes in HEPRUP so that PYTHIA only rarely needs to be reinitialized in case events with higher process ids are generated.

⟨*HEP common: public*⟩+≡
```
public :: assure_heprup
```

⟨*HEP common: procedures*⟩+≡
```
subroutine assure_heprup (pset)
  type(particle_set_t), intent(in) :: pset
  integer :: i, num_id
  integer, parameter :: min_processes = 10
  num_id = 1
  if (LPRUP (num_id) /= 0)  return
  call heprup_init ( &
       [pset%prt(1)%get_pdg (), pset%prt(2)%get_pdg ()] , &
       [pset%prt(1)%p%p(0), pset%prt(2)%p%p(0)], &
       num_id, .false., .false.)
  do i = 1, (num_id / min_processes + 1) * min_processes
     call heprup_set_process_parameters (i = i, process_id = &
          i, cross_section = 1._default, error = 1._default)
  end do
end subroutine assure_heprup
```

Read in the LHE file opened in unit `u` and add the final particles to the `particle_set`, the outgoing particles of the existing `particle_set` are compared to the particles that are read in. When they are equal in flavor and

momenta, they are erased and their mother-daughter relations are transferred to the existing particles.

⟨*HEP common: public*⟩+≡

```
public :: combine_lhef_with_particle_set
```

⟨*HEP common: procedures*⟩+≡

```
subroutine combine_lhef_with_particle_set &
     (particle_set, u, model_in, model_hadrons)
  type(particle_set_t), intent(inout) :: particle_set
  integer, intent(in) :: u
  class(model_data_t), intent(in), target :: model_in
  class(model_data_t), intent(in), target :: model_hadrons
  type(flavor_t) :: flv
  type(color_t) :: col
  class(model_data_t), pointer :: model
  type(particle_t), dimension(:), allocatable :: prt_tmp, prt
  integer :: i, j
  type(vector4_t) :: mom, d_mom
  integer, PARAMETER :: MAXLEN=200
  character(len=maxlen) :: string
  integer :: ibeg, n_tot, n_entries
  integer, dimension(:), allocatable :: relations, mothers, tbd
  INTEGER :: NUP,IDPRUP,IDUP,ISTUP
  real(kind=double) :: XWGTUP,SCALUP,AQEDUP,AQCDUP,VTIMUP,SPINUP
  integer :: MOTHUP(1:2), ICOLUP(1:2)
  real(kind=double) :: PUP(1:5)
  real(kind=default) :: pup_dum(1:5)
  character(len=5) :: buffer
  character(len=6) :: strfmt
  logical :: not_found
  logical :: debug_lhef = .false.
  STRFMT='(A000)'
  WRITE (STRFMT(3:5),'(I3)') MAXLEN

  if (debug_lhef)  call particle_set%write ()

  rewind (u)

  do
     read (u,*, END=501, ERR=502) STRING
     IBEG = 0
     do
        if (signal_is_pending ()) return
        IBEG = IBEG + 1
        ! Allow indentation.
        IF (STRING (IBEG:IBEG) .EQ. ' ' .and. IBEG < MAXLEN-6) cycle
        exit
     end do
     IF (string(IBEG:IBEG+6) /= '<event>' .and. &
          string(IBEG:IBEG+6) /= '<event ') cycle
     exit
  end do
  !!! Read first line of event info -> number of entries
  read (u, *, END=503, ERR=504) NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP
```

```
n_tot = particle_set%get_n_tot ()
allocate (prt_tmp (1:n_tot+NUP))
allocate (relations (1:NUP), mothers (1:NUP), tbd(1:NUP))
do i = 1, n_tot
   if (signal_is_pending ()) return
   prt_tmp (i) = particle_set%get_particle (i)
end do
!!! transfer particles from lhef to particle_set
!!!...Read NUP subsequent lines with information on each particle.
n_entries = 1
mothers = 0
relations = 0
PARTICLE_LOOP: do I = 1, NUP
   read (u,*, END=200, ERR=505) IDUP, ISTUP, MOTHUP(1), MOTHUP(2), &
        ICOLUP(1), ICOLUP(2), (PUP (J),J=1,5), VTIMUP, SPINUP
   if (model_in%test_field (IDUP)) then
      model => model_in
   else if (model_hadrons%test_field (IDUP)) then
      model => model_hadrons
   else
      write (buffer, "(I5)") IDUP
      call msg_error ("Parton " // buffer // &
           " found neither in given model file nor in SM_hadrons")
      return
   end if
   if (debug_lhef) then
      print *, "IDUP, ISTUP, MOTHUP, PUP = ", IDUP, ISTUP, MOTHUP(1), &
         MOTHUP(2), PUP
   end if
   call flv%init (IDUP, model)
   if (IABS(IDUP) == 2212 .or. IABS(IDUP) == 2112) then
      ! PYTHIA sometimes sets color indices for protons and neutrons (?)
      ICOLUP (1) = 0
      ICOLUP (2) = 0
   end if
   call col%init_col_acl (ICOLUP (1), ICOLUP (2))
   !!! Settings for unpolarized particles
   ! particle_set%prt (oldsize+i)%hel = ??
   ! particle_set%prt (oldsize+i)%pol = ??
   if (MOTHUP(1) /= 0) then
      mothers(i) = MOTHUP(1)
   end if
   pup_dum = PUP
   if (pup_dum(4) < 1E-10_default)  cycle
   mom = vector4_moving (pup_dum (4), &
        vector3_moving ([pup_dum (1), pup_dum (2), pup_dum (3)]))
   not_found = .true.
   SCAN_PARTICLES: do j = 1, n_tot
      d_mom = prt_tmp(j)%get_momentum ()
      if (all (nearly_equal &
           (mom%p, d_mom%p, abs_smallness = 1.E-4_default)) .and. &
            (prt_tmp(j)%get_pdg () == IDUP)) then
         if (.not. prt_tmp(j)%get_status () == PRT_BEAM .or. &
              .not. prt_tmp(j)%get_status () == PRT_BEAM_REMNANT) &
```

```
                relations(i) = j
                not_found = .false.
         end if
      end do SCAN_PARTICLES
      if (not_found) then
         if (debug_lhef) &
            print *, "Not found: adding particle"
         call prt_tmp(n_tot+n_entries)%set_flavor (flv)
         call prt_tmp(n_tot+n_entries)%set_color (col)
         call prt_tmp(n_tot+n_entries)%set_momentum (mom)
         if (MOTHUP(1) /= 0) then
            if (relations(MOTHUP(1)) /= 0) then
               call prt_tmp(n_tot+n_entries)%set_parents &
                    ([relations(MOTHUP(1))])
               call prt_tmp(relations(MOTHUP(1)))%add_child (n_tot+n_entries)
               if (prt_tmp(relations(MOTHUP(1)))%get_status () &
                    == PRT_OUTGOING) &
                    call prt_tmp(relations(MOTHUP(1)))%reset_status &
                    (PRT_VIRTUAL)
            end if
         end if
         call prt_tmp(n_tot+n_entries)%set_status (PRT_OUTGOING)
         if (debug_lhef) call prt_tmp(n_tot+n_entries)%write ()
         n_entries = n_entries + 1
      end if
   end do PARTICLE_LOOP
   do i = 1, n_tot
      if (prt_tmp(i)%get_status () == PRT_OUTGOING .and. &
          prt_tmp(i)%get_n_children () /= 0) then
               call prt_tmp(i)%reset_status (PRT_VIRTUAL)
      end if
   end do

   allocate (prt (1:n_tot+n_entries-1))
   prt = prt_tmp (1:n_tot+n_entries-1)
   ! transfer to particle_set
   call particle_set%replace (prt)
   deallocate (prt, prt_tmp)

   if (debug_lhef) then
      call particle_set%write ()
      print *, "combine_lhef_with_particle_set"
      ! stop
   end if

200 continue
   return

501 write(*,*) "READING LHEF failed 501"
   return
502 write(*,*) "READING LHEF failed 502"
   return
503 write(*,*) "READING LHEF failed 503"
   return
```

```
  504 write(*,*) "READING LHEF failed 504"
      return
  505 write(*,*) "READING LHEF failed 505"
      return
    end subroutine combine_lhef_with_particle_set
```

⟨*HEP common: public*⟩+≡

```
    public :: w2p_write_lhef_event
```

⟨*HEP common: procedures*⟩+≡

```
    subroutine w2p_write_lhef_event (unit)
      integer, intent(in) :: unit
      type(xml_tag_t), allocatable :: tag_lhef, tag_head, tag_init, &
           tag_event, tag_gen_n, tag_gen_v
      call msg_debug (D_EVENTS, "w2p_write_lhef_event")
      allocate (tag_lhef, tag_head, tag_init, tag_event, &
           tag_gen_n, tag_gen_v)
      call tag_lhef%init (var_str ("LesHouchesEvents"), &
           [xml_attribute (var_str ("version"), var_str ("1.0"))], .true.)
      call tag_head%init (var_str ("header"), .true.)
      call tag_init%init (var_str ("init"), .true.)
      call tag_event%init (var_str ("event"), .true.)
      call tag_gen_n%init (var_str ("generator_name"), .true.)
      call tag_gen_v%init (var_str ("generator_version"), .true.)
      call tag_lhef%write (unit); write (unit, *)
      call tag_head%write (unit); write (unit, *)
      write (unit, "(2x)", advance = "no")
      call tag_gen_n%write (var_str ("WHIZARD"), unit)
      write (unit, *)
      write (unit, "(2x)", advance = "no")
      call tag_gen_v%write (var_str ("⟨Version⟩"), unit)
      write (unit, *)
      call tag_head%close (unit); write (unit, *)
      call tag_init%write (unit); write (unit, *)
      call heprup_write_lhef (unit)
      call tag_init%close (unit); write (unit, *)
      call tag_event%write (unit); write (unit, *)
      call hepeup_write_lhef (unit)
      call tag_event%close (unit); write (unit, *)
      call tag_lhef%close (unit); write (unit, *)
      deallocate (tag_lhef, tag_head, tag_init, tag_event, &
           tag_gen_n, tag_gen_v)
    end subroutine w2p_write_lhef_event
```

Extract parameters from the common block. We leave it to the caller to specify which parameters it actually needs.

`PDFGUP` and `PDFSUP` are not extracted. `IDWTUP=1,2` are not supported by `WHIZARD`, but correspond to weighted events.

⟨*HEP common: public*⟩+≡

```
    public :: heprup_get_run_parameters
```

⟨*HEP common: procedures*⟩+≡

```
    subroutine heprup_get_run_parameters &
         (beam_pdg, beam_energy, n_processes, unweighted, negative_weights)
```

```
    integer, dimension(2), intent(out), optional :: beam_pdg
    real(default), dimension(2), intent(out), optional :: beam_energy
    integer, intent(out), optional :: n_processes
    logical, intent(out), optional :: unweighted
    logical, intent(out), optional :: negative_weights
    if (present (beam_pdg))  beam_pdg = IDBMUP
    if (present (beam_energy))  beam_energy = EBMUP
    if (present (n_processes))  n_processes = NPRUP
    if (present (unweighted)) then
       select case (abs (IDWTUP))
       case (3)
          unweighted = .true.
       case (4)
          unweighted = .false.
       case (1,2)  !!! not supported by WHIZARD
          unweighted = .false.
       case default
          call msg_fatal ("HEPRUP: unsupported IDWTUP value")
       end select
    end if
    if (present (negative_weights)) then
       negative_weights = IDWTUP < 0
    end if
  end subroutine heprup_get_run_parameters
```

Specify PDF set info. Since we support only LHAPDF, the group entry is zero.

⟨*HEP common: public*⟩+≡

```
  public :: heprup_set_lhapdf_id
```

⟨*HEP common: procedures*⟩+≡

```
  subroutine heprup_set_lhapdf_id (i_beam, pdf_id)
    integer, intent(in) :: i_beam, pdf_id
    PDFGUP(i_beam) = 0
    PDFSUP(i_beam) = pdf_id
  end subroutine heprup_set_lhapdf_id
```

Fill the characteristics for a particular process. Only the process ID is mandatory. Note that WHIZARD computes cross sections in fb, so we have to rescale to pb. The maximum weight is meaningless for unweighted events.

⟨*HEP common: public*⟩+≡

```
  public :: heprup_set_process_parameters
```

⟨*HEP common: procedures*⟩+≡

```
  subroutine heprup_set_process_parameters &
       (i, process_id, cross_section, error, max_weight)
    integer, intent(in) :: i, process_id
    real(default), intent(in), optional :: cross_section, error, max_weight
    real(default), parameter :: pb_per_fb = 1.e-3_default
    LPRUP(i) = process_id
    if (present (cross_section)) then
       XSECUP(i) = cross_section * pb_per_fb
    else
       XSECUP(i) = 0
    end if
```

```
    if (present (error)) then
       XERRUP(i) = error * pb_per_fb
    else
       XERRUP(i) = 0
    end if
    select case (IDWTUP)
    case (3);  XMAXUP(i) = 1
    case (4)
       if (present (max_weight)) then
          XMAXUP(i) = max_weight * pb_per_fb
       else
          XMAXUP(i) = 0
       end if
    end select
  end subroutine heprup_set_process_parameters
```

Extract the process parameters, as far as needed.

⟨*HEP common: public*⟩+≡

```
  public :: heprup_get_process_parameters
```

⟨*HEP common: procedures*⟩+≡

```
  subroutine heprup_get_process_parameters  &
       (i, process_id, cross_section, error, max_weight)
    integer, intent(in) :: i
    integer, intent(out), optional :: process_id
    real(default), intent(out), optional :: cross_section, error, max_weight
    real(default), parameter :: pb_per_fb = 1.e-3_default
    if (present (process_id))  process_id = LPRUP(i)
    if (present (cross_section)) then
       cross_section = XSECUP(i) / pb_per_fb
    end if
    if (present (error)) then
       error = XERRUP(i) / pb_per_fb
    end if
    if (present (max_weight)) then
       select case (IDWTUP)
       case (3)
          max_weight = 1
       case (4)
          max_weight = XMAXUP(i) / pb_per_fb
       case (1,2)   !!! not supported by WHIZARD
          max_weight = 0
       case default
          call msg_fatal ("HEPRUP: unsupported IDWTUP value")
       end select
    end if
  end subroutine heprup_get_process_parameters
```

17.9.4 Run parameter output (verbose)

This is a verbose output of the HEPRUP block.

⟨*HEP common: public*⟩+≡

```
  public :: heprup_write_verbose
```

⟨*HEP common: procedures*⟩+≡

```
subroutine heprup_write_verbose (unit)
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(A)")  "HEPRUP Common Block"
  write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "IDBMUP", IDBMUP, &
       "PDG code of beams"
  write (u, "(3x,A6,' = ',G12.5,1x,G12.5,8x,A)")  "EBMUP ", EBMUP, &
       "Energy of beams in GeV"
  write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "PDFGUP", PDFGUP, &
       "PDF author group [-1 = undefined]"
  write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "PDFSUP", PDFSUP, &
       "PDF set ID       [-1 = undefined]"
  write (u, "(3x,A6,' = ',I9,3x,1x,9x,3x,8x,A)")  "IDWTUP", IDWTUP, &
       "LHA code for event weight mode"
  write (u, "(3x,A6,' = ',I9,3x,1x,9x,3x,8x,A)")  "NPRUP ", NPRUP, &
       "Number of user subprocesses"
  do i = 1, NPRUP
     write (u, "(1x,A,I0)")  "Subprocess #", i
     write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "XSECUP", XSECUP(i), &
          "Cross section in pb"
     write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "XERRUP", XERRUP(i), &
          "Cross section error in pb"
     write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "XMAXUP", XMAXUP(i), &
          "Maximum event weight (cf. IDWTUP)"
     write (u, "(3x,A6,' = ',I9,3x,1x,12x,8x,A)")  "LPRUP ", LPRUP(i), &
          "Subprocess ID"
  end do
end subroutine heprup_write_verbose
```

17.9.5 Run parameter output (other formats)

This routine writes the initialization block according to the LHEF standard. It uses the current contents of the HEPRUP block.

⟨*HEP common: public*⟩+≡

```
public :: heprup_write_lhef
```

⟨*HEP common: procedures*⟩+≡

```
subroutine heprup_write_lhef (unit)
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(2(1x,I0),2(1x,ES17.10),6(1x,I0))") &
       IDBMUP, EBMUP, PDFGUP, PDFSUP, IDWTUP, NPRUP
  do i = 1, NPRUP
     write (u, "(3(1x,ES17.10),1x,I0)") &
          XSECUP(i), XERRUP(i), XMAXUP(i), LPRUP(i)
  end do
end subroutine heprup_write_lhef
```

This routine is a complete dummy at the moment. It uses the current contents of the HEPRUP block. At the end, it should depend on certain input flags for the different ASCII event formats.

⟨*HEP common: public*⟩+≡
```
public :: heprup_write_ascii
```

⟨*HEP common: procedures*⟩+≡
```
subroutine heprup_write_ascii (unit)
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(2(1x,I0),2(1x,ES17.10),6(1x,I0))") &
       IDBMUP, EBMUP, PDFGUP, PDFSUP, IDWTUP, NPRUP
  do i = 1, NPRUP
     write (u, "(3(1x,ES17.10),1x,I0)") &
          XSECUP(i), XERRUP(i), XMAXUP(i), LPRUP(i)
  end do
end subroutine heprup_write_ascii
```

Run parameter input (LHEF)

In a LHEF file, the parameters are written in correct order on separate lines, but we should not count on the precise format. List-directed input should just work.

⟨*HEP common: public*⟩+≡
```
public :: heprup_read_lhef
```

⟨*HEP common: procedures*⟩+≡
```
subroutine heprup_read_lhef (u)
  integer, intent(in) :: u
  integer :: i
  read (u, *) &
       IDBMUP, EBMUP, PDFGUP, PDFSUP, IDWTUP, NPRUP
  do i = 1, NPRUP
     read (u, *) &
          XSECUP(i), XERRUP(i), XMAXUP(i), LPRUP(i)
  end do
end subroutine heprup_read_lhef
```

17.9.6 The HEPEUP common block

⟨*HEP common: common blocks*⟩+≡
```
common /HEPEUP/ &
     NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP, &
     IDUP, ISTUP, MOTHUP, ICOLUP, PUP, VTIMUP, SPINUP
save /HEPEUP/
```

Initialization

Fill the event characteristics of the common block. The initialization sets only the number of particles and initializes the rest with default values. The other routine sets the optional parameters.

⟨*HEP common: public*⟩+≡
```
public :: hepeup_init
public :: hepeup_set_event_parameters
```

⟨*HEP common: procedures*⟩+≡
```
subroutine hepeup_init (n_tot)
  integer, intent(in) :: n_tot
  NUP = n_tot
  IDPRUP = 0
  XWGTUP = 1
  SCALUP = -1
  AQEDUP = -1
  AQCDUP = -1
end subroutine hepeup_init

subroutine hepeup_set_event_parameters &
     (proc_id, weight, scale, alpha_qed, alpha_qcd)
  integer, intent(in), optional :: proc_id
  real(default), intent(in), optional :: weight, scale, alpha_qed, alpha_qcd
  if (present (proc_id))   IDPRUP = proc_id
  if (present (weight))    XWGTUP = weight
  if (present (scale))     SCALUP = scale
  if (present (alpha_qed)) AQEDUP = alpha_qed
  if (present (alpha_qcd)) AQCDUP = alpha_qcd
end subroutine hepeup_set_event_parameters
```

Extract event information. The caller determines the parameters.

⟨*HEP common: public*⟩+≡
```
public :: hepeup_get_event_parameters
```

⟨*HEP common: procedures*⟩+≡
```
subroutine hepeup_get_event_parameters &
     (proc_id, weight, scale, alpha_qed, alpha_qcd)
  integer, intent(out), optional :: proc_id
  real(default), intent(out), optional :: weight, scale, alpha_qed, alpha_qcd
  if (present (proc_id))   proc_id   = IDPRUP
  if (present (weight))    weight    = XWGTUP
  if (present (scale))     scale     = SCALUP
  if (present (alpha_qed)) alpha_qed = AQEDUP
  if (present (alpha_qcd)) alpha_qcd = AQCDUP
end subroutine hepeup_get_event_parameters
```

Particle data

Below we need the particle status codes which are actually defined in the `subevents` module.

Set the entry for a specific particle. All parameters are set with the exception of lifetime and spin, where default values are stored.

⟨*HEP common: public*⟩+≡

```
public :: hepeup_set_particle
```

⟨*HEP common: procedures*⟩+≡

```
subroutine hepeup_set_particle (i, pdg, status, parent, col, p, m2)
  integer, intent(in) :: i
  integer, intent(in) :: pdg, status
  integer, dimension(:), intent(in) :: parent
  type(vector4_t), intent(in) :: p
  integer, dimension(2), intent(in) :: col
  real(default), intent(in) :: m2
  if (i > MAXNUP) then
     call msg_error (arr=[ &
          var_str ("Too many particles in HEPEUP common block. " // &
                           "If this happened "), &
          var_str ("during event output, your events will be " // &
                           "invalid; please consider "), &
          var_str ("switching to a modern event format like HEPMC. " // &
                           "If you are not "), &
          var_str ("using an old, HEPEUP based format and " // &
                           "nevertheless get this error,"), &
          var_str ("please notify the WHIZARD developers,") ])
     return
  end if
  IDUP(i) = pdg
  select case (status)
  case (PRT_BEAM);         ISTUP(i) = -9
  case (PRT_INCOMING);     ISTUP(i) = -1
  case (PRT_BEAM_REMNANT); ISTUP(i) =  3
  case (PRT_OUTGOING);     ISTUP(i) =  1
  case (PRT_RESONANT);     ISTUP(i) =  2
  case (PRT_VIRTUAL);      ISTUP(i) =  3
  case default;            ISTUP(i) =  0
  end select
  select case (size (parent))
  case (0);      MOTHUP(:,i) = 0
  case (1);      MOTHUP(1,i) = parent(1); MOTHUP(2,i) = 0
  case default;  MOTHUP(:,i) = [ parent(1), parent(size (parent)) ]
  end select
  if (col(1) > 0) then
     ICOLUP(1,i) = 500 + col(1)
  else
     ICOLUP(1,i) = 0
  end if
  if (col(2) > 0) then
     ICOLUP(2,i) = 500 + col(2)
  else
     ICOLUP(2,i) = 0
  end if
  PUP(1:3,i) = vector3_get_components (space_part (p))
  PUP(4,i) = energy (p)
  PUP(5,i) = sign (sqrt (abs (m2)), m2)
```

```
    VTIMUP(i) = 0
    SPINUP(i) = 9
  end subroutine hepeup_set_particle
```

Set the lifetime, actually $c\tau$ measured im mm, where τ is the invariant lifetime.

⟨*HEP common: public*⟩+≡

```
  public :: hepeup_set_particle_lifetime
```

⟨*HEP common: procedures*⟩+≡

```
  subroutine hepeup_set_particle_lifetime (i, lifetime)
    integer, intent(in) :: i
    real(default), intent(in) :: lifetime
    VTIMUP(i) = lifetime
  end subroutine hepeup_set_particle_lifetime
```

Set the particle spin entry. We need the cosine of the angle of the spin axis with respect to the three-momentum of the parent particle.

If the particle has a full polarization density matrix given, we need the particle momentum and polarization as well as the mother-particle momentum. The polarization is transformed into a spin vector (which is sensible only for spin-1/2 or massless particles), which then is transformed into the lab frame (by a rotation of the 3-axis to the particle momentum axis). Finally, we compute the scalar product of this vector with the mother-particle three-momentum.

This puts severe restrictions on the applicability of this definition, and Lorentz invariance is lost. Unfortunately, the Les Houches Accord requires this computation.

⟨*HEP common: public*⟩+≡

```
  public :: hepeup_set_particle_spin
```

⟨*HEP common: interfaces*⟩≡

```
  interface hepeup_set_particle_spin
     module procedure hepeup_set_particle_spin_pol
  end interface
```

⟨*HEP common: procedures*⟩+≡

```
  subroutine hepeup_set_particle_spin_pol (i, p, pol, p_mother)
    integer, intent(in) :: i
    type(vector4_t), intent(in) :: p
    type(polarization_t), intent(in) :: pol
    type(vector4_t), intent(in) :: p_mother
    type(vector3_t) :: s3, p3
    type(vector4_t) :: s4
    s3 = vector3_moving (pol%get_axis ())
    p3 = space_part (p)
    s4 = rotation_to_2nd (3, p3) * vector4_moving (0._default, s3)
    SPINUP(i) = enclosed_angle_ct (s4, p_mother)
  end subroutine hepeup_set_particle_spin_pol
```

Extract particle data. The caller decides which ones to retrieve.

Status codes: beam remnants share the status code with virtual particles. However, for the purpose of WHIZARD we should identify them. We use the PDG code for this.

⟨*HEP common: public*⟩+≡

```
public :: hepeup_get_particle
```

⟨*HEP common: procedures*⟩+≡

```
subroutine hepeup_get_particle (i, pdg, status, parent, col, p, m2)
  integer, intent(in) :: i
  integer, intent(out), optional :: pdg, status
  integer, dimension(:), intent(out), optional :: parent
  type(vector4_t), intent(out), optional :: p
  integer, dimension(2), intent(out), optional :: col
  real(default), dimension(5,MAXNUP) :: pup_def
  real(default), intent(out), optional :: m2
  if (present (pdg))  pdg = IDUP(i)
  if (present (status)) then
     select case (ISTUP(i))
     case (-9);  status = PRT_BEAM
     case (-1);  status = PRT_INCOMING
     case (1);   status = PRT_OUTGOING
     case (2);   status = PRT_RESONANT
     case (3);
        select case (abs (IDUP(i)))
        case (HADRON_REMNANT, HADRON_REMNANT_SINGLET, &
             HADRON_REMNANT_TRIPLET, HADRON_REMNANT_OCTET)
           status = PRT_BEAM_REMNANT
        case default
           status = PRT_VIRTUAL
        end select
     case default
        status = PRT_UNDEFINED
     end select
  end if
  if (present (parent)) then
     select case (size (parent))
     case (0)
     case (1);    parent(1) = MOTHUP(1,i)
     case (2);    parent = MOTHUP(:,i)
     end select
  end if
  if (present (col)) then
     col = ICOLUP(:,i)
  end if
  if (present (p)) then
     pup_def = PUP
     p = vector4_moving (pup_def(4,i), vector3_moving (pup_def(1:3,i)))
  end if
  if (present (m2)) then
     m2 = sign (PUP(5,i) ** 2, PUP(5,i))
  end if
end subroutine hepeup_get_particle
```

17.9.7 The HEPEVT and HEPEV4 common block

For the LEP Monte Carlos, a standard common block has been proposed in AKV89. We strongly recommend its use. (The description is an abbreviated

transcription of AKV89, Vol. 3, pp. 327-330).

NMXHEP is the maximum number of entries:

⟨*HEP common: variables*⟩+≡
```
integer, parameter :: NMXHEP = 4000
```

NEVHEP is normally the event number, but may take special values as follows:

0 the program does not keep track of event numbers. -1 a special initialization record. -2 a special final record.

⟨*HEP common: variables*⟩+≡
```
integer :: NEVHEP
```

NHEP holds the number of entries for this event.

⟨*HEP common: variables*⟩+≡
```
integer, public :: NHEP
```

The entry ISTHEP(N) gives the status code for the Nth entry, with the following semantics: 0 a null entry. 1 an existing entry, which has not decayed or fragmented. 2 a decayed or fragmented entry, which is retained for event history information. 3 documentation line. 4- 10 reserved for future standards. 11-200 at the disposal of each model builder. 201- at the disposal of users.

⟨*HEP common: variables*⟩+≡
```
integer, dimension(NMXHEP), public :: ISTHEP
```

The Particle Data Group has proposed standard particle codes, which are to be stored in IDHEP(N).

⟨*HEP common: variables*⟩+≡
```
integer, dimension(NMXHEP), public :: IDHEP
```

JMOHEP(1,N) points to the mother of the Nth entry, if any. It is set to zero for initial entries. JMOHEP(2,N) points to the second mother, if any.

⟨*HEP common: variables*⟩+≡
```
integer, dimension(2, NMXHEP), public :: JMOHEP
```

JDAHEP(1,N) and JDAHEP(2,N) point to the first and last daughter of the Nth entry, if any. These are zero for entries which have not yet decayed. The other daughters are stored in between these two.

⟨*HEP common: variables*⟩+≡
```
integer, dimension(2, NMXHEP), public :: JDAHEP
```

In PHEP we store the momentum of the particle, more specifically this means that PHEP(1,N), PHEP(2,N), and PHEP(3,N) contain the momentum in the x, y, and z direction (as defined by the machine people), measured in GeV/c. PHEP(4,N) contains the energy in GeV and PHEP(5,N) the mass in GeV/c^2. The latter may be negative for spacelike partons.

⟨*HEP common: variables*⟩+≡
```
double precision, dimension(5, NMXHEP), public :: PHEP
```

Finally VHEP is the place to store the position of the production vertex. VHEP(1,N), VHEP(2,N), and VHEP(3,N) contain the x, y, and z coordinate (as defined by the machine people), measured in mm. VHEP(4,N) contains the production time in mm/c.

⟨*HEP common: variables*⟩+≡
```
double precision, dimension(4, NMXHEP) :: VHEP
```

As an amendment to the proposed standard common block HEPEVT, we also have a polarisation common block HEPSPN, as described in AKV89. SHEP(1,N), SHEP(2,N), and SHEP(3,N) give the x, y, and z component of the spinvector s of a fermion in the fermions restframe.

Furthermore, we add the polarization of the corresponding outgoing particles:

⟨*HEP common: variables*⟩+≡
```
integer, dimension(NMXHEP) :: hepevt_pol
```

By this variable the identity of the current process is given, defined via the LPRUP codes.

⟨*HEP common: variables*⟩+≡
```
integer, public :: idruplh
```

This is the event weight, i.e. the cross section divided by the total number of generated events for the output of the parton shower programs.

⟨*HEP common: variables*⟩+≡
```
double precision, public :: eventweightlh
```

There are the values for the electromagnetic and the strong coupling constants, α_{em} and α_s.

⟨*HEP common: variables*⟩+≡
```
double precision, public :: alphaqedlh, alphaqcdlh
```

This is the squared scale Q of the event.

⟨*HEP common: variables*⟩+≡
```
double precision, dimension(10), public :: scalelh
```

Finally, these variables contain the spin information and the color/anticolor flow of the particles.

⟨*HEP common: variables*⟩+≡
```
double precision, dimension (3,NMXHEP), public :: spinlh
integer, dimension (2,NMXHEP), public :: icolorflowlh
```

By convention, SHEP(4,N) is always 1. All this is taken from StdHep 4.06 manual and written using Fortran90 conventions.

⟨*HEP common: common blocks*⟩+≡
```
common /HEPEVT/ &
     NEVHEP, NHEP, ISTHEP, IDHEP, &
     JMOHEP, JDAHEP, PHEP, VHEP
save /HEPEVT/
```

Here we store HEPEVT parameters of the WHIZARD 1 realization which are not part of the HEPEVT common block.

⟨*HEP common: variables*⟩+≡
```
integer :: hepevt_n_out, hepevt_n_remnants
```

⟨*HEP common: variables*⟩+≡
```
double precision :: hepevt_weight, hepevt_function_value
double precision :: hepevt_function_ratio
```

The HEPEV4 common block is an extension of the HEPEVT common block to allow for partonic colored events, including especially the color flow etc.

⟨*HEP common: common blocks*⟩+≡
```
common /HEPEV4/ &
     eventweightlh, alphaqedlh, alphaqcdlh, scalelh, &
     spinlh, icolorflowlh, idruplh
save /HEPEV4/
```

Filling HEPEVT: If the event count is not provided, set `NEVHEP` to zero. If the event count is `-1` or `-2`, the record corresponds to initialization and finalization, and the event is irrelevant.

Note that the event count may be larger than 2^{31} (2 GEvents). In that case, cut off the upper bits since `NEVHEP` is probably limited to default integer.

For the HEPEV4 common block, it is unclear why the `scalelh` variable is 10-dimensional. We choose to only set the first value of the array.

⟨*HEP common: public*⟩+≡
```
public :: hepevt_init
public :: hepevt_set_event_parameters
```

⟨*HEP common: procedures*⟩+≡
```
subroutine hepevt_init (n_tot, n_out)
  integer, intent(in) :: n_tot, n_out
  NHEP              = n_tot
  NEVHEP            = 0
  idruplh           = 0
  hepevt_n_out      = n_out
  hepevt_n_remnants = 0
  hepevt_weight     = 1
  eventweightlh     = 1
  hepevt_function_value = 0
  hepevt_function_ratio = 1
  alphaqcdlh        = -1
  alphaqedlh        = -1
  scalelh           = -1
end subroutine hepevt_init

subroutine hepevt_set_event_parameters &
     (proc_id, weight, function_value, function_ratio, &
     alpha_qcd, alpha_qed, scale, i_evt)
  integer, intent(in), optional :: proc_id
  integer, intent(in), optional :: i_evt
  real(default), intent(in), optional :: weight, function_value, &
     function_ratio, alpha_qcd, alpha_qed, scale
```

```
    if (present (proc_id))  idruplh = proc_id
    if (present (i_evt))  NEVHEP = i_evt
    if (present (weight)) then
       hepevt_weight = weight
       eventweightlh = weight
    end if
    if (present (function_value)) hepevt_function_value = &
         function_value
    if (present (function_ratio)) hepevt_function_ratio = &
         function_ratio
    if (present (alpha_qcd))  alphaqcdlh = alpha_qcd
    if (present (alpha_qed))  alphaqedlh = alpha_qed
    if (present (scale))  scalelh(1) = scale
  end subroutine hepevt_set_event_parameters
```

Set the entry for a specific particle. All parameters are set with the exception of lifetime and spin, where default values are stored.

⟨*HEP common: public*⟩+≡

```
  public :: hepevt_set_particle
```

⟨*HEP common: procedures*⟩+≡

```
  subroutine hepevt_set_particle &
       (i, pdg, status, parent, child, p, m2, hel, vtx, &
       col, pol_status, pol, fill_hepev4)
    integer, intent(in) :: i
    integer, intent(in) :: pdg, status
    integer, dimension(:), intent(in) :: parent
    integer, dimension(:), intent(in) :: child
    logical, intent(in), optional :: fill_hepev4
    type(vector4_t), intent(in) :: p
    real(default), intent(in) :: m2
    integer, dimension(2), intent(in) :: col
    integer, intent(in) :: pol_status
    integer, intent(in) :: hel
    type(polarization_t), intent(in), optional :: pol
    type(vector4_t), intent(in) :: vtx
    logical :: hepev4
    hepev4 = .false.; if (present (fill_hepev4))  hepev4 = fill_hepev4
    IDHEP(i) = pdg
    select case (status)
      case (PRT_BEAM);      ISTHEP(i) = 2
      case (PRT_INCOMING);  ISTHEP(i) = 2
      case (PRT_OUTGOING);  ISTHEP(i) = 1
      case (PRT_VIRTUAL);   ISTHEP(i) = 2
      case (PRT_RESONANT);  ISTHEP(i) = 2
      case default;         ISTHEP(i) = 0
    end select
    select case (size (parent))
    case (0);      JMOHEP(:,i) = 0
    case (1);      JMOHEP(1,i) = parent(1); JMOHEP(2,i) = 0
    case default;  JMOHEP(:,i) = [ parent(1), parent(size (parent)) ]
    end select
    select case (size (child))
    case (0);      JDAHEP(:,i) = 0
```

```
    case (1);      JDAHEP(:,i) = child(1)
    case default;  JDAHEP(:,i) = [ child(1), child(size (child)) ]
    end select
    PHEP(1:3,i) = vector3_get_components (space_part (p))
    PHEP(4,i) = energy (p)
    PHEP(5,i) = sign (sqrt (abs (m2)), m2)
    VHEP(1:3,i) = vtx%p(1:3)
    VHEP(4,i) = vtx%p(0)
    hepevt_pol(i) = hel
    if (hepev4) then
       if (col(1) > 0) then
          icolorflowlh(1,i) = 500 + col(1)
       else
          icolorflowlh(1,i) = 0
       end if
       if (col(2) > 0) then
          icolorflowlh(2,i) = 500 + col(2)
       else
          icolorflowlh(2,i) = 0
       end if
       if (present (pol) .and. &
            pol_status == PRT_GENERIC_POLARIZATION) then
          if (pol%is_polarized ()) &
             spinlh(:,i) = pol%get_axis ()
       else
          spinlh(:,i) = zero
          spinlh(3,i) = hel
       end if
    end if
  end subroutine hepevt_set_particle
```

17.9.8 Event output

This is a verbose output of the HEPEVT block.

⟨*HEP common: public*⟩+≡

```
  public :: hepevt_write_verbose
```

⟨*HEP common: procedures*⟩+≡

```
  subroutine hepevt_write_verbose (unit)
    integer, intent(in), optional :: unit
    integer :: u, i
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(A)")  "HEPEVT Common Block"
    write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)")  "NEVHEP", NEVHEP, &
         "Event number"
    write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)")  "NHEP  ", NHEP, &
         "Number of particles in event"
    do i = 1, NHEP
       write (u, "(1x,A,I0)")  "Particle #", i
       write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)", advance="no") &
            "ISTHEP", ISTHEP(i), "Status code: "
       select case (ISTHEP(i))
       case ( 0);  write (u, "(A)")  "null entry"
```

```
      case ( 1);  write (u, "(A)")  "outgoing"
      case ( 2);  write (u, "(A)")  "decayed"
      case ( 3);  write (u, "(A)")  "documentation"
      case (4:10);  write (u, "(A)")  "[unspecified]"
      case (11:200);  write (u, "(A)")  "[model-specific]"
      case (201:);  write (u, "(A)")  "[user-defined]"
      case default;  write (u, "(A)")  "[undefined]"
      end select
      write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)")  "IDHEP ", IDHEP(i), &
           "PDG code of particle"
      write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "JMOHEP", JMOHEP(:,i), &
           "Index of first/second mother"
      write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "JDAHEP", JDAHEP(:,i), &
           "Index of first/last daughter"
      write (u, "(3x,A6,' = ',ES12.5,1x,ES12.5,8x,A)")  "PHEP12", &
           PHEP(1:2,i), "Transversal momentum (x/y) in GeV"
      write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "PHEP3 ", PHEP(3,i), &
           "Longitudinal momentum (z) in GeV"
      write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "PHEP4 ", PHEP(4,i), &
           "Energy in GeV"
      write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "PHEP5 ", PHEP(5,i), &
           "Invariant mass in GeV"
      write (u, "(3x,A6,' = ',ES12.5,1x,ES12.5,8x,A)")  "VHEP12", VHEP(1:2,i), &
           "Transversal displacement (xy) in mm"
      write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "VHEP3 ", VHEP(3,i), &
           "Longitudinal displacement (z) in mm"
      write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "VHEP4 ", VHEP(4,i), &
           "Production time in mm"
    end do
  end subroutine hepevt_write_verbose
```

This is a verbose output of the HEPEUP block.

⟨*HEP common: public*⟩+≡

```
  public :: hepeup_write_verbose
```

⟨*HEP common: procedures*⟩+≡

```
  subroutine hepeup_write_verbose (unit)
    integer, intent(in), optional :: unit
    integer :: u, i
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(A)")  "HEPEUP Common Block"
    write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)")  "NUP   ", NUP, &
         "Number of particles in event"
    write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)")  "IDPRUP", IDPRUP, &
         "Subprocess ID"
    write (u, "(3x,A6,' = ',ES12.5,1x,20x,A)")  "XWGTUP", XWGTUP, &
         "Event weight"
    write (u, "(3x,A6,' = ',ES12.5,1x,20x,A)")  "SCALUP", SCALUP, &
         "Event energy scale in GeV"
    write (u, "(3x,A6,' = ',ES12.5,1x,20x,A)")  "AQEDUP", AQEDUP, &
         "QED coupling [-1 = undefined]"
    write (u, "(3x,A6,' = ',ES12.5,1x,20x,A)")  "AQCDUP", AQCDUP, &
         "QCD coupling [-1 = undefined]"
    do i = 1, NUP
```

```
      write (u, "(1x,A,I0)")  "Particle #", i
      write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)")  "IDUP  ", IDUP(i), &
           "PDG code of particle"
      write (u, "(3x,A6,' = ',I9,3x,1x,20x,A)", advance="no") &
           "ISTUP ", ISTUP(i), "Status code: "
      select case (ISTUP(i))
      case (-1);  write (u, "(A)")  "incoming"
      case ( 1);  write (u, "(A)")  "outgoing"
      case (-2);  write (u, "(A)")  "spacelike"
      case ( 2);  write (u, "(A)")  "resonance"
      case ( 3);  write (u, "(A)")  "resonance (doc)"
      case (-9);  write (u, "(A)")  "beam"
      case default;  write (u, "(A)")  "[undefined]"
      end select
      write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "MOTHUP", MOTHUP(:,i), &
           "Index of first/last mother"
      write (u, "(3x,A6,' = ',I9,3x,1x,I9,3x,8x,A)")  "ICOLUP", ICOLUP(:,i), &
           "Color/anticolor flow index"
      write (u, "(3x,A6,' = ',ES12.5,1x,ES12.5,8x,A)")  "PUP1/2", PUP(1:2,i), &
           "Transversal momentum (x/y) in GeV"
      write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "PUP3  ", PUP(3,i), &
           "Longitudinal momentum (z) in GeV"
      write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "PUP4  ", PUP(4,i), &
           "Energy in GeV"
      write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "PUP5  ", PUP(5,i), &
           "Invariant mass in GeV"
      write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "VTIMUP", VTIMUP(i), &
           "Invariant lifetime in mm"
      write (u, "(3x,A6,' = ',ES12.5,1x,12x,8x,A)")  "SPINUP", SPINUP(i), &
           "cos(spin angle) [9 = undefined]"
    end do
  end subroutine hepeup_write_verbose
```

17.9.9 Event output in various formats

This routine writes event output according to the LHEF standard. It uses the current contents of the HEPEUP block.

⟨*HEP common: public*⟩+≡

```
  public :: hepeup_write_lhef
  public :: hepeup_write_lha
```

⟨*HEP common: procedures*⟩+≡

```
  subroutine hepeup_write_lhef (unit)
    integer, intent(in), optional :: unit
    integer :: u, i
    u = given_output_unit (unit);  if (u < 0)  return
    call msg_debug (D_EVENTS, "hepeup_write_lhef")
    call msg_debug2 (D_EVENTS, "ID IST MOTH ICOL P VTIM SPIN")
    write (u, "(2(1x,I0),4(1x,ES17.10))") &
         NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP
    do i = 1, NUP
       write (u, "(6(1x,I0),7(1x,ES17.10))") &
            IDUP(i), ISTUP(i), MOTHUP(:,i), ICOLUP(:,i), &
```

```
          PUP(:,i), VTIMUP(i), SPINUP(i)
      if (debug2_active (D_EVENTS)) then
         write (msg_buffer, "(6(1x,I0),7(1x,ES17.10))") &
              IDUP(i), ISTUP(i), MOTHUP(:,i), ICOLUP(:,i), &
              PUP(:,i), VTIMUP(i), SPINUP(i)
         call msg_message ()
      end if
    end do
  end subroutine hepeup_write_lhef

  subroutine hepeup_write_lha (unit)
    integer, intent(in), optional :: unit
    integer :: u, i
    integer, dimension(MAXNUP) :: spin_up
    spin_up = int(SPINUP)
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(2(1x,I5),1x,ES17.10,3(1x,ES13.6))") &
         NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP
    write (u, "(500(1x,I5))") IDUP(:NUP)
    write (u, "(500(1x,I5))") MOTHUP(1,:NUP)
    write (u, "(500(1x,I5))") MOTHUP(2,:NUP)
    write (u, "(500(1x,I5))") ICOLUP(1,:NUP)
    write (u, "(500(1x,I5))") ICOLUP(2,:NUP)
    write (u, "(500(1x,I5))") ISTUP(:NUP)
    write (u, "(500(1x,I5))") spin_up(:NUP)
    do i = 1, NUP
            write (u, "(1x,I5,4(1x,ES17.10))") i, PUP([ 4,1,2,3 ], i)
    end do

  end subroutine hepeup_write_lha
```

This routine writes event output according to the HEPEVT standard. It uses the current contents of the HEPEVT block and some additional parameters according to the standard in WHIZARD 1. For the long ASCII format, the value of the sample function (i.e. the product of squared matrix element, structure functions and phase space factor is printed out). The option of reweighting matrix elements with respect to some reference cross section is not implemented in WHIZARD 2 for this event format, therefore the second entry in the long ASCII format (the function ratio) is always one. The ATHENA format is an implementation of the HEPEVT format that is readable by the ATLAS ATHENA software framework. It is very similar to the WHIZARD 1 HEPEVT format, except that it contains an event counter, a particle counter inside the event, and has the HEPEVT `ISTHEP` status before the PDG code. The MOKKA format is a special ASCII format that contains the information to be parsed to the MOKKA LC fast simulation software.

⟨*HEP common: public*⟩+≡

```
  public :: hepevt_write_hepevt
  public :: hepevt_write_ascii
  public :: hepevt_write_athena
  public :: hepevt_write_mokka
```

⟨*HEP common: procedures*⟩+≡

```
  subroutine hepevt_write_hepevt (unit)
```

```
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(3(1x,I0),(1x,ES17.10))") &
       NHEP, hepevt_n_out, hepevt_n_remnants, hepevt_weight
  do i = 1, NHEP
     write (u, "(7(1x,I0))") &
          ISTHEP(i), IDHEP(i), JMOHEP(:,i), JDAHEP(:,i), hepevt_pol(i)
     write (u, "(5(1x,ES17.10))") PHEP(:,i)
     write (u, "(5(1x,ES17.10))") VHEP(:,i), 0.d0
  end do
end subroutine hepevt_write_hepevt

subroutine hepevt_write_ascii (unit, long)
  integer, intent(in), optional :: unit
  logical, intent(in) :: long
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(3(1x,I0),(1x,ES17.10))") &
       NHEP, hepevt_n_out, hepevt_n_remnants, hepevt_weight
  do i = 1, NHEP
     if (ISTHEP(i) /= 1)  cycle
     write (u, "(2(1x,I0))") IDHEP(i), hepevt_pol(i)
     write (u, "(5(1x,ES17.10))") PHEP(:,i)
  end do
  if (long) then
     write (u, "(2(1x,ES17.10))") &
          hepevt_function_value, hepevt_function_ratio
  end if
end subroutine hepevt_write_ascii

subroutine hepevt_write_athena (unit, i_evt)
  integer, intent(in), optional :: unit, i_evt
  integer :: u, i, num_event
  num_event = 0
  if (present (i_evt)) num_event = i_evt
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(2(1x,I0))") num_event, NHEP
  do i = 1, NHEP
     write (u, "(7(1x,I0))") &
          i, ISTHEP(i), IDHEP(i), JMOHEP(:,i), JDAHEP(:,i)
     write (u, "(5(1x,ES17.10))") PHEP(:,i)
     write (u, "(5(1x,ES17.10))") VHEP(1:4,i)
  end do
end subroutine hepevt_write_athena

subroutine hepevt_write_mokka (unit)
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(3(1x,I0),(1x,ES17.10))") &
       NHEP, hepevt_n_out, hepevt_n_remnants, hepevt_weight
  do i = 1, NHEP
     write (u, "(4(1x,I0),4(1x,ES17.10))") &
```

```
            ISTHEP(i), IDHEP(i), JDAHEP(1,i), JDAHEP(2,i), &
            PHEP(1:3,i), PHEP(5,i)
      end do
    end subroutine hepevt_write_mokka
```

17.9.10 Event input in various formats

This routine writes event output according to the LHEF standard. It uses the current contents of the HEPEUP block.

⟨*HEP common: public*⟩+≡

```
    public :: hepeup_read_lhef
```

⟨*HEP common: procedures*⟩+≡

```
    subroutine hepeup_read_lhef (u)
      integer, intent(in) :: u
      integer :: i
      read (u, *) &
           NUP, IDPRUP, XWGTUP, SCALUP, AQEDUP, AQCDUP
      do i = 1, NUP
         read (u, *) &
              IDUP(i), ISTUP(i), MOTHUP(:,i), ICOLUP(:,i), &
              PUP(:,i), VTIMUP(i), SPINUP(i)
      end do
    end subroutine hepeup_read_lhef
```

17.9.11 Data Transfer: particle sets

The `WHIZARD` format for handling particle data in events is `particle_set_t`. We have to interface this to the common blocks.

We first create a new particle set that contains only the particles that are supported by the LHEF format. These are: beam, incoming, resonant, outgoing. We drop particles with unknown, virtual or beam-remnant status.

From this set we fill the common block. Event information such as process ID and weight is not transferred here; this has to be done by the caller. The spin information is set only if the particle has a unique mother, and if its polarization is fully defined.

We use this routine also to hand over information to Pythia which lets Tauola access SPINUP. Tauola expects in SPINUP the helicity and not the LHA convention. We switch to this mode with `tauola_convention`.

⟨*HEP common: public*⟩+≡

```
    public :: hepeup_from_particle_set
```

⟨*HEP common: procedures*⟩+≡

```
    subroutine hepeup_from_particle_set (pset_in, &
       keep_beams, keep_remnants, tauola_convention)
      type(particle_set_t), intent(in) :: pset_in
      type(particle_set_t), target :: pset
      logical, intent(in), optional :: keep_beams
      logical, intent(in), optional :: keep_remnants
      logical, intent(in), optional :: tauola_convention
```

```
  integer :: i, n_parents, status, n_tot
  integer, dimension(1) :: i_mother
  logical :: kr, tc
  kr = .true.;  if (present (keep_remnants))  kr = keep_remnants
  tc = .false.;  if (present (tauola_convention))  tc = tauola_convention
  call pset_in%filter_particles (pset, real_parents = .true. , &
     keep_beams = keep_beams, keep_virtuals = .false.)
  n_tot = pset%get_n_tot ()
  call hepeup_init (n_tot)
  do i = 1, n_tot
     associate (prt => pset%prt(i))
       status = prt%get_status ()
       if (kr .and. status == PRT_BEAM_REMNANT &
              .and. prt%get_n_children () == 0) &
            status = PRT_OUTGOING
       call hepeup_set_particle (i, &
            prt%get_pdg (), &
            status, &
            prt%get_parents (), &
            prt%get_color (), &
            prt%get_momentum (), &
            prt%get_p2 ())
       n_parents = prt%get_n_parents ()
       call hepeup_set_particle_lifetime (i, &
            prt%get_lifetime ())
       if (.not. tc) then
          if (n_parents == 1) then
             i_mother = prt%get_parents ()
             select case (prt%get_polarization_status ())
             case (PRT_GENERIC_POLARIZATION)
                call hepeup_set_particle_spin (i, &
                     prt%get_momentum (), &
                     prt%get_polarization (), &
                     pset%prt(i_mother(1))%get_momentum ())
             end select
          end if
       else
          select case (prt%get_polarization_status ())
          case (PRT_DEFINITE_HELICITY)
            SPINUP(i) = prt%get_helicity()
          end select
       end if
     end associate
  end do
end subroutine hepeup_from_particle_set
```

Input. The particle set should be allocated properly, but we replace the particle contents.

If there are no beam particles in the event, we try to reconstruct beam particles and beam remnants. We assume for simplicity that the beam particles, if any, are the first two particles. If they are absent, the first two particles should be the incoming partons.

⟨*HEP common: public*⟩+≡

```
public :: hepeup_to_particle_set
```

⟨*HEP common: procedures*⟩+≡

```
subroutine hepeup_to_particle_set &
    (particle_set, recover_beams, model, alt_model)
  type(particle_set_t), intent(inout), target :: particle_set
  logical, intent(in), optional :: recover_beams
  class(model_data_t), intent(in), target :: model, alt_model
  type(particle_t), dimension(:), allocatable :: prt
  integer, dimension(2) :: parent
  integer, dimension(:), allocatable :: child
  integer :: i, j, k, pdg, status
  type(flavor_t) :: flv
  type(color_t) :: col
  integer, dimension(2) :: c
  type(vector4_t) :: p
  real(default) :: p2
  logical :: reconstruct
  integer :: off
  if (present (recover_beams)) then
     reconstruct = recover_beams .and. .not. all (ISTUP(1:2) == PRT_BEAM)
  else
     reconstruct = .false.
  end if
  if (reconstruct) then
     off = 4
  else
     off = 0
  end if
  allocate (prt (NUP + off), child (NUP + off))
  do i = 1, NUP
     k = i + off
     call hepeup_get_particle (i, pdg, status, col = c, p = p, m2 = p2)
     call flv%init (pdg, model, alt_model)
     call prt(k)%set_flavor (flv)
     call prt(k)%reset_status (status)
     call col%init (c)
     call prt(k)%set_color (col)
     call prt(k)%set_momentum (p, p2)
     where (MOTHUP(:,i) /= 0)
        parent = MOTHUP(:,i) + off
     elsewhere
        parent = 0
     end where
     call prt(k)%set_parents (parent)
     child = [(j, j = 1 + off, NUP + off)]
     where (MOTHUP(1,:NUP) /= i .and. MOTHUP(2,:NUP) /= i)  child = 0
     call prt(k)%set_children (child)
  end do
  if (reconstruct) then
     do k = 1, 2
        call prt(k)%reset_status (PRT_BEAM)
        call prt(k)%set_children ([k+2,k+4])
     end do
     do k = 3, 4
```

```
         call prt(k)%reset_status (PRT_BEAM_REMNANT)
         call prt(k)%set_parents ([k-2])
      end do
      do k = 5, 6
         call prt(k)%set_parents ([k-4])
      end do
   end if
   call particle_set%replace (prt)
end subroutine hepeup_to_particle_set
```

The HEPEVT common block is quite similar, but does contain less information, e.g. no color flows (it was LEP time). The spin information is set only if the particle has a unique mother, and if its polarization is fully defined.

⟨*HEP common: public*⟩+≡

```
public :: hepevt_from_particle_set
```

⟨*HEP common: procedures*⟩+≡

```
subroutine hepevt_from_particle_set &
     (particle_set, keep_beams, keep_remnants, ensure_order, fill_hepev4)
  type(particle_set_t), intent(in) :: particle_set
  type(particle_set_t), target :: pset_hepevt, pset_tmp
  logical, intent(in), optional :: keep_beams
  logical, intent(in), optional :: keep_remnants
  logical, intent(in), optional :: ensure_order
  logical, intent(in), optional :: fill_hepev4
  integer :: i, status, n_tot
  logical :: activate_remnants, ensure
  activate_remnants = .true.
  if (present (keep_remnants))  activate_remnants = keep_remnants
  ensure = .false.
  if (present (ensure_order))  ensure = ensure_order
  call particle_set%filter_particles (pset_tmp, real_parents = .true., &
     keep_virtuals = .false., keep_beams = keep_beams)
  if (ensure) then
     call pset_tmp%to_hepevt_form (pset_hepevt)
  else
     pset_hepevt = pset_tmp
  end if
  n_tot = pset_hepevt%get_n_tot ()
  call hepevt_init (n_tot, pset_hepevt%get_n_out ())
  do i = 1, n_tot
     associate (prt => pset_hepevt%prt(i))
       status = prt%get_status ()
       if (activate_remnants &
            .and. status == PRT_BEAM_REMNANT &
            .and. prt%get_n_children () == 0) &
            status = PRT_OUTGOING
       select case (prt%get_polarization_status ())
       case (PRT_GENERIC_POLARIZATION)
          call hepevt_set_particle (i, &
               prt%get_pdg (), status, &
               prt%get_parents (), &
               prt%get_children (), &
               prt%get_momentum (), &
```

```
                prt%get_p2 (), &
                prt%get_helicity (), &
                prt%get_vertex (), &
                prt%get_color (), &
                prt%get_polarization_status (), &
                pol = prt%get_polarization (), &
                fill_hepev4 = fill_hepev4)
         case default
            call hepevt_set_particle (i, &
                prt%get_pdg (), status, &
                prt%get_parents (), &
                prt%get_children (), &
                prt%get_momentum (), &
                prt%get_p2 (), &
                prt%get_helicity (), &
                prt%get_vertex (), &
                prt%get_color (), &
                prt%get_polarization_status (), &
                fill_hepev4 = fill_hepev4)
         end select
      end associate
   end do
   call pset_hepevt%final ()
end subroutine hepevt_from_particle_set
```

17.10 HepMC events

This section provides the interface to the HepMC C++ library for handling Monte-Carlo events.

Each C++ class of HepMC that we use is mirrored by a Fortran type, which contains as its only component the C pointer to the C++ object.

Each C++ method of HepMC that we use has a C wrapper function. This function takes a pointer to the host object as its first argument. Further arguments are either C pointers, or in the case of simple types (integer, real), interoperable C/Fortran objects.

The C wrapper functions have explicit interfaces in the Fortran module. They are called by Fortran wrapper procedures. These are treated as methods of the corresponding Fortran type.

⟨hepmc_interface.f90⟩≡

```
⟨File header⟩

module hepmc_interface

  use, intrinsic :: iso_c_binding !NODEP!

⟨Use kinds⟩
⟨Use strings⟩
  use constants, only: PI
  use lorentz
  use flavors
  use colors
```

```
  use helicities
  use polarizations
```

⟨*Standard module head*⟩

⟨*HepMC interface: public*⟩

⟨*HepMC interface: types*⟩

⟨*HepMC interface: parameters*⟩

⟨*HepMC interface: interfaces*⟩

```
contains
```

⟨*HepMC interface: procedures*⟩

```
end module hepmc_interface
```

17.10.1 Interface check

This function can be called in order to verify that we are using the actual HepMC library, and not the dummy version.

⟨*HepMC interface: interfaces*⟩≡

```
interface
   logical(c_bool) function hepmc_available () bind(C)
     import
   end function hepmc_available
end interface
```

⟨*HepMC interface: public*⟩≡

```
public :: hepmc_is_available
```

⟨*HepMC interface: procedures*⟩≡

```
function hepmc_is_available () result (flag)
  logical :: flag
  flag = hepmc_available ()
end function hepmc_is_available
```

17.10.2 FourVector

The C version of four-vectors is often transferred by value, and the associated procedures are all inlined. The wrapper needs to transfer by reference, so we create FourVector objects on the heap which have to be deleted explicitly. The input is a `vector4_t` or `vector3_t` object from the `lorentz` module.

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_four_vector_t
```

⟨*HepMC interface: types*⟩≡

```
type :: hepmc_four_vector_t
   private
   type(c_ptr) :: obj
end type hepmc_four_vector_t
```

In the C constructor, the zero-component (fourth argument) is optional; if missing, it is set to zero. The Fortran version has initializer form and takes either a three-vector or a four-vector. A further version extracts the four-vector from a HepMC particle object.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function new_four_vector_xyz (x, y, z) bind(C)
     import
     real(c_double), value :: x, y, z
   end function new_four_vector_xyz
end interface
interface
   type(c_ptr) function new_four_vector_xyzt (x, y, z, t) bind(C)
     import
     real(c_double), value :: x, y, z, t
   end function new_four_vector_xyzt
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_four_vector_init
```

⟨*HepMC interface: interfaces*⟩+≡

```
interface hepmc_four_vector_init
   module procedure hepmc_four_vector_init_v4
   module procedure hepmc_four_vector_init_v3
   module procedure hepmc_four_vector_init_hepmc_prt
end interface
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_four_vector_init_v4 (pp, p)
  type(hepmc_four_vector_t), intent(out) :: pp
  type(vector4_t), intent(in) :: p
  real(default), dimension(0:3) :: pa
  pa = vector4_get_components (p)
  pp%obj = new_four_vector_xyzt &
       (real (pa(1), c_double), &
        real (pa(2), c_double), &
        real (pa(3), c_double), &
        real (pa(0), c_double))
end subroutine hepmc_four_vector_init_v4

subroutine hepmc_four_vector_init_v3 (pp, p)
  type(hepmc_four_vector_t), intent(out) :: pp
  type(vector3_t), intent(in) :: p
  real(default), dimension(3) :: pa
  pa = vector3_get_components (p)
  pp%obj = new_four_vector_xyz &
       (real (pa(1), c_double), &
        real (pa(2), c_double), &
        real (pa(3), c_double))
end subroutine hepmc_four_vector_init_v3

subroutine hepmc_four_vector_init_hepmc_prt (pp, prt)
  type(hepmc_four_vector_t), intent(out) :: pp
  type(hepmc_particle_t), intent(in) :: prt
```

```
  pp%obj = gen_particle_momentum (prt%obj)
end subroutine hepmc_four_vector_init_hepmc_prt
```

Here, the destructor is explicitly needed.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine four_vector_delete (p_obj) bind(C)
     import
     type(c_ptr), value :: p_obj
   end subroutine four_vector_delete
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_four_vector_final
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_four_vector_final (p)
  type(hepmc_four_vector_t), intent(inout) :: p
  call four_vector_delete (p%obj)
end subroutine hepmc_four_vector_final
```

Convert to a Lorentz vector.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   function four_vector_px (p_obj) result (px) bind(C)
     import
     real(c_double) :: px
     type(c_ptr), value :: p_obj
   end function four_vector_px
end interface
interface
   function four_vector_py (p_obj) result (py) bind(C)
     import
     real(c_double) :: py
     type(c_ptr), value :: p_obj
   end function four_vector_py
end interface
interface
   function four_vector_pz (p_obj) result (pz) bind(C)
     import
     real(c_double) :: pz
     type(c_ptr), value :: p_obj
   end function four_vector_pz
end interface
interface
   function four_vector_e (p_obj) result (e) bind(C)
     import
     real(c_double) :: e
     type(c_ptr), value :: p_obj
   end function four_vector_e
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_four_vector_to_vector4
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_four_vector_to_vector4 (pp, p)
  type(hepmc_four_vector_t), intent(in) :: pp
  type(vector4_t), intent(out) :: p
  real(default) :: E
  real(default), dimension(3) :: p3
  E = four_vector_e (pp%obj)
  p3(1) = four_vector_px (pp%obj)
  p3(2) = four_vector_py (pp%obj)
  p3(3) = four_vector_pz (pp%obj)
  p = vector4_moving (E, vector3_moving (p3))
end subroutine hepmc_four_vector_to_vector4
```

17.10.3 Polarization

Polarization objects are temporarily used for assigning particle polarization. We add a flag `polarized`. If this is false, the polarization is not set and should not be transferred to `hepmc_particle` objects.

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_polarization_t
```

⟨*HepMC interface: types*⟩+≡

```
type :: hepmc_polarization_t
   private
   logical :: polarized = .false.
   type(c_ptr) :: obj
end type hepmc_polarization_t
```

Constructor. The C wrapper takes polar and azimuthal angle as arguments. The Fortran version allows for either a complete polarization density matrix, or for a definite (diagonal) helicity.

HepMC does not allow to specify the degree of polarization, therefore we have to map it to either 0 or 1. We choose 0 for polarization less than 0.5 *and 1 for polarization greater than* 0.5. *Even this simplification works only for spin-1/2 and for massless particles; massive vector bosons cannot be treated this way. In particular, zero helicity is always translated as unpolarized.*

For massive vector bosons, we arbitrarily choose the convention that the longitudinal (zero) helicity state is mapped to the theta angle $\pi/2$. *This works under the condition that helicity is projected onto one of the basis states.*

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function new_polarization (theta, phi) bind(C)
     import
     real(c_double), value :: theta, phi
   end function new_polarization
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_polarization_init
```

⟨*HepMC interface: interfaces*⟩+≡

```
interface hepmc_polarization_init
   module procedure hepmc_polarization_init_pol
   module procedure hepmc_polarization_init_hel
   module procedure hepmc_polarization_init_int
end interface
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_polarization_init_pol (hpol, pol)
  type(hepmc_polarization_t), intent(out) :: hpol
  type(polarization_t), intent(in) :: pol
  real(default) :: r, theta, phi
  if (pol%is_polarized ()) then
     call pol%to_angles (r, theta, phi)
     if (r >= 0.5) then
        hpol%polarized = .true.
        hpol%obj = new_polarization &
              (real (theta, c_double), real (phi, c_double))
     end if
  end if
end subroutine hepmc_polarization_init_pol

subroutine hepmc_polarization_init_hel (hpol, hel)
  type(hepmc_polarization_t), intent(out) :: hpol
  type(helicity_t), intent(in) :: hel
  integer, dimension(2) :: h
  if (hel%is_defined ()) then
     h = hel%to_pair ()
     select case (h(1))
     case (1:)
        hpol%polarized = .true.
        hpol%obj = new_polarization (0._c_double, 0._c_double)
     case (:-1)
        hpol%polarized = .true.
        hpol%obj = new_polarization (real (pi, c_double), 0._c_double)
     case (0)
        hpol%polarized = .true.
        hpol%obj = new_polarization (real (pi/2, c_double), 0._c_double)
     end select
  end if
end subroutine hepmc_polarization_init_hel

subroutine hepmc_polarization_init_int (hpol, hel)
  type(hepmc_polarization_t), intent(out) :: hpol
  integer, intent(in) :: hel
  select case (hel)
  case (1:)
     hpol%polarized = .true.
     hpol%obj = new_polarization (0._c_double, 0._c_double)
  case (:-1)
     hpol%polarized = .true.
     hpol%obj = new_polarization (real (pi, c_double), 0._c_double)
  case (0)
     hpol%polarized = .true.
     hpol%obj = new_polarization (real (pi/2, c_double), 0._c_double)
```

```
    end select
  end subroutine hepmc_polarization_init_int
```

Destructor. The C object is deallocated only if the `polarized` flag is set.

⟨*HepMC interface: interfaces*⟩+≡

```
  interface
     subroutine polarization_delete (pol_obj) bind(C)
       import
       type(c_ptr), value :: pol_obj
     end subroutine polarization_delete
  end interface
```

⟨*HepMC interface: public*⟩+≡

```
  public :: hepmc_polarization_final
```

⟨*HepMC interface: procedures*⟩+≡

```
  subroutine hepmc_polarization_final (hpol)
    type(hepmc_polarization_t), intent(inout) :: hpol
    if (hpol%polarized)  call polarization_delete (hpol%obj)
  end subroutine hepmc_polarization_final
```

Recover polarization from HepMC polarization object (with the abovementioned deficiencies).

⟨*HepMC interface: interfaces*⟩+≡

```
  interface
     function polarization_theta (pol_obj) result (theta) bind(C)
       import
       real(c_double) :: theta
       type(c_ptr), value :: pol_obj
     end function polarization_theta
  end interface
  interface
     function polarization_phi (pol_obj) result (phi) bind(C)
       import
       real(c_double) :: phi
       type(c_ptr), value :: pol_obj
     end function polarization_phi
  end interface
```

⟨*HepMC interface: public*⟩+≡

```
  public :: hepmc_polarization_to_pol
```

⟨*HepMC interface: procedures*⟩+≡

```
  subroutine hepmc_polarization_to_pol (hpol, flv, pol)
    type(hepmc_polarization_t), intent(in) :: hpol
    type(flavor_t), intent(in) :: flv
    type(polarization_t), intent(out) :: pol
    real(default) :: theta, phi
    theta = polarization_theta (hpol%obj)
    phi = polarization_phi (hpol%obj)
    call pol%init_angles (flv, 1._default, theta, phi)
  end subroutine hepmc_polarization_to_pol
```

Recover helicity. Here, ϕ is ignored and only the sign of $\cos\theta$ is relevant, mapped to positive/negative helicity.

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_polarization_to_hel
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_polarization_to_hel (hpol, flv, hel)
  type(hepmc_polarization_t), intent(in) :: hpol
  type(flavor_t), intent(in) :: flv
  type(helicity_t), intent(out) :: hel
  real(default) :: theta
  integer :: hmax
  theta = polarization_theta (hpol%obj)
  hmax = flv%get_spin_type () / 2
  call hel%init (sign (hmax, nint (cos (theta))))
end subroutine hepmc_polarization_to_hel
```

17.10.4 GenParticle

Particle objects have the obvious meaning.

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_particle_t
```

⟨*HepMC interface: types*⟩+≡
```
type :: hepmc_particle_t
   private
   type(c_ptr) :: obj
end type hepmc_particle_t
```

Constructor. The C version takes a FourVector object, which in the Fortran wrapper is created on the fly from a `vector4` Lorentz vector.

No destructor is needed as long as all particles are entered into vertex containers.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   type(c_ptr) function new_gen_particle (prt_obj, pdg_id, status) bind(C)
     import
     type(c_ptr), value :: prt_obj
     integer(c_int), value :: pdg_id, status
   end function new_gen_particle
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_particle_init
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_particle_init (prt, p, pdg, status)
  type(hepmc_particle_t), intent(out) :: prt
  type(vector4_t), intent(in) :: p
  integer, intent(in) :: pdg, status
  type(hepmc_four_vector_t) :: pp
  call hepmc_four_vector_init (pp, p)
  prt%obj = new_gen_particle (pp%obj, int (pdg, c_int), int (status, c_int))
```

```
    call hepmc_four_vector_final (pp)
  end subroutine hepmc_particle_init
```

Set the particle color flow.

⟨*HepMC interface: interfaces*⟩+≡

```
  interface
     subroutine gen_particle_set_flow (prt_obj, code_index, code) bind(C)
       import
       type(c_ptr), value :: prt_obj
       integer(c_int), value :: code_index, code
     end subroutine gen_particle_set_flow
  end interface
```

Set the particle color. Either from a `color_t` object or directly from a pair of integers.

⟨*HepMC interface: interfaces*⟩+≡

```
  interface hepmc_particle_set_color
     module procedure hepmc_particle_set_color_col
     module procedure hepmc_particle_set_color_int
  end interface hepmc_particle_set_color
```

⟨*HepMC interface: public*⟩+≡

```
  public :: hepmc_particle_set_color
```

⟨*HepMC interface: procedures*⟩+≡

```
  subroutine hepmc_particle_set_color_col (prt, col)
    type(hepmc_particle_t), intent(inout) :: prt
    type(color_t), intent(in) :: col
    integer(c_int) :: c
    c = col%get_col ()
    if (c /= 0)  call gen_particle_set_flow (prt%obj, 1_c_int, c)
    c = col%get_acl ()
    if (c /= 0)  call gen_particle_set_flow (prt%obj, 2_c_int, c)
  end subroutine hepmc_particle_set_color_col

  subroutine hepmc_particle_set_color_int (prt, col)
    type(hepmc_particle_t), intent(inout) :: prt
    integer, dimension(2), intent(in) :: col
    integer(c_int) :: c
    c = col(1)
    if (c /= 0)  call gen_particle_set_flow (prt%obj, 1_c_int, c)
    c = col(2)
    if (c /= 0)  call gen_particle_set_flow (prt%obj, 2_c_int, c)
  end subroutine hepmc_particle_set_color_int
```

Set the particle polarization. For the restrictions on particle polarization in HepMC, see above `hepmc_polarization_init`.

⟨*HepMC interface: interfaces*⟩+≡

```
  interface
     subroutine gen_particle_set_polarization (prt_obj, pol_obj) bind(C)
       import
       type(c_ptr), value :: prt_obj, pol_obj
     end subroutine gen_particle_set_polarization
  end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_particle_set_polarization
```
⟨*HepMC interface: interfaces*⟩+≡
```
interface hepmc_particle_set_polarization
   module procedure hepmc_particle_set_polarization_pol
   module procedure hepmc_particle_set_polarization_hel
   module procedure hepmc_particle_set_polarization_int
end interface
```
⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_particle_set_polarization_pol (prt, pol)
  type(hepmc_particle_t), intent(inout) :: prt
  type(polarization_t), intent(in) :: pol
  type(hepmc_polarization_t) :: hpol
  call hepmc_polarization_init (hpol, pol)
  if (hpol%polarized)  call gen_particle_set_polarization (prt%obj, hpol%obj)
  call hepmc_polarization_final (hpol)
end subroutine hepmc_particle_set_polarization_pol

subroutine hepmc_particle_set_polarization_hel (prt, hel)
  type(hepmc_particle_t), intent(inout) :: prt
  type(helicity_t), intent(in) :: hel
  type(hepmc_polarization_t) :: hpol
  call hepmc_polarization_init (hpol, hel)
  if (hpol%polarized)  call gen_particle_set_polarization (prt%obj, hpol%obj)
  call hepmc_polarization_final (hpol)
end subroutine hepmc_particle_set_polarization_hel

subroutine hepmc_particle_set_polarization_int (prt, hel)
  type(hepmc_particle_t), intent(inout) :: prt
  integer, intent(in) :: hel
  type(hepmc_polarization_t) :: hpol
  call hepmc_polarization_init (hpol, hel)
  if (hpol%polarized)  call gen_particle_set_polarization (prt%obj, hpol%obj)
  call hepmc_polarization_final (hpol)
end subroutine hepmc_particle_set_polarization_int
```

Return the HepMC barcode (unique integer ID) of the particle.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   function gen_particle_barcode (prt_obj) result (barcode) bind(C)
     import
     integer(c_int) :: barcode
     type(c_ptr), value :: prt_obj
   end function gen_particle_barcode
end interface
```
⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_particle_get_barcode
```
⟨*HepMC interface: procedures*⟩+≡
```
function hepmc_particle_get_barcode (prt) result (barcode)
  integer :: barcode
  type(hepmc_particle_t), intent(in) :: prt
  barcode = gen_particle_barcode (prt%obj)
```

```
end function hepmc_particle_get_barcode
```

Return the four-vector component of the particle object as a `vector4_t` Lorentz vector.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function gen_particle_momentum (prt_obj) bind(C)
     import
     type(c_ptr), value :: prt_obj
   end function gen_particle_momentum
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_particle_get_momentum
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_particle_get_momentum (prt) result (p)
  type(vector4_t) :: p
  type(hepmc_particle_t), intent(in) :: prt
  type(hepmc_four_vector_t) :: pp
  call hepmc_four_vector_init (pp, prt)
  call hepmc_four_vector_to_vector4 (pp, p)
  call hepmc_four_vector_final (pp)
end function hepmc_particle_get_momentum
```

Return the invariant mass squared of the particle object. HepMC stores the signed invariant mass (no squaring).

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   function gen_particle_generated_mass (prt_obj) result (mass) bind(C)
     import
     real(c_double) :: mass
     type(c_ptr), value :: prt_obj
   end function gen_particle_generated_mass
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_particle_get_mass_squared
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_particle_get_mass_squared (prt) result (m2)
  real(default) :: m2
  type(hepmc_particle_t), intent(in) :: prt
  real(default) :: m
  m = gen_particle_generated_mass (prt%obj)
  m2 = sign (m**2, m)
end function hepmc_particle_get_mass_squared
```

Return the PDG ID:

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   function gen_particle_pdg_id (prt_obj) result (pdg_id) bind(C)
     import
     integer(c_int) :: pdg_id
```

```
      type(c_ptr), value :: prt_obj
    end function gen_particle_pdg_id
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_particle_get_pdg
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_particle_get_pdg (prt) result (pdg)
  integer :: pdg
  type(hepmc_particle_t), intent(in) :: prt
  pdg = gen_particle_pdg_id (prt%obj)
end function hepmc_particle_get_pdg
```

Return the status code:

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   function gen_particle_status (prt_obj) result (status) bind(C)
     import
     integer(c_int) :: status
     type(c_ptr), value :: prt_obj
   end function gen_particle_status
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_particle_get_status
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_particle_get_status (prt) result (status)
  integer :: status
  type(hepmc_particle_t), intent(in) :: prt
  status = gen_particle_status (prt%obj)
end function hepmc_particle_get_status
```

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   function gen_particle_is_beam (prt_obj) result (is_beam) bind(C)
     import
     logical(c_bool) :: is_beam
     type(c_ptr), value :: prt_obj
   end function gen_particle_is_beam
end interface
```

Determine whether a particle is a beam particle.

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_particle_is_beam
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_particle_is_beam (prt) result (is_beam)
  logical :: is_beam
  type(hepmc_particle_t), intent(in) :: prt
  is_beam = gen_particle_is_beam (prt%obj)
end function hepmc_particle_is_beam
```

Return the production/decay vertex (as a pointer, no finalization necessary).

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function gen_particle_production_vertex (prt_obj) bind(C)
     import
     type(c_ptr), value :: prt_obj
   end function gen_particle_production_vertex
end interface
interface
   type(c_ptr) function gen_particle_end_vertex (prt_obj) bind(C)
     import
     type(c_ptr), value :: prt_obj
   end function gen_particle_end_vertex
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_particle_get_production_vertex
public :: hepmc_particle_get_decay_vertex
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_particle_get_production_vertex (prt) result (v)
  type(hepmc_vertex_t) :: v
  type(hepmc_particle_t), intent(in) :: prt
  v%obj = gen_particle_production_vertex (prt%obj)
end function hepmc_particle_get_production_vertex

function hepmc_particle_get_decay_vertex (prt) result (v)
  type(hepmc_vertex_t) :: v
  type(hepmc_particle_t), intent(in) :: prt
  v%obj = gen_particle_end_vertex (prt%obj)
end function hepmc_particle_get_decay_vertex
```

Return the number of parents/children.

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_particle_get_n_parents
public :: hepmc_particle_get_n_children
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_particle_get_n_parents (prt) result (n_parents)
  integer :: n_parents
  type(hepmc_particle_t), intent(in) :: prt
  type(hepmc_vertex_t) :: v
  v = hepmc_particle_get_production_vertex (prt)
  if (hepmc_vertex_is_valid (v)) then
     n_parents = hepmc_vertex_get_n_in (v)
  else
     n_parents = 0
  end if
end function hepmc_particle_get_n_parents

function hepmc_particle_get_n_children (prt) result (n_children)
  integer :: n_children
  type(hepmc_particle_t), intent(in) :: prt
  type(hepmc_vertex_t) :: v
  v = hepmc_particle_get_decay_vertex (prt)
```

```
  if (hepmc_vertex_is_valid (v)) then
     n_children = hepmc_vertex_get_n_out (v)
  else
     n_children = 0
  end if
end function hepmc_particle_get_n_children
```

Convenience function: Return the array of parent particles for a given HepMC particle. The contents are HepMC barcodes that still have to be mapped to the particle indices.

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_particle_get_parent_barcodes
public :: hepmc_particle_get_child_barcodes
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_particle_get_parent_barcodes (prt) result (parent_barcode)
  type(hepmc_particle_t), intent(in) :: prt
  integer, dimension(:), allocatable :: parent_barcode
  type(hepmc_vertex_t) :: v
  type(hepmc_vertex_particle_in_iterator_t) :: it
  integer :: i
  v = hepmc_particle_get_production_vertex (prt)
  if (hepmc_vertex_is_valid (v)) then
     allocate (parent_barcode (hepmc_vertex_get_n_in (v)))
     if (size (parent_barcode) /= 0) then
        call hepmc_vertex_particle_in_iterator_init (it, v)
        do i = 1, size (parent_barcode)
           parent_barcode(i) = hepmc_particle_get_barcode &
                (hepmc_vertex_particle_in_iterator_get (it))
           call hepmc_vertex_particle_in_iterator_advance (it)
        end do
        call hepmc_vertex_particle_in_iterator_final (it)
     end if
  else
     allocate (parent_barcode (0))
  end if
end function hepmc_particle_get_parent_barcodes

function hepmc_particle_get_child_barcodes (prt) result (child_barcode)
  type(hepmc_particle_t), intent(in) :: prt
  integer, dimension(:), allocatable :: child_barcode
  type(hepmc_vertex_t) :: v
  type(hepmc_vertex_particle_out_iterator_t) :: it
  integer :: i
  v = hepmc_particle_get_decay_vertex (prt)
  if (hepmc_vertex_is_valid (v)) then
     allocate (child_barcode (hepmc_vertex_get_n_out (v)))
     call hepmc_vertex_particle_out_iterator_init (it, v)
     if (size (child_barcode) /= 0) then
        do i = 1, size (child_barcode)
           child_barcode(i) = hepmc_particle_get_barcode &
                (hepmc_vertex_particle_out_iterator_get (it))
           call hepmc_vertex_particle_out_iterator_advance (it)
        end do
```

```
          call hepmc_vertex_particle_out_iterator_final (it)
        end if
      else
        allocate (child_barcode (0))
      end if
    end function hepmc_particle_get_child_barcodes
```

Return the polarization (assuming that the particle is completely polarized). Note that the generated polarization object needs finalization.

⟨*HepMC interface: interfaces*⟩+≡

```
    interface
       type(c_ptr) function gen_particle_polarization (prt_obj) bind(C)
         import
         type(c_ptr), value :: prt_obj
       end function gen_particle_polarization
    end interface
```

⟨*HepMC interface: public*⟩+≡

```
    public :: hepmc_particle_get_polarization
```

⟨*HepMC interface: procedures*⟩+≡

```
    function hepmc_particle_get_polarization (prt) result (pol)
      type(hepmc_polarization_t) :: pol
      type(hepmc_particle_t), intent(in) :: prt
      pol%obj = gen_particle_polarization (prt%obj)
    end function hepmc_particle_get_polarization
```

Return the particle color as a two-dimensional array (color, anticolor).

⟨*HepMC interface: interfaces*⟩+≡

```
    interface
       function gen_particle_flow (prt_obj, code_index) result (code) bind(C)
         import
         integer(c_int) :: code
         type(c_ptr), value :: prt_obj
         integer(c_int), value :: code_index
       end function gen_particle_flow
    end interface
```

⟨*HepMC interface: public*⟩+≡

```
    public :: hepmc_particle_get_color
```

⟨*HepMC interface: procedures*⟩+≡

```
    function hepmc_particle_get_color (prt) result (col)
      integer, dimension(2) :: col
      type(hepmc_particle_t), intent(in) :: prt
      col(1) = gen_particle_flow (prt%obj, 1)
      col(2) = - gen_particle_flow (prt%obj, 2)
    end function hepmc_particle_get_color
```

⟨*HepMC interface: interfaces*⟩+≡

```
    interface
       function gen_vertex_pos_x (v_obj) result (x) bind(C)
         import
         type(c_ptr), value :: v_obj
```

```
      real(c_double) :: x
   end function gen_vertex_pos_x
end interface
interface
   function gen_vertex_pos_y (v_obj) result (y) bind(C)
      import
      type(c_ptr), value :: v_obj
      real(c_double) :: y
   end function gen_vertex_pos_y
end interface
interface
   function gen_vertex_pos_z (v_obj) result (z) bind(C)
      import
      type(c_ptr), value :: v_obj
      real(c_double) :: z
   end function gen_vertex_pos_z
end interface
interface
   function gen_vertex_time (v_obj) result (t) bind(C)
      import
      type(c_ptr), value :: v_obj
      real(c_double) :: t
   end function gen_vertex_time
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_to_vertex
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_vertex_to_vertex (vtx) result (v)
  type(hepmc_vertex_t), intent(in) :: vtx
  type(vector4_t) :: v
  real(default) :: t, vx, vy, vz
  if (hepmc_vertex_is_valid (vtx)) then
     t = gen_vertex_time (vtx%obj)
     vx = gen_vertex_pos_x (vtx%obj)
     vy = gen_vertex_pos_y (vtx%obj)
     vz = gen_vertex_pos_z (vtx%obj)
     v = vector4_moving (t, &
          vector3_moving ([vx, vy, vz]))
  end if
end function hepmc_vertex_to_vertex
```

17.10.5 GenVertex

Vertices are made of particles (incoming and outgoing).

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_t
```

⟨*HepMC interface: types*⟩+≡

```
type :: hepmc_vertex_t
   private
   type(c_ptr) :: obj
end type hepmc_vertex_t
```

Constructor. Two versions, one plain, one with the position in space and time (measured in mm) as argument. The Fortran version has initializer form, and the vertex position is an optional argument.

A destructor is unnecessary as long as all vertices are entered into an event container.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   type(c_ptr) function new_gen_vertex () bind(C)
     import
   end function new_gen_vertex
end interface
interface
   type(c_ptr) function new_gen_vertex_pos (prt_obj) bind(C)
     import
     type(c_ptr), value :: prt_obj
   end function new_gen_vertex_pos
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_vertex_init
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_vertex_init (v, x)
  type(hepmc_vertex_t), intent(out) :: v
  type(vector4_t), intent(in), optional :: x
  type(hepmc_four_vector_t) :: pos
  if (present (x)) then
     call hepmc_four_vector_init (pos, x)
     v%obj = new_gen_vertex_pos (pos%obj)
     call hepmc_four_vector_final (pos)
  else
     v%obj = new_gen_vertex ()
  end if
end subroutine hepmc_vertex_init
```

Return true if the vertex pointer is non-null:

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   function gen_vertex_is_valid (v_obj) result (flag) bind(C)
     import
     logical(c_bool) :: flag
     type(c_ptr), value :: v_obj
   end function gen_vertex_is_valid
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_vertex_is_valid
```

⟨*HepMC interface: procedures*⟩+≡
```
function hepmc_vertex_is_valid (v) result (flag)
  logical :: flag
  type(hepmc_vertex_t), intent(in) :: v
  flag = gen_vertex_is_valid (v%obj)
end function hepmc_vertex_is_valid
```

Add a particle to a vertex, incoming or outgoing.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine gen_vertex_add_particle_in (v_obj, prt_obj) bind(C)
     import
     type(c_ptr), value :: v_obj, prt_obj
   end subroutine gen_vertex_add_particle_in
end interface
interface
   subroutine gen_vertex_add_particle_out (v_obj, prt_obj) bind(C)
     import
     type(c_ptr), value :: v_obj, prt_obj
   end subroutine gen_vertex_add_particle_out
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_add_particle_in
public :: hepmc_vertex_add_particle_out
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_vertex_add_particle_in (v, prt)
  type(hepmc_vertex_t), intent(inout) :: v
  type(hepmc_particle_t), intent(in) :: prt
  call gen_vertex_add_particle_in (v%obj, prt%obj)
end subroutine hepmc_vertex_add_particle_in

subroutine hepmc_vertex_add_particle_out (v, prt)
  type(hepmc_vertex_t), intent(inout) :: v
  type(hepmc_particle_t), intent(in) :: prt
  call gen_vertex_add_particle_out (v%obj, prt%obj)
end subroutine hepmc_vertex_add_particle_out
```

Return the number of incoming/outgoing particles.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   function gen_vertex_particles_in_size (v_obj) result (size) bind(C)
     import
     integer(c_int) :: size
     type(c_ptr), value :: v_obj
   end function gen_vertex_particles_in_size
end interface
interface
   function gen_vertex_particles_out_size (v_obj) result (size) bind(C)
     import
     integer(c_int) :: size
     type(c_ptr), value :: v_obj
   end function gen_vertex_particles_out_size
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_get_n_in
public :: hepmc_vertex_get_n_out
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_vertex_get_n_in (v) result (n_in)
  integer :: n_in
  type(hepmc_vertex_t), intent(in) :: v
  n_in = gen_vertex_particles_in_size (v%obj)
end function hepmc_vertex_get_n_in

function hepmc_vertex_get_n_out (v) result (n_out)
  integer :: n_out
  type(hepmc_vertex_t), intent(in) :: v
  n_out = gen_vertex_particles_out_size (v%obj)
end function hepmc_vertex_get_n_out
```

17.10.6 Vertex-particle-in iterator

This iterator iterates over all incoming particles in an vertex. We store a pointer to the vertex in addition to the iterator. This allows for simple end checking.

The iterator is actually a constant iterator; it can only read.

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_in_iterator_t
```

⟨*HepMC interface: types*⟩+≡

```
type :: hepmc_vertex_particle_in_iterator_t
   private
   type(c_ptr) :: obj
   type(c_ptr) :: v_obj
end type hepmc_vertex_particle_in_iterator_t
```

Constructor. The iterator is initialized at the first particle in the vertex.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function &
        new_vertex_particles_in_const_iterator (v_obj) bind(C)
     import
     type(c_ptr), value :: v_obj
   end function new_vertex_particles_in_const_iterator
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_in_iterator_init
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_vertex_particle_in_iterator_init (it, v)
  type(hepmc_vertex_particle_in_iterator_t), intent(out) :: it
  type(hepmc_vertex_t), intent(in) :: v
  it%obj = new_vertex_particles_in_const_iterator (v%obj)
  it%v_obj = v%obj
end subroutine hepmc_vertex_particle_in_iterator_init
```

Destructor. Necessary because the iterator is allocated on the heap.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
```

```
    subroutine vertex_particles_in_const_iterator_delete (it_obj) bind(C)
      import
      type(c_ptr), value :: it_obj
    end subroutine vertex_particles_in_const_iterator_delete
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_in_iterator_final
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_vertex_particle_in_iterator_final (it)
  type(hepmc_vertex_particle_in_iterator_t), intent(inout) :: it
  call vertex_particles_in_const_iterator_delete (it%obj)
end subroutine hepmc_vertex_particle_in_iterator_final
```

Increment

⟨*HepMC interface: interfaces*⟩+≡

```
interface
    subroutine vertex_particles_in_const_iterator_advance (it_obj) bind(C)
      import
      type(c_ptr), value :: it_obj
    end subroutine vertex_particles_in_const_iterator_advance
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_in_iterator_advance
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_vertex_particle_in_iterator_advance (it)
  type(hepmc_vertex_particle_in_iterator_t), intent(inout) :: it
  call vertex_particles_in_const_iterator_advance (it%obj)
end subroutine hepmc_vertex_particle_in_iterator_advance
```

Reset to the beginning

⟨*HepMC interface: interfaces*⟩+≡

```
interface
    subroutine vertex_particles_in_const_iterator_reset &
         (it_obj, v_obj) bind(C)
      import
      type(c_ptr), value :: it_obj, v_obj
    end subroutine vertex_particles_in_const_iterator_reset
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_in_iterator_reset
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_vertex_particle_in_iterator_reset (it)
  type(hepmc_vertex_particle_in_iterator_t), intent(inout) :: it
  call vertex_particles_in_const_iterator_reset (it%obj, it%v_obj)
end subroutine hepmc_vertex_particle_in_iterator_reset
```

Test: return true as long as we are not past the end.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
```

```
    function vertex_particles_in_const_iterator_is_valid &
         (it_obj, v_obj) result (flag) bind(C)
      import
      logical(c_bool) :: flag
      type(c_ptr), value :: it_obj, v_obj
    end function vertex_particles_in_const_iterator_is_valid
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_in_iterator_is_valid
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_vertex_particle_in_iterator_is_valid (it) result (flag)
  logical :: flag
  type(hepmc_vertex_particle_in_iterator_t), intent(in) :: it
  flag = vertex_particles_in_const_iterator_is_valid (it%obj, it%v_obj)
end function hepmc_vertex_particle_in_iterator_is_valid
```

Return the particle pointed to by the iterator. (The particle object should not be finalized, since it contains merely a pointer to the particle which is owned by the vertex.)

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function &
        vertex_particles_in_const_iterator_get (it_obj) bind(C)
     import
     type(c_ptr), value :: it_obj
   end function vertex_particles_in_const_iterator_get
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_in_iterator_get
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_vertex_particle_in_iterator_get (it) result (prt)
  type(hepmc_particle_t) :: prt
  type(hepmc_vertex_particle_in_iterator_t), intent(in) :: it
  prt%obj = vertex_particles_in_const_iterator_get (it%obj)
end function hepmc_vertex_particle_in_iterator_get
```

17.10.7 Vertex-particle-out iterator

This iterator iterates over all incoming particles in an vertex. We store a pointer to the vertex in addition to the iterator. This allows for simple end checking.

The iterator is actually a constant iterator; it can only read.

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_out_iterator_t
```

⟨*HepMC interface: types*⟩+≡

```
type :: hepmc_vertex_particle_out_iterator_t
   private
   type(c_ptr) :: obj
   type(c_ptr) :: v_obj
end type hepmc_vertex_particle_out_iterator_t
```

Constructor. The iterator is initialized at the first particle in the vertex.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function &
        new_vertex_particles_out_const_iterator (v_obj) bind(C)
     import
     type(c_ptr), value :: v_obj
   end function new_vertex_particles_out_const_iterator
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_out_iterator_init
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_vertex_particle_out_iterator_init (it, v)
  type(hepmc_vertex_particle_out_iterator_t), intent(out) :: it
  type(hepmc_vertex_t), intent(in) :: v
  it%obj = new_vertex_particles_out_const_iterator (v%obj)
  it%v_obj = v%obj
end subroutine hepmc_vertex_particle_out_iterator_init
```

Destructor. Necessary because the iterator is allocated on the heap.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine vertex_particles_out_const_iterator_delete (it_obj) bind(C)
     import
     type(c_ptr), value :: it_obj
   end subroutine vertex_particles_out_const_iterator_delete
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_out_iterator_final
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_vertex_particle_out_iterator_final (it)
  type(hepmc_vertex_particle_out_iterator_t), intent(inout) :: it
  call vertex_particles_out_const_iterator_delete (it%obj)
end subroutine hepmc_vertex_particle_out_iterator_final
```

Increment

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine vertex_particles_out_const_iterator_advance (it_obj) bind(C)
     import
     type(c_ptr), value :: it_obj
   end subroutine vertex_particles_out_const_iterator_advance
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_out_iterator_advance
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_vertex_particle_out_iterator_advance (it)
  type(hepmc_vertex_particle_out_iterator_t), intent(inout) :: it
  call vertex_particles_out_const_iterator_advance (it%obj)
end subroutine hepmc_vertex_particle_out_iterator_advance
```

Reset to the beginning

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine vertex_particles_out_const_iterator_reset &
        (it_obj, v_obj) bind(C)
     import
     type(c_ptr), value :: it_obj, v_obj
   end subroutine vertex_particles_out_const_iterator_reset
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_out_iterator_reset
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_vertex_particle_out_iterator_reset (it)
  type(hepmc_vertex_particle_out_iterator_t), intent(inout) :: it
  call vertex_particles_out_const_iterator_reset (it%obj, it%v_obj)
end subroutine hepmc_vertex_particle_out_iterator_reset
```

Test: return true as long as we are not past the end.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   function vertex_particles_out_const_iterator_is_valid &
        (it_obj, v_obj) result (flag) bind(C)
     import
     logical(c_bool) :: flag
     type(c_ptr), value :: it_obj, v_obj
   end function vertex_particles_out_const_iterator_is_valid
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_out_iterator_is_valid
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_vertex_particle_out_iterator_is_valid (it) result (flag)
  logical :: flag
  type(hepmc_vertex_particle_out_iterator_t), intent(in) :: it
  flag = vertex_particles_out_const_iterator_is_valid (it%obj, it%v_obj)
end function hepmc_vertex_particle_out_iterator_is_valid
```

Return the particle pointed to by the iterator. (The particle object should not be finalized, since it contains merely a pointer to the particle which is owned by the vertex.)

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function &
        vertex_particles_out_const_iterator_get (it_obj) bind(C)
     import
     type(c_ptr), value :: it_obj
   end function vertex_particles_out_const_iterator_get
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_vertex_particle_out_iterator_get
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_vertex_particle_out_iterator_get (it) result (prt)
  type(hepmc_particle_t) :: prt
  type(hepmc_vertex_particle_out_iterator_t), intent(in) :: it
  prt%obj = vertex_particles_out_const_iterator_get (it%obj)
end function hepmc_vertex_particle_out_iterator_get
```

17.10.8 GenEvent

The main object of HepMC is a GenEvent. The object is filled by GenVertex objects, which in turn contain GenParticle objects.

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_t
```

⟨*HepMC interface: types*⟩+≡

```
type :: hepmc_event_t
   private
   type(c_ptr) :: obj
end type hepmc_event_t
```

Constructor. Arguments are process ID (integer) and event ID (integer).

The Fortran version has initializer form.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function new_gen_event (proc_id, event_id) bind(C)
     import
     integer(c_int), value :: proc_id, event_id
   end function new_gen_event
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_init
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_event_init (evt, proc_id, event_id)
  type(hepmc_event_t), intent(out) :: evt
  integer, intent(in), optional :: proc_id, event_id
  integer(c_int) :: pid, eid
  pid = 0;  if (present (proc_id))  pid = proc_id
  eid = 0;  if (present (event_id)) eid = event_id
  evt%obj = new_gen_event (pid, eid)
end subroutine hepmc_event_init
```

Destructor.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine gen_event_delete (evt_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
   end subroutine gen_event_delete
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_final
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_event_final (evt)
  type(hepmc_event_t), intent(inout) :: evt
  call gen_event_delete (evt%obj)
end subroutine hepmc_event_final
```

Screen output. Printing to file is possible in principle (using a C++ output channel), by allowing an argument. Printing to an open Fortran unit is obviously not possible.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   subroutine gen_event_print (evt_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
   end subroutine gen_event_print
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_print
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_event_print (evt)
  type(hepmc_event_t), intent(in) :: evt
  call gen_event_print (evt%obj)
end subroutine hepmc_event_print
```

Get the event number.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   integer(c_int) function gen_event_event_number (evt_obj) bind(C)
     use iso_c_binding !NODEP!
     type(c_ptr), value :: evt_obj
   end function gen_event_event_number
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_get_event_index
```

⟨*HepMC interface: procedures*⟩+≡
```
function hepmc_event_get_event_index (evt) result (i_proc)
  integer :: i_proc
  type(hepmc_event_t), intent(in) :: evt
  i_proc = gen_event_event_number (evt%obj)
end function hepmc_event_get_event_index
```

Set the numeric signal process ID

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   subroutine gen_event_set_signal_process_id (evt_obj, proc_id) bind(C)
     import
     type(c_ptr), value :: evt_obj
```

```
      integer(c_int), value :: proc_id
    end subroutine gen_event_set_signal_process_id
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_set_process_id
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_event_set_process_id (evt, proc)
  type(hepmc_event_t), intent(in) :: evt
  integer, intent(in) :: proc
  integer(c_int) :: i_proc
  i_proc = proc
  call gen_event_set_signal_process_id (evt%obj, i_proc)
end subroutine hepmc_event_set_process_id
```

Get the numeric signal process ID

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   integer(c_int) function gen_event_signal_process_id (evt_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
   end function gen_event_signal_process_id
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_get_process_id
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_event_get_process_id (evt) result (i_proc)
  integer :: i_proc
  type(hepmc_event_t), intent(in) :: evt
  i_proc = gen_event_signal_process_id (evt%obj)
end function hepmc_event_get_process_id
```

Set the event energy scale

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine gen_event_set_event_scale (evt_obj, scale) bind(C)
     import
     type(c_ptr), value :: evt_obj
     real(c_double), value :: scale
   end subroutine gen_event_set_event_scale
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_set_scale
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_event_set_scale (evt, scale)
  type(hepmc_event_t), intent(in) :: evt
  real(default), intent(in) :: scale
  real(c_double) :: cscale
  cscale = scale
  call gen_event_set_event_scale (evt%obj, cscale)
end subroutine hepmc_event_set_scale
```

Get the event energy scale

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   real(c_double) function gen_event_event_scale (evt_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
   end function gen_event_event_scale
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_get_scale
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_event_get_scale (evt) result (scale)
  real(default) :: scale
  type(hepmc_event_t), intent(in) :: evt
  scale = gen_event_event_scale (evt%obj)
end function hepmc_event_get_scale
```

Set the value of α_{QCD}.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine gen_event_set_alpha_qcd (evt_obj, a) bind(C)
     import
     type(c_ptr), value :: evt_obj
     real(c_double), value :: a
   end subroutine gen_event_set_alpha_qcd
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_set_alpha_qcd
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_event_set_alpha_qcd (evt, alpha)
  type(hepmc_event_t), intent(in) :: evt
  real(default), intent(in) :: alpha
  real(c_double) :: a
  a = alpha
  call gen_event_set_alpha_qcd (evt%obj, a)
end subroutine hepmc_event_set_alpha_qcd
```

Get the value of α_{QCD}.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   real(c_double) function gen_event_alpha_qcd (evt_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
   end function gen_event_alpha_qcd
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_get_alpha_qcd
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_event_get_alpha_qcd (evt) result (alpha)
  real(default) :: alpha
  type(hepmc_event_t), intent(in) :: evt
  alpha = gen_event_alpha_qcd (evt%obj)
end function hepmc_event_get_alpha_qcd
```

Set the value of α_{QED}.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine gen_event_set_alpha_qed (evt_obj, a) bind(C)
     import
     type(c_ptr), value :: evt_obj
     real(c_double), value :: a
   end subroutine gen_event_set_alpha_qed
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_set_alpha_qed
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_event_set_alpha_qed (evt, alpha)
  type(hepmc_event_t), intent(in) :: evt
  real(default), intent(in) :: alpha
  real(c_double) :: a
  a = alpha
  call gen_event_set_alpha_qed (evt%obj, a)
end subroutine hepmc_event_set_alpha_qed
```

Get the value of α_{QED}.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   real(c_double) function gen_event_alpha_qed (evt_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
   end function gen_event_alpha_qed
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_get_alpha_qed
```

⟨*HepMC interface: procedures*⟩+≡

```
function hepmc_event_get_alpha_qed (evt) result (alpha)
  real(default) :: alpha
  type(hepmc_event_t), intent(in) :: evt
  alpha = gen_event_alpha_qed (evt%obj)
end function hepmc_event_get_alpha_qed
```

Clear a weight value to the end of the weight container.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine gen_event_clear_weights (evt_obj) bind(C)
     use iso_c_binding !NODEP!
     type(c_ptr), value :: evt_obj
```

```
      end subroutine gen_event_clear_weights
   end interface
```

The HepMC weights are measured in pb.

⟨*HepMC interface: parameters*⟩≡

```
   real(default), parameter :: pb_per_fb = 1.e-3_default
```

⟨*HepMC interface: public*⟩+≡

```
   public :: hepmc_event_clear_weights
```

⟨*HepMC interface: procedures*⟩+≡

```
   subroutine hepmc_event_clear_weights (evt)
     type(hepmc_event_t), intent(in) :: evt
     call gen_event_clear_weights (evt%obj)
   end subroutine hepmc_event_clear_weights
```

Add a weight value to the end of the weight container.

⟨*HepMC interface: interfaces*⟩+≡

```
   interface
      subroutine gen_event_add_weight (evt_obj, w) bind(C)
        use iso_c_binding !NODEP!
        type(c_ptr), value :: evt_obj
        real(c_double), value :: w
      end subroutine gen_event_add_weight
   end interface
```

⟨*HepMC interface: public*⟩+≡

```
   public :: hepmc_event_add_weight
```

⟨*HepMC interface: procedures*⟩+≡

```
   subroutine hepmc_event_add_weight (evt, weight)
     type(hepmc_event_t), intent(in) :: evt
     real(default), intent(in) :: weight
     real(c_double) :: w
     w = weight * pb_per_fb
     call gen_event_add_weight (evt%obj, w)
   end subroutine hepmc_event_add_weight
```

Get the size of the weight container (the number of valid elements).

⟨*HepMC interface: interfaces*⟩+≡

```
   interface
      integer(c_int) function gen_event_weights_size (evt_obj) bind(C)
        use iso_c_binding !NODEP!
        type(c_ptr), value :: evt_obj
      end function gen_event_weights_size
   end interface
```

⟨*HepMC interface: public*⟩+≡

```
   public :: hepmc_event_get_weights_size
```

⟨*HepMC interface: procedures*⟩+≡

```
   function hepmc_event_get_weights_size (evt) result (n)
     integer :: n
     type(hepmc_event_t), intent(in) :: evt
     n = gen_event_weights_size (evt%obj)
   end function hepmc_event_get_weights_size
```

Get the value of the weight with index `i`. (Count from 1, while C counts from zero.)

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   real(c_double) function gen_event_weight (evt_obj, i) bind(C)
     use iso_c_binding !NODEP!
     type(c_ptr), value :: evt_obj
     integer(c_int), value :: i
   end function gen_event_weight
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_get_weight
```

⟨*HepMC interface: procedures*⟩+≡
```
function hepmc_event_get_weight (evt, index) result (weight)
  real(default) :: weight
  type(hepmc_event_t), intent(in) :: evt
  integer, intent(in) :: index
  integer(c_int) :: i
  i = index - 1
  weight = gen_event_weight (evt%obj, i) / pb_per_fb
end function hepmc_event_get_weight
```

Add a vertex to the event container.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   subroutine gen_event_add_vertex (evt_obj, v_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
     type(c_ptr), value :: v_obj
   end subroutine gen_event_add_vertex
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_add_vertex
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_event_add_vertex (evt, v)
  type(hepmc_event_t), intent(inout) :: evt
  type(hepmc_vertex_t), intent(in) :: v
  call gen_event_add_vertex (evt%obj, v%obj)
end subroutine hepmc_event_add_vertex
```

Mark a particular vertex as the signal process (hard interaction).

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   subroutine gen_event_set_signal_process_vertex (evt_obj, v_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
     type(c_ptr), value :: v_obj
   end subroutine gen_event_set_signal_process_vertex
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_set_signal_process_vertex
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_event_set_signal_process_vertex (evt, v)
  type(hepmc_event_t), intent(inout) :: evt
  type(hepmc_vertex_t), intent(in) :: v
  call gen_event_set_signal_process_vertex (evt%obj, v%obj)
end subroutine hepmc_event_set_signal_process_vertex
```

Return the the signal process (hard interaction).

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   function gen_event_get_signal_process_vertex (evt_obj) &
        result (v_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
     type(c_ptr) :: v_obj
   end function gen_event_get_signal_process_vertex
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_get_signal_process_vertex
```

⟨*HepMC interface: procedures*⟩+≡
```
function hepmc_event_get_signal_process_vertex (evt) result (v)
  type(hepmc_event_t), intent(in) :: evt
  type(hepmc_vertex_t) :: v
  v%obj = gen_event_get_signal_process_vertex (evt%obj)
end function hepmc_event_get_signal_process_vertex
```

Set the beam particles explicitly.

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_set_beam_particles
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_event_set_beam_particles (evt, prt1, prt2)
  type(hepmc_event_t), intent(inout) :: evt
  type(hepmc_particle_t), intent(in) :: prt1, prt2
  logical(c_bool) :: flag
  flag = gen_event_set_beam_particles (evt%obj, prt1%obj, prt2%obj)
end subroutine hepmc_event_set_beam_particles
```

The C function returns a boolean which we do not use.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   logical(c_bool) function gen_event_set_beam_particles &
        (evt_obj, prt1_obj, prt2_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj, prt1_obj, prt2_obj
   end function gen_event_set_beam_particles
end interface
```

Set the cross section and error explicitly. Note that HepMC uses pb, while WHIZARD uses fb.

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_set_cross_section
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_event_set_cross_section (evt, xsec, xsec_err)
  type(hepmc_event_t), intent(inout) :: evt
  real(default), intent(in) :: xsec, xsec_err
  call gen_event_set_cross_section &
       (evt%obj, &
       real (xsec * 1e-3_default, c_double), &
       real (xsec_err * 1e-3_default, c_double))
end subroutine hepmc_event_set_cross_section
```

The C function returns a boolean which we do not use.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine gen_event_set_cross_section (evt_obj, xs, xs_err) bind(C)
     import
     type(c_ptr), value :: evt_obj
     real(c_double), value :: xs, xs_err
   end subroutine gen_event_set_cross_section
end interface
```

17.10.9 Event-particle iterator

This iterator iterates over all particles in an event. We store a pointer to the event in addition to the iterator. This allows for simple end checking.

The iterator is actually a constant iterator; it can only read.

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_particle_iterator_t
```

⟨*HepMC interface: types*⟩+≡

```
type :: hepmc_event_particle_iterator_t
   private
   type(c_ptr) :: obj
   type(c_ptr) :: evt_obj
end type hepmc_event_particle_iterator_t
```

Constructor. The iterator is initialized at the first particle in the event.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function new_event_particle_const_iterator (evt_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
   end function new_event_particle_const_iterator
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_particle_iterator_init
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_event_particle_iterator_init (it, evt)
  type(hepmc_event_particle_iterator_t), intent(out) :: it
  type(hepmc_event_t), intent(in) :: evt
  it%obj = new_event_particle_const_iterator (evt%obj)
```

```
  it%evt_obj = evt%obj
end subroutine hepmc_event_particle_iterator_init
```

Destructor. Necessary because the iterator is allocated on the heap.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine event_particle_const_iterator_delete (it_obj) bind(C)
     import
     type(c_ptr), value :: it_obj
   end subroutine event_particle_const_iterator_delete
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_particle_iterator_final
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_event_particle_iterator_final (it)
  type(hepmc_event_particle_iterator_t), intent(inout) :: it
  call event_particle_const_iterator_delete (it%obj)
end subroutine hepmc_event_particle_iterator_final
```

Increment

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine event_particle_const_iterator_advance (it_obj) bind(C)
     import
     type(c_ptr), value :: it_obj
   end subroutine event_particle_const_iterator_advance
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_particle_iterator_advance
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_event_particle_iterator_advance (it)
  type(hepmc_event_particle_iterator_t), intent(inout) :: it
  call event_particle_const_iterator_advance (it%obj)
end subroutine hepmc_event_particle_iterator_advance
```

Reset to the beginning

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine event_particle_const_iterator_reset (it_obj, evt_obj) bind(C)
     import
     type(c_ptr), value :: it_obj, evt_obj
   end subroutine event_particle_const_iterator_reset
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_event_particle_iterator_reset
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_event_particle_iterator_reset (it)
  type(hepmc_event_particle_iterator_t), intent(inout) :: it
  call event_particle_const_iterator_reset (it%obj, it%evt_obj)
end subroutine hepmc_event_particle_iterator_reset
```

Test: return true as long as we are not past the end.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   function event_particle_const_iterator_is_valid &
        (it_obj, evt_obj) result (flag) bind(C)
     import
     logical(c_bool) :: flag
     type(c_ptr), value :: it_obj, evt_obj
   end function event_particle_const_iterator_is_valid
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_particle_iterator_is_valid
```

⟨*HepMC interface: procedures*⟩+≡
```
function hepmc_event_particle_iterator_is_valid (it) result (flag)
  logical :: flag
  type(hepmc_event_particle_iterator_t), intent(in) :: it
  flag = event_particle_const_iterator_is_valid (it%obj, it%evt_obj)
end function hepmc_event_particle_iterator_is_valid
```

Return the particle pointed to by the iterator. (The particle object should not be finalized, since it contains merely a pointer to the particle which is owned by the vertex.)

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   type(c_ptr) function event_particle_const_iterator_get (it_obj) bind(C)
     import
     type(c_ptr), value :: it_obj
   end function event_particle_const_iterator_get
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_event_particle_iterator_get
```

⟨*HepMC interface: procedures*⟩+≡
```
function hepmc_event_particle_iterator_get (it) result (prt)
  type(hepmc_particle_t) :: prt
  type(hepmc_event_particle_iterator_t), intent(in) :: it
  prt%obj = event_particle_const_iterator_get (it%obj)
end function hepmc_event_particle_iterator_get
```

17.10.10 I/O streams

There is a specific I/O stream type for handling the output of GenEvent objects (i.e., Monte Carlo event samples) to file. Opening the file is done by the constructor, closing by the destructor.

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_iostream_t
```

⟨*HepMC interface: types*⟩+≡
```
type :: hepmc_iostream_t
   private
   type(c_ptr) :: obj
```

```
end type hepmc_iostream_t
```

Constructor for an output stream associated to a file.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function new_io_gen_event_out (filename) bind(C)
     import
     character(c_char), dimension(*), intent(in) :: filename
   end function new_io_gen_event_out
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_iostream_open_out
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_iostream_open_out (iostream, filename)
  type(hepmc_iostream_t), intent(out) :: iostream
  type(string_t), intent(in) :: filename
  iostream%obj = new_io_gen_event_out (char (filename) // c_null_char)
end subroutine hepmc_iostream_open_out
```

Constructor for an input stream associated to a file.

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function new_io_gen_event_in (filename) bind(C)
     import
     character(c_char), dimension(*), intent(in) :: filename
   end function new_io_gen_event_in
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_iostream_open_in
```

⟨*HepMC interface: procedures*⟩+≡

```
subroutine hepmc_iostream_open_in (iostream, filename)
  type(hepmc_iostream_t), intent(out) :: iostream
  type(string_t), intent(in) :: filename
  iostream%obj = new_io_gen_event_in (char (filename) // c_null_char)
end subroutine hepmc_iostream_open_in
```

Destructor:

⟨*HepMC interface: interfaces*⟩+≡

```
interface
   subroutine io_gen_event_delete (io_obj) bind(C)
     import
     type(c_ptr), value :: io_obj
   end subroutine io_gen_event_delete
end interface
```

⟨*HepMC interface: public*⟩+≡

```
public :: hepmc_iostream_close
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_iostream_close (iostream)
  type(hepmc_iostream_t), intent(inout) :: iostream
  call io_gen_event_delete (iostream%obj)
end subroutine hepmc_iostream_close
```

Write a single event to the I/O stream.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   subroutine io_gen_event_write_event (io_obj, evt_obj) bind(C)
     import
     type(c_ptr), value :: io_obj, evt_obj
   end subroutine io_gen_event_write_event
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_iostream_write_event
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_iostream_write_event (iostream, evt)
  type(hepmc_iostream_t), intent(inout) :: iostream
  type(hepmc_event_t), intent(in) :: evt
  call io_gen_event_write_event (iostream%obj, evt%obj)
end subroutine hepmc_iostream_write_event
```

Read a single event from the I/O stream. Return true if successful.

⟨*HepMC interface: interfaces*⟩+≡
```
interface
   logical(c_bool) function io_gen_event_read_event (io_obj, evt_obj) bind(C)
     import
     type(c_ptr), value :: io_obj, evt_obj
   end function io_gen_event_read_event
end interface
```

⟨*HepMC interface: public*⟩+≡
```
public :: hepmc_iostream_read_event
```

⟨*HepMC interface: procedures*⟩+≡
```
subroutine hepmc_iostream_read_event (iostream, evt, ok)
  type(hepmc_iostream_t), intent(inout) :: iostream
  type(hepmc_event_t), intent(inout) :: evt
  logical, intent(out) :: ok
  ok = io_gen_event_read_event (iostream%obj, evt%obj)
end subroutine hepmc_iostream_read_event
```

17.10.11 Unit tests

Test module, followed by the corresponding implementation module.

⟨`hepmc_interface_ut.f90`⟩≡
```
⟨File header⟩

module hepmc_interface_ut
  use unit_tests
```

```
  use hepmc_interface_uti

⟨Standard module head⟩

⟨HepMC interface: public test⟩

contains

⟨HepMC interface: test driver⟩

end module hepmc_interface_ut
```

⟨hepmc_interface_uti.f90⟩≡

```
⟨File header⟩

module hepmc_interface_uti

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use lorentz
  use flavors
  use colors
  use polarizations

  use hepmc_interface

⟨Standard module head⟩

⟨HepMC interface: test declarations⟩

contains

⟨HepMC interface: tests⟩

end module hepmc_interface_uti
```

API: driver for the unit tests below.

⟨*HepMC interface: public test*⟩≡

```
  public :: hepmc_interface_test
```

⟨*HepMC interface: test driver*⟩≡

```
  subroutine hepmc_interface_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨HepMC interface: execute tests⟩
  end subroutine hepmc_interface_test
```

This test example is an abridged version from the build-from-scratch example in the HepMC distribution. We create two vertices for $p \to q$ PDF splitting, then a vertex for a $qq \to W^- g$ hard-interaction process, and finally a vertex for $W^- \to qq$ decay. The setup is for LHC kinematics.

Extending the original example, we set color flow for the incoming quarks and polarization for the outgoing photon. For the latter, we have to define a

particle-data object for the photon, so a flavor object can be correctly initialized.

⟨*HepMC interface: execute tests*⟩≡

```
call test (hepmc_interface_1, "hepmc_interface_1", &
     "check HepMC interface", &
     u, results)
```

⟨*HepMC interface: test declarations*⟩≡

```
public :: hepmc_interface_1
```

⟨*HepMC interface: tests*⟩≡

```
subroutine hepmc_interface_1 (u)
  use physics_defs, only: VECTOR
  use model_data, only: field_data_t
  integer, intent(in) :: u
  integer :: u_file, iostat
  type(hepmc_event_t) :: evt
  type(hepmc_vertex_t) :: v1, v2, v3, v4
  type(hepmc_particle_t) :: prt1, prt2, prt3, prt4, prt5, prt6, prt7, prt8
  type(hepmc_iostream_t) :: iostream
  type(flavor_t) :: flv
  type(color_t) :: col
  type(polarization_t) :: pol
  type(field_data_t), target :: photon_data
  character(80) :: buffer

  write (u, "(A)")  "* Test output: HepMC interface"
  write (u, "(A)")  "*   Purpose: test HepMC interface"
  write (u, "(A)")

  write (u, "(A)")  "* Initialization"
  write (u, "(A)")

  ! Initialize a photon flavor object and some polarization
  call photon_data%init (var_str ("PHOTON"), 22)
  call photon_data%set (spin_type=VECTOR)
  call photon_data%freeze ()
  call flv%init (photon_data)
  call pol%init_angles &
       (flv, 0.6_default, 1._default, 0.5_default)

  ! Event initialization
  call hepmc_event_init (evt, 20, 1)

  write (u, "(A)")  "* p -> q splitting"
  write (u, "(A)")

  ! $p\to q$ splittings
  call hepmc_vertex_init (v1)
  call hepmc_event_add_vertex (evt, v1)
  call hepmc_vertex_init (v2)
  call hepmc_event_add_vertex (evt, v2)
  call particle_init (prt1, &
       0._default, 0._default, 7000._default, 7000._default, &
       2212, 3)
  call hepmc_vertex_add_particle_in (v1, prt1)
```

```
call particle_init (prt2, &
     0._default, 0._default,-7000._default, 7000._default, &
     2212, 3)
call hepmc_vertex_add_particle_in (v2, prt2)
call particle_init (prt3, &
     .750_default, -1.569_default, 32.191_default, 32.238_default, &
     1, 3)
call color_init_from_array (col, [501])
call hepmc_particle_set_color (prt3, col)
call hepmc_vertex_add_particle_out (v1, prt3)
call particle_init (prt4, &
     -3.047_default, -19._default, -54.629_default, 57.920_default, &
     -2, 3)
call color_init_from_array (col, [-501])
call hepmc_particle_set_color (prt4, col)
call hepmc_vertex_add_particle_out (v2, prt4)

write (u, "(A)")  "* Hard interaction"
write (u, "(A)")

! Hard interaction
call hepmc_vertex_init (v3)
call hepmc_event_add_vertex (evt, v3)
call hepmc_vertex_add_particle_in (v3, prt3)
call hepmc_vertex_add_particle_in (v3, prt4)
call particle_init (prt6, &
     -3.813_default, 0.113_default, -1.833_default, 4.233_default, &
     22, 1)
call hepmc_particle_set_polarization (prt6, pol)
call hepmc_vertex_add_particle_out (v3, prt6)
call particle_init (prt5, &
     1.517_default, -20.68_default, -20.605_default, 85.925_default, &
     -24, 3)
call hepmc_vertex_add_particle_out (v3, prt5)
call hepmc_event_set_signal_process_vertex (evt, v3)

! $W^-$ decay
call vertex_init_pos (v4, &
     0.12_default, -0.3_default, 0.05_default, 0.004_default)
call hepmc_event_add_vertex (evt, v4)
call hepmc_vertex_add_particle_in (v4, prt5)
call particle_init (prt7, &
     -2.445_default, 28.816_default, 6.082_default, 29.552_default, &
     1, 1)
call hepmc_vertex_add_particle_out (v4, prt7)
call particle_init (prt8, &
     3.962_default, -49.498_default, -26.687_default, 56.373_default, &
     -2, 1)
call hepmc_vertex_add_particle_out (v4, prt8)

! Event output
call hepmc_event_print (evt)
write (u, "(A)")  "Writing to file 'hepmc_test.hepmc'"
write (u, "(A)")
```

```
  call hepmc_iostream_open_out (iostream , var_str ("hepmc_test.hepmc"))
  call hepmc_iostream_write_event (iostream, evt)
  call hepmc_iostream_close (iostream)

  write (u, "(A)")  "Writing completed"

  write (u, "(A)")
  write (u, "(A)")  "* File contents:"
  write (u, "(A)")

  u_file = free_unit ()
  open (u_file, file = "hepmc_test.hepmc", &
       action = "read", status = "old")
  do
     read (u_file, "(A)", iostat = iostat)  buffer
     if (buffer(1:14) == "HepMC::Version")  buffer = "[...]"
     if (iostat /= 0)  exit
     write (u, "(A)") trim (buffer)
  end do
  close (u_file)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"
  write (u, "(A)")

  ! Wrapup
  ! call pol%final ()
  call hepmc_event_final (evt)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: hepmc_interface_1"

contains

  subroutine vertex_init_pos (v, x, y, z, t)
    type(hepmc_vertex_t), intent(out) :: v
    real(default), intent(in) :: x, y, z, t
    type(vector4_t) :: xx
    xx = vector4_moving (t, vector3_moving ([x, y, z]))
    call hepmc_vertex_init (v, xx)
  end subroutine vertex_init_pos

  subroutine particle_init (prt, px, py, pz, E, pdg, status)
    type(hepmc_particle_t), intent(out) :: prt
    real(default), intent(in) :: px, py, pz, E
    integer, intent(in) :: pdg, status
    type(vector4_t) :: p
    p = vector4_moving (E, vector3_moving ([px, py, pz]))
    call hepmc_particle_init (prt, p, pdg, status)
  end subroutine particle_init

end subroutine hepmc_interface_1
```

17.11 LCIO events

This section provides the interface to the LCIO C++ library for handling Monte-Carlo events.

Each C++ class of LCIO that we use is mirrored by a Fortran type, which contains as its only component the C pointer to the C++ object.

Each C++ method of LCIO that we use has a C wrapper function. This function takes a pointer to the host object as its first argument. Further arguments are either C pointers, or in the case of simple types (integer, real), interoperable C/Fortran objects.

The C wrapper functions have explicit interfaces in the Fortran module. They are called by Fortran wrapper procedures. These are treated as methods of the corresponding Fortran type.

⟨`lcio_interface.f90`⟩≡

```
⟨File header⟩

module lcio_interface

  use, intrinsic :: iso_c_binding !NODEP!

⟨Use kinds⟩
⟨Use strings⟩
  use constants, only: PI
  use diagnostics
  use lorentz
  use flavors
  use colors
  use helicities
  use polarizations

⟨Standard module head⟩

⟨LCIO interface: public⟩

⟨LCIO interface: types⟩

⟨LCIO interface: interfaces⟩

contains

⟨LCIO interface: procedures⟩

end module lcio_interface
```

17.11.1 Interface check

This function can be called in order to verify that we are using the actual LCIO library, and not the dummy version.

⟨*LCIO interface: interfaces*⟩≡

```
interface
   logical(c_bool) function lcio_available () bind(C)
     import
```

```
      end function lcio_available
   end interface
```

⟨*LCIO interface: public*⟩≡

```
   public :: lcio_is_available
```

⟨*LCIO interface: procedures*⟩≡

```
   function lcio_is_available () result (flag)
     logical :: flag
     flag = lcio_available ()
   end function lcio_is_available
```

17.11.2 LCIO Run Header

This is a type for the run header of the LCIO file.

⟨*LCIO interface: public*⟩+≡

```
   public :: lcio_run_header_t
```

⟨*LCIO interface: types*⟩≡

```
   type :: lcio_run_header_t
      private
      type(c_ptr) :: obj
   end type lcio_run_header_t
```

The Fortran version has initializer form.

⟨*LCIO interface: interfaces*⟩+≡

```
   interface
      type(c_ptr) function new_lcio_run_header (proc_id) bind(C)
        import
        integer(c_int), value :: proc_id
      end function new_lcio_run_header
   end interface
```

⟨*LCIO interface: interfaces*⟩+≡

```
   interface
      subroutine run_header_set_simstring &
           (runhdr_obj, simstring) bind(C)
        import
        type(c_ptr), value :: runhdr_obj
        character(c_char), dimension(*), intent(in) :: simstring
      end subroutine run_header_set_simstring
   end interface
```

⟨*LCIO interface: public*⟩+≡

```
   public :: lcio_run_header_init
```

⟨*LCIO interface: procedures*⟩+≡

```
   subroutine lcio_run_header_init (runhdr, proc_id, run_id)
     type(lcio_run_header_t), intent(out) :: runhdr
     integer, intent(in), optional :: proc_id, run_id
     integer(c_int) :: rid
     rid = 0; if (present (run_id))  rid = run_id
     runhdr%obj = new_lcio_run_header (rid)
     call run_header_set_simstring (runhdr%obj, &
          "WHIZARD version:" // "⟨Version⟩")
```

```
end subroutine lcio_run_header_init
```

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   subroutine write_run_header (lcwrt_obj, runhdr_obj) bind(C)
     import
     type(c_ptr), value :: lcwrt_obj
     type(c_ptr), value :: runhdr_obj
   end subroutine write_run_header
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_run_header_write
```

⟨*LCIO interface: procedures*⟩+≡

```
subroutine lcio_run_header_write (wrt, hdr)
  type(lcio_writer_t), intent(inout) :: wrt
  type(lcio_run_header_t), intent(inout) :: hdr
  call write_run_header (wrt%obj, hdr%obj)
end subroutine lcio_run_header_write
```

17.11.3 LCIO Event and LC Collection

The main object of LCIO is a LCEventImpl. The object is filled by MCParticle objects, which are set as LCCollection.

⟨*LCIO interface: public*⟩+≡

```
public :: lccollection_t
```

⟨*LCIO interface: types*⟩+≡

```
type :: lccollection_t
   private
   type(c_ptr) :: obj
end type lccollection_t
```

Initializer.

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function new_lccollection () bind(C)
     import
   end function new_lccollection
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_event_t
```

⟨*LCIO interface: types*⟩+≡

```
type :: lcio_event_t
   private
   type(c_ptr) :: obj
   type(lccollection_t) :: lccoll
end type lcio_event_t
```

Constructor. Arguments are process ID (integer) and event ID (integer).

The Fortran version has initializer form.

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function new_lcio_event (proc_id, event_id, run_id) bind(C)
     import
     integer(c_int), value :: proc_id, event_id, run_id
   end function new_lcio_event
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_event_init
```

⟨*LCIO interface: procedures*⟩+≡

```
subroutine lcio_event_init (evt, proc_id, event_id, run_id)
  type(lcio_event_t), intent(out) :: evt
  integer, intent(in), optional :: proc_id, event_id, run_id
  integer(c_int) :: pid, eid, rid
  pid = 0;  if (present (proc_id))  pid = proc_id
  eid = 0;  if (present (event_id)) eid = event_id
  rid = 0;  if (present (run_id))   rid = run_id
  evt%obj = new_lcio_event (pid, eid, rid)
  evt%lccoll%obj = new_lccollection ()
end subroutine lcio_event_init
```

Destructor.

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   subroutine lcio_event_delete (evt_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
   end subroutine lcio_event_delete
end interface
```

Show event on screen.

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   subroutine dump_lcio_event (evt_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
   end subroutine dump_lcio_event
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: show_lcio_event
```

⟨*LCIO interface: procedures*⟩+≡

```
subroutine show_lcio_event (evt)
  type(lcio_event_t), intent(in) :: evt
  if (c_associated (evt%obj)) then
     call dump_lcio_event (evt%obj)
  else
     call msg_error ("LCIO event is not allocated.")
  end if
end subroutine show_lcio_event
```

Put a single event to file.

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   subroutine lcio_event_to_file (evt_obj, filename) bind(C)
     import
     type(c_ptr), value :: evt_obj
     character(c_char), dimension(*), intent(in) :: filename
   end subroutine lcio_event_to_file
end interface
```

⟨*LCIO interface: public*⟩+≡
```
public :: write_lcio_event
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine write_lcio_event (evt, filename)
  type(lcio_event_t), intent(in) :: evt
  type(string_t), intent(in) :: filename
  call lcio_event_to_file (evt%obj, char (filename) // c_null_char)
end subroutine write_lcio_event
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_event_final
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_event_final (evt)
  type(lcio_event_t), intent(inout) :: evt
  call lcio_event_delete (evt%obj)
end subroutine lcio_event_final
```

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   subroutine lcio_set_alpha_qcd (evt_obj, alphas) bind(C)
     import
     type(c_ptr), value :: evt_obj
     real(c_double), value :: alphas
   end subroutine lcio_set_alpha_qcd
end interface
interface
   subroutine lcio_set_scale (evt_obj, scale) bind(C)
     import
     type(c_ptr), value :: evt_obj
     real(c_double), value :: scale
   end subroutine lcio_set_scale
end interface
interface
   subroutine lcio_set_sqrts (evt_obj, sqrts) bind(C)
     import
     type(c_ptr), value :: evt_obj
     real(c_double), value :: sqrts
   end subroutine lcio_set_sqrts
end interface
interface
   subroutine lcio_set_xsec (evt_obj, xsec, xsec_err) bind(C)
     import
```

```
      type(c_ptr), value :: evt_obj
      real(c_double), value :: xsec, xsec_err
   end subroutine lcio_set_xsec
end interface
interface
   subroutine lcio_set_beam (evt_obj, pdg, beam) bind(C)
      import
      type(c_ptr), value :: evt_obj
      integer(c_int), value :: pdg, beam
   end subroutine lcio_set_beam
end interface
interface
   subroutine lcio_set_pol (evt_obj, pol1, pol2) bind(C)
      import
      type(c_ptr), value :: evt_obj
      real(c_double), value :: pol1, pol2
   end subroutine lcio_set_pol
end interface
interface
   subroutine lcio_set_beam_file (evt_obj, file) bind(C)
      import
      type(c_ptr), value :: evt_obj
      character(len=1, kind=c_char), dimension(*), intent(in) :: file
   end subroutine lcio_set_beam_file
end interface
interface
   subroutine lcio_set_process_name (evt_obj, name) bind(C)
      import
      type(c_ptr), value :: evt_obj
      character(len=1, kind=c_char), dimension(*), intent(in) :: name
   end subroutine lcio_set_process_name
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_event_set_alpha_qcd
```

⟨*LCIO interface: procedures*⟩+≡

```
subroutine lcio_event_set_alpha_qcd (evt, alphas)
  type(lcio_event_t), intent(inout) :: evt
  real(default), intent(in) :: alphas
  call lcio_set_alpha_qcd (evt%obj, real (alphas, c_double))
end subroutine lcio_event_set_alpha_qcd
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_event_set_scale
```

⟨*LCIO interface: procedures*⟩+≡

```
subroutine lcio_event_set_scale (evt, scale)
  type(lcio_event_t), intent(inout) :: evt
  real(default), intent(in) :: scale
  call lcio_set_scale (evt%obj, real (scale, c_double))
end subroutine lcio_event_set_scale
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_event_set_sqrts
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_event_set_sqrts (evt, sqrts)
  type(lcio_event_t), intent(inout) :: evt
  real(default), intent(in) :: sqrts
  call lcio_set_sqrts (evt%obj, real (sqrts, c_double))
end subroutine lcio_event_set_sqrts
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_event_set_xsec
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_event_set_xsec (evt, xsec, xsec_err)
  type(lcio_event_t), intent(inout) :: evt
  real(default), intent(in) :: xsec, xsec_err
  call lcio_set_xsec (evt%obj, &
       real (xsec, c_double), real (xsec_err, c_double))
end subroutine lcio_event_set_xsec
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_event_set_beam
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_event_set_beam (evt, pdg, beam)
  type(lcio_event_t), intent(inout) :: evt
  integer, intent(in) :: pdg, beam
  call lcio_set_beam (evt%obj, &
       int (pdg, c_int), int (beam, c_int))
end subroutine lcio_event_set_beam
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_event_set_polarization
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_event_set_polarization (evt, pol)
  type(lcio_event_t), intent(inout) :: evt
  real(default), intent(in), dimension(2) :: pol
  call lcio_set_pol (evt%obj, &
       real (pol(1), c_double), real (pol(2), c_double))
end subroutine lcio_event_set_polarization
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_event_set_beam_file
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_event_set_beam_file (evt, file)
  type(lcio_event_t), intent(inout) :: evt
  type(string_t), intent(in) :: file
  call lcio_set_beam_file (evt%obj, &
       char (file) // c_null_char)
end subroutine lcio_event_set_beam_file
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_event_set_process_name
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_event_set_process_name (evt, name)
  type(lcio_event_t), intent(inout) :: evt
  type(string_t), intent(in) :: name
  call lcio_set_process_name (evt%obj, &
       char (name) // c_null_char)
end subroutine lcio_event_set_process_name
```

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   subroutine lcio_event_add_collection &
        (evt_obj, lccoll_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj, lccoll_obj
   end subroutine lcio_event_add_collection
end interface
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_event_add_coll
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_event_add_coll (evt)
  type(lcio_event_t), intent(inout) :: evt
  call lcio_event_add_collection (evt%obj, &
       evt%lccoll%obj)
end subroutine lcio_event_add_coll
```

17.11.4 LCIO Particle

Particle objects have the obvious meaning.

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_particle_t
```

⟨*LCIO interface: types*⟩+≡
```
type :: lcio_particle_t
   private
   type(c_ptr) :: obj
end type lcio_particle_t
```

Constructor.

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   type(c_ptr) function new_lcio_particle &
        (px, py, pz, pdg_id, mass, charge, status) bind(C)
     import
     integer(c_int), value :: pdg_id, status
     real(c_double), value :: px, py, pz, mass, charge
   end function new_lcio_particle
end interface
```

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   subroutine add_particle_to_collection &
        (prt_obj, lccoll_obj) bind(C)
     import
     type(c_ptr), value :: prt_obj, lccoll_obj
   end subroutine add_particle_to_collection
end interface
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_particle_add_to_evt_coll
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_particle_add_to_evt_coll &
     (lprt, evt)
  type(lcio_particle_t), intent(in) :: lprt
  type(lcio_event_t), intent(inout) :: evt
  call add_particle_to_collection (lprt%obj, evt%lccoll%obj)
end subroutine lcio_particle_add_to_evt_coll
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_particle_init
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_particle_init (prt, p, pdg, charge, status)
  type(lcio_particle_t), intent(out) :: prt
  type(vector4_t), intent(in) :: p
  real(default), intent(in) :: charge
  real(default) :: mass
  real(default) :: px, py, pz
  integer, intent(in) :: pdg, status
  px = vector4_get_component (p, 1)
  py = vector4_get_component (p, 2)
  pz = vector4_get_component (p, 3)
  mass = p**1
  prt%obj = new_lcio_particle (real (px, c_double), real (py, c_double), &
       real (pz, c_double), int (pdg, c_int), &
       real (mass, c_double), real (charge, c_double), int (status, c_int))
end subroutine lcio_particle_init
```

Set the particle color flow.

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   subroutine lcio_set_color_flow (prt_obj, col1, col2) bind(C)
     import
     type(c_ptr), value :: prt_obj
     integer(c_int), value :: col1, col2
   end subroutine lcio_set_color_flow
end interface
```

Set the particle color. Either from a `color_t` object or directly from a pair of integers.

⟨*LCIO interface: interfaces*⟩+≡
```
interface lcio_particle_set_color
   module procedure lcio_particle_set_color_col
```

```
      module procedure lcio_particle_set_color_int
   end interface lcio_particle_set_color
```

⟨*LCIO interface: public*⟩+≡

```
   public :: lcio_particle_set_color
```

⟨*LCIO interface: procedures*⟩+≡

```
   subroutine lcio_particle_set_color_col (prt, col)
     type(lcio_particle_t), intent(inout) :: prt
     type(color_t), intent(in) :: col
     integer(c_int), dimension(2) :: c
     c(1) = col%get_col ()
     c(2) = col%get_acl ()
     if (c(1) /= 0 .or. c(2) /= 0)  then
        call lcio_set_color_flow (prt%obj, c(1), c(2))
     end  if
   end subroutine lcio_particle_set_color_col

   subroutine lcio_particle_set_color_int (prt, col)
     type(lcio_particle_t), intent(inout) :: prt
     integer, dimension(2), intent(in) :: col
     integer(c_int), dimension(2) :: c
     c = col
     if (c(1) /= 0 .or. c(2) /= 0) then
        call lcio_set_color_flow (prt%obj, c(1), c(2))
     end if
   end subroutine lcio_particle_set_color_int
```

Return the particle color as a two-dimensional array (color, anticolor).

⟨*LCIO interface: interfaces*⟩+≡

```
   interface
      integer(c_int) function lcio_particle_flow (prt_obj, col_index) bind(C)
        use iso_c_binding !NODEP!
        type(c_ptr), value :: prt_obj
        integer(c_int), value :: col_index
      end function lcio_particle_flow
   end interface
```

⟨*LCIO interface: public*⟩+≡

```
   public :: lcio_particle_get_flow
```

⟨*LCIO interface: procedures*⟩+≡

```
   function lcio_particle_get_flow (prt) result (col)
     integer, dimension(2) :: col
     type(lcio_particle_t), intent(in) :: prt
     col(1) = lcio_particle_flow (prt%obj, 0_c_int)
     col(2) = - lcio_particle_flow (prt%obj, 1_c_int)
   end function lcio_particle_get_flow
```

Return the four-momentum of a LCIO particle.

⟨*LCIO interface: interfaces*⟩+≡

```
   interface
      real(c_double) function lcio_three_momentum (prt_obj, p_index) bind(C)
        use iso_c_binding !NODEP!
        type(c_ptr), value :: prt_obj
```

```
      integer(c_int), value :: p_index
    end function lcio_three_momentum
end interface
```

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   real(c_double) function lcio_energy (prt_obj) bind(C)
     import
     type(c_ptr), intent(in), value :: prt_obj
   end function lcio_energy
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_particle_get_momentum
```

⟨*LCIO interface: procedures*⟩+≡

```
function lcio_particle_get_momentum (prt) result (p)
  type(vector4_t) :: p
  type(lcio_particle_t), intent(in) :: prt
  real(default) :: E, px, py, pz
  E = lcio_energy (prt%obj)
  px = lcio_three_momentum (prt%obj, 0_c_int)
  py = lcio_three_momentum (prt%obj, 1_c_int)
  pz = lcio_three_momentum (prt%obj, 2_c_int)
  p = vector4_moving ( E, vector3_moving ([ px, py, pz ]))
end function lcio_particle_get_momentum
```

Return the invariant mass squared of the particle object. LCIO stores the signed invariant mass (no squaring).

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   function lcio_mass (prt_obj) result (mass) bind(C)
     import
     real(c_double) :: mass
     type(c_ptr), value :: prt_obj
   end function lcio_mass
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_particle_get_mass_squared
```

⟨*LCIO interface: procedures*⟩+≡

```
function lcio_particle_get_mass_squared (prt) result (m2)
  real(default) :: m2
  type(lcio_particle_t), intent(in) :: prt
  real(default) :: m
  m = lcio_mass (prt%obj)
  m2 = sign (m**2, m)
end function lcio_particle_get_mass_squared
```

Return vertex and production time of a LCIO particle.

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   real(c_double) function lcio_vtx_x (prt) bind(C)
     import
```

```
      type(c_ptr), value :: prt
    end function lcio_vtx_x
end interface
interface
    real(c_double) function lcio_vtx_y (prt) bind(C)
      import
      type(c_ptr), value :: prt
    end function lcio_vtx_y
end interface
interface
    real(c_double) function lcio_vtx_z (prt) bind(C)
      import
      type(c_ptr), value :: prt
    end function lcio_vtx_z
end interface
interface
    real(c_double) function lcio_prt_time (prt) bind(C)
      import
      type(c_ptr), value :: prt
    end function lcio_prt_time
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_particle_get_vertex
public :: lcio_particle_get_time
```

⟨*LCIO interface: procedures*⟩+≡

```
function lcio_particle_get_vertex (prt) result (vtx)
  type(vector3_t) :: vtx
  type(lcio_particle_t), intent(in) :: prt
  real(default) :: vx, vy, vz
  vx = lcio_vtx_x (prt%obj)
  vy = lcio_vtx_y (prt%obj)
  vz = lcio_vtx_z (prt%obj)
  vtx = vector3_moving ([vx, vy, vz])
end function lcio_particle_get_vertex

function lcio_particle_get_time (prt) result (time)
  real(default) :: time
  type(lcio_particle_t), intent(in) :: prt
  time = lcio_prt_time (prt%obj)
end function lcio_particle_get_time
```

17.11.5 Polarization

For polarization there is a three-component float entry foreseen in the LCIO format. Completely generic density matrices can in principle be attached to events as float vectors added to `LCCollection` of the `LCEvent`. This is not yet implemented currently. Here, we restrict ourselves to the same implementation as in HepMC format: we use two entries as the polarization angles, while the first entry gives the degree of polarization (something not specified in the HepMC format). *For massive vector bosons, we arbitrarily choose the convention that*

the longitudinal (zero) helicity state is mapped to the theta angle $\pi/2$. This works under the condition that helicity is projected onto one of the basis states.

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   subroutine lcio_particle_set_spin (prt_obj, s1, s2, s3) bind(C)
     import
     type(c_ptr), value :: prt_obj
     real(c_double), value :: s1, s2, s3
   end subroutine lcio_particle_set_spin
end interface
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_polarization_init
```

⟨*LCIO interface: interfaces*⟩+≡
```
interface lcio_polarization_init
   module procedure lcio_polarization_init_pol
   module procedure lcio_polarization_init_hel
   module procedure lcio_polarization_init_int
end interface
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_polarization_init_pol (prt, pol)
  type(lcio_particle_t), intent(inout) :: prt
  type(polarization_t), intent(in) :: pol
  real(default) :: r, theta, phi
  if (pol%is_polarized ()) then
     call pol%to_angles (r, theta, phi)
     call lcio_particle_set_spin (prt%obj, &
          real(r, c_double), real (theta, c_double), real (phi, c_double))
  end if
end subroutine lcio_polarization_init_pol

subroutine lcio_polarization_init_hel (prt, hel)
  type(lcio_particle_t), intent(inout) :: prt
  type(helicity_t), intent(in) :: hel
  integer, dimension(2) :: h
  if (hel%is_defined ()) then
     h = hel%to_pair ()
     select case (h(1))
     case (1:)
        call lcio_particle_set_spin (prt%obj, 1._c_double, &
             0._c_double, 0._c_double)
     case (:-1)
        call lcio_particle_set_spin (prt%obj, 1._c_double, &
             real (pi, c_double), 0._c_double)
     case (0)
        call lcio_particle_set_spin (prt%obj, 1._c_double, &
             real (pi/2, c_double), 0._c_double)
     end select
  end if
end subroutine lcio_polarization_init_hel

subroutine lcio_polarization_init_int (prt, hel)
  type(lcio_particle_t), intent(inout) :: prt
```

```
    integer, intent(in) :: hel
    select case (hel)
    case (1:)
       call lcio_particle_set_spin (prt%obj, 1._c_double, &
            0._c_double, 0._c_double)
    case (:-1)
       call lcio_particle_set_spin (prt%obj, 1._c_double, &
            real (pi, c_double), 0._c_double)
    case (0)
       call lcio_particle_set_spin (prt%obj, 1._c_double, &
            real (pi/2, c_double), 0._c_double)
    end select
  end subroutine lcio_polarization_init_int
```

Recover polarization from LCIO particle (with the abovementioned deficiencies).

⟨*LCIO interface: interfaces*⟩+≡

```
  interface
     function lcio_polarization_degree (prt_obj) result (degree) bind(C)
       import
       real(c_double) :: degree
       type(c_ptr), value :: prt_obj
     end function lcio_polarization_degree
  end interface
  interface
     function lcio_polarization_theta (prt_obj) result (theta) bind(C)
       import
       real(c_double) :: theta
       type(c_ptr), value :: prt_obj
     end function lcio_polarization_theta
  end interface
  interface
     function lcio_polarization_phi (prt_obj) result (phi) bind(C)
       import
       real(c_double) :: phi
       type(c_ptr), value :: prt_obj
     end function lcio_polarization_phi
  end interface
```

⟨*LCIO interface: public*⟩+≡

```
  public :: lcio_particle_to_pol
```

⟨*LCIO interface: procedures*⟩+≡

```
  subroutine lcio_particle_to_pol (prt, flv, pol)
    type(lcio_particle_t), intent(in) :: prt
    type(flavor_t), intent(in) :: flv
    type(polarization_t), intent(out) :: pol
    real(default) :: degree, theta, phi
    degree = lcio_polarization_degree (prt%obj)
    theta = lcio_polarization_theta (prt%obj)
    phi = lcio_polarization_phi (prt%obj)
    call pol%init_angles (flv, degree, theta, phi)
  end subroutine lcio_particle_to_pol
```

Recover helicity. Here, ϕ and `degree` is ignored and only the sign of $\cos\theta$ is relevant, mapped to positive/negative helicity.

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_particle_to_hel
```
⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_particle_to_hel (prt, flv, hel)
  type(lcio_particle_t), intent(in) :: prt
  type(flavor_t), intent(in) :: flv
  type(helicity_t), intent(out) :: hel
  real(default) :: theta
  integer :: hmax
  theta = lcio_polarization_theta (prt%obj)
  hmax = flv%get_spin_type () / 2
  call hel%init (sign (hmax, nint (cos (theta))))
end subroutine lcio_particle_to_hel
```

Set the vertex of a particle.

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   subroutine lcio_particle_set_vertex (prt_obj, vx, vy, vz) bind(C)
     import
     type(c_ptr), value :: prt_obj
     real(c_double), value :: vx, vy, vz
   end subroutine lcio_particle_set_vertex
end interface
interface
   subroutine lcio_particle_set_time (prt_obj, t) bind(C)
     import
     type(c_ptr), value :: prt_obj
     real(c_double), value :: t
   end subroutine lcio_particle_set_time
end interface
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_particle_set_vtx
```
⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_particle_set_vtx (prt, vtx)
  type(lcio_particle_t), intent(inout) :: prt
  type(vector3_t), intent(in) :: vtx
  call lcio_particle_set_vertex (prt%obj, real(vtx%p(1), c_double), &
       real(vtx%p(2), c_double), real(vtx%p(3), c_double))
end subroutine lcio_particle_set_vtx
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_particle_set_t
```
⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_particle_set_t (prt, t)
  type(lcio_particle_t), intent(inout) :: prt
  real(default), intent(in) :: t
  call lcio_particle_set_time (prt%obj, real(t, c_double))
end subroutine lcio_particle_set_t
```

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   subroutine lcio_particle_add_parent (prt_obj1, prt_obj2) bind(C)
     import
     type(c_ptr), value :: prt_obj1, prt_obj2
   end subroutine lcio_particle_add_parent
end interface
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_particle_set_parent
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_particle_set_parent (daughter, parent)
  type(lcio_particle_t), intent(inout) :: daughter, parent
  call lcio_particle_add_parent (daughter%obj, parent%obj)
end subroutine lcio_particle_set_parent
```

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   integer(c_int) function lcio_particle_get_generator_status &
        (prt_obj) bind(C)
     import
     type(c_ptr), value :: prt_obj
   end function lcio_particle_get_generator_status
end interface
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_particle_get_status
```

⟨*LCIO interface: procedures*⟩+≡
```
function lcio_particle_get_status (lptr) result (status)
  integer :: status
  type(lcio_particle_t), intent(in) :: lptr
  status = lcio_particle_get_generator_status (lptr%obj)
end function lcio_particle_get_status
```

Getting the PDG code.

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   integer(c_int) function lcio_particle_get_pdg_code (prt_obj) bind(C)
     import
     type(c_ptr), value :: prt_obj
   end function lcio_particle_get_pdg_code
end interface
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_particle_get_pdg
```

⟨*LCIO interface: procedures*⟩+≡
```
function lcio_particle_get_pdg (lptr) result (pdg)
  integer :: pdg
  type(lcio_particle_t), intent(in) :: lptr
  pdg = lcio_particle_get_pdg_code (lptr%obj)
end function lcio_particle_get_pdg
```

Obtaining the number of parents and daughters of an LCIO particle.

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   integer(c_int) function lcio_n_parents (prt_obj) bind(C)
     import
     type(c_ptr), value :: prt_obj
   end function lcio_n_parents
end interface
```

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   integer(c_int) function lcio_n_daughters (prt_obj) bind(C)
     import
     type(c_ptr), value :: prt_obj
   end function lcio_n_daughters
end interface
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_particle_get_n_parents
```

⟨*LCIO interface: procedures*⟩+≡
```
function lcio_particle_get_n_parents (lptr) result (n_parents)
  integer :: n_parents
  type(lcio_particle_t), intent(in) :: lptr
  n_parents = lcio_n_parents (lptr%obj)
end function lcio_particle_get_n_parents
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_particle_get_n_children
```

⟨*LCIO interface: procedures*⟩+≡
```
function lcio_particle_get_n_children (lptr) result (n_children)
  integer :: n_children
  type(lcio_particle_t), intent(in) :: lptr
  n_children = lcio_n_daughters (lptr%obj)
end function lcio_particle_get_n_children
```

This provides access from the LCIO event `lcio_event_t` to the array entries of the parent and daughter arrays of the LCIO particles.

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   integer(c_int) function lcio_event_parent_k &
        (evt_obj, num_part, k_parent) bind (C)
     use iso_c_binding !NODEP!
     type(c_ptr), value :: evt_obj
     integer(c_int), value :: num_part, k_parent
   end function lcio_event_parent_k
end interface
```

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   integer(c_int) function lcio_event_daughter_k &
        (evt_obj, num_part, k_daughter) bind (C)
     use iso_c_binding !NODEP!
     type(c_ptr), value :: evt_obj
```

```
      integer(c_int), value :: num_part, k_daughter
    end function lcio_event_daughter_k
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_get_n_parents
```

⟨*LCIO interface: procedures*⟩+≡

```
function lcio_get_n_parents (evt, num_part, k_parent) result (index_parent)
  type(lcio_event_t), intent(in) :: evt
  integer, intent(in) :: num_part, k_parent
  integer :: index_parent
  index_parent = lcio_event_parent_k (evt%obj, int (num_part, c_int), &
       int (k_parent, c_int))
end function lcio_get_n_parents
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_get_n_children
```

⟨*LCIO interface: procedures*⟩+≡

```
function lcio_get_n_children (evt, num_part, k_daughter) result (index_daughter)
  type(lcio_event_t), intent(in) :: evt
  integer, intent(in) :: num_part, k_daughter
  integer :: index_daughter
  index_daughter = lcio_event_daughter_k (evt%obj, int (num_part, c_int), &
       int (k_daughter, c_int))
end function lcio_get_n_children
```

17.11.6 LCIO Writer type

There is a specific LCIO Writer type for handling the output of LCEventImpl objects (i.e., Monte Carlo event samples) to file. Opening the file is done by the constructor, closing by the destructor.

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_writer_t
```

⟨*LCIO interface: types*⟩+≡

```
type :: lcio_writer_t
   private
   type(c_ptr) :: obj
end type lcio_writer_t
```

Constructor for an output associated to a file.

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function open_lcio_writer_new (filename, complevel) bind(C)
     import
     character(c_char), dimension(*), intent(in) :: filename
     integer(c_int), intent(in) :: complevel
   end function open_lcio_writer_new
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_writer_open_out
```

⟨*LCIO interface: procedures*⟩+≡

```
subroutine lcio_writer_open_out (lcio_writer, filename)
  type(lcio_writer_t), intent(out) :: lcio_writer
  type(string_t), intent(in) :: filename
  lcio_writer%obj = open_lcio_writer_new (char (filename) // &
       c_null_char, 9_c_int)
end subroutine lcio_writer_open_out
```

Destructor:

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   subroutine lcio_writer_delete (io_obj) bind(C)
     import
     type(c_ptr), value :: io_obj
   end subroutine lcio_writer_delete
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_writer_close
```

⟨*LCIO interface: procedures*⟩+≡

```
subroutine lcio_writer_close (lciowriter)
  type(lcio_writer_t), intent(inout) :: lciowriter
  call lcio_writer_delete (lciowriter%obj)
end subroutine lcio_writer_close
```

Write a single event to the LCIO writer.

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   subroutine lcio_write_event (io_obj, evt_obj) bind(C)
     import
     type(c_ptr), value :: io_obj, evt_obj
   end subroutine lcio_write_event
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_event_write
```

⟨*LCIO interface: procedures*⟩+≡

```
subroutine lcio_event_write (wrt, evt)
  type(lcio_writer_t), intent(inout) :: wrt
  type(lcio_event_t), intent(in) :: evt
  call lcio_write_event (wrt%obj, evt%obj)
end subroutine lcio_event_write
```

17.11.7 LCIO Reader type

There is a specific LCIO Reader type for handling the input of LCEventImpl objects (i.e., Monte Carlo event samples) from file. Opening the file is done by the constructor, closing by the destructor.

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_reader_t
```

⟨*LCIO interface: types*⟩+≡

```
type :: lcio_reader_t
   private
   type(c_ptr) :: obj
end type lcio_reader_t
```

Constructor for an output associated to a file.

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function open_lcio_reader (filename) bind(C)
     import
     character(c_char), dimension(*), intent(in) :: filename
   end function open_lcio_reader
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_open_file
```

⟨*LCIO interface: procedures*⟩+≡

```
subroutine lcio_open_file (lcio_reader, filename)
  type(lcio_reader_t), intent(out) :: lcio_reader
  type(string_t), intent(in) :: filename
  lcio_reader%obj = open_lcio_reader (char (filename) // c_null_char)
end subroutine lcio_open_file
```

Destructor:

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   subroutine lcio_reader_delete (io_obj) bind(C)
     import
     type(c_ptr), value :: io_obj
   end subroutine lcio_reader_delete
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_reader_close
```

⟨*LCIO interface: procedures*⟩+≡

```
subroutine lcio_reader_close (lcioreader)
  type(lcio_reader_t), intent(inout) :: lcioreader
  call lcio_reader_delete (lcioreader%obj)
end subroutine lcio_reader_close
```

Read a single event from the event file. Return true if successful.

⟨*LCIO interface: interfaces*⟩+≡

```
interface
   type(c_ptr) function read_lcio_event (io_obj) bind(C)
     import
     type(c_ptr), value :: io_obj
   end function read_lcio_event
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_read_event
```

⟨*LCIO interface: procedures*⟩+≡
```
subroutine lcio_read_event (lcrdr, evt, ok)
  type(lcio_reader_t), intent(inout) :: lcrdr
  type(lcio_event_t), intent(out) :: evt
  logical, intent(out) :: ok
  evt%obj = read_lcio_event (lcrdr%obj)
  ok = c_associated (evt%obj)
end subroutine lcio_read_event
```

Extract the process ID. This is stored (at the moment abusively) in the RUN ID as well as in an additional event parameter.

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   integer(c_int) function lcio_event_signal_process_id (evt_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
   end function lcio_event_signal_process_id
end interface
```

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_event_get_process_id
```

⟨*LCIO interface: procedures*⟩+≡
```
function lcio_event_get_process_id (evt) result (i_proc)
  integer :: i_proc
  type(lcio_event_t), intent(in) :: evt
  i_proc = lcio_event_signal_process_id (evt%obj)
end function lcio_event_get_process_id
```

Number of particles in an LCIO event.

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   integer(c_int) function lcio_event_get_n_particles (evt_obj) bind(C)
     import
     type(c_ptr), value :: evt_obj
   end function lcio_event_get_n_particles
end interface
```

⟨*LCIO interface:*⟩≡

⟨*LCIO interface: public*⟩+≡
```
public :: lcio_event_get_n_tot
```

⟨*LCIO interface: procedures*⟩+≡
```
function lcio_event_get_n_tot (evt) result (n_tot)
  integer :: n_tot
  type(lcio_event_t), intent(in) :: evt
  n_tot = lcio_event_get_n_particles (evt%obj)
end function lcio_event_get_n_tot
```

Extracting α_s and the scale.

⟨*LCIO interface: interfaces*⟩+≡
```
interface
   function lcio_event_get_alpha_qcd (evt_obj) result (as) bind(C)
```

```
      import
      real(c_double) :: as
      type(c_ptr), value :: evt_obj
    end function lcio_event_get_alpha_qcd
end interface
interface
    function lcio_event_get_scale (evt_obj) result (scale) bind(C)
      import
      real(c_double) :: scale
      type(c_ptr), value :: evt_obj
    end function lcio_event_get_scale
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_event_get_alphas
```

⟨*LCIO interface: procedures*⟩+≡

```
function lcio_event_get_alphas (evt) result (as)
  type(lcio_event_t), intent(in) :: evt
  real(default) :: as
  as = lcio_event_get_alpha_qcd (evt%obj)
end function lcio_event_get_alphas
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_event_get_scaleval
```

⟨*LCIO interface: procedures*⟩+≡

```
function lcio_event_get_scaleval (evt) result (scale)
  type(lcio_event_t), intent(in) :: evt
  real(default) :: scale
  scale = lcio_event_get_scale (evt%obj)
end function lcio_event_get_scaleval
```

Extracting particles by index from an LCIO event.

⟨*LCIO interface: interfaces*⟩+≡

```
interface
    type(c_ptr) function lcio_event_particle_k (evt_obj, k) bind(C)
      import
      type(c_ptr), value :: evt_obj
      integer(c_int), value :: k
    end function lcio_event_particle_k
end interface
```

⟨*LCIO interface: public*⟩+≡

```
public :: lcio_event_get_particle
```

⟨*LCIO interface: procedures*⟩+≡

```
function lcio_event_get_particle (evt, n) result (prt)
  type(lcio_event_t), intent(in) :: evt
  integer, intent(in) :: n
  type(lcio_particle_t) :: prt
  prt%obj = lcio_event_particle_k (evt%obj, int (n, c_int))
end function lcio_event_get_particle
```

17.11.8 Unit tests

Test module, followed by the corresponding implementation module.

⟨lcio_interface_ut.f90⟩≡

```
⟨File header⟩

module lcio_interface_ut
  use unit_tests
  use lcio_interface_uti

⟨Standard module head⟩

⟨LCIO interface: public test⟩

contains

⟨LCIO interface: test driver⟩

end module lcio_interface_ut
```

⟨lcio_interface_uti.f90⟩≡

```
⟨File header⟩

module lcio_interface_uti

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use lorentz
  use flavors
  use colors
  use polarizations

  use lcio_interface

⟨Standard module head⟩

⟨LCIO interface: test declarations⟩

contains

⟨LCIO interface: tests⟩

end module lcio_interface_uti
```

API: driver for the unit tests below.

⟨*LCIO interface: public test*⟩≡

```
  public :: lcio_interface_test
```

⟨*LCIO interface: test driver*⟩≡

```
  subroutine lcio_interface_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨LCIO interface: execute tests⟩
  end subroutine lcio_interface_test
```

⟨*LCIO interface: execute tests*⟩≡

```
call test (lcio_interface_1, "lcio_interface_1", &
     "check LCIO interface", &
     u, results)
```

⟨*LCIO interface: test declarations*⟩≡

```
public :: lcio_interface_1
```

⟨*LCIO interface: tests*⟩≡

```
subroutine lcio_interface_1 (u)
  use physics_defs, only: VECTOR
  use model_data, only: field_data_t
  integer, intent(in) :: u
  integer :: u_file, iostat
  type(lcio_event_t) :: evt
  type(lcio_particle_t) :: prt1, prt2, prt3, prt4, prt5, prt6, prt7, prt8
  type(flavor_t) :: flv
  type(color_t) :: col
  type(polarization_t) :: pol
  type(field_data_t), target :: photon_data
  character(220) :: buffer

  write (u, "(A)")  "* Test output: LCIO interface"
  write (u, "(A)")  "*   Purpose: test LCIO interface"
  write (u, "(A)")

  write (u, "(A)")  "* Initialization"
  write (u, "(A)")

  ! Initialize a photon flavor object and some polarization
  call photon_data%init (var_str ("PHOTON"), 22)
  call photon_data%set (spin_type=VECTOR)
  call photon_data%freeze ()
  call flv%init (photon_data)
  call pol%init_angles &
       (flv, 0.6_default, 1._default, 0.5_default)

  ! Event initialization
  call lcio_event_init (evt, 20, 1, 42)

  write (u, "(A)")  "* p -> q splitting"
  write (u, "(A)")

  ! $p\to q$ splittings
  call particle_init (prt1, &
       0._default, 0._default, 7000._default, 7000._default, &
       2212, 1._default, 3)
  call particle_init (prt2, &
       0._default, 0._default,-7000._default, 7000._default, &
       2212, 1._default, 3)
  call particle_init (prt3, &
        .750_default, -1.569_default, 32.191_default, 32.238_default, &
        1, -1._default/3._default, 3)
  call color_init_from_array (col, [501])
  call lcio_particle_set_color (prt3, col)
```

```
call lcio_particle_set_parent (prt3, prt1)
call lcio_particle_set_parent (prt3, prt2)
call particle_init (prt4, &
     -3.047_default, -19._default, -54.629_default, 57.920_default, &
     -2, -2._default/3._default, 3)
call color_init_from_array (col, [-501])
call lcio_particle_set_color (prt4, col)
call lcio_particle_set_parent (prt4, prt1)
call lcio_particle_set_parent (prt4, prt2)

write (u, "(A)")  "* Hard interaction"
write (u, "(A)")

! Hard interaction
call particle_init (prt6, &
     -3.813_default, 0.113_default, -1.833_default, 4.233_default, &
     22, 0._default, 1)
call lcio_polarization_init (prt6, pol)
call particle_init (prt5, &
     1.517_default, -20.68_default, -20.605_default, 85.925_default, &
     -24, -1._default, 3)
call lcio_particle_set_parent (prt5, prt3)
call lcio_particle_set_parent (prt5, prt4)
call lcio_particle_set_parent (prt6, prt3)
call lcio_particle_set_parent (prt6, prt4)

! $W^-$ decay
call particle_init (prt7, &
     -2.445_default, 28.816_default, 6.082_default, 29.552_default, &
     1, -1._default/3._default, 1)
call particle_init (prt8, &
     3.962_default, -49.498_default, -26.687_default, 56.373_default, &
     -2, -2._default/3._default, 1)
call lcio_particle_set_t (prt7, 0.12_default)
call lcio_particle_set_t (prt8, 0.12_default)
call lcio_particle_set_vtx &
     (prt7, vector3_moving ([-0.3_default, 0.05_default, 0.004_default]))
call lcio_particle_set_vtx &
     (prt8, vector3_moving ([-0.3_default, 0.05_default, 0.004_default]))
call lcio_particle_set_parent (prt7, prt5)
call lcio_particle_set_parent (prt8, prt5)
call lcio_particle_add_to_evt_coll (prt1, evt)
call lcio_particle_add_to_evt_coll (prt2, evt)
call lcio_particle_add_to_evt_coll (prt3, evt)
call lcio_particle_add_to_evt_coll (prt4, evt)
call lcio_particle_add_to_evt_coll (prt5, evt)
call lcio_particle_add_to_evt_coll (prt6, evt)
call lcio_particle_add_to_evt_coll (prt7, evt)
call lcio_particle_add_to_evt_coll (prt8, evt)
call lcio_event_add_coll (evt)

! Event output
write (u, "(A)")  "Writing in ASCII form to file 'lcio_test.slcio'"
write (u, "(A)")
```

```
  call write_lcio_event (evt, var_str ("lcio_test.slcio"))

  write (u, "(A)")  "Writing completed"

  write (u, "(A)")
  write (u, "(A)")  "* File contents:"
  write (u, "(A)")

  u_file = free_unit ()
  open (u_file, file = "lcio_test.slcio", &
       action = "read", status = "old")
  do
     read (u_file, "(A)", iostat = iostat)  buffer
     if (trim (buffer) == "")  cycle
     if (buffer(1:12) == " - timestamp")  buffer = "[...]"
     if (buffer(1:6) == " date:")  buffer = "[...]"
     if (iostat /= 0)  exit
     write (u, "(A)") trim (buffer)
  end do
  close (u_file)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"
  write (u, "(A)")

  ! Wrapup
  ! call pol%final ()
  call lcio_event_final (evt)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: lcio_interface_1"

contains

  subroutine particle_init &
       (prt, px, py, pz, E, pdg, charge, status)
    type(lcio_particle_t), intent(out) :: prt
    real(default), intent(in) :: px, py, pz, E, charge
    integer, intent(in) :: pdg, status
    type(vector4_t) :: p
    p = vector4_moving (E, vector3_moving ([px, py, pz]))
    call lcio_particle_init (prt, p, pdg, charge, status)
  end subroutine particle_init

end subroutine lcio_interface_1
```

17.12 HEP Common and Events

This is a separate module that manages data exchange between the common blocks and `event_t` objects. We separate this from the previous module in order to avoid a circular module dependency. It also contains the functions necessary

for communication between hepmc_event_t and event_t or lcio_event_t and event_t as well as particle_set_t and particle_t objects.

⟨hep_events.f90⟩≡
```
⟨File header⟩

module hep_events

⟨Use kinds⟩
⟨Use strings⟩
  use diagnostics
  use lorentz
  use numeric_utils
  use flavors
  use colors
  use helicities
  use polarizations
  use model_data
  use subevents, only: PRT_BEAM, PRT_INCOMING, PRT_OUTGOING
  use subevents, only: PRT_UNDEFINED
  use subevents, only: PRT_VIRTUAL, PRT_RESONANT, PRT_BEAM_REMNANT
  use particles
  use hep_common
  use hepmc_interface
  use lcio_interface
  use event_base

⟨Standard module head⟩

⟨HEP events: public⟩

contains

⟨HEP events: procedures⟩

end module hep_events
```

17.12.1 Data Transfer: events

Fill the HEPEUP block, given a WHIZARD event object.

⟨HEP events: public⟩≡
```
  public :: hepeup_from_event
```

⟨HEP events: procedures⟩≡
```
  subroutine hepeup_from_event &
       (event, keep_beams, keep_remnants, process_index)
    class(generic_event_t), intent(in), target :: event
    logical, intent(in), optional :: keep_beams
    logical, intent(in), optional :: keep_remnants
    integer, intent(in), optional :: process_index
    type(particle_set_t), pointer :: particle_set
    real(default) :: scale, alpha_qcd
    if (event%has_valid_particle_set ()) then
       particle_set => event%get_particle_set_ptr ()
       call hepeup_from_particle_set (particle_set, keep_beams, keep_remnants)
```

```
      if (present (process_index)) then
         call hepeup_set_event_parameters (proc_id = process_index)
      end if
      scale = event%get_fac_scale ()
      if (.not. vanishes (scale)) then
         call hepeup_set_event_parameters (scale = scale)
      end if
      alpha_qcd = event%get_alpha_s ()
      if (.not. vanishes (alpha_qcd)) then
         call hepeup_set_event_parameters (alpha_qcd = alpha_qcd)
      end if
      if (event%weight_prc_is_known ()) then
         call hepeup_set_event_parameters (weight = event%get_weight_prc ())
      end if
   else
      call msg_bug ("HEPEUP: event incomplete")
   end if
end subroutine hepeup_from_event
```

Reverse.

Note: The current implementation sets the particle set of the hard process and is therefore not useful if the event on file is dressed. This should be reconsidered.

Note: setting of scale or alpha is not yet supported by the `event_t` object. Ticket #628.

⟨*HEP events: public*⟩+≡

```
public :: hepeup_to_event
```

⟨*HEP events: procedures*⟩+≡

```
subroutine hepeup_to_event &
     (event, fallback_model, process_index, recover_beams, &
     use_alpha_s, use_scale)
   class(generic_event_t), intent(inout), target :: event
   class(model_data_t), intent(in), target :: fallback_model
   integer, intent(out), optional :: process_index
   logical, intent(in), optional :: recover_beams
   logical, intent(in), optional :: use_alpha_s
   logical, intent(in), optional :: use_scale
   class(model_data_t), pointer :: model
   real(default) :: weight, scale, alpha_qcd
   type(particle_set_t) :: particle_set
   model => event%get_model_ptr ()
   call hepeup_to_particle_set &
        (particle_set, recover_beams, model, fallback_model)
   call event%set_hard_particle_set (particle_set)
   call particle_set%final ()
   if (present (process_index)) then
      call hepeup_get_event_parameters (proc_id = process_index)
   end if
   call hepeup_get_event_parameters (weight = weight, &
        scale = scale, alpha_qcd = alpha_qcd)
   call event%set_weight_ref (weight)
   if (present (use_alpha_s)) then
      if (use_alpha_s .and. alpha_qcd > 0) &
```

```
            call event%set_alpha_qcd_forced (alpha_qcd)
      end if
      if (present (use_scale)) then
         if (use_scale .and. scale > 0) &
            call event%set_scale_forced (scale)
      end if
   end subroutine hepeup_to_event
```

Fill the HEPEVT (event) common block.

⟨*HEP events: public*⟩+≡

```
   public :: hepevt_from_event
```

⟨*HEP events: procedures*⟩+≡

```
   subroutine hepevt_from_event  &
         (event, process_index, i_evt, keep_beams, keep_remnants, &
          ensure_order, fill_hepev4)
      class(generic_event_t), intent(in), target :: event
      integer, intent(in), optional :: i_evt, process_index
      logical, intent(in), optional :: keep_beams
      logical, intent(in), optional :: keep_remnants
      logical, intent(in), optional :: ensure_order
      logical, intent(in), optional :: fill_hepev4
      type(particle_set_t), pointer :: particle_set
      real(default) :: alpha_qcd, scale
      if (event%has_valid_particle_set ()) then
         particle_set => event%get_particle_set_ptr ()
         call hepevt_from_particle_set (particle_set, keep_beams, &
              keep_remnants, ensure_order, fill_hepev4)
         if (present (process_index)) then
            call hepevt_set_event_parameters (proc_id = process_index)
         end if
         if (event%weight_prc_is_known ()) then
            call hepevt_set_event_parameters (weight = event%get_weight_prc ())
         end if
         if (event%sqme_prc_is_known ()) then
            call hepevt_set_event_parameters &
                 (function_value = event%get_sqme_prc ())
         end if
         scale = event%get_fac_scale ()
         if (.not. vanishes (scale)) then
            call hepevt_set_event_parameters (scale = scale)
         end if
         alpha_qcd = event%get_alpha_s ()
         if (.not. vanishes (alpha_qcd)) then
            call hepevt_set_event_parameters (alpha_qcd = alpha_qcd)
         end if
         if (present (i_evt)) then
            call hepevt_set_event_parameters (i_evt = i_evt)
         end if
      else
         call msg_bug ("HEPEVT: event incomplete")
      end if
   end subroutine hepevt_from_event
```

HepMC format

The master output function fills a HepMC GenEvent object that is already initialized, but has no vertices in it.

We first set up the vertex lists and enter the vertices into the HepMC event. Then, we assign first all incoming particles and then all outgoing particles to their associated vertices. Particles which have neither parent nor children entries (this should not happen) are dropped.

Finally, we insert the beam particles. If there are none, use the incoming particles instead. Transform a particle into a `hepmc_particle` object, including color and polarization. The HepMC status is equivalent to the HEPEVT status, in particular: 0 = null entry, 1 = physical particle, 2 = decayed/fragmented SM hadron, tau or muon, 3 = other unphysical particle entry, 4 = incoming particles, 11 = intermediate resonance such as squarks. The use of 11 for intermediate resonances is as done by HERWIG, see http://herwig.hepforge.org/trac/wiki/FaQs.

⟨*HEP events: procedures*⟩+≡

```
subroutine particle_to_hepmc (prt, hprt)
  type(particle_t), intent(in) :: prt
  type(hepmc_particle_t), intent(out) :: hprt
  integer :: hepmc_status
  select case (prt%get_status ())
  case (PRT_UNDEFINED)
     hepmc_status = 0
  case (PRT_OUTGOING)
     hepmc_status = 1
  case (PRT_BEAM)
     hepmc_status = 4
  case (PRT_RESONANT)
     if (abs(prt%get_pdg()) == 13 .or. &
          abs(prt%get_pdg()) == 15) then
        hepmc_status = 2
     else
        hepmc_status = 11
     end if
  case default
     hepmc_status = 3
  end select
  call hepmc_particle_init (hprt, &
       prt%get_momentum (), prt%get_pdg (), &
       hepmc_status)
  call hepmc_particle_set_color (hprt, prt%get_color ())
  select case (prt%get_polarization_status ())
  case (PRT_DEFINITE_HELICITY)
     call hepmc_particle_set_polarization (hprt, &
          prt%get_helicity ())
  case (PRT_GENERIC_POLARIZATION)
     call hepmc_particle_set_polarization (hprt, &
          prt%get_polarization ())
  end select
end subroutine particle_to_hepmc
```

⟨*HEP events: public*⟩+≡

```
public :: hepmc_event_from_particle_set
```

⟨*HEP events: procedures*⟩+≡

```
subroutine hepmc_event_from_particle_set &
      (evt, particle_set, cross_section, error)
  type(hepmc_event_t), intent(inout) :: evt
  type(particle_set_t), intent(in) :: particle_set
  real(default), intent(in), optional :: cross_section, error
  type(hepmc_vertex_t), dimension(:), allocatable :: v
  type(hepmc_particle_t), dimension(:), allocatable :: hprt
  type(hepmc_particle_t), dimension(2) :: hbeam
  type(vector4_t), dimension(:), allocatable :: vtx
  logical, dimension(:), allocatable :: is_beam
  integer, dimension(:), allocatable :: v_from, v_to
  integer :: n_vertices, n_tot, i
  n_tot = particle_set%get_n_tot ()
  allocate (v_from (n_tot), v_to (n_tot))
  call particle_set%assign_vertices (v_from, v_to, n_vertices)
  allocate (hprt (n_tot))
  allocate (vtx (n_vertices))
  vtx = vector4_null
  do i = 1, n_tot
     if (v_to(i) /= 0 .or. v_from(i) /= 0) then
        call particle_to_hepmc (particle_set%prt(i), hprt(i))
        if (v_to(i) /= 0) then
           vtx(v_to(i)) = particle_set%prt(i)%get_vertex ()
        end if
     end if
  end do
  if (present (cross_section) .and. present(error)) &
     call hepmc_event_set_cross_section (evt, cross_section, error)
  allocate (v (n_vertices))
  do i = 1, n_vertices
     call hepmc_vertex_init (v(i), vtx(i))
     call hepmc_event_add_vertex (evt, v(i))
  end do
  allocate (is_beam (n_tot))
  !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
  do i = 1, n_tot
     is_beam(i) = particle_set%prt(i)%get_status () == PRT_BEAM
  end do
  !!! is_beam = particle_set%prt(1:n_tot)%get_status () == PRT_BEAM
  if (.not. any (is_beam)) then
     do i = 1, n_tot
        is_beam(i) = particle_set%prt(i)%get_status () == PRT_INCOMING
     end do
     !!! is_beam = particle_set%prt(1:n_tot)%get_status () == PRT_INCOMING
  end if
  if (count (is_beam) == 2) then
     hbeam = pack (hprt, is_beam)
     call hepmc_event_set_beam_particles (evt, hbeam(1), hbeam(2))
  end if
  do i = 1, n_tot
     if (v_to(i) /= 0) then
        call hepmc_vertex_add_particle_in (v(v_to(i)), hprt(i))
     end if
```

```
    end do
    do i = 1, n_tot
       if (v_from(i) /= 0) then
          call hepmc_vertex_add_particle_out (v(v_from(i)), hprt(i))
       end if
    end do
    FIND_SIGNAL_PROCESS: do i = 1, n_tot
       if (particle_set%prt(i)%get_status () == PRT_INCOMING) then
          call hepmc_event_set_signal_process_vertex (evt, v(v_to(i)))
          exit FIND_SIGNAL_PROCESS
       end if
    end do FIND_SIGNAL_PROCESS
  end subroutine hepmc_event_from_particle_set
```

Initialize a particle from a HepMC particle object. The model is necessary for making a fully qualified flavor component. We have the additional flag `polarized` which tells whether the polarization information should be interpreted or ignored, and the lookup array of barcodes. Note that the lookup array is searched linearly, a possible bottleneck for large particle arrays. If necessary, the barcode array could be replaced by a hash table.

⟨*HEP events: procedures*⟩+≡

```
  subroutine particle_from_hepmc_particle &
       (prt, hprt, model, fallback_model, polarization, barcode)
    type(particle_t), intent(out) :: prt
    type(hepmc_particle_t), intent(in) :: hprt
    type(model_data_t), intent(in), target :: model
    type(model_data_t), intent(in), target :: fallback_model
    type(hepmc_vertex_t) :: vtx
    integer, intent(in) :: polarization
    integer, dimension(:), intent(in) :: barcode
    type(hepmc_polarization_t) :: hpol
    type(flavor_t) :: flv
    type(color_t) :: col
    type(helicity_t) :: hel
    type(polarization_t) :: pol
    type(vector4_t) :: vertex
    integer :: n_parents, n_children
    integer, dimension(:), allocatable :: &
         parent_barcode, child_barcode, parent, child
    integer :: i
    select case (hepmc_particle_get_status (hprt))
    case (1);  call prt%set_status (PRT_OUTGOING)
    case (2);  call prt%set_status (PRT_RESONANT)
    case (3);  call prt%set_status (PRT_VIRTUAL)
    end select
    if (hepmc_particle_is_beam (hprt)) call prt%set_status (PRT_BEAM)
    call flv%init (hepmc_particle_get_pdg (hprt), model, fallback_model)
    call col%init (hepmc_particle_get_color (hprt))
    call prt%set_flavor (flv)
    call prt%set_color (col)
    call prt%set_polarization (polarization)
    select case (polarization)
    case (PRT_DEFINITE_HELICITY)
```

```
      hpol = hepmc_particle_get_polarization (hprt)
      call hepmc_polarization_to_hel (hpol, prt%get_flv (), hel)
      call prt%set_helicity (hel)
      call hepmc_polarization_final (hpol)
   case (PRT_GENERIC_POLARIZATION)
      hpol = hepmc_particle_get_polarization (hprt)
      call hepmc_polarization_to_pol (hpol, prt%get_flv (), pol)
      call prt%set_pol (pol)
      call hepmc_polarization_final (hpol)
   end select
   call prt%set_momentum (hepmc_particle_get_momentum (hprt), &
        hepmc_particle_get_mass_squared (hprt))
   n_parents  = hepmc_particle_get_n_parents  (hprt)
   n_children = hepmc_particle_get_n_children (hprt)
   allocate (parent_barcode (n_parents),  parent (n_parents))
   allocate (child_barcode  (n_children), child  (n_children))
   parent_barcode = hepmc_particle_get_parent_barcodes (hprt)
   child_barcode  = hepmc_particle_get_child_barcodes  (hprt)
   do i = 1, size (barcode)
      where (parent_barcode == barcode(i))  parent = i
      where (child_barcode  == barcode(i))  child  = i
   end do
   call prt%set_parents (parent)
   call prt%set_children (child)
   if (prt%get_status () == PRT_VIRTUAL .and. n_parents == 0) &
        call prt%set_status (PRT_INCOMING)
   vtx = hepmc_particle_get_decay_vertex (hprt)
   if (hepmc_vertex_is_valid (vtx)) then
      vertex = hepmc_vertex_to_vertex (vtx)
      if (vertex /= vector4_null)  call prt%set_vertex (vertex)
   end if
end subroutine particle_from_hepmc_particle
```

If a particle set is initialized from a HepMC event record, we have to specify the treatment of polarization (unpolarized or density matrix) which is common to all particles. Correlated polarization information is not available.

There is some complication in reconstructing incoming particles and beam remnants. First of all, they all will be tagged as virtual. We then define an incoming particle as

⟨*HEP events: public*⟩+≡

```
public :: hepmc_event_to_particle_set
```

⟨*HEP events: procedures*⟩+≡

```
subroutine hepmc_event_to_particle_set &
     (particle_set, evt, model, fallback_model, polarization)
  type(particle_set_t), intent(inout), target :: particle_set
  type(hepmc_event_t), intent(in) :: evt
  class(model_data_t), intent(in), target :: model, fallback_model
  integer, intent(in) :: polarization
  type(hepmc_event_particle_iterator_t) :: it
  type(hepmc_vertex_t) :: v
  type(hepmc_vertex_particle_in_iterator_t) :: v_it
  type(hepmc_particle_t) :: prt
  integer, dimension(:), allocatable :: barcode
```

```
  integer :: n_tot, i, bc
  n_tot = 0
  call hepmc_event_particle_iterator_init (it, evt)
  do while (hepmc_event_particle_iterator_is_valid (it))
     n_tot = n_tot + 1
     call hepmc_event_particle_iterator_advance (it)
  end do
  allocate (barcode (n_tot))
  call hepmc_event_particle_iterator_reset (it)
  do i = 1, n_tot
     barcode(i) = hepmc_particle_get_barcode &
          (hepmc_event_particle_iterator_get (it))
     call hepmc_event_particle_iterator_advance (it)
  end do
  allocate (particle_set%prt (n_tot))
  call hepmc_event_particle_iterator_reset (it)
  do i = 1, n_tot
     prt = hepmc_event_particle_iterator_get (it)
     call particle_from_hepmc_particle (particle_set%prt(i), &
          prt, model, fallback_model, polarization, barcode)
     call hepmc_event_particle_iterator_advance (it)
  end do
  call hepmc_event_particle_iterator_final (it)
  v = hepmc_event_get_signal_process_vertex (evt)
  if (hepmc_vertex_is_valid (v)) then
     call hepmc_vertex_particle_in_iterator_init (v_it, v)
     do while (hepmc_vertex_particle_in_iterator_is_valid (v_it))
        prt = hepmc_vertex_particle_in_iterator_get (v_it)
        bc = hepmc_particle_get_barcode &
             (hepmc_vertex_particle_in_iterator_get (v_it))
        do i = 1, size(barcode)
           if (bc == barcode(i))  &
             call particle_set%prt(i)%set_status (PRT_INCOMING)
        end do
        call hepmc_vertex_particle_in_iterator_advance (v_it)
     end do
     call hepmc_vertex_particle_in_iterator_final (v_it)
  end if
  do i = 1, n_tot
     if (particle_set%prt(i)%get_status () == PRT_VIRTUAL &
          .and. particle_set%prt(i)%get_n_children () == 0) &
          call particle_set%prt(i)%set_status (PRT_OUTGOING)
  end do
  particle_set%n_tot = n_tot
  particle_set%n_beam  = &
       count (particle_set%prt%get_status () == PRT_BEAM)
  particle_set%n_in  = &
       count (particle_set%prt%get_status () == PRT_INCOMING)
  particle_set%n_out = &
       count (particle_set%prt%get_status () == PRT_OUTGOING)
  particle_set%n_vir = &
       particle_set%n_tot - particle_set%n_in - particle_set%n_out
end subroutine hepmc_event_to_particle_set
```

Fill a WHIZARD event from a HepMC event record. In HepMC the weights are in a weight container. If the size of this container is larger than one, it is ambiguous to assign the event a specific weight. For now we only allow to read in unweighted events.

⟨*HEP events: public*⟩+≡
```
public :: hepmc_to_event
```

⟨*HEP events: procedures*⟩+≡
```
subroutine hepmc_to_event &
     (event, hepmc_event, fallback_model, process_index, &
     recover_beams, use_alpha_s, use_scale)
  class(generic_event_t), intent(inout), target :: event
  type(hepmc_event_t), intent(inout) :: hepmc_event
  class(model_data_t), intent(in), target :: fallback_model
  integer, intent(out), optional :: process_index
  logical, intent(in), optional :: recover_beams
  logical, intent(in), optional :: use_alpha_s
  logical, intent(in), optional :: use_scale
  class(model_data_t), pointer :: model
  real(default) :: scale, alpha_qcd
  type(particle_set_t) :: particle_set
  model => event%get_model_ptr ()
  call hepmc_event_to_particle_set (particle_set, &
       hepmc_event, model, fallback_model, PRT_DEFINITE_HELICITY)
  call event%set_hard_particle_set (particle_set)
  call particle_set%final ()
  call event%set_weight_ref (1._default)
  alpha_qcd = hepmc_event_get_alpha_qcd (hepmc_event)
  scale = hepmc_event_get_scale (hepmc_event)
  if (present (use_alpha_s)) then
     if (use_alpha_s .and. alpha_qcd > 0) &
          call event%set_alpha_qcd_forced (alpha_qcd)
  end if
  if (present (use_scale)) then
     if (use_scale .and. scale > 0) &
          call event%set_scale_forced (scale)
  end if
end subroutine hepmc_to_event
```

LCIO event format

The master output function fills a LCIO event object that is already initialized, but has no particles in it.

In contrast to HepMC in LCIO there are no vertices (except for tracker and other detector specifications). So we assign first all incoming particles and then all outgoing particles to LCIO particle types. Particles which have neither parent nor children entries (this should not happen) are dropped. Finally, we insert the beam particles. If there are none, use the incoming particles instead.

Transform a particle into a `lcio_particle` object, including color and polarization. The LCIO status is equivalent to the HepMC status, in particular: 0 = null entry, 1 = physical particle, 2 = decayed/fragmented SM hadron, tau or

muon, 3 = other unphysical particle entry, 4 = incoming particles, 11 = intermediate resonance such as squarks. The use of 11 for intermediate resonances is as done by HERWIG, see http://herwig.hepforge.org/trac/wiki/FaQs.

A beam-remnant particle (e.g., ISR photon) that has no children is tagged as outgoing, otherwise unphysical.

⟨*HEP events: public*⟩+≡
```
public :: particle_to_lcio
```

⟨*HEP events: procedures*⟩+≡
```
subroutine particle_to_lcio (prt, lprt)
  type(particle_t), intent(in) :: prt
  type(lcio_particle_t), intent(out) :: lprt
  integer :: lcio_status
  type(vector4_t) :: vtx
  select case (prt%get_status ())
  case (PRT_UNDEFINED)
     lcio_status = 0
  case (PRT_OUTGOING)
     lcio_status = 1
  case (PRT_BEAM_REMNANT)
     if (prt%get_n_children () == 0) then
        lcio_status = 1
     else
        lcio_status = 3
     end if
  case (PRT_BEAM)
     lcio_status = 4
  case (PRT_RESONANT)
     lcio_status = 2
  case default
     lcio_status = 3
  end select
  call lcio_particle_init (lprt, &
       prt%get_momentum (), &
       prt%get_pdg (), &
       prt%flv%get_charge (), &
       lcio_status)
  call lcio_particle_set_color (lprt, prt%get_color ())
  vtx = prt%get_vertex ()
  call lcio_particle_set_vtx (lprt, space_part (vtx))
  call lcio_particle_set_t (lprt, vtx%p(0))
  select case (prt%get_polarization_status ())
  case (PRT_DEFINITE_HELICITY)
     call lcio_polarization_init (lprt, prt%get_helicity ())
  case (PRT_GENERIC_POLARIZATION)
     call lcio_polarization_init (lprt, prt%get_polarization ())
  end select
end subroutine particle_to_lcio
```

Initialize a particle from a LCIO particle object. The model is necessary for making a fully qualified flavor component.

⟨*HEP events: public*⟩+≡
```
public :: particle_from_lcio_particle
```

⟨*HEP events: procedures*⟩+≡

```
subroutine particle_from_lcio_particle &
   (prt, lprt, model, daughters, parents, polarization)
  type(particle_t), intent(out) :: prt
  type(lcio_particle_t), intent(in) :: lprt
  type(model_data_t), intent(in), target :: model
  integer, dimension(:), intent(in) :: daughters, parents
  type(vector4_t) :: vtx4
  type(flavor_t) :: flv
  type(color_t) :: col
  type(helicity_t) :: hel
  type(polarization_t) :: pol
  integer, intent(in) :: polarization
  select case (lcio_particle_get_status (lprt))
  case (1);  call prt%set_status (PRT_OUTGOING)
  case (2);  call prt%set_status (PRT_RESONANT)
  case (3);  call prt%set_status (PRT_VIRTUAL)
  end select
  call flv%init (lcio_particle_get_pdg (lprt), model)
  call col%init (lcio_particle_get_flow (lprt))
  if (flv%is_beam_remnant ())  call prt%set_status (PRT_BEAM_REMNANT)
  call prt%set_flavor (flv)
  call prt%set_color (col)
  call prt%set_polarization (polarization)
  select case (polarization)
  case (PRT_DEFINITE_HELICITY)
     call lcio_particle_to_hel (lprt, prt%get_flv (), hel)
     call prt%set_helicity (hel)
  case (PRT_GENERIC_POLARIZATION)
     call lcio_particle_to_pol (lprt, prt%get_flv (), pol)
     call prt%set_pol (pol)
  end select
  call prt%set_momentum (lcio_particle_get_momentum (lprt), &
       lcio_particle_get_mass_squared (lprt))
  call prt%set_parents (parents)
  call prt%set_children (daughters)
  if (prt%get_status () == PRT_VIRTUAL .and. size(parents) == 0) &
       call prt%set_status (PRT_INCOMING)
  vtx4 = vector4_moving (lcio_particle_get_time (lprt), &
       lcio_particle_get_vertex (lprt))
  if (vtx4 /= vector4_null)  call prt%set_vertex (vtx4)
end subroutine particle_from_lcio_particle
```

⟨*HEP events: public*⟩+≡

```
public :: lcio_event_from_particle_set
```

⟨*HEP events: procedures*⟩+≡

```
subroutine lcio_event_from_particle_set (evt, particle_set)
  type(lcio_event_t), intent(inout) :: evt
  type(particle_set_t), intent(in) :: particle_set
  type(lcio_particle_t), dimension(:), allocatable :: lprt
  integer, dimension(:), allocatable :: parent
  integer :: n_tot, i, j, n_beam, n_parents, type, beam_count
  n_tot = particle_set%n_tot
```

```
    n_beam = count (particle_set%prt%get_status () == PRT_BEAM)
    if (n_beam == 0) then
       type = PRT_INCOMING
    else
       type = PRT_BEAM
    end if
    beam_count = 0
    allocate (lprt (n_tot))
    do i = 1, n_tot
       call particle_to_lcio (particle_set%prt(i), lprt(i))
       n_parents = particle_set%prt(i)%get_n_parents ()
       if (n_parents /= 0) then
          allocate (parent (n_parents))
          parent = particle_set%prt(i)%get_parents ()
          do j = 1, n_parents
             call lcio_particle_set_parent (lprt(i), lprt(parent(j)))
          end do
          deallocate (parent)
       end if
       if (particle_set%prt(i)%get_status () == type) then
          beam_count = beam_count + 1
          call lcio_event_set_beam &
               (evt, particle_set%prt(i)%get_pdg (), beam_count)
       end if
       call lcio_particle_add_to_evt_coll (lprt(i), evt)
    end do
    call lcio_event_add_coll (evt)
  end subroutine lcio_event_from_particle_set
```

If a particle set is initialized from a LCIO event record, we have to specify the treatment of polarization (unpolarized or density matrix) which is common to all particles. Correlated polarization information is not available.

⟨*HEP events: public*⟩+≡

```
  public :: lcio_event_to_particle_set
```

⟨*HEP events: procedures*⟩+≡

```
  subroutine lcio_event_to_particle_set &
       (particle_set, evt, model, fallback_model, polarization)
    type(particle_set_t), intent(inout), target :: particle_set
    type(lcio_event_t), intent(in) :: evt
    class(model_data_t), intent(in), target :: model, fallback_model
    integer, intent(in) :: polarization
    type(lcio_particle_t) :: prt
    integer, dimension(:), allocatable :: parents, daughters
    integer :: n_tot, i, j, n_parents, n_children
    n_tot = lcio_event_get_n_tot (evt)
    allocate (particle_set%prt (n_tot))
    do i = 1, n_tot
       prt = lcio_event_get_particle (evt, i-1)
       n_parents = lcio_particle_get_n_parents (prt)
       n_children = lcio_particle_get_n_children (prt)
       allocate (daughters (n_children))
       allocate (parents (n_parents))
       if (n_children > 0) then
```

```
            do j = 1, n_children
               daughters(j) = lcio_get_n_children (evt,i,j)
            end do
         end if
         if (n_parents > 0) then
            do j = 1, n_parents
               parents(j) = lcio_get_n_parents (evt,i,j)
            end do
         end if
         call particle_from_lcio_particle (particle_set%prt(i), prt, model, &
              daughters, parents, polarization)
         deallocate (daughters, parents)
      end do
      do i = 1, n_tot
         if (particle_set%prt(i)%get_status () == PRT_VIRTUAL) then
            CHECK_BEAM: do j = 1, particle_set%prt(i)%get_n_parents ()
               if (particle_set%prt(j)%get_status () == PRT_BEAM) &
                    call particle_set%prt(i)%set_status (PRT_INCOMING)
               exit CHECK_BEAM
            end do CHECK_BEAM
         end if
      end do
      particle_set%n_tot = n_tot
      particle_set%n_beam  = &
           count (particle_set%prt%get_status () == PRT_BEAM)
      particle_set%n_in  = &
           count (particle_set%prt%get_status () == PRT_INCOMING)
      particle_set%n_out = &
           count (particle_set%prt%get_status () == PRT_OUTGOING)
      particle_set%n_vir = &
           particle_set%n_tot - particle_set%n_in - particle_set%n_out
   end subroutine lcio_event_to_particle_set
```

⟨*HEP events: public*⟩+≡

```
   public :: lcio_to_event
```

⟨*HEP events: procedures*⟩+≡

```
   subroutine lcio_to_event &
        (event, lcio_event, fallback_model, process_index, recover_beams, &
         use_alpha_s, use_scale)
      class(generic_event_t), intent(inout), target :: event
      type(lcio_event_t), intent(inout) :: lcio_event
      class(model_data_t), intent(in), target :: fallback_model
      integer, intent(out), optional :: process_index
      logical, intent(in), optional :: recover_beams
      logical, intent(in), optional :: use_alpha_s
      logical, intent(in), optional :: use_scale
      class(model_data_t), pointer :: model
      real(default) :: scale, alpha_qcd
      type(particle_set_t) :: particle_set
      model => event%get_model_ptr ()
      call lcio_event_to_particle_set (particle_set, &
           lcio_event, model, fallback_model, PRT_DEFINITE_HELICITY)
      call event%set_hard_particle_set (particle_set)
```

```
      call particle_set%final ()
      alpha_qcd = lcio_event_get_alphas (lcio_event)
      scale = lcio_event_get_scaleval (lcio_event)
      if (present (use_alpha_s)) then
         if (use_alpha_s .and. alpha_qcd > 0) &
              call event%set_alpha_qcd_forced (alpha_qcd)
      end if
      if (present (use_scale)) then
         if (use_scale .and. scale > 0) &
              call event%set_scale_forced (scale)
      end if
    end subroutine lcio_to_event
```

17.12.2 Unit tests

Test module, followed by the corresponding implementation module.

⟨hep_events_ut.f90⟩≡

```
  ⟨File header⟩

  module hep_events_ut
    use unit_tests
    use hepmc_interface, only: HEPMC_IS_AVAILABLE
    use hep_events_uti

  ⟨Standard module head⟩

  ⟨HEP events: public test⟩

  contains

  ⟨HEP events: test driver⟩

  end module hep_events_ut
```

⟨hep_events_uti.f90⟩≡

```
  ⟨File header⟩

  module hep_events_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use lorentz
    use flavors
    use colors
    use helicities
    use quantum_numbers
    use state_matrices, only: FM_SELECT_HELICITY, FM_FACTOR_HELICITY
    use interactions
    use evaluators
    use model_data
    use particles
    use subevents
    use hepmc_interface
```

```
  use hep_events

⟨Standard module head⟩

⟨HEP events: test declarations⟩

contains

⟨HEP events: tests⟩

end module hep_events_uti
```

API: driver for the unit tests below.

⟨*HEP events: public test*⟩≡
```
  public :: hep_events_test
```

⟨*HEP events: test driver*⟩≡
```
  subroutine hep_events_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨HEP events: execute tests⟩
  end subroutine hep_events_test
```

If `HepMC` is available, check the routines via `HepMC`.

Set up a chain of production and decay and factorize the result into particles. The process is $d\bar{d} \to Z \to q\bar{q}$.

⟨*HEP events: execute tests*⟩≡
```
  if (hepmc_is_available ()) then
     call test (hep_events_1, "hep_events_1", &
          "check HepMC event routines", &
          u, results)
  end if
```

⟨*HEP events: test declarations*⟩≡
```
  public :: hep_events_1
```

⟨*HEP events: tests*⟩≡
```
  subroutine hep_events_1 (u)
    use os_interface
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t), dimension(3) :: flv
    type(color_t), dimension(3) :: col
    type(helicity_t), dimension(3) :: hel
    type(quantum_numbers_t), dimension(3) :: qn
    type(vector4_t), dimension(3) :: p
    type(interaction_t), target :: int1, int2
    type(quantum_numbers_mask_t) :: qn_mask_conn
    type(evaluator_t), target :: eval
    type(interaction_t), pointer :: int
    type(particle_set_t) :: particle_set1, particle_set2
    type(hepmc_event_t) :: hepmc_event
    type(hepmc_iostream_t) :: iostream
    real(default) :: cross_section, error, weight
```

```
logical :: ok

write (u, "(A)")  "* Test output: HEP events"
write (u, "(A)")  "*   Purpose: test HepMC event routines"
write (u, "(A)")

write (u, "(A)")  "* Reading model file"

call model%init_sm_test ()

write (u, "(A)")
write (u, "(A)")  "* Initializing production process"

call int1%basic_init (2, 0, 1, set_relations=.true.)
call flv%init ([1, -1, 23], model)
call col%init_col_acl ([0, 0, 0], [0, 0, 0])
call hel(3)%init ( 1, 1)
call qn%init (flv, col, hel)
call int1%add_state (qn, value=(0.25_default, 0._default))
call hel(3)%init ( 1,-1)
call qn%init (flv, col, hel)
call int1%add_state (qn, value=(0._default, 0.25_default))
call hel(3)%init (-1, 1)
call qn%init (flv, col, hel)
call int1%add_state (qn, value=(0._default,-0.25_default))
call hel(3)%init (-1,-1)
call qn%init (flv, col, hel)
call int1%add_state (qn, value=(0.25_default, 0._default))
call hel(3)%init ( 0, 0)
call qn%init (flv, col, hel)
call int1%add_state (qn, value=(0.5_default, 0._default))
call int1%freeze ()
p(1) = vector4_moving (45._default, 45._default, 3)
p(2) = vector4_moving (45._default,-45._default, 3)
p(3) = p(1) + p(2)
call int1%set_momenta (p)

write (u, "(A)")
write (u, "(A)")  "* Setup decay process"

call int2%basic_init (1, 0, 2, set_relations=.true.)
call flv%init ([23, 1, -1], model)
call col%init_col_acl ([0, 501, 0], [0, 0, 501])
call hel%init ([1, 1, 1], [1, 1, 1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(1._default, 0._default))
call hel%init ([1, 1, 1], [-1,-1,-1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(0._default, 0.1_default))
call hel%init ([-1,-1,-1], [1, 1, 1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(0._default,-0.1_default))
call hel%init ([-1,-1,-1], [-1,-1,-1])
call qn%init (flv, col, hel)
```

```
call int2%add_state (qn, value=(1._default, 0._default))
call hel%init ([0, 1,-1], [0, 1,-1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(4._default, 0._default))
call hel%init ([0,-1, 1], [0, 1,-1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(2._default, 0._default))
call hel%init ([0, 1,-1], [0,-1, 1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(2._default, 0._default))
call hel%init ([0,-1, 1], [0,-1, 1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(4._default, 0._default))
call flv%init ([23, 2, -2], model)
call hel%init ([0, 1,-1], [0, 1,-1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(0.5_default, 0._default))
call hel%init ([0,-1, 1], [0,-1, 1])
call qn%init (flv, col, hel)
call int2%add_state (qn, value=(0.5_default, 0._default))
call int2%freeze ()
p(2) = vector4_moving (45._default, 45._default, 2)
p(3) = vector4_moving (45._default,-45._default, 2)
call int2%set_momenta (p)
call int2%set_source_link (1, int1, 3)
call int1%basic_write (u)
call int2%basic_write (u)

write (u, "(A)")
write (u, "(A)")  "* Concatenate production and decay"

call eval%init_product (int1, int2, qn_mask_conn, &
     connections_are_resonant=.true.)
call eval%receive_momenta ()
call eval%evaluate ()
call eval%write (u)

write (u, "(A)")
write (u, "(A)")  "* Factorize as subevent (complete, polarized)"
write (u, "(A)")

int => eval%interaction_t
call particle_set1%init &
     (ok, int, int, FM_FACTOR_HELICITY, &
      [0.2_default, 0.2_default], .false., .true.)
call particle_set1%write (u)

write (u, "(A)")
write (u, "(A)")  "* Factorize as subevent (in/out only, selected helicity)"
write (u, "(A)")

int => eval%interaction_t
call particle_set2%init &
     (ok, int, int, FM_SELECT_HELICITY, &
```

```
      [0.9_default, 0.9_default], .false., .false.)
call particle_set2%write (u)
call particle_set2%final ()

write (u, "(A)")
write (u, "(A)")  "* Factorize as subevent (complete, selected helicity)"
write (u, "(A)")

int => eval%interaction_t
call particle_set2%init &
     (ok, int, int, FM_SELECT_HELICITY, &
      [0.7_default, 0.7_default], .false., .true.)
call particle_set2%write (u)

write (u, "(A)")
write (u, "(A)")  "* Transfer particle_set to HepMC, print, and output to"
write (u, "(A)")  "        hep_events.hepmc.dat"
write (u, "(A)")

cross_section = 42.0_default
error = 17.0_default
weight = 1.0_default
call hepmc_event_init (hepmc_event, 11, 127)
call hepmc_event_from_particle_set (hepmc_event, particle_set2, &
     cross_section, error)
call hepmc_event_add_weight (hepmc_event, weight)
call hepmc_event_print (hepmc_event)
call hepmc_iostream_open_out &
     (iostream , var_str ("hep_events.hepmc.dat"))
call hepmc_iostream_write_event (iostream, hepmc_event)
call hepmc_iostream_close (iostream)

write (u, "(A)")
write (u, "(A)")  "* Recover from HepMC file"
write (u, "(A)")

call particle_set2%final ()
call hepmc_event_final (hepmc_event)
call hepmc_event_init (hepmc_event)
call hepmc_iostream_open_in &
     (iostream , var_str ("hep_events.hepmc.dat"))
call hepmc_iostream_read_event (iostream, hepmc_event, ok)
call hepmc_iostream_close (iostream)
call hepmc_event_to_particle_set (particle_set2, &
     hepmc_event, model, model, PRT_DEFINITE_HELICITY)
call particle_set2%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call particle_set1%final ()
call particle_set2%final ()
call eval%final ()
call int1%final ()
```

```
    call int2%final ()
    call hepmc_event_final (hepmc_event)
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: hep_events_1"

  end subroutine hep_events_1
```

17.13 LHEF Input/Output

The LHEF event record is standardized. It is an ASCII format. We try our best at using it for both input and output.

⟨eio_lhef.f90⟩≡
⟨*File header*⟩

```
module eio_lhef
```

⟨*Use kinds*⟩
⟨*Use strings*⟩

```
  use io_units
  use string_utils
  use numeric_utils
  use diagnostics
  use os_interface
  use xml
  use event_base
  use eio_data
  use eio_base
  use hep_common
  use hep_events
```

⟨*Standard module head*⟩

⟨*EIO LHEF: public*⟩

⟨*EIO LHEF: types*⟩

```
contains
```

⟨*EIO LHEF: procedures*⟩

```
end module eio_lhef
```

17.13.1 Type

With sufficient confidence that it will always be three characters, we can store the version string with a default value.

⟨*EIO LHEF: public*⟩≡

```
  public :: eio_lhef_t
```

⟨*EIO LHEF: types*⟩≡

```
type, extends (eio_t) :: eio_lhef_t
   logical :: writing = .false.
   logical :: reading = .false.
   integer :: unit = 0
   type(event_sample_data_t) :: data
   type(cstream_t) :: cstream
   character(3) :: version = "1.0"
   logical :: keep_beams = .false.
   logical :: keep_remnants = .true.
   logical :: keep_virtuals = .false.
   logical :: recover_beams = .true.
   logical :: unweighted = .true.
   logical :: write_sqme_ref = .false.
   logical :: write_sqme_prc = .false.
   logical :: write_sqme_alt = .false.
   logical :: use_alpha_s_from_file = .false.
   logical :: use_scale_from_file = .false.
   integer :: n_alt = 0
   integer, dimension(:), allocatable :: proc_num_id
   integer :: i_weight_sqme = 0
   type(xml_tag_t) :: tag_lhef, tag_head, tag_init, tag_event
   type(xml_tag_t), allocatable :: tag_gen_n, tag_gen_v
   type(xml_tag_t), allocatable :: tag_generator, tag_xsecinfo
   type(xml_tag_t), allocatable :: tag_sqme_ref, tag_sqme_prc
   type(xml_tag_t), dimension(:), allocatable :: tag_sqme_alt, tag_wgts_alt
   type(xml_tag_t), allocatable :: tag_weight, tag_weightinfo, tag_weights
 contains
 ⟨EIO LHEF: eio lhef: TBP⟩
end type eio_lhef_t
```

17.13.2 Specific Methods

Set parameters that are specifically used with LHEF.

⟨*EIO LHEF: eio lhef: TBP*⟩≡

```
procedure :: set_parameters => eio_lhef_set_parameters
```

⟨*EIO LHEF: procedures*⟩≡

```
subroutine eio_lhef_set_parameters (eio, &
     keep_beams, keep_remnants, recover_beams, &
     use_alpha_s_from_file, use_scale_from_file, &
     version, extension, write_sqme_ref, write_sqme_prc, write_sqme_alt)
  class(eio_lhef_t), intent(inout) :: eio
  logical, intent(in), optional :: keep_beams
  logical, intent(in), optional :: keep_remnants
  logical, intent(in), optional :: recover_beams
  logical, intent(in), optional :: use_alpha_s_from_file
  logical, intent(in), optional :: use_scale_from_file
  character(*), intent(in), optional :: version
  type(string_t), intent(in), optional :: extension
  logical, intent(in), optional :: write_sqme_ref
  logical, intent(in), optional :: write_sqme_prc
  logical, intent(in), optional :: write_sqme_alt
```

```
  if (present (keep_beams))  eio%keep_beams = keep_beams
  if (present (keep_remnants))  eio%keep_remnants = keep_remnants
  if (present (recover_beams))  eio%recover_beams = recover_beams
  if (present (use_alpha_s_from_file)) &
       eio%use_alpha_s_from_file = use_alpha_s_from_file
  if (present (use_scale_from_file)) &
       eio%use_scale_from_file = use_scale_from_file
  if (present (version)) then
     select case (version)
     case ("1.0", "2.0", "3.0")
        eio%version = version
     case default
        call msg_error ("LHEF version " // version &
             // " is not supported.  Inserting 2.0")
        eio%version = "2.0"
     end select
  end if
  if (present (extension)) then
     eio%extension = extension
  else
     eio%extension = "lhe"
  end if
  if (present (write_sqme_ref))  eio%write_sqme_ref = write_sqme_ref
  if (present (write_sqme_prc))  eio%write_sqme_prc = write_sqme_prc
  if (present (write_sqme_alt))  eio%write_sqme_alt = write_sqme_alt
end subroutine eio_lhef_set_parameters
```

17.13.3 Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
procedure :: write => eio_lhef_write
```

⟨*EIO LHEF: procedures*⟩+≡

```
subroutine eio_lhef_write (object, unit)
  class(eio_lhef_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "LHEF event stream:"
  if (object%writing) then
     write (u, "(3x,A,A)")  "Writing to file   = ", char (object%filename)
  else if (object%reading) then
     write (u, "(3x,A,A)")  "Reading from file = ", char (object%filename)
  else
     write (u, "(3x,A)")  "[closed]"
  end if
  write (u, "(3x,A,L1)")    "Keep beams        = ", object%keep_beams
  write (u, "(3x,A,L1)")    "Keep remnants     = ", object%keep_remnants
  write (u, "(3x,A,L1)")    "Recover beams     = ", object%recover_beams
  write (u, "(3x,A,L1)")    "Alpha_s from file = ", &
       object%use_alpha_s_from_file
```

```
    write (u, "(3x,A,L1)")    "Scale from file   = ", &
         object%use_scale_from_file
    write (u, "(3x,A,A)")     "Version           = ", object%version
    write (u, "(3x,A,A,A)")     "File extension    = '", &
         char (object%extension), "'"
    if (allocated (object%proc_num_id)) then
       write (u, "(3x,A)")  "Numerical process IDs:"
       do i = 1, size (object%proc_num_id)
          write (u, "(5x,I0,': ',I0)")  i, object%proc_num_id(i)
       end do
    end if
  end subroutine eio_lhef_write
```

Finalizer: close any open file.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
  procedure :: final => eio_lhef_final
```

⟨*EIO LHEF: procedures*⟩+≡

```
  subroutine eio_lhef_final (object)
    class(eio_lhef_t), intent(inout) :: object
    if (allocated (object%proc_num_id))  deallocate (object%proc_num_id)
    if (object%writing) then
       write (msg_buffer, "(A,A,A)")  "Events: closing LHEF file '", &
            char (object%filename), "'"
       call msg_message ()
       call object%write_footer ()
       close (object%unit)
       object%writing = .false.
    else if (object%reading) then
       write (msg_buffer, "(A,A,A)")  "Events: closing LHEF file '", &
            char (object%filename), "'"
       call msg_message ()
       call object%cstream%final ()
       close (object%unit)
       object%reading = .false.
    end if
  end subroutine eio_lhef_final
```

Common initialization for input and output.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
  procedure :: common_init => eio_lhef_common_init
```

⟨*EIO LHEF: procedures*⟩+≡

```
  subroutine eio_lhef_common_init (eio, sample, data, extension)
    class(eio_lhef_t), intent(inout) :: eio
    type(string_t), intent(in) :: sample
    type(string_t), intent(in), optional :: extension
    type(event_sample_data_t), intent(in), optional :: data
    if (.not. present (data)) &
         call msg_bug ("LHEF initialization: missing data")
    eio%data = data
    if (data%n_beam /= 2) &
         call msg_fatal ("LHEF: defined for scattering processes only")
    eio%unweighted = data%unweighted
```

```
    if (eio%unweighted) then
       select case (data%norm_mode)
       case (NORM_UNIT)
       case default;  call msg_fatal &
            ("LHEF: normalization for unweighted events must be '1'")
       end select
    else
       select case (data%norm_mode)
       case (NORM_SIGMA)
       case default;  call msg_fatal &
            ("LHEF: normalization for weighted events must be 'sigma'")
       end select
    end if
    eio%n_alt = data%n_alt
    eio%sample = sample
    if (present (extension)) then
       eio%extension = extension
    end if
    call eio%set_filename ()
    eio%unit = free_unit ()
    call eio%init_tags (data)
    allocate (eio%proc_num_id (data%n_proc), source = data%proc_num_id)
  end subroutine eio_lhef_common_init
```

Initialize the tag objects. Some tags depend on the LHEF version. In particular, the tags that in LHEF 2.0 identify individual weights by name in each event block, in LHEF 3.0 are replaced by info tags in the init block and a single `weights` tag in the event block. The name attributes of those tags are specific for `WHIZARD`.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
  procedure :: init_tags => eio_lhef_init_tags
```

⟨*EIO LHEF: procedures*⟩+≡

```
  subroutine eio_lhef_init_tags (eio, data)
    class(eio_lhef_t), intent(inout) :: eio
    type(event_sample_data_t), intent(in) :: data
    real(default), parameter :: pb_per_fb = 1.e-3_default
    integer :: i
    call eio%tag_lhef%init ( &
         var_str ("LesHouchesEvents"), &
         [xml_attribute (var_str ("version"), var_str (eio%version))], &
         .true.)
    call eio%tag_head%init ( &
         var_str ("header"), &
         .true.)
    call eio%tag_init%init ( &
         var_str ("init"), &
         .true.)
    call eio%tag_event%init (var_str ("event"), &
         .true.)
    select case (eio%version)
    case ("1.0")
       allocate (eio%tag_gen_n)
       call eio%tag_gen_n%init ( &
```

```
        var_str ("generator_name"), &
        .true.)
   allocate (eio%tag_gen_v)
   call eio%tag_gen_v%init ( &
        var_str ("generator_version"), &
        .true.)
end select
select case (eio%version)
case ("2.0", "3.0")
   allocate (eio%tag_generator)
   call eio%tag_generator%init ( &
        var_str ("generator"), &
        [xml_attribute (var_str ("version"), var_str ("⟨Version⟩"))], &
        .true.)
   allocate (eio%tag_xsecinfo)
   call eio%tag_xsecinfo%init ( &
        var_str ("xsecinfo"), &
        [xml_attribute (var_str ("neve"), str (data%n_evt)), &
         xml_attribute (var_str ("totxsec"), &
                        str (data%total_cross_section * pb_per_fb))])
end select
select case (eio%version)
case ("2.0")
   allocate (eio%tag_weight)
   call eio%tag_weight%init (var_str ("weight"), &
        [xml_attribute (var_str ("name"))])
   if (eio%write_sqme_ref) then
      allocate (eio%tag_sqme_ref)
      call eio%tag_sqme_ref%init (var_str ("weight"), &
           [xml_attribute (var_str ("name"), var_str ("sqme_ref"))], &
           .true.)
   end if
   if (eio%write_sqme_prc) then
      allocate (eio%tag_sqme_prc)
      call eio%tag_sqme_prc%init (var_str ("weight"), &
           [xml_attribute (var_str ("name"), var_str ("sqme_prc"))], &
           .true.)
   end if
   if (eio%n_alt > 0) then
      if (eio%write_sqme_alt) then
         allocate (eio%tag_sqme_alt (1))
         call eio%tag_sqme_alt(1)%init (var_str ("weight"), &
              [xml_attribute (var_str ("name"), var_str ("sqme_alt"))], &
              .true.)
      end if
      allocate (eio%tag_wgts_alt (1))
      call eio%tag_wgts_alt(1)%init (var_str ("weight"), &
           [xml_attribute (var_str ("name"), var_str ("wgts_alt"))], &
           .true.)
   end if
case ("3.0")
   if (eio%write_sqme_ref) then
      allocate (eio%tag_sqme_ref)
      call eio%tag_sqme_ref%init (var_str ("weightinfo"), &
```

```
            [xml_attribute (var_str ("name"), var_str ("sqme_ref"))])
      end if
      if (eio%write_sqme_prc) then
         allocate (eio%tag_sqme_prc)
         call eio%tag_sqme_prc%init (var_str ("weightinfo"), &
              [xml_attribute (var_str ("name"), var_str ("sqme_prc"))])
      end if
      if (eio%n_alt > 0) then
         if (eio%write_sqme_alt) then
            allocate (eio%tag_sqme_alt (eio%n_alt))
            do i = 1, eio%n_alt
               call eio%tag_sqme_alt(i)%init (var_str ("weightinfo"), &
                    [xml_attribute (var_str ("name"), &
                                    var_str ("sqme_alt") // str (i))])
            end do
         end if
         allocate (eio%tag_wgts_alt (eio%n_alt))
         do i = 1, eio%n_alt
            call eio%tag_wgts_alt(i)%init (var_str ("weightinfo"), &
                 [xml_attribute (var_str ("name"), &
                                 var_str ("wgts_alt") // str (i))])
         end do
      end if
      allocate (eio%tag_weightinfo)
      call eio%tag_weightinfo%init (var_str ("weightinfo"), &
           [xml_attribute (var_str ("name"))])
      allocate (eio%tag_weights)
      call eio%tag_weights%init (var_str ("weights"), .true.)
    end select
  end subroutine eio_lhef_init_tags
```

Initialize event writing.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
  procedure :: init_out => eio_lhef_init_out
```

⟨*EIO LHEF: procedures*⟩+≡

```
  subroutine eio_lhef_init_out (eio, sample, data, success, extension)
    class(eio_lhef_t), intent(inout) :: eio
    type(string_t), intent(in) :: sample
    type(string_t), intent(in), optional :: extension
    type(event_sample_data_t), intent(in), optional :: data
    logical, intent(out), optional :: success
    integer :: u, i
    call eio%set_splitting (data)
    call eio%common_init (sample, data, extension)
    write (msg_buffer, "(A,A,A)")  "Events: writing to LHEF file '", &
         char (eio%filename), "'"
    call msg_message ()
    eio%writing = .true.
    u = eio%unit
    open (u, file = char (eio%filename), &
         action = "write", status = "replace")
    call eio%write_header ()
    call heprup_init &
```

```
      (data%pdg_beam, &
       data%energy_beam, &
       n_processes = data%n_proc, &
       unweighted = data%unweighted, &
       negative_weights = data%negative_weights)
  do i = 1, data%n_proc
     call heprup_set_process_parameters (i = i, &
          process_id = data%proc_num_id(i), &
          cross_section = data%cross_section(i), &
          error = data%error(i))
  end do
  call eio%tag_init%write (u);  write (u, *)
  call heprup_write_lhef (u)
  select case (eio%version)
  case ("2.0");  call eio%write_init_20 (data)
  case ("3.0");  call eio%write_init_30 (data)
  end select
  call eio%tag_init%close (u);  write (u, *)
  if (present (success))  success = .true.
end subroutine eio_lhef_init_out
```

Initialize event reading. First read the LHEF tag and version, then read the header and skip over its contents, then read the init block. (We require the opening and closing tags of the init block to be placed on separate lines without extra stuff.)

For input, we do not (yet?) support split event files.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
procedure :: init_in => eio_lhef_init_in
```

⟨*EIO LHEF: procedures*⟩+≡

```
subroutine eio_lhef_init_in (eio, sample, data, success, extension)
  class(eio_lhef_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(string_t), intent(in), optional :: extension
  type(event_sample_data_t), intent(inout), optional :: data
  logical, intent(out), optional :: success
  logical :: exist, ok, closing
  type(event_sample_data_t) :: data_file
  type(string_t) :: string
  integer :: u
  eio%split = .false.
  call eio%common_init (sample, data, extension)
  write (msg_buffer, "(A,A,A)")  "Events: reading from LHEF file '", &
       char (eio%filename), "'"
  call msg_message ()
  inquire (file = char (eio%filename), exist = exist)
  if (.not. exist)  call msg_fatal ("Events: LHEF file not found.")
  eio%reading = .true.
  u = eio%unit
  open (u, file = char (eio%filename), &
       action = "read", status = "old")
  call eio%cstream%init (u)
  call eio%read_header ()
  call eio%tag_init%read (eio%cstream, ok)
```

```
    if (.not. ok)  call err_init
    select case (eio%version)
    case ("1.0");  call eio%read_init_10 (data_file)
       call eio%tag_init%read_content (eio%cstream, string, closing)
       if (string /= "" .or. .not. closing)  call err_init
    case ("2.0");  call eio%read_init_20 (data_file)
    case ("3.0");  call eio%read_init_30 (data_file)
    end select
    call eio%merge_data (data, data_file)
    if (present (success))  success = .true.

  contains

    subroutine err_init
      call msg_fatal ("LHEF: syntax error in init tag")
    end subroutine err_init

  end subroutine eio_lhef_init_in
```

Merge event sample data: we can check the data in the file against our assumptions and set or reset parameters.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
  procedure :: merge_data => eio_merge_data
```

⟨*EIO LHEF: procedures*⟩+≡

```
  subroutine eio_merge_data (eio, data, data_file)
    class(eio_lhef_t), intent(inout) :: eio
    type(event_sample_data_t), intent(inout) :: data
    type(event_sample_data_t), intent(in) :: data_file
    real, parameter :: tolerance = 1000 * epsilon (1._default)
    if (data%unweighted .neqv. data_file%unweighted)  call err_weights
    if (data%negative_weights .neqv. data_file%negative_weights) &
         call err_weights
    if (data%norm_mode /= data_file%norm_mode)  call err_norm
    if (data%n_beam /= data_file%n_beam)  call err_beams
    if (any (data%pdg_beam /= data_file%pdg_beam))  call err_beams
    if (any (abs ((data%energy_beam - data_file%energy_beam)) &
         > (data%energy_beam + data_file%energy_beam) * tolerance)) &
         call err_beams
    if (data%n_proc /= data_file%n_proc)  call err_proc
    if (any (data%proc_num_id /= data_file%proc_num_id))  call err_proc
    where (data%cross_section == 0)
       data%cross_section = data_file%cross_section
       data%error = data_file%error
    end where
    data%total_cross_section = sum (data%cross_section)
    if (data_file%n_evt > 0) then
       if (data%n_evt > 0 .and. data_file%n_evt /= data%n_evt)  call err_n_evt
       data%n_evt = data_file%n_evt
    end if
  contains
    subroutine err_weights
      call msg_fatal ("LHEF: mismatch in event weight properties")
    end subroutine err_weights
```

```
    subroutine err_norm
      call msg_fatal ("LHEF: mismatch in event normalization")
    end subroutine err_norm
    subroutine err_beams
      call msg_fatal ("LHEF: mismatch in beam properties")
    end subroutine err_beams
    subroutine err_proc
      call msg_fatal ("LHEF: mismatch in process definitions")
    end subroutine err_proc
    subroutine err_n_evt
      call msg_error ("LHEF: mismatch in specified number of events (ignored)")
    end subroutine err_n_evt
  end subroutine eio_merge_data
```

Switch from input to output: reopen the file for reading.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
  procedure :: switch_inout => eio_lhef_switch_inout
```

⟨*EIO LHEF: procedures*⟩+≡

```
  subroutine eio_lhef_switch_inout (eio, success)
    class(eio_lhef_t), intent(inout) :: eio
    logical, intent(out), optional :: success
    call msg_bug ("LHEF: in-out switch not supported")
    if (present (success))  success = .false.
  end subroutine eio_lhef_switch_inout
```

Split event file: increment the counter, close the current file, open a new one. If the file needs a header, repeat it for the new file. (We assume that the common block contents are still intact.)

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
  procedure :: split_out => eio_lhef_split_out
```

⟨*EIO LHEF: procedures*⟩+≡

```
  subroutine eio_lhef_split_out (eio)
    class(eio_lhef_t), intent(inout) :: eio
    integer :: u
    if (eio%split) then
       eio%split_index = eio%split_index + 1
       call eio%set_filename ()
       write (msg_buffer, "(A,A,A)")  "Events: writing to LHEF file '", &
            char (eio%filename), "'"
       call msg_message ()
       call eio%write_footer ()
       u = eio%unit
       close (u)
       open (u, file = char (eio%filename), &
            action = "write", status = "replace")
       call eio%write_header ()
       call eio%tag_init%write (u);  write (u, *)
       call heprup_write_lhef (u)
       select case (eio%version)
       case ("2.0");  call eio%write_init_20 (eio%data)
       case ("3.0");  call eio%write_init_30 (eio%data)
       end select
```

```
      call eio%tag_init%close (u);  write (u, *)
    end if
  end subroutine eio_lhef_split_out
```

Output an event. Write first the event indices, then weight and squared matrix element, then the particle set.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
  procedure :: output => eio_lhef_output
```

⟨*EIO LHEF: procedures*⟩+≡

```
  subroutine eio_lhef_output (eio, event, i_prc, reading, passed, pacify)
    class(eio_lhef_t), intent(inout) :: eio
    class(generic_event_t), intent(in), target :: event
    integer, intent(in) :: i_prc
    logical, intent(in), optional :: reading, passed, pacify
    integer :: u
    u = given_output_unit (eio%unit);  if (u < 0)  return
    if (present (passed)) then
       if (.not. passed)  return
    end if
    if (eio%writing) then
       call hepeup_from_event (event, &
            process_index = eio%proc_num_id (i_prc), &
            keep_beams = eio%keep_beams, &
            keep_remnants = eio%keep_remnants)
       write (u, '(A)') "<event>"
       call hepeup_write_lhef (eio%unit)
       select case (eio%version)
         case ("2.0");  call eio%write_event_20 (event)
         case ("3.0");  call eio%write_event_30 (event)
       end select
       write (u, '(A)') "</event>"
    else
       call eio%write ()
       call msg_fatal ("LHEF file is not open for writing")
    end if
  end subroutine eio_lhef_output
```

Input an event. Upon input of `i_prc`, we can just read in the whole HEPEUP common block. These data are known to come first. The `i_prc` value can be deduced from the IDPRUP value by a table lookup.

Reading the common block bypasses the `cstream` which accesses the input unit. This is consistent with the LHEF specification. After the common-block data have been swallowed, we can resume reading from stream.

We don't catch actual I/O errors. However, we return a negative value in `iostat` if we reached the terminating `</LesHouchesEvents>` tag.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
  procedure :: input_i_prc => eio_lhef_input_i_prc
```

⟨*EIO LHEF: procedures*⟩+≡

```
  subroutine eio_lhef_input_i_prc (eio, i_prc, iostat)
    class(eio_lhef_t), intent(inout) :: eio
    integer, intent(out) :: i_prc
```

```
    integer, intent(out) :: iostat
    integer :: i, proc_num_id
    type(string_t) :: s
    logical :: ok
    iostat = 0
    call eio%tag_lhef%read_content (eio%cstream, s, ok)
    if (ok) then
       if (s == "") then
          iostat = -1
       else
          call err_close
       end if
       return
    else
       call eio%cstream%revert_record (s)
    end if
    call eio%tag_event%read (eio%cstream, ok)
    if (.not. ok) then
       call err_evt1
       return
    end if
    call hepeup_read_lhef (eio%unit)
    call hepeup_get_event_parameters (proc_id = proc_num_id)
    i_prc = 0
    FIND_I_PRC: do i = 1, size (eio%proc_num_id)
       if (eio%proc_num_id(i) == proc_num_id) then
          i_prc = i
          exit FIND_I_PRC
       end if
    end do FIND_I_PRC
    if (i_prc == 0)  call err_index
  contains
    subroutine err_close
      call msg_error ("LHEF: reading events: syntax error in closing tag")
      iostat = 1
    end subroutine
    subroutine err_evt1
      call msg_error ("LHEF: reading events: invalid event tag, &
           &aborting read")
      iostat = 2
    end subroutine err_evt1
    subroutine err_index
      call msg_error ("LHEF: reading events: undefined process ID " &
           // char (str (proc_num_id)) // ", aborting read")
      iostat = 3
    end subroutine err_index
  end subroutine eio_lhef_input_i_prc
```

Since we have already read the event information from file, this input routine can transfer the common-block contents to the event record. Also, we read any further information in the event record.

Since LHEF doesn't give this information, we must assume that the MCI group, term, and channel can all be safely set to 1. This works if there is only

one MCI group and term. The channel doesn't matter for the matrix element.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
procedure :: input_event => eio_lhef_input_event
```

⟨*EIO LHEF: procedures*⟩+≡
```
subroutine eio_lhef_input_event (eio, event, iostat)
  class(eio_lhef_t), intent(inout) :: eio
  class(generic_event_t), intent(inout), target :: event
  integer, intent(out) :: iostat
  type(string_t) :: s
  logical :: closing
  iostat = 0
  call event%reset ()
  call event%select (1, 1, 1)
  call hepeup_to_event (event, eio%fallback_model, &
       recover_beams = eio%recover_beams, &
       use_alpha_s = eio%use_alpha_s_from_file, &
       use_scale = eio%use_scale_from_file)
  select case (eio%version)
  case ("1.0")
     call eio%tag_event%read_content (eio%cstream, s, closing = closing)
     if (s /= "" .or. .not. closing)  call err_evt2
  case ("2.0");  call eio%read_event_20 (event)
  case ("3.0");  call eio%read_event_30 (event)
  end select
contains
  subroutine err_evt2
    call msg_error ("LHEF: reading events: syntax error in event record, &
         &aborting read")
    iostat = 2
  end subroutine err_evt2

end subroutine eio_lhef_input_event
```

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
procedure :: skip => eio_lhef_skip
```

⟨*EIO LHEF: procedures*⟩+≡
```
subroutine eio_lhef_skip (eio, iostat)
  class(eio_lhef_t), intent(inout) :: eio
  integer, intent(out) :: iostat
  if (eio%reading) then
     read (eio%unit, iostat = iostat)
  else
     call eio%write ()
     call msg_fatal ("Raw event file is not open for reading")
  end if
end subroutine eio_lhef_skip
```

17.13.4 Les Houches Event File: header/footer

These two routines write the header and footer for the Les Houches Event File format (LHEF).

The current version writes no information except for the generator name and version (v.1.0 only).

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
procedure :: write_header => eio_lhef_write_header
procedure :: write_footer => eio_lhef_write_footer
```

⟨*EIO LHEF: procedures*⟩+≡

```
subroutine eio_lhef_write_header (eio)
  class(eio_lhef_t), intent(in) :: eio
  integer :: u
  u = given_output_unit (eio%unit);  if (u < 0)  return
  call eio%tag_lhef%write (u);  write (u, *)
  call eio%tag_head%write (u);  write (u, *)
  select case (eio%version)
  case ("1.0")
     write (u, "(2x)", advance = "no")
     call eio%tag_gen_n%write (var_str ("WHIZARD"), u)
     write (u, *)
     write (u, "(2x)", advance = "no")
     call eio%tag_gen_v%write (var_str ("⟨Version⟩"), u)
     write (u, *)
  end select
  call eio%tag_head%close (u);  write (u, *)
end subroutine eio_lhef_write_header

subroutine eio_lhef_write_footer (eio)
  class(eio_lhef_t), intent(in) :: eio
  integer :: u
  u = given_output_unit (eio%unit);  if (u < 0)  return
  call eio%tag_lhef%close (u)
end subroutine eio_lhef_write_footer
```

Reading the header just means finding the tags and ignoring any contents. When done, we should stand just after the header tag.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
procedure :: read_header => eio_lhef_read_header
```

⟨*EIO LHEF: procedures*⟩+≡

```
subroutine eio_lhef_read_header (eio)
  class(eio_lhef_t), intent(inout) :: eio
  logical :: success, closing
  type(string_t) :: content
  call eio%tag_lhef%read (eio%cstream, success)
  if (.not. success .or. .not. eio%tag_lhef%has_content)  call err_lhef
  if (eio%tag_lhef%get_attribute (1) /= eio%version)  call err_version
  call eio%tag_head%read (eio%cstream, success)
  if (.not. success)  call err_header
  if (eio%tag_head%has_content) then
     SKIP_HEADER_CONTENT: do
        call eio%tag_head%read_content (eio%cstream, content, closing)
        if (closing)  exit SKIP_HEADER_CONTENT
     end do SKIP_HEADER_CONTENT
  end if
contains
```

```
    subroutine err_lhef
      call msg_fatal ("LHEF: LesHouchesEvents tag absent or corrupted")
    end subroutine err_lhef
    subroutine err_header
      call msg_fatal ("LHEF: header tag absent or corrupted")
    end subroutine err_header
    subroutine err_version
       call msg_error ("LHEF: version mismatch: expected " &
            // eio%version // ", found " &
            // char (eio%tag_lhef%get_attribute (1)))
    end subroutine err_version
  end subroutine eio_lhef_read_header
```

17.13.5 Version-Specific Code: 1.0

In version 1.0, the init tag contains just HEPRUP data. While a `cstream` is connected to the input unit, we bypass it temporarily for the purpose of reading the HEPRUP contents. This is consistent with the LHEF standard.

This routine does not read the closing tag of the init block.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
  procedure :: read_init_10 => eio_lhef_read_init_10
```

⟨*EIO LHEF: procedures*⟩+≡

```
  subroutine eio_lhef_read_init_10 (eio, data)
    class(eio_lhef_t), intent(in) :: eio
    type(event_sample_data_t), intent(out) :: data
    integer :: n_proc, i
    call heprup_read_lhef (eio%unit)
    call heprup_get_run_parameters (n_processes = n_proc)
    call data%init (n_proc)
    data%n_beam = 2
    call heprup_get_run_parameters ( &
         unweighted = data%unweighted, &
         negative_weights = data%negative_weights, &
         beam_pdg = data%pdg_beam, &
         beam_energy = data%energy_beam)
    if (data%unweighted) then
       data%norm_mode = NORM_UNIT
    else
       data%norm_mode = NORM_SIGMA
    end if
    do i = 1, n_proc
       call heprup_get_process_parameters (i, &
            process_id = data%proc_num_id(i), &
            cross_section = data%cross_section(i), &
            error = data%error(i))
    end do
  end subroutine eio_lhef_read_init_10
```

17.13.6 Version-Specific Code: 2.0

This is the init information for the 2.0 format, after the HEPRUP data. We have the following tags:

- `generator` Generator name and version.
- `xsecinfo` Cross section and weights data. We have the total cross section and number of events (assuming that the event file is intact), but information on minimum and maximum weights is not available before the file is complete. We just write the mandatory tags. (Note that the default values of the other tags describe a uniform unit weight, but we can determine most values only after the sample is complete.)
- `cutsinfo` This optional tag is too specific to represent the possibilities of WHIZARD, so we skip it.
- `procinfo` This optional tag is useful for giving details of NLO calculations. Skipped.
- `mergetype` Optional, also not applicable.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
procedure :: write_init_20 => eio_lhef_write_init_20
```

⟨*EIO LHEF: procedures*⟩+≡
```
subroutine eio_lhef_write_init_20 (eio, data)
  class(eio_lhef_t), intent(in) :: eio
  type(event_sample_data_t), intent(in) :: data
  integer :: u
  u = eio%unit
  call eio%tag_generator%write (u)
  write (u, "(A)", advance="no")  "WHIZARD"
  call eio%tag_generator%close (u);  write (u, *)
  call eio%tag_xsecinfo%write (u);  write (u, *)
end subroutine eio_lhef_write_init_20
```

When reading the init block, we first call the 1.0 routine that fills HEPRUP. Then we consider the possible tags. Only the `generator` and `xsecinfo` tags are of interest. We skip everything else except for the closing tag.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡
```
procedure :: read_init_20 => eio_lhef_read_init_20
```

⟨*EIO LHEF: procedures*⟩+≡
```
subroutine eio_lhef_read_init_20 (eio, data)
  class(eio_lhef_t), intent(inout) :: eio
  type(event_sample_data_t), intent(out) :: data
  real(default), parameter :: pb_per_fb = 1.e-3_default
  type(string_t) :: content
  logical :: found, closing
  call eio_lhef_read_init_10 (eio, data)
  SCAN_INIT_TAGS: do
     call eio%tag_generator%read (eio%cstream, found)
     if (found) then
        if (.not. eio%tag_generator%has_content)  call err_generator
```

```
            call eio%tag_generator%read_content (eio%cstream, content, closing)
            call msg_message ("LHEF: Event file has been generated by " &
                 // char (content) // " " &
                 // char (eio%tag_generator%get_attribute (1)))
            cycle SCAN_INIT_TAGS
         end if
         call eio%tag_xsecinfo%read (eio%cstream, found)
         if (found) then
            if (eio%tag_xsecinfo%has_content)  call err_xsecinfo
            cycle SCAN_INIT_TAGS
         end if
         call eio%tag_init%read_content (eio%cstream, content, closing)
         if (closing) then
            if (content /= "")  call err_init
            exit SCAN_INIT_TAGS
         end if
      end do SCAN_INIT_TAGS
      data%n_evt = &
           read_ival (eio%tag_xsecinfo%get_attribute (1))
      data%total_cross_section = &
           read_rval (eio%tag_xsecinfo%get_attribute (2)) / pb_per_fb
   contains
      subroutine err_generator
        call msg_fatal ("LHEF: invalid generator tag")
      end subroutine err_generator
      subroutine err_xsecinfo
        call msg_fatal ("LHEF: invalid xsecinfo tag")
      end subroutine err_xsecinfo
      subroutine err_init
        call msg_fatal ("LHEF: syntax error after init tag")
      end subroutine err_init
   end subroutine eio_lhef_read_init_20
```

This is additional event-specific information for the 2.0 format, after the HEPEUP data. We can specify weights, starting from the master weight and adding alternative weights. The alternative weights are collected in a common tag.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
   procedure :: write_event_20 => eio_lhef_write_event_20
```

⟨*EIO LHEF: procedures*⟩+≡

```
   subroutine eio_lhef_write_event_20 (eio, event)
      class(eio_lhef_t), intent(in) :: eio
      class(generic_event_t), intent(in) :: event
      type(string_t) :: s
      integer :: i, u
      u = eio%unit
      if (eio%write_sqme_ref) then
         s = str (event%get_sqme_ref ())
         call eio%tag_sqme_ref%write (s, u);  write (u, *)
      end if
      if (eio%write_sqme_prc) then
         s = str (event%get_sqme_prc ())
         call eio%tag_sqme_prc%write (s, u);  write (u, *)
      end if
```

```
    if (eio%n_alt > 0) then
       if (eio%write_sqme_alt) then
          s = str (event%get_sqme_alt(1))
          do i = 2, eio%n_alt
             s = s // " " // str (event%get_sqme_alt(i));  write (u, *)
          end do
          call eio%tag_sqme_alt(1)%write (s, u)
       end if
       s = str (event%get_weight_alt(1))
       do i = 2, eio%n_alt
          s = s // " " // str (event%get_weight_alt(i));  write (u, *)
       end do
       call eio%tag_wgts_alt(1)%write (s, u)
    end if
  end subroutine eio_lhef_write_event_20
```

Read extra event data. If there is a weight entry labeled `sqme_prc`, we take this as the squared matrix-element value (the new *reference* value `sqme_ref`). Other tags, including tags written by the above writer, are skipped.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
  procedure :: read_event_20 => eio_lhef_read_event_20
```

⟨*EIO LHEF: procedures*⟩+≡

```
  subroutine eio_lhef_read_event_20 (eio, event)
    class(eio_lhef_t), intent(inout) :: eio
    class(generic_event_t), intent(inout) :: event
    type(string_t) :: content
    logical :: found, closing
    SCAN_EVENT_TAGS: do
       call eio%tag_weight%read (eio%cstream, found)
       if (found) then
          if (.not. eio%tag_weight%has_content)  call err_weight
          call eio%tag_weight%read_content (eio%cstream, content, closing)
          if (.not. closing)  call err_weight
          if (eio%tag_weight%get_attribute (1) == "sqme_prc") then
             call event%set_sqme_ref (read_rval (content))
          end if
          cycle SCAN_EVENT_TAGS
       end if
       call eio%tag_event%read_content (eio%cstream, content, closing)
       if (closing) then
          if (content /= "")  call err_event
          exit SCAN_EVENT_TAGS
       end if
    end do SCAN_EVENT_TAGS
  contains
    subroutine err_weight
      call msg_fatal ("LHEF: invalid weight tag in event record")
    end subroutine err_weight
    subroutine err_event
      call msg_fatal ("LHEF: syntax error after event tag")
    end subroutine err_event
  end subroutine eio_lhef_read_event_20
```

17.13.7 Version-Specific Code: 3.0

This is the init information for the 3.0 format, after the HEPRUP data. We have the following tags:

- `generator` Generator name and version.
- `xsecinfo` Cross section and weights data. We have the total cross section and number of events (assuming that the event file is intact), but information on minimum and maximum weights is not available before the file is complete. We just write the mandatory tags. (Note that the default values of the other tags describe a uniform unit weight, but we can determine most values only after the sample is complete.)
- `cutsinfo` This optional tag is too specific to represent the possibilities of WHIZARD, so we skip it.
- `procinfo` This optional tag is useful for giving details of NLO calculations. Skipped.
- `weightinfo` Determine the meaning of optional weights, whose values are given in the event record.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
procedure :: write_init_30 => eio_lhef_write_init_30
```

⟨*EIO LHEF: procedures*⟩+≡

```
subroutine eio_lhef_write_init_30 (eio, data)
  class(eio_lhef_t), intent(in) :: eio
  type(event_sample_data_t), intent(in) :: data
  integer :: u, i
  u = given_output_unit (eio%unit)
  call eio%tag_generator%write (u)
  write (u, "(A)", advance="no")  "WHIZARD"
  call eiO%tag_generator%close (u);  write (u, *)
  call eio%tag_xsecinfo%write (u);  write (u, *)
  if (eio%write_sqme_ref) then
     call eio%tag_sqme_ref%write (u);  write (u, *)
  end if
  if (eio%write_sqme_prc) then
     call eio%tag_sqme_prc%write (u);  write (u, *)
  end if
  if (eio%write_sqme_alt) then
     do i = 1, eio%n_alt
        call eio%tag_sqme_alt(i)%write (u);  write (u, *)
     end do
  end if
  do i = 1, eio%n_alt
     call eio%tag_wgts_alt(i)%write (u);  write (u, *)
  end do
end subroutine eio_lhef_write_init_30
```

When reading the init block, we first call the 1.0 routine that fills HEPRUP. Then we consider the possible tags. Only the `generator` and `xsecinfo` tags are of interest. We skip everything else except for the closing tag.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
procedure :: read_init_30 => eio_lhef_read_init_30
```

⟨*EIO LHEF: procedures*⟩+≡

```
subroutine eio_lhef_read_init_30 (eio, data)
  class(eio_lhef_t), intent(inout) :: eio
  type(event_sample_data_t), intent(out) :: data
  real(default), parameter :: pb_per_fb = 1.e-3_default
  type(string_t) :: content
  logical :: found, closing
  integer :: n_weightinfo
  call eio_lhef_read_init_10 (eio, data)
  n_weightinfo = 0
  eio%i_weight_sqme = 0
  SCAN_INIT_TAGS: do
     call eio%tag_generator%read (eio%cstream, found)
     if (found) then
        if (.not. eio%tag_generator%has_content)  call err_generator
        call eio%tag_generator%read_content (eio%cstream, content, closing)
        call msg_message ("LHEF: Event file has been generated by " &
             // char (content) // " " &
             // char (eio%tag_generator%get_attribute (1)))
        cycle SCAN_INIT_TAGS
     end if
     call eio%tag_xsecinfo%read (eio%cstream, found)
     if (found) then
        if (eio%tag_xsecinfo%has_content)  call err_xsecinfo
        cycle SCAN_INIT_TAGS
     end if
     call eio%tag_weightinfo%read (eio%cstream, found)
     if (found) then
        if (eio%tag_weightinfo%has_content)  call err_xsecinfo
        n_weightinfo = n_weightinfo + 1
        if (eio%tag_weightinfo%get_attribute (1) == "sqme_prc") then
           eio%i_weight_sqme = n_weightinfo
        end if
        cycle SCAN_INIT_TAGS
     end if
     call eio%tag_init%read_content (eio%cstream, content, closing)
     if (closing) then
        if (content /= "")  call err_init
        exit SCAN_INIT_TAGS
     end if
  end do SCAN_INIT_TAGS
  data%n_evt = &
       read_ival (eio%tag_xsecinfo%get_attribute (1))
  data%total_cross_section = &
       read_rval (eio%tag_xsecinfo%get_attribute (2)) / pb_per_fb
contains
  subroutine err_generator
    call msg_fatal ("LHEF: invalid generator tag")
  end subroutine err_generator
  subroutine err_xsecinfo
    call msg_fatal ("LHEF: invalid xsecinfo tag")
  end subroutine err_xsecinfo
  subroutine err_init
```

```
      call msg_fatal ("LHEF: syntax error after init tag")
    end subroutine err_init
  end subroutine eio_lhef_read_init_30
```

This is additional event-specific information for the 3.0 format, after the HEPEUP data. We can specify weights, starting from the master weight and adding alternative weights. The weight tags are already allocated, so we just have to transfer the weight values to strings, assemble them and write them to file. All weights are collected in a single tag.

Note: If efficiency turns out to be an issue, we may revert to traditional character buffer writing. However, we need to know the maximum length.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
  procedure :: write_event_30 => eio_lhef_write_event_30
```

⟨*EIO LHEF: procedures*⟩+≡

```
  subroutine eio_lhef_write_event_30 (eio, event)
    class(eio_lhef_t), intent(in) :: eio
    class(generic_event_t), intent(in) :: event
    type(string_t) :: s
    integer :: u, i
    u = eio%unit
    s = ""
    if (eio%write_sqme_ref) then
       s = s // str (event%get_sqme_ref ()) // " "
    end if
    if (eio%write_sqme_prc) then
       s = s // str (event%get_sqme_prc ()) // " "
    end if
    if (eio%n_alt > 0) then
       if (eio%write_sqme_alt) then
          s = s // str (event%get_sqme_alt(1)) // " "
          do i = 2, eio%n_alt
             s = s // str (event%get_sqme_alt(i)) // " "
          end do
       end if
       s = s // str (event%get_weight_alt(1)) // " "
       do i = 2, eio%n_alt
          s = s // str (event%get_weight_alt(i)) // " "
       end do
    end if
    if (len_trim (s) > 0) then
       call eio%tag_weights%write (trim (s), u);  write (u, *)
    end if
  end subroutine eio_lhef_write_event_30
```

Read extra event data. If there is a `weights` tag and if there was a `weightinfo` entry labeled `sqme_prc`, we extract the corresponding entry from the weights string and store this as the event's squared matrix-element value. Other tags, including tags written by the above writer, are skipped.

⟨*EIO LHEF: eio lhef: TBP*⟩+≡

```
  procedure :: read_event_30 => eio_lhef_read_event_30
```

⟨*EIO LHEF: procedures*⟩+≡

```
subroutine eio_lhef_read_event_30 (eio, event)
  class(eio_lhef_t), intent(inout) :: eio
  class(generic_event_t), intent(inout) :: event
  type(string_t) :: content, string
  logical :: found, closing
  integer :: i
  SCAN_EVENT_TAGS: do
     call eio%tag_weights%read (eio%cstream, found)
     if (found) then
        if (.not. eio%tag_weights%has_content)  call err_weights
        call eio%tag_weights%read_content (eio%cstream, content, closing)
        if (.not. closing)  call err_weights
        if (eio%i_weight_sqme > 0) then
           SCAN_WEIGHTS: do i = 1, eio%i_weight_sqme
              call split (content, string, " ")
              content = adjustl (content)
              if (i == eio%i_weight_sqme) then
                 call event%set_sqme_ref (read_rval (string))
                 exit SCAN_WEIGHTS
              end if
           end do SCAN_WEIGHTS
        end if
        cycle SCAN_EVENT_TAGS
     end if
     call eio%tag_event%read_content (eio%cstream, content, closing)
     if (closing) then
        if (content /= "")  call err_event
        exit SCAN_EVENT_TAGS
     end if
  end do SCAN_EVENT_TAGS
contains
  subroutine err_weights
    call msg_fatal ("LHEF: invalid weights tag in event record")
  end subroutine err_weights
  subroutine err_event
    call msg_fatal ("LHEF: syntax error after event tag")
  end subroutine err_event
end subroutine eio_lhef_read_event_30
```

17.13.8 Unit tests

Test module, followed by the corresponding implementation module.

⟨eio_lhef_ut.f90⟩≡

```
⟨File header⟩

module eio_lhef_ut
  use unit_tests
  use eio_lhef_uti

⟨Standard module head⟩
```

```
  ⟨EIO LHEF: public test⟩

  contains

  ⟨EIO LHEF: test driver⟩

  end module eio_lhef_ut
```

⟨eio_lhef_uti.f90⟩≡

```
  ⟨File header⟩

  module eio_lhef_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use io_units
    use model_data
    use event_base
    use eio_data
    use eio_base

    use eio_lhef

    use eio_base_ut, only: eio_prepare_test, eio_cleanup_test
    use eio_base_ut, only: eio_prepare_fallback_model, eio_cleanup_fallback_model

  ⟨Standard module head⟩

  ⟨EIO LHEF: test declarations⟩

  contains

  ⟨EIO LHEF: tests⟩

  end module eio_lhef_uti
```

API: driver for the unit tests below.

⟨*EIO LHEF: public test*⟩≡

```
    public :: eio_lhef_test
```

⟨*EIO LHEF: test driver*⟩≡

```
    subroutine eio_lhef_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨EIO LHEF: execute tests⟩
    end subroutine eio_lhef_test
```

Version 1.0 Output

We test the implementation of all I/O methods. We start with output according to version 1.0.

⟨*EIO LHEF: execute tests*⟩≡

```
    call test (eio_lhef_1, "eio_lhef_1", &
         "write version 1.0", &
```

```
        u, results)
```

⟨*EIO LHEF: test declarations*⟩≡

```
    public :: eio_lhef_1
```

⟨*EIO LHEF: tests*⟩≡

```
    subroutine eio_lhef_1 (u)
      integer, intent(in) :: u
      class(generic_event_t), pointer :: event
      type(event_sample_data_t) :: data
      class(eio_t), allocatable :: eio
      type(string_t) :: sample
      integer :: u_file, iostat
      character(80) :: buffer

      write (u, "(A)")  "* Test output: eio_lhef_1"
      write (u, "(A)")  "*   Purpose: generate an event and write weight to file"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize test process"

      call eio_prepare_test (event, unweighted = .false.)

      call data%init (1)
      data%n_evt = 1
      data%n_beam = 2
      data%unweighted = .true.
      data%norm_mode = NORM_UNIT
      data%pdg_beam = 25
      data%energy_beam = 500
      data%proc_num_id = [42]
      data%cross_section(1) = 100
      data%error(1) = 1
      data%total_cross_section = sum (data%cross_section)

      write (u, "(A)")
      write (u, "(A)")  "* Generate and write an event"
      write (u, "(A)")

      sample = "eio_lhef_1"

      allocate (eio_lhef_t :: eio)
      select type (eio)
      type is (eio_lhef_t)
         call eio%set_parameters ()
      end select

      call eio%init_out (sample, data)
      call event%generate (1, [0._default, 0._default])

      call eio%output (event, i_prc = 1)
      call eio%write (u)
      call eio%final ()

      write (u, "(A)")
```

```
    write (u, "(A)")  "* File contents:"
    write (u, "(A)")

    u_file = free_unit ()
    open (u_file, file = char (sample // "." // eio%extension), &
         action = "read", status = "old")
    do
       read (u_file, "(A)", iostat = iostat)  buffer
       if (buffer(1:21) == "  <generator_version>")  buffer = "[...]"
       if (iostat /= 0)  exit
       write (u, "(A)") trim (buffer)
    end do
    close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Reset data"
    write (u, "(A)")

    deallocate (eio)
    allocate (eio_lhef_t :: eio)
    select type (eio)
    type is (eio_lhef_t)
       call eio%set_parameters ()
    end select

    select type (eio)
    type is (eio_lhef_t)
       call eio%set_parameters (keep_beams = .true.)
    end select
    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_lhef_1"

  end subroutine eio_lhef_1
```

Version 2.0 Output

Version 2.0 has added a lot of options to the LHEF format. We implement some of them.

⟨*EIO LHEF: execute tests*⟩+≡

```
  call test (eio_lhef_2, "eio_lhef_2", &
       "write version 2.0", &
       u, results)
```

⟨*EIO LHEF: test declarations*⟩+≡

```
  public :: eio_lhef_2
```

⟨*EIO LHEF: tests*⟩+≡

```
subroutine eio_lhef_2 (u)
  integer, intent(in) :: u
  class(generic_event_t), pointer :: event
  type(event_sample_data_t) :: data
  class(eio_t), allocatable :: eio
  type(string_t) :: sample
  integer :: u_file, iostat
  character(80) :: buffer

  write (u, "(A)")  "* Test output: eio_lhef_2"
  write (u, "(A)")  "*   Purpose: generate an event and write weight to file"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize test process"

  call eio_prepare_test (event, unweighted = .false.)

  call data%init (1)
  data%unweighted = .false.
  data%norm_mode = NORM_SIGMA
  data%n_evt = 1
  data%n_beam = 2
  data%pdg_beam = 25
  data%energy_beam = 500
  data%proc_num_id = [42]
  data%cross_section(1) = 100
  data%error(1) = 1
  data%total_cross_section = sum (data%cross_section)

  write (u, "(A)")
  write (u, "(A)")  "* Generate and write an event"
  write (u, "(A)")

  sample = "eio_lhef_2"

  allocate (eio_lhef_t :: eio)
  select type (eio)
  type is (eio_lhef_t)
     call eio%set_parameters (version = "2.0", write_sqme_prc = .true.)
  end select

  call eio%init_out (sample, data)
  call event%generate (1, [0._default, 0._default])


  call eio%output (event, i_prc = 1)
  call eio%write (u)
  call eio%final ()

  write (u, "(A)")
  write (u, "(A)")  "* File contents:"
  write (u, "(A)")
```

```
    u_file = free_unit ()
    open (u_file, file = char (sample // "." // eio%extension), &
         action = "read", status = "old")
    do
       read (u_file, "(A)", iostat = iostat)  buffer
       if (buffer(1:10) == "<generator")  buffer = "[...]"
       if (iostat /= 0)  exit
       write (u, "(A)") trim (buffer)
    end do
    close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_lhef_2"

  end subroutine eio_lhef_2
```

Version 3.0 Output

Version 3.0 is an update which removes some tags (which we didn't use anyway) and suggests a new treatment of weights.

⟨*EIO LHEF: execute tests*⟩+≡

```
  call test (eio_lhef_3, "eio_lhef_3", &
       "write version 3.0", &
       u, results)
```

⟨*EIO LHEF: test declarations*⟩+≡

```
  public :: eio_lhef_3
```

⟨*EIO LHEF: tests*⟩+≡

```
  subroutine eio_lhef_3 (u)
    integer, intent(in) :: u
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat
    character(80) :: buffer

    write (u, "(A)")  "* Test output: eio_lhef_3"
    write (u, "(A)")  "*   Purpose: generate an event and write weight to file"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize test process"

    call eio_prepare_test (event, unweighted = .false.)

    call data%init (1)
    data%unweighted = .false.
```

```
  data%norm_mode = NORM_SIGMA
  data%n_evt = 1
  data%n_beam = 2
  data%pdg_beam = 25
  data%energy_beam = 500
  data%proc_num_id = [42]
  data%cross_section(1) = 100
  data%error(1) = 1
  data%total_cross_section = sum (data%cross_section)

  write (u, "(A)")
  write (u, "(A)")  "* Generate and write an event"
  write (u, "(A)")

  sample = "eio_lhef_3"

  allocate (eio_lhef_t :: eio)
  select type (eio)
  type is (eio_lhef_t)
     call eio%set_parameters (version = "3.0", write_sqme_prc = .true.)
  end select

  call eio%init_out (sample, data)
  call event%generate (1, [0._default, 0._default])


  call eio%output (event, i_prc = 1)
  call eio%write (u)
  call eio%final ()

  write (u, "(A)")
  write (u, "(A)")  "* File contents:"
  write (u, "(A)")

  u_file = free_unit ()
  open (u_file, file = char (sample // ".lhe"), &
       action = "read", status = "old")
  do
     read (u_file, "(A)", iostat = iostat)  buffer
     if (buffer(1:10) == "<generator")  buffer = "[...]"
     if (iostat /= 0)  exit
     write (u, "(A)") trim (buffer)
  end do
  close (u_file)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call eio_cleanup_test (event)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: eio_lhef_3"

end subroutine eio_lhef_3
```

Version 1.0 Input

Check input of a version-1.0 conforming LHEF file.

⟨*EIO LHEF: execute tests*⟩+≡

```
call test (eio_lhef_4, "eio_lhef_4", &
     "read version 1.0", &
     u, results)
```

⟨*EIO LHEF: test declarations*⟩+≡

```
public :: eio_lhef_4
```

⟨*EIO LHEF: tests*⟩+≡

```
subroutine eio_lhef_4 (u)
  integer, intent(in) :: u
  class(model_data_t), pointer :: fallback_model
  class(generic_event_t), pointer :: event
  type(event_sample_data_t) :: data
  class(eio_t), allocatable :: eio
  type(string_t) :: sample
  integer :: u_file, iostat, i_prc

  write (u, "(A)")  "* Test output: eio_lhef_4"
  write (u, "(A)")  "*   Purpose: read a LHEF 1.0 file"
  write (u, "(A)")

  write (u, "(A)")  "* Write a LHEF data file"
  write (u, "(A)")

  u_file = free_unit ()
  sample = "eio_lhef_4"
  open (u_file, file = char (sample // ".lhe"), &
       status = "replace", action = "readwrite")

  write (u_file, "(A)")  '<LesHouchesEvents version="1.0">'
  write (u_file, "(A)")  '<header>'
  write (u_file, "(A)")  '  <arbitrary_tag opt="foo">content</arbitrary_tag>'
  write (u_file, "(A)")  '  Text'
  write (u_file, "(A)")  '  <another_tag />'
  write (u_file, "(A)")  '</header>'
  write (u_file, "(A)")  '<init>'
  write (u_file, "(A)")  ' 25 25  5.0000000000E+02  5.0000000000E+02 &
       & -1 -1 -1 -1 3 1'
  write (u_file, "(A)")  '  1.0000000000E-01  1.0000000000E-03 &
       & 1.0000000000E+00 42'
  write (u_file, "(A)")  '</init>'
  write (u_file, "(A)")  '<event>'
  write (u_file, "(A)")  ' 4 42  3.0574068604E+08  1.0000000000E+03 &
       & -1.0000000000E+00 -1.0000000000E+00'
  write (u_file, "(A)")  ' 25 -1 0 0 0 0  0.0000000000E+00  0.0000000000E+00 &
       & 4.8412291828E+02  5.0000000000E+02  1.2500000000E+02 &
       & 0.0000000000E+00  9.0000000000E+00'
  write (u_file, "(A)")  ' 25 -1 0 0 0 0  0.0000000000E+00  0.0000000000E+00 &
       &-4.8412291828E+02  5.0000000000E+02  1.2500000000E+02 &
```

```
     & 0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  ' 25 1 1 2 0 0 -1.4960220911E+02 -4.6042825611E+02 &
     & 0.0000000000E+00  5.0000000000E+02  1.2500000000E+02 &
     & 0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  ' 25 1 1 2 0 0  1.4960220911E+02  4.6042825611E+02 &
     & 0.0000000000E+00  5.0000000000E+02  1.2500000000E+02 &
     & 0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  '</event>'
write (u_file, "(A)")  '</LesHouchesEvents>'
close (u_file)


write (u, "(A)")  "* Initialize test process"
write (u, "(A)")

call eio_prepare_fallback_model (fallback_model)
call eio_prepare_test (event, unweighted = .false.)

allocate (eio_lhef_t :: eio)
select type (eio)
type is (eio_lhef_t)
   call eio%set_parameters (recover_beams = .false.)
end select
call eio%set_fallback_model (fallback_model)

call data%init (1)
data%n_beam = 2
data%unweighted = .true.
data%norm_mode = NORM_UNIT
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
call data%write (u)
write (u, *)

write (u, "(A)")  "* Initialize and read header"
write (u, "(A)")

call eio%init_in (sample, data)
call eio%write (u)

write (u, *)

select type (eio)
type is (eio_lhef_t)
   call eio%tag_lhef%write (u);  write (u, *)
end select

write (u, *)
call data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Read event"
write (u, "(A)")
```

```
    call eio%input_i_prc (i_prc, iostat)

    select type (eio)
    type is (eio_lhef_t)
       write (u, "(A,I0,A,I0)")  "Found process #", i_prc, &
            " with ID = ", eio%proc_num_id(i_prc)
    end select

    call eio%input_event (event, iostat)

    call event%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Read closing"
    write (u, "(A)")

    call eio%input_i_prc (i_prc, iostat)
    write (u, "(A,I0)")  "iostat = ", iostat

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio%final ()

    call eio_cleanup_test (event)
    call eio_cleanup_fallback_model (fallback_model)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_lhef_4"

  end subroutine eio_lhef_4
```

Version 2.0 Input

Check input of a version-2.0 conforming LHEF file.

⟨*EIO LHEF: execute tests*⟩+≡

```
  call test (eio_lhef_5, "eio_lhef_5", &
       "read version 2.0", &
       u, results)
```

⟨*EIO LHEF: test declarations*⟩+≡

```
  public :: eio_lhef_5
```

⟨*EIO LHEF: tests*⟩+≡

```
  subroutine eio_lhef_5 (u)
    integer, intent(in) :: u
    class(model_data_t), pointer :: fallback_model
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat, i_prc
```

```
write (u, "(A)")  "* Test output: eio_lhef_5"
write (u, "(A)")  "*   Purpose: read a LHEF 2.0 file"
write (u, "(A)")

write (u, "(A)")  "* Write a LHEF data file"
write (u, "(A)")

u_file = free_unit ()
sample = "eio_lhef_5"
open (u_file, file = char (sample // ".lhe"), &
     status = "replace", action = "readwrite")

write (u_file, "(A)")  '<LesHouchesEvents version="2.0">'
write (u_file, "(A)")  '<header>'
write (u_file, "(A)")  '</header>'
write (u_file, "(A)")  '<init>'
write (u_file, "(A)")  ' 25 25  5.0000000000E+02  5.0000000000E+02 &
     &-1 -1 -1 -1 4 1'
write (u_file, "(A)")  '  1.0000000000E-01  1.0000000000E-03 &
     & 0.0000000000E+00 42'
write (u_file, "(A)")  '<generator version="2.2.3">WHIZARD&
     &</generator>'
write (u_file, "(A)")  '<xsecinfo neve="1" totxsec="1.0000000000E-01" />'
write (u_file, "(A)")  '</init>'
write (u_file, "(A)")  '<event>'
write (u_file, "(A)")  ' 4 42  3.0574068604E+08  1.0000000000E+03 &
     &-1.0000000000E+00 -1.0000000000E+00'
write (u_file, "(A)")  ' 25 -1 0 0 0 0  0.0000000000E+00 &
     & 0.0000000000E+00  4.8412291828E+02  5.0000000000E+02 &
     & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  ' 25 -1 0 0 0 0  0.0000000000E+00 &
     & 0.0000000000E+00 -4.8412291828E+02  5.0000000000E+02 &
     & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  ' 25 1 1 2 0 0 -1.4960220911E+02 &
     &-4.6042825611E+02  0.0000000000E+00  5.0000000000E+02 &
     & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  ' 25 1 1 2 0 0  1.4960220911E+02 &
     & 4.6042825611E+02  0.0000000000E+00  5.0000000000E+02 &
     & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  '<weight name="sqme_prc">1.0000000000E+00</weight>'
write (u_file, "(A)")  '</event>'
write (u_file, "(A)")  '</LesHouchesEvents>'
close (u_file)

write (u, "(A)")  "* Initialize test process"
write (u, "(A)")

call eio_prepare_fallback_model (fallback_model)
call eio_prepare_test (event, unweighted = .false.)

allocate (eio_lhef_t :: eio)
select type (eio)
type is (eio_lhef_t)
   call eio%set_parameters (version = "2.0", recover_beams = .false.)
```

```
end select
call eio%set_fallback_model (fallback_model)

call data%init (1)
data%unweighted = .false.
data%norm_mode = NORM_SIGMA
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
call data%write (u)
write (u, *)

write (u, "(A)")  "* Initialize and read header"
write (u, "(A)")

call eio%init_in (sample, data)
call eio%write (u)

write (u, *)

select type (eio)
type is (eio_lhef_t)
   call eio%tag_lhef%write (u);  write (u, *)
end select

write (u, *)
call data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Read event"
write (u, "(A)")

call eio%input_i_prc (i_prc, iostat)

select type (eio)
type is (eio_lhef_t)
   write (u, "(A,I0,A,I0)")  "Found process #", i_prc, &
        " with ID = ", eio%proc_num_id(i_prc)
end select

call eio%input_event (event, iostat)

call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Read closing"
write (u, "(A)")

call eio%input_i_prc (i_prc, iostat)
write (u, "(A,I0)")  "iostat = ", iostat

write (u, "(A)")
write (u, "(A)")  "* Cleanup"
```

```
    call eio%final ()

    call eio_cleanup_test (event)
    call eio_cleanup_fallback_model (fallback_model)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_lhef_5"

  end subroutine eio_lhef_5
```

Version 3.0 Input

Check input of a version-3.0 conforming LHEF file.

⟨*EIO LHEF: execute tests*⟩+≡

```
  call test (eio_lhef_6, "eio_lhef_6", &
       "read version 3.0", &
       u, results)
```

⟨*EIO LHEF: test declarations*⟩+≡

```
  public :: eio_lhef_6
```

⟨*EIO LHEF: tests*⟩+≡

```
  subroutine eio_lhef_6 (u)
    integer, intent(in) :: u
    class(model_data_t), pointer :: fallback_model
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat, i_prc

    write (u, "(A)")  "* Test output: eio_lhef_6"
    write (u, "(A)")  "*   Purpose: read a LHEF 3.0 file"
    write (u, "(A)")

    write (u, "(A)")  "* Write a LHEF data file"
    write (u, "(A)")

    u_file = free_unit ()
    sample = "eio_lhef_6"
    open (u_file, file = char (sample // ".lhe"), &
         status = "replace", action = "readwrite")

    write (u_file, "(A)")  '<LesHouchesEvents version="3.0">'
    write (u_file, "(A)")  '<header>'
    write (u_file, "(A)")  '</header>'
    write (u_file, "(A)")  '<init>'
    write (u_file, "(A)")  ' 25 25  5.0000000000E+02  5.0000000000E+02 &
         &-1 -1 -1 -1 4 1'
    write (u_file, "(A)")  '  1.0000000000E-01  1.0000000000E-03 &
         & 0.0000000000E+00 42'
    write (u_file, "(A)")  '<generator version="2.2.3">WHIZARD&
         &</generator>'
```

```
write (u_file, "(A)")  '<xsecinfo neve="1" totxsec="1.0000000000E-01" />'
write (u_file, "(A)")  '<weightinfo name="sqme_prc" />'
write (u_file, "(A)")  '</init>'
write (u_file, "(A)")  '<event>'
write (u_file, "(A)")  ' 4 42  3.0574068604E+08  1.0000000000E+03 &
     &-1.0000000000E+00 -1.0000000000E+00'
write (u_file, "(A)")  ' 25 -1 0 0 0 0  0.0000000000E+00 &
     & 0.0000000000E+00  4.8412291828E+02  5.0000000000E+02 &
     & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  ' 25 -1 0 0 0 0  0.0000000000E+00 &
     & 0.0000000000E+00 -4.8412291828E+02  5.0000000000E+02 &
     & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  ' 25 1 1 2 0 0 -1.4960220911E+02 &
     &-4.6042825611E+02  0.0000000000E+00  5.0000000000E+02 &
     & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  ' 25 1 1 2 0 0  1.4960220911E+02 &
     & 4.6042825611E+02  0.0000000000E+00  5.0000000000E+02 &
     & 1.2500000000E+02  0.0000000000E+00  9.0000000000E+00'
write (u_file, "(A)")  '<weights>1.0000000000E+00</weights>'
write (u_file, "(A)")  '</event>'
write (u_file, "(A)")  '</LesHouchesEvents>'
close (u_file)

write (u, "(A)")  "* Initialize test process"
write (u, "(A)")

call eio_prepare_fallback_model (fallback_model)
call eio_prepare_test (event, unweighted = .false.)

allocate (eio_lhef_t :: eio)
select type (eio)
type is (eio_lhef_t)
   call eio%set_parameters (version = "3.0", recover_beams = .false.)
end select
call eio%set_fallback_model (fallback_model)

call data%init (1)
data%unweighted = .false.
data%norm_mode = NORM_SIGMA
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
call data%write (u)
write (u, *)

write (u, "(A)")  "* Initialize and read header"
write (u, "(A)")

call eio%init_in (sample, data)
call eio%write (u)

write (u, *)
```

```
    select type (eio)
    type is (eio_lhef_t)
       call eio%tag_lhef%write (u);  write (u, *)
    end select

    write (u, *)
    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Read event"
    write (u, "(A)")

    call eio%input_i_prc (i_prc, iostat)

    select type (eio)
    type is (eio_lhef_t)
       write (u, "(A,I0,A,I0)")  "Found process #", i_prc, &
            " with ID = ", eio%proc_num_id(i_prc)
    end select

    call eio%input_event (event, iostat)

    call event%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Read closing"
    write (u, "(A)")

    call eio%input_i_prc (i_prc, iostat)
    write (u, "(A,I0)")  "iostat = ", iostat

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio%final ()

    call eio_cleanup_test (event)
    call eio_cleanup_fallback_model (fallback_model)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_lhef_6"

  end subroutine eio_lhef_6
```

17.14 STDHEP File Formats

Here, we implement the two existing STDHEP file formats, one based on the HEPRUP/HEPEUP common blocks, the other based on the HEPEVT common block. The second one is actually the standard STDHEP format.

⟨eio_stdhep.f90⟩≡
  ⟨*File header*⟩

```
module eio_stdhep

  use kinds, only: i32, i64
⟨Use strings⟩
  use io_units
  use string_utils
  use diagnostics
  use event_base
  use hep_common
  use hep_events
  use eio_data
  use eio_base

⟨Standard module head⟩

⟨EIO stdhep: public⟩

⟨EIO stdhep: types⟩

⟨EIO stdhep: variables⟩

contains

⟨EIO stdhep: procedures⟩

end module eio_stdhep
```

17.14.1 Type

⟨*EIO stdhep: public*⟩≡

```
public :: eio_stdhep_t
```

⟨*EIO stdhep: types*⟩≡

```
type, abstract, extends (eio_t) :: eio_stdhep_t
   logical :: writing = .false.
   logical :: reading = .false.
   integer :: unit = 0
   logical :: keep_beams = .false.
   logical :: keep_remnants = .true.
   logical :: ensure_order = .false.
   logical :: recover_beams = .false.
   logical :: use_alpha_s_from_file = .false.
   logical :: use_scale_from_file = .false.
   integer, dimension(:), allocatable :: proc_num_id
   integer(i64) :: n_events_expected = 0
 contains
 ⟨EIO stdhep: eio stdhep: TBP⟩
end type eio_stdhep_t
```

⟨*EIO stdhep: public*⟩+≡

```
public :: eio_stdhep_hepevt_t
```

⟨*EIO stdhep: types*⟩+≡
```
type, extends (eio_stdhep_t) :: eio_stdhep_hepevt_t
end type eio_stdhep_hepevt_t
```

⟨*EIO stdhep: public*⟩+≡
```
public :: eio_stdhep_hepeup_t
```
⟨*EIO stdhep: types*⟩+≡
```
type, extends (eio_stdhep_t) :: eio_stdhep_hepeup_t
end type eio_stdhep_hepeup_t
```

⟨*EIO stdhep: public*⟩+≡
```
public :: eio_stdhep_hepev4_t
```
⟨*EIO stdhep: types*⟩+≡
```
type, extends (eio_stdhep_t) :: eio_stdhep_hepev4_t
end type eio_stdhep_hepev4_t
```

17.14.2 Specific Methods

Set parameters that are specifically used with STDHEP file formats.

⟨*EIO stdhep: eio stdhep: TBP*⟩≡
```
procedure :: set_parameters => eio_stdhep_set_parameters
```
⟨*EIO stdhep: procedures*⟩≡
```
subroutine eio_stdhep_set_parameters (eio, &
     keep_beams, keep_remnants, ensure_order, recover_beams, &
     use_alpha_s_from_file, use_scale_from_file, extension)
  class(eio_stdhep_t), intent(inout) :: eio
  logical, intent(in), optional :: keep_beams
  logical, intent(in), optional :: keep_remnants
  logical, intent(in), optional :: ensure_order
  logical, intent(in), optional :: recover_beams
  logical, intent(in), optional :: use_alpha_s_from_file
  logical, intent(in), optional :: use_scale_from_file
  type(string_t), intent(in), optional :: extension
  if (present (keep_beams))  eio%keep_beams = keep_beams
  if (present (keep_remnants))  eio%keep_remnants = keep_remnants
  if (present (ensure_order))  eio%ensure_order = ensure_order
  if (present (recover_beams))  eio%recover_beams = recover_beams
  if (present (use_alpha_s_from_file)) &
       eio%use_alpha_s_from_file = use_alpha_s_from_file
  if (present (use_scale_from_file))  &
       eio%use_scale_from_file = use_scale_from_file
  if (present (extension)) then
     eio%extension = extension
  else
     select type (eio)
     type is (eio_stdhep_hepevt_t)
        eio%extension = "hep"
     type is (eio_stdhep_hepev4_t)
        eio%extension = "ev4.hep"
     type is (eio_stdhep_hepeup_t)
```

```
        eio%extension = "up.hep"
      end select
    end if
  end subroutine eio_stdhep_set_parameters
```

17.14.3 Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡

```
  procedure :: write => eio_stdhep_write
```

⟨*EIO stdhep: procedures*⟩+≡

```
  subroutine eio_stdhep_write (object, unit)
    class(eio_stdhep_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u, i
    u = given_output_unit (unit)
    write (u, "(1x,A)")  "STDHEP event stream:"
    if (object%writing) then
       write (u, "(3x,A,A)")  "Writing to file   = ", char (object%filename)
    else if (object%reading) then
       write (u, "(3x,A,A)")  "Reading from file = ", char (object%filename)
    else
       write (u, "(3x,A)")  "[closed]"
    end if
    write (u, "(3x,A,L1)")    "Keep beams        = ", object%keep_beams
    write (u, "(3x,A,L1)")    "Keep remnants     = ", object%keep_remnants
    write (u, "(3x,A,L1)")    "Recover beams     = ", object%recover_beams
    write (u, "(3x,A,L1)")    "Alpha_s from file = ", &
         object%use_alpha_s_from_file
    write (u, "(3x,A,L1)")    "Scale from file   = ", &
         object%use_scale_from_file
    if (allocated (object%proc_num_id)) then
       write (u, "(3x,A)")  "Numerical process IDs:"
       do i = 1, size (object%proc_num_id)
          write (u, "(5x,I0,': ',I0)")  i, object%proc_num_id(i)
       end do
    end if
  end subroutine eio_stdhep_write
```

Finalizer: close any open file.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡

```
  procedure :: final => eio_stdhep_final
```

⟨*EIO stdhep: procedures*⟩+≡

```
  subroutine eio_stdhep_final (object)
    class(eio_stdhep_t), intent(inout) :: object
    if (allocated (object%proc_num_id))  deallocate (object%proc_num_id)
    if (object%writing) then
       write (msg_buffer, "(A,A,A)")  "Events: closing STDHEP file '", &
            char (object%filename), "'"
       call msg_message ()
```

```
      call stdhep_write (200)
      call stdhep_end ()
      object%writing = .false.
   else if (object%reading) then
      write (msg_buffer, "(A,A,A)")  "Events: closing STDHEP file '", &
           char (object%filename), "'"
      call msg_message ()
      object%reading = .false.
   end if
end subroutine eio_stdhep_final
```

Common initialization for input and output.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡

```
procedure :: common_init => eio_stdhep_common_init
```

⟨*EIO stdhep: procedures*⟩+≡

```
subroutine eio_stdhep_common_init (eio, sample, data, extension)
  class(eio_stdhep_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(string_t), intent(in), optional :: extension
  type(event_sample_data_t), intent(in), optional :: data
  if (.not. present (data)) &
       call msg_bug ("STDHEP initialization: missing data")
  if (data%n_beam /= 2) &
       call msg_fatal ("STDHEP: defined for scattering processes only")
  if (present (extension)) then
     eio%extension = extension
  end if
  eio%sample = sample
  call eio%set_filename ()
  eio%unit = free_unit ()
  allocate (eio%proc_num_id (data%n_proc), source = data%proc_num_id)
end subroutine eio_stdhep_common_init
```

Split event file: increment the counter, close the current file, open a new one. If the file needs a header, repeat it for the new file. (We assume that the common block contents are still intact.)

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡

```
procedure :: split_out => eio_stdhep_split_out
```

⟨*EIO stdhep: procedures*⟩+≡

```
subroutine eio_stdhep_split_out (eio)
  class(eio_stdhep_t), intent(inout) :: eio
  if (eio%split) then
     eio%split_index = eio%split_index + 1
     call eio%set_filename ()
     write (msg_buffer, "(A,A,A)")  "Events: writing to STDHEP file '", &
          char (eio%filename), "'"
     call msg_message ()
     call stdhep_write (200)
     call stdhep_end ()
     select type (eio)
     type is (eio_stdhep_hepeup_t)
        call stdhep_init_out (char (eio%filename), &
```

```
            "WHIZARD ⟨Version⟩", eio%n_events_expected)
         call stdhep_write (100)
         call stdhep_write (STDHEP_HEPRUP)
      type is (eio_stdhep_hepevt_t)
         call stdhep_init_out (char (eio%filename), &
            "WHIZARD ⟨Version⟩", eio%n_events_expected)
         call stdhep_write (100)
      type is (eio_stdhep_hepev4_t)
         call stdhep_init_out (char (eio%filename), &
            "WHIZARD ⟨Version⟩", eio%n_events_expected)
         call stdhep_write (100)
      end select
   end if
end subroutine eio_stdhep_split_out
```

Initialize event writing.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡

```
procedure :: init_out => eio_stdhep_init_out
```

⟨*EIO stdhep: procedures*⟩+≡

```
subroutine eio_stdhep_init_out (eio, sample, data, success, extension)
  class(eio_stdhep_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(string_t), intent(in), optional :: extension
  type(event_sample_data_t), intent(in), optional :: data
  logical, intent(out), optional :: success
  integer :: i
  if (.not. present (data)) &
       call msg_bug ("STDHEP initialization: missing data")
  call eio%set_splitting (data)
  call eio%common_init (sample, data, extension)
  eio%n_events_expected = data%n_evt
  write (msg_buffer, "(A,A,A)")  "Events: writing to STDHEP file '", &
       char (eio%filename), "'"
  call msg_message ()
  eio%writing = .true.
  select type (eio)
  type is (eio_stdhep_hepeup_t)
     call heprup_init &
          (data%pdg_beam, &
          data%energy_beam, &
          n_processes = data%n_proc, &
          unweighted = data%unweighted, &
          negative_weights = data%negative_weights)
     do i = 1, data%n_proc
        call heprup_set_process_parameters (i = i, &
             process_id = data%proc_num_id(i), &
             cross_section = data%cross_section(i), &
             error = data%error(i))
     end do
     call stdhep_init_out (char (eio%filename), &
          "WHIZARD ⟨Version⟩", eio%n_events_expected)
     call stdhep_write (100)
     call stdhep_write (STDHEP_HEPRUP)
```

```
    type is (eio_stdhep_hepevt_t)
       call stdhep_init_out (char (eio%filename), &
            "WHIZARD ⟨Version⟩", eio%n_events_expected)
       call stdhep_write (100)
    type is (eio_stdhep_hepev4_t)
       call stdhep_init_out (char (eio%filename), &
            "WHIZARD ⟨Version⟩", eio%n_events_expected)
       call stdhep_write (100)
    end select
    if (present (success))  success = .true.
  end subroutine eio_stdhep_init_out
```

Initialize event reading.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡

```
  procedure :: init_in => eio_stdhep_init_in
```

⟨*EIO stdhep: procedures*⟩+≡

```
  subroutine eio_stdhep_init_in (eio, sample, data, success, extension)
    class(eio_stdhep_t), intent(inout) :: eio
    type(string_t), intent(in) :: sample
    type(string_t), intent(in), optional :: extension
    type(event_sample_data_t), intent(inout), optional :: data
    logical, intent(out), optional :: success
    integer :: ilbl, lok
    logical :: exist
    call eio%common_init (sample, data, extension)
    write (msg_buffer, "(A,A,A)")  "Events: reading from STDHEP file '", &
         char (eio%filename), "'"
    call msg_message ()
    inquire (file = char (eio%filename), exist = exist)
    if (.not. exist)  call msg_fatal ("Events: STDHEP file not found.")
    eio%reading = .true.
    call stdhep_init_in (char (eio%filename), eio%n_events_expected)
    call stdhep_read (ilbl, lok)
    if (lok /= 0) then
       call stdhep_end ()
       write (msg_buffer, "(A)")  "Events: STDHEP file appears to" // &
            " be empty."
       call msg_message ()
    end if
    if (ilbl == 100) then
       write (msg_buffer, "(A)")  "Events: reading in STDHEP events"
       call msg_message ()
    end if
    if (present (success))  success = .false.
  end subroutine eio_stdhep_init_in
```

Switch from input to output: reopen the file for reading.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡

```
  procedure :: switch_inout => eio_stdhep_switch_inout
```

⟨*EIO stdhep: procedures*⟩+≡

```
  subroutine eio_stdhep_switch_inout (eio, success)
    class(eio_stdhep_t), intent(inout) :: eio
```

```
    logical, intent(out), optional :: success
    call msg_bug ("STDHEP: in-out switch not supported")
    if (present (success))  success = .false.
  end subroutine eio_stdhep_switch_inout
```

Output an event. Write first the event indices, then weight and squared matrix element, then the particle set.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡

```
  procedure :: output => eio_stdhep_output
```

⟨*EIO stdhep: procedures*⟩+≡

```
  subroutine eio_stdhep_output (eio, event, i_prc, reading, passed, pacify)
    class(eio_stdhep_t), intent(inout) :: eio
    class(generic_event_t), intent(in), target :: event
    integer, intent(in) :: i_prc
    logical, intent(in), optional :: reading, passed, pacify
    if (present (passed)) then
       if (.not. passed)  return
    end if
    if (eio%writing) then
       select type (eio)
       type is (eio_stdhep_hepeup_t)
          call hepeup_from_event (event, &
               process_index = eio%proc_num_id (i_prc), &
               keep_beams = eio%keep_beams, &
               keep_remnants = eio%keep_remnants)
          call stdhep_write (STDHEP_HEPEUP)
       type is (eio_stdhep_hepevt_t)
          call hepevt_from_event (event, &
               i_evt = event%get_index (), &
               keep_beams = eio%keep_beams, &
               keep_remnants = eio%keep_remnants, &
               ensure_order = eio%ensure_order)
          call stdhep_write (STDHEP_HEPEVT)
       type is (eio_stdhep_hepev4_t)
          call hepevt_from_event (event, &
               process_index = eio%proc_num_id (i_prc), &
               i_evt = event%get_index (), &
               keep_beams = eio%keep_beams, &
               keep_remnants = eio%keep_remnants, &
               ensure_order = eio%ensure_order, &
               fill_hepev4 = .true.)
          call stdhep_write (STDHEP_HEPEV4)
       end select
    else
       call eio%write ()
       call msg_fatal ("STDHEP file is not open for writing")
    end if
  end subroutine eio_stdhep_output
```

Input an event. We do not allow to read in STDHEP files written via the HEPEVT common block as there is no control on the process ID.

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡

```
procedure :: input_i_prc => eio_stdhep_input_i_prc
procedure :: input_event => eio_stdhep_input_event
```

⟨*EIO stdhep: procedures*⟩+≡

```
subroutine eio_stdhep_input_i_prc (eio, i_prc, iostat)
  class(eio_stdhep_t), intent(inout) :: eio
  integer, intent(out) :: i_prc
  integer, intent(out) :: iostat
  integer :: i, ilbl, proc_num_id
  iostat = 0
  select type (eio)
  type is (eio_stdhep_hepevt_t)
     if (size (eio%proc_num_id) > 1) then
        call msg_fatal ("Events: only single processes allowed " // &
             "with the STDHEP HEPEVT format.")
     else
        proc_num_id = eio%proc_num_id (1)
        call stdhep_read (ilbl, lok)
     end if
  type is (eio_stdhep_hepev4_t)
     call stdhep_read (ilbl, lok)
     proc_num_id = idruplh
  type is (eio_stdhep_hepeup_t)
     call stdhep_read (ilbl, lok)
     if (lok /= 0)  call msg_error ("Events: STDHEP appears to be " // &
          "empty or corrupted.")
     if (ilbl == 12) then
        call stdhep_read (ilbl, lok)
     end if
     if (ilbl == 11) then
        proc_num_id = IDPRUP
     end if
  end select
  FIND_I_PRC: do i = 1, size (eio%proc_num_id)
     if (eio%proc_num_id(i) == proc_num_id) then
        i_prc = i
        exit FIND_I_PRC
     end if
  end do FIND_I_PRC
  if (i_prc == 0)  call err_index
contains
  subroutine err_index
    call msg_error ("STDHEP: reading events: undefined process ID " &
         // char (str (proc_num_id)) // ", aborting read")
    iostat = 1
  end subroutine err_index
end subroutine eio_stdhep_input_i_prc

subroutine eio_stdhep_input_event (eio, event, iostat)
  class(eio_stdhep_t), intent(inout) :: eio
  class(generic_event_t), intent(inout), target :: event
  integer, intent(out) :: iostat
  iostat = 0
  call event%reset ()
  call event%select (1, 1, 1)
```

```
      call hepeup_to_event (event, eio%fallback_model, &
           recover_beams = eio%recover_beams, &
           use_alpha_s = eio%use_alpha_s_from_file, &
           use_scale = eio%use_scale_from_file)
    end subroutine eio_stdhep_input_event
```

⟨*EIO stdhep: eio stdhep: TBP*⟩+≡

```
    procedure :: skip => eio_stdhep_skip
```

⟨*EIO stdhep: procedures*⟩+≡

```
    subroutine eio_stdhep_skip (eio, iostat)
      class(eio_stdhep_t), intent(inout) :: eio
      integer, intent(out) :: iostat
      if (eio%reading) then
         read (eio%unit, iostat = iostat)
      else
         call eio%write ()
         call msg_fatal ("Raw event file is not open for reading")
      end if
    end subroutine eio_stdhep_skip
```

STDHEP speficic routines.

⟨*EIO stdhep: public*⟩+≡

```
    public :: stdhep_init_out
    public :: stdhep_init_in
    public :: stdhep_write
    public :: stdhep_end
```

⟨*EIO stdhep: procedures*⟩+≡

```
    subroutine stdhep_init_out (file, title, nevt)
      character(len=*), intent(in) :: file, title
      integer(i64), intent(in) :: nevt
      integer(i32) :: nevt32
      nevt32 = min (nevt, int (huge (1_i32), i64))
      call stdxwinit (file, title, nevt32, istr, lok)
    end subroutine stdhep_init_out

    subroutine stdhep_init_in (file, nevt)
      character(len=*), intent(in) :: file
      integer(i64), intent(out) :: nevt
      integer(i32) :: nevt32
      call stdxrinit (file, nevt32, istr, lok)
      if (lok /= 0)  call msg_fatal ("STDHEP: error in reading file '" // &
           file // "'.")
      nevt = int (nevt32, i64)
    end subroutine stdhep_init_in

    subroutine stdhep_write (ilbl)
      integer, intent(in) :: ilbl
      call stdxwrt (ilbl, istr, lok)
    end subroutine stdhep_write

    subroutine stdhep_read (ilbl, lok)
      integer, intent(out) :: ilbl, lok
```

```
    call stdxrd (ilbl, istr, lok)
    if (lok /= 0)  return
  end subroutine stdhep_read

  subroutine stdhep_end
    call stdxend (istr)
  end subroutine stdhep_end
```

17.14.4 Variables

⟨*EIO stdhep: variables*⟩≡

```
  integer, save :: istr, lok
  integer, parameter :: &
       STDHEP_HEPEVT = 1, STDHEP_HEPEV4 = 4, &
       STDHEP_HEPEUP = 11, STDHEP_HEPRUP = 12
```

17.14.5 Unit tests

Test module, followed by the corresponding implementation module.

⟨eio_stdhep_ut.f90⟩≡

```
  ⟨File header⟩

  module eio_stdhep_ut
    use unit_tests
    use eio_stdhep_uti

  ⟨Standard module head⟩

  ⟨EIO stdhep: public test⟩

  contains

  ⟨EIO stdhep: test driver⟩

  end module eio_stdhep_ut
```

⟨eio_stdhep_uti.f90⟩≡

```
  ⟨File header⟩

  module eio_stdhep_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use io_units
    use model_data
    use event_base
    use eio_data
    use eio_base
    use xdr_wo_stdhep

    use eio_stdhep
```

```
  use eio_base_ut, only: eio_prepare_test, eio_cleanup_test
  use eio_base_ut, only: eio_prepare_fallback_model, eio_cleanup_fallback_model
```

⟨*Standard module head*⟩

⟨*EIO stdhep: test declarations*⟩

```
contains
```

⟨*EIO stdhep: tests*⟩

```
end module eio_stdhep_uti
```

API: driver for the unit tests below.

⟨*EIO stdhep: public test*⟩≡

```
  public :: eio_stdhep_test
```

⟨*EIO stdhep: test driver*⟩≡

```
  subroutine eio_stdhep_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```

⟨*EIO stdhep: execute tests*⟩

```
  end subroutine eio_stdhep_test
```

Test I/O methods

We test the implementation of the STDHEP HEPEVT I/O method:

⟨*EIO stdhep: execute tests*⟩≡

```
  call test (eio_stdhep_1, "eio_stdhep_1", &
       "read and write event contents, format [stdhep]", &
       u, results)
```

⟨*EIO stdhep: test declarations*⟩≡

```
  public :: eio_stdhep_1
```

⟨*EIO stdhep: tests*⟩≡

```
  subroutine eio_stdhep_1 (u)
    integer, intent(in) :: u
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat
    character(215) :: buffer

    write (u, "(A)")  "* Test output: eio_stdhep_1"
    write (u, "(A)")  "*   Purpose: generate an event in STDHEP HEPEVT format"
    write (u, "(A)")  "*      and write weight to file"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize test process"

    call eio_prepare_test (event)
```

```
call data%init (1)
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_stdhep_1"

allocate (eio_stdhep_hepevt_t :: eio)
select type (eio)
type is (eio_stdhep_hepevt_t)
   call eio%set_parameters ()
end select

call eio%init_out (sample, data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()
call event%pacify_particle_set ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* Write STDHEP file contents to ASCII file"
write (u, "(A)")

call write_stdhep_event &
     (sample // ".hep", var_str ("test_1.hep"), 1)

write (u, "(A)")
write (u, "(A)")  "* Read in ASCII contents of STDHEP file"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = "test_1.hep", &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (iostat /= 0)  exit
   if (trim (buffer) == "")  cycle
   if (buffer(1:18) == "    total blocks: ")  &
        buffer = "    total blocks: [...]"
   if (buffer(1:25) == "           title: WHIZARD")  &
        buffer = "           title: WHIZARD [version]"
   if (buffer(1:17) == "            date:")  &
```

```
          buffer = "            date: [...]"
      if (buffer(1:17) == "    closing date:")  &
          buffer = "    closing date: [...]"
      write (u, "(A)") trim (buffer)
    end do
    close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Reset data"
    write (u, "(A)")

    deallocate (eio)
    allocate (eio_stdhep_hepevt_t :: eio)

    select type (eio)
    type is (eio_stdhep_hepevt_t)
      call eio%set_parameters (keep_beams = .true.)
    end select
    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_stdhep_1"

  end subroutine eio_stdhep_1
```

We test the implementation of the STDHEP HEPEUP I/O method:

⟨*EIO stdhep: execute tests*⟩+≡

```
  call test (eio_stdhep_2, "eio_stdhep_2", &
      "read and write event contents, format [stdhep]", &
      u, results)
```

⟨*EIO stdhep: test declarations*⟩+≡

```
  public :: eio_stdhep_2
```

⟨*EIO stdhep: tests*⟩+≡

```
  subroutine eio_stdhep_2 (u)
    integer, intent(in) :: u
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(model_data_t), pointer :: fallback_model
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat
    character(215) :: buffer

    write (u, "(A)")  "* Test output: eio_stdhep_2"
    write (u, "(A)")  "*   Purpose: generate an event in STDHEP HEPEUP format"
    write (u, "(A)")  "*      and write weight to file"
    write (u, "(A)")
```

```
write (u, "(A)")  "* Initialize test process"

call eio_prepare_fallback_model (fallback_model)
call eio_prepare_test (event, unweighted = .false.)

call data%init (1)
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_stdhep_2"

allocate (eio_stdhep_hepeup_t :: eio)
select type (eio)
type is (eio_stdhep_hepeup_t)
   call eio%set_parameters ()
end select
call eio%set_fallback_model (fallback_model)

call eio%init_out (sample, data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* Write STDHEP file contents to ASCII file"
write (u, "(A)")

call write_stdhep_event &
     (sample // ".up.hep", var_str ("test_2.hep"), 2)

write (u, "(A)")
write (u, "(A)")  "* Read in ASCII contents of STDHEP file"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = "test_2.hep", &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (iostat /= 0)  exit
   if (trim (buffer) == "")  cycle
```

```
      if (buffer(1:18) == "    total blocks: ")  &
           buffer = "    total blocks: [...]"
      if (buffer(1:25) == "           title: WHIZARD")  &
           buffer = "           title: WHIZARD [version]"
      if (buffer(1:17) == "            date:")  &
           buffer = "            date: [...]"
      if (buffer(1:17) == "    closing date:")  &
           buffer = "    closing date: [...]"
      write (u, "(A)") trim (buffer)
   end do
   close (u_file)

   write (u, "(A)")
   write (u, "(A)")  "* Reset data"
   write (u, "(A)")

   deallocate (eio)
   allocate (eio_stdhep_hepeup_t :: eio)

   select type (eio)
   type is (eio_stdhep_hepeup_t)
      call eio%set_parameters (keep_beams = .true.)
   end select
   call eio%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call eio_cleanup_test (event)
   call eio_cleanup_fallback_model (fallback_model)

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: eio_stdhep_2"

end subroutine eio_stdhep_2
```

Check input from a StdHep file, HEPEVT block.

⟨*EIO stdhep: execute tests*⟩+≡

```
call test (eio_stdhep_3, "eio_stdhep_3", &
     "read StdHep file, HEPEVT block", &
     u, results)
```

⟨*EIO stdhep: test declarations*⟩+≡

```
public :: eio_stdhep_3
```

⟨*EIO stdhep: tests*⟩+≡

```
subroutine eio_stdhep_3 (u)
  integer, intent(in) :: u
  class(model_data_t), pointer :: fallback_model
  class(generic_event_t), pointer :: event
  type(event_sample_data_t) :: data
  class(eio_t), allocatable :: eio
  type(string_t) :: sample
  integer :: iostat, i_prc
```

```
write (u, "(A)")  "* Test output: eio_stdhep_3"
write (u, "(A)")  "*   Purpose: read a StdHep file, HEPEVT block"
write (u, "(A)")

write (u, "(A)")  "* Write a StdHep data file, HEPEVT block"
write (u, "(A)")

call eio_prepare_fallback_model (fallback_model)
call eio_prepare_test (event)

call data%init (1)
data%n_evt = 1
data%n_beam = 2
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_stdhep_3"

allocate (eio_stdhep_hepevt_t :: eio)
select type (eio)
type is (eio_stdhep_hepevt_t)
   call eio%set_parameters ()
end select
call eio%set_fallback_model (fallback_model)

call eio%init_out (sample, data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

call eio_cleanup_test (event)
call eio_cleanup_fallback_model (fallback_model)
deallocate (eio)

write (u, "(A)")  "* Initialize test process"
write (u, "(A)")

call eio_prepare_fallback_model (fallback_model)
call eio_prepare_test (event, unweighted = .false.)

allocate (eio_stdhep_hepevt_t :: eio)
select type (eio)
type is (eio_stdhep_hepevt_t)
```

```
   call eio%set_parameters (recover_beams = .false.)
end select
call eio%set_fallback_model (fallback_model)

call data%init (1)
data%n_beam = 2
data%unweighted = .true.
data%norm_mode = NORM_UNIT
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
call data%write (u)
write (u, *)

write (u, "(A)")  "* Initialize"
write (u, "(A)")

call eio%init_in (sample, data)
call eio%write (u)

write (u, "(A)")
write (u, "(A)")  "* Read event"
write (u, "(A)")

call eio%input_i_prc (i_prc, iostat)

select type (eio)
type is (eio_stdhep_hepevt_t)
   write (u, "(A,I0,A,I0)")  "Found process #", i_prc, &
        " with ID = ", eio%proc_num_id(i_prc)
end select

call eio%input_event (event, iostat)

call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Read closing"
write (u, "(A)")

call eio%input_i_prc (i_prc, iostat)
write (u, "(A,I0)")  "iostat = ", iostat

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call eio%final ()

call eio_cleanup_test (event)
call eio_cleanup_fallback_model (fallback_model)

write (u, "(A)")
write (u, "(A)")  "* Test output end: eio_stdhep_3"
```

```
end subroutine eio_stdhep_3
```

Check input from a StdHep file, HEPEVT block.

⟨*EIO stdhep: execute tests*⟩+≡

```
call test (eio_stdhep_4, "eio_stdhep_4", &
     "read StdHep file, HEPRUP/HEPEUP block", &
     u, results)
```

⟨*EIO stdhep: test declarations*⟩+≡

```
public :: eio_stdhep_4
```

⟨*EIO stdhep: tests*⟩+≡

```
subroutine eio_stdhep_4 (u)
  integer, intent(in) :: u
  class(model_data_t), pointer :: fallback_model
  class(generic_event_t), pointer :: event
  type(event_sample_data_t) :: data
  class(eio_t), allocatable :: eio
  type(string_t) :: sample
  integer :: iostat, i_prc

  write (u, "(A)")  "* Test output: eio_stdhep_3"
  write (u, "(A)")  "*   Purpose: read a StdHep file, HEPRUP/HEPEUP block"
  write (u, "(A)")

  write (u, "(A)")  "* Write a StdHep data file, HEPRUP/HEPEUP block"
  write (u, "(A)")

  call eio_prepare_fallback_model (fallback_model)
  call eio_prepare_test (event)

  call data%init (1)
  data%n_evt = 1
  data%n_beam = 2
  data%pdg_beam = 25
  data%energy_beam = 500
  data%proc_num_id = [42]
  data%cross_section(1) = 100
  data%error(1) = 1
  data%total_cross_section = sum (data%cross_section)

  write (u, "(A)")
  write (u, "(A)")  "* Generate and write an event, HEPEUP/HEPRUP"
  write (u, "(A)")

  sample = "eio_stdhep_4"

  allocate (eio_stdhep_hepeup_t :: eio)
  select type (eio)
  type is (eio_stdhep_hepeup_t)
     call eio%set_parameters ()
  end select
  call eio%set_fallback_model (fallback_model)

  call eio%init_out (sample, data)
```

```
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()
call event%pacify_particle_set ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

call eio_cleanup_test (event)
call eio_cleanup_fallback_model (fallback_model)
deallocate (eio)

write (u, "(A)")  "* Initialize test process"
write (u, "(A)")

call eio_prepare_fallback_model (fallback_model)
call eio_prepare_test (event, unweighted = .false.)

allocate (eio_stdhep_hepeup_t :: eio)
select type (eio)
type is (eio_stdhep_hepeup_t)
   call eio%set_parameters (recover_beams = .false.)
end select
call eio%set_fallback_model (fallback_model)

call data%init (1)
data%n_beam = 2
data%unweighted = .true.
data%norm_mode = NORM_UNIT
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
call data%write (u)
write (u, *)

write (u, "(A)")  "* Initialize"
write (u, "(A)")

call eio%init_in (sample, data)
call eio%write (u)

write (u, "(A)")
write (u, "(A)")  "* Read event"
write (u, "(A)")

call eio%input_i_prc (i_prc, iostat)

select type (eio)
type is (eio_stdhep_hepeup_t)
   write (u, "(A,I0,A,I0)")  "Found process #", i_prc, &
        " with ID = ", eio%proc_num_id(i_prc)
end select

call eio%input_event (event, iostat)
```

```
    call event%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Read closing"
    write (u, "(A)")

    call eio%input_i_prc (i_prc, iostat)
    write (u, "(A,I0)")  "iostat = ", iostat

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio%final ()

    call eio_cleanup_test (event)
    call eio_cleanup_fallback_model (fallback_model)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_stdhep_4"

  end subroutine eio_stdhep_4
```

17.15 HepMC Output

The HepMC event record is standardized. It is an ASCII format. We try our best at using it for both input and output.

⟨eio_hepmc.f90⟩≡

```
⟨File header⟩

module eio_hepmc

⟨Use strings⟩
  use io_units
  use string_utils
  use diagnostics
  use particles
  use model_data
  use event_base
  use hep_events
  use eio_data
  use eio_base
  use hepmc_interface

⟨Standard module head⟩

⟨EIO HepMC: public⟩

⟨EIO HepMC: types⟩

contains
```

⟨*EIO HepMC: procedures*⟩

```
end module eio_hepmc
```

17.15.1 Type

A type `hepmc_event` is introduced as container to store HepMC event data, particularly for splitting the reading into read out of the process index and the proper event data.

Note: the `keep_beams` flag is not supported. Beams will always be written. Tools like `Rivet` can use the cross section information of a HepMC file for scaling plots. As there is no header in HepMC and this is written for every event, we make it optional with `output_cross_section`.

⟨*EIO HepMC: public*⟩≡

```
public :: eio_hepmc_t
```

⟨*EIO HepMC: types*⟩≡

```
type, extends (eio_t) :: eio_hepmc_t
   logical :: writing = .false.
   logical :: reading = .false.
   type(event_sample_data_t) :: data
   ! logical :: keep_beams = .false.
   logical :: recover_beams = .false.
   logical :: use_alpha_s_from_file = .false.
   logical :: use_scale_from_file = .false.
   logical :: output_cross_section = .false.
   type(hepmc_iostream_t) :: iostream
   type(hepmc_event_t) :: hepmc_event
   integer, dimension(:), allocatable :: proc_num_id
 contains
 ⟨EIO HepMC: eio hepmc: TBP⟩
end type eio_hepmc_t
```

17.15.2 Specific Methods

Set parameters that are specifically used with HepMC.

⟨*EIO HepMC: eio hepmc: TBP*⟩≡

```
procedure :: set_parameters => eio_hepmc_set_parameters
```

⟨*EIO HepMC: procedures*⟩≡

```
subroutine eio_hepmc_set_parameters &
     (eio, &
     recover_beams, use_alpha_s_from_file, use_scale_from_file, &
     extension, output_cross_section)
  class(eio_hepmc_t), intent(inout) :: eio
  logical, intent(in), optional :: recover_beams
  logical, intent(in), optional :: use_alpha_s_from_file
  logical, intent(in), optional :: use_scale_from_file
  logical, intent(in), optional :: output_cross_section
  type(string_t), intent(in), optional :: extension
  if (present (recover_beams)) &
```

```
      eio%recover_beams = recover_beams
  if (present (use_alpha_s_from_file)) &
      eio%use_alpha_s_from_file = use_alpha_s_from_file
  if (present (use_scale_from_file)) &
      eio%use_scale_from_file = use_scale_from_file
  if (present (extension)) then
     eio%extension = extension
  else
     eio%extension = "hepmc"
  end if
  if (present (output_cross_section)) &
      eio%output_cross_section = output_cross_section
end subroutine eio_hepmc_set_parameters
```

17.15.3 Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡

```
procedure :: write => eio_hepmc_write
```

⟨*EIO HepMC: procedures*⟩+≡

```
subroutine eio_hepmc_write (object, unit)
  class(eio_hepmc_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "HepMC event stream:"
  if (object%writing) then
     write (u, "(3x,A,A)")  "Writing to file   = ", char (object%filename)
  else if (object%reading) then
     write (u, "(3x,A,A)")  "Reading from file = ", char (object%filename)
  else
     write (u, "(3x,A)")  "[closed]"
  end if
  write (u, "(3x,A,L1)")    "Recover beams     = ", object%recover_beams
  write (u, "(3x,A,L1)")    "Alpha_s from file = ", &
      object%use_alpha_s_from_file
  write (u, "(3x,A,L1)")    "Scale from file   = ", &
      object%use_scale_from_file
  write (u, "(3x,A,A,A)")     "File extension    = '", &
      char (object%extension), "'"
  if (allocated (object%proc_num_id)) then
     write (u, "(3x,A)")  "Numerical process IDs:"
     do i = 1, size (object%proc_num_id)
        write (u, "(5x,I0,': ',I0)")  i, object%proc_num_id(i)
     end do
  end if
end subroutine eio_hepmc_write
```

Finalizer: close any open file.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡

```
procedure :: final => eio_hepmc_final
```

⟨*EIO HepMC: procedures*⟩+≡

```
subroutine eio_hepmc_final (object)
  class(eio_hepmc_t), intent(inout) :: object
  if (allocated (object%proc_num_id))  deallocate (object%proc_num_id)
  if (object%writing) then
     write (msg_buffer, "(A,A,A)")  "Events: closing HepMC file '", &
          char (object%filename), "'"
     call msg_message ()
     call hepmc_iostream_close (object%iostream)
     object%writing = .false.
  else if (object%reading) then
     write (msg_buffer, "(A,A,A)")  "Events: closing HepMC file '", &
          char (object%filename), "'"
     call msg_message ()
     call hepmc_iostream_close (object%iostream)
     object%reading = .false.
  end if
end subroutine eio_hepmc_final
```

Split event file: increment the counter, close the current file, open a new one. If the file needs a header, repeat it for the new file.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡

```
procedure :: split_out => eio_hepmc_split_out
```

⟨*EIO HepMC: procedures*⟩+≡

```
subroutine eio_hepmc_split_out (eio)
  class(eio_hepmc_t), intent(inout) :: eio
  if (eio%split) then
     eio%split_index = eio%split_index + 1
     call eio%set_filename ()
     write (msg_buffer, "(A,A,A)")  "Events: writing to HepMC file '", &
          char (eio%filename), "'"
     call msg_message ()
     call hepmc_iostream_close (eio%iostream)
     call hepmc_iostream_open_out (eio%iostream, eio%filename)
  end if
end subroutine eio_hepmc_split_out
```

Common initialization for input and output.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡

```
procedure :: common_init => eio_hepmc_common_init
```

⟨*EIO HepMC: procedures*⟩+≡

```
subroutine eio_hepmc_common_init (eio, sample, data, extension)
  class(eio_hepmc_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(string_t), intent(in), optional :: extension
  type(event_sample_data_t), intent(in), optional :: data
  if (.not. present (data)) &
       call msg_bug ("HepMC initialization: missing data")
  eio%data = data
  if (data%n_beam /= 2) &
```

```
        call msg_fatal ("HepMC: defined for scattering processes only")
    ! We could relax this condition now with weighted hepmc events
    if (data%unweighted) then
       select case (data%norm_mode)
       case (NORM_UNIT)
       case default; call msg_fatal &
            ("HepMC: normalization for unweighted events must be '1'")
       end select
    end if
    eio%sample = sample
    if (present (extension)) then
       eio%extension = extension
    end if
    call eio%set_filename ()
    allocate (eio%proc_num_id (data%n_proc), source = data%proc_num_id)
  end subroutine eio_hepmc_common_init
```

Initialize event writing.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡

```
  procedure :: init_out => eio_hepmc_init_out
```

⟨*EIO HepMC: procedures*⟩+≡

```
  subroutine eio_hepmc_init_out (eio, sample, data, success, extension)
    class(eio_hepmc_t), intent(inout) :: eio
    type(string_t), intent(in) :: sample
    type(string_t), intent(in), optional :: extension
    type(event_sample_data_t), intent(in), optional :: data
    logical, intent(out), optional :: success
    call eio%set_splitting (data)
    call eio%common_init (sample, data, extension)
    write (msg_buffer, "(A,A,A)")  "Events: writing to HepMC file '", &
         char (eio%filename), "'"
    call msg_message ()
    eio%writing = .true.
    call hepmc_iostream_open_out (eio%iostream, eio%filename)
    if (present (success))  success = .true.
  end subroutine eio_hepmc_init_out
```

Initialize event reading. For input, we do not (yet) support split event files.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡

```
  procedure :: init_in => eio_hepmc_init_in
```

⟨*EIO HepMC: procedures*⟩+≡

```
  subroutine eio_hepmc_init_in (eio, sample, data, success, extension)
    class(eio_hepmc_t), intent(inout) :: eio
    type(string_t), intent(in) :: sample
    type(string_t), intent(in), optional :: extension
    type(event_sample_data_t), intent(inout), optional :: data
    logical, intent(out), optional :: success
    logical :: exist
    eio%split = .false.
    call eio%common_init (sample, data, extension)
    write (msg_buffer, "(A,A,A)")  "Events: reading from HepMC file '", &
         char (eio%filename), "'"
```

```
    call msg_message ()
    inquire (file = char (eio%filename), exist = exist)
    if (.not. exist)  call msg_fatal ("Events: HepMC file not found.")
    eio%reading = .true.
    call hepmc_iostream_open_in (eio%iostream, eio%filename)
    if (present (success))  success = .true.
  end subroutine eio_hepmc_init_in
```

Switch from input to output: reopen the file for reading.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡

```
  procedure :: switch_inout => eio_hepmc_switch_inout
```

⟨*EIO HepMC: procedures*⟩+≡

```
  subroutine eio_hepmc_switch_inout (eio, success)
    class(eio_hepmc_t), intent(inout) :: eio
    logical, intent(out), optional :: success
    call msg_bug ("HepMC: in-out switch not supported")
    if (present (success))  success = .false.
  end subroutine eio_hepmc_switch_inout
```

Output an event to the allocated HepMC output stream.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡

```
  procedure :: output => eio_hepmc_output
```

⟨*EIO HepMC: procedures*⟩+≡

```
  subroutine eio_hepmc_output (eio, event, i_prc, reading, passed, pacify)
    class(eio_hepmc_t), intent(inout) :: eio
    class(generic_event_t), intent(in), target :: event
    integer, intent(in) :: i_prc
    logical, intent(in), optional :: reading, passed, pacify
    type(particle_set_t), pointer :: pset_ptr
    if (present (passed)) then
       if (.not. passed)  return
    end if
    if (eio%writing) then
       pset_ptr => event%get_particle_set_ptr ()
       call hepmc_event_init (eio%hepmc_event, &
            proc_id = eio%proc_num_id(i_prc), &
            event_id = event%get_index ())
       if (eio%output_cross_section) then
          call hepmc_event_from_particle_set (eio%hepmc_event, pset_ptr, &
               eio%data%cross_section(i_prc), eio%data%error(i_prc))
       else
          call hepmc_event_from_particle_set (eio%hepmc_event, pset_ptr)
       end if
       call hepmc_event_set_scale (eio%hepmc_event, event%get_fac_scale ())
       call hepmc_event_set_alpha_qcd (eio%hepmc_event, event%get_alpha_s ())
       if (.not. eio%data%unweighted) &
          call hepmc_event_add_weight (eio%hepmc_event, event%weight_prc)
       call hepmc_iostream_write_event (eio%iostream, eio%hepmc_event)
       call hepmc_event_final (eio%hepmc_event)
    else
       call eio%write ()
       call msg_fatal ("HepMC file is not open for writing")
```

```
    end if
  end subroutine eio_hepmc_output
```

Input an event.

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡

```
  procedure :: input_i_prc => eio_hepmc_input_i_prc
  procedure :: input_event => eio_hepmc_input_event
```

⟨*EIO HepMC: procedures*⟩+≡

```
  subroutine eio_hepmc_input_i_prc (eio, i_prc, iostat)
    class(eio_hepmc_t), intent(inout) :: eio
    integer, intent(out) :: i_prc
    integer, intent(out) :: iostat
    logical :: ok
    integer :: i, proc_num_id
    iostat = 0
    call hepmc_event_init (eio%hepmc_event)
    call hepmc_iostream_read_event (eio%iostream, eio%hepmc_event, ok)
    proc_num_id = hepmc_event_get_process_id (eio%hepmc_event)
    if (.not. ok) then
       iostat = -1
       return
    end if
    i_prc = 0
    FIND_I_PRC: do i = 1, size (eio%proc_num_id)
       if (eio%proc_num_id(i) == proc_num_id) then
          i_prc = i
          exit FIND_I_PRC
       end if
    end do FIND_I_PRC
    if (i_prc == 0)  call err_index
  contains
    subroutine err_index
      call msg_error ("HepMC: reading events: undefined process ID " &
           // char (str (proc_num_id)) // ", aborting read")
      iostat = 1
    end subroutine err_index
  end subroutine eio_hepmc_input_i_prc

  subroutine eio_hepmc_input_event (eio, event, iostat)
    class(eio_hepmc_t), intent(inout) :: eio
    class(generic_event_t), intent(inout), target :: event
    integer, intent(out) :: iostat
    iostat = 0
    call event%reset ()
    call event%select (1, 1, 1)
    call hepmc_to_event (event, eio%hepmc_event, &
         eio%fallback_model, &
         recover_beams = eio%recover_beams, &
         use_alpha_s = eio%use_alpha_s_from_file, &
         use_scale = eio%use_scale_from_file)
    call hepmc_event_final (eio%hepmc_event)
  end subroutine eio_hepmc_input_event
```

⟨*EIO HepMC: eio hepmc: TBP*⟩+≡
```
procedure :: skip => eio_hepmc_skip
```

⟨*EIO HepMC: procedures*⟩+≡
```
subroutine eio_hepmc_skip (eio, iostat)
  class(eio_hepmc_t), intent(inout) :: eio
  integer, intent(out) :: iostat
  iostat = 0
end subroutine eio_hepmc_skip
```

17.15.4 Unit tests

Test module, followed by the corresponding implementation module.

⟨eio_hepmc_ut.f90⟩≡
```
⟨File header⟩

module eio_hepmc_ut
  use unit_tests
  use eio_hepmc_uti

⟨Standard module head⟩

⟨EIO HepMC: public test⟩

contains

⟨EIO HepMC: test driver⟩

end module eio_hepmc_ut
```

⟨eio_hepmc_uti.f90⟩≡
```
⟨File header⟩

module eio_hepmc_uti

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use model_data
  use event_base
  use eio_data
  use eio_base

  use eio_hepmc

  use eio_base_ut, only: eio_prepare_test, eio_cleanup_test
  use eio_base_ut, only: eio_prepare_fallback_model, eio_cleanup_fallback_model

⟨Standard module head⟩

⟨EIO HepMC: test declarations⟩

contains
```

⟨*EIO HepMC: tests*⟩

```
end module eio_hepmc_uti
```

API: driver for the unit tests below.

⟨*EIO HepMC: public test*⟩≡

```
public :: eio_hepmc_test
```

⟨*EIO HepMC: test driver*⟩≡

```
subroutine eio_hepmc_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```

⟨*EIO HepMC: execute tests*⟩

```
end subroutine eio_hepmc_test
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO HepMC: execute tests*⟩≡

```
call test (eio_hepmc_1, "eio_hepmc_1", &
     "write event contents", &
     u, results)
```

⟨*EIO HepMC: test declarations*⟩≡

```
public :: eio_hepmc_1
```

⟨*EIO HepMC: tests*⟩≡

```
subroutine eio_hepmc_1 (u)
  integer, intent(in) :: u
  class(generic_event_t), pointer :: event
  type(event_sample_data_t) :: data
  class(eio_t), allocatable :: eio
  type(string_t) :: sample
  integer :: u_file, iostat
  character(116) :: buffer

  write (u, "(A)")  "* Test output: eio_hepmc_1"
  write (u, "(A)")  "*   Purpose: write a HepMC file"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize test process"

  call eio_prepare_test (event, unweighted=.false.)

  call data%init (1)
  data%n_beam = 2
  data%unweighted = .true.
  data%norm_mode = NORM_UNIT
  data%pdg_beam = 25
  data%energy_beam = 500
  data%proc_num_id = [42]
  data%cross_section(1) = 100
  data%error(1) = 1
  data%total_cross_section = sum (data%cross_section)
```

```
write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_hepmc_1"

allocate (eio_hepmc_t :: eio)
select type (eio)
type is (eio_hepmc_t)
   call eio%set_parameters ()
end select

call eio%init_out (sample, data)
call event%generate (1, [0._default, 0._default])

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* File contents (blanking out last two digits):"
write (u, "(A)")

u_file = free_unit ()
open (u_file, file = char (sample // ".hepmc"), &
     action = "read", status = "old")
do
   read (u_file, "(A)", iostat = iostat)  buffer
   if (iostat /= 0)  exit
   if (trim (buffer) == "")  cycle
   if (buffer(1:14) == "HepMC::Version")  cycle
   if (buffer(1:10) == "P 10001 25") &
        call buffer_blanker (buffer, 32, 55, 78)
   if (buffer(1:10) == "P 10002 25") &
        call buffer_blanker (buffer, 33, 56, 79)
   if (buffer(1:10) == "P 10003 25") &
        call buffer_blanker (buffer, 29, 53, 78, 101)
   if (buffer(1:10) == "P 10004 25") &
        call buffer_blanker (buffer, 28, 51, 76, 99)
   write (u, "(A)") trim (buffer)
end do
close (u_file)

write (u, "(A)")
write (u, "(A)")  "* Reset data"
write (u, "(A)")

deallocate (eio)
allocate (eio_hepmc_t :: eio)

select type (eio)
type is (eio_hepmc_t)
   call eio%set_parameters ()
```

```
    end select
    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_hepmc_1"

  contains

    subroutine buffer_blanker (buf, pos1, pos2, pos3, pos4)
      character(len=*), intent(inout) :: buf
      integer, intent(in) :: pos1, pos2, pos3
      integer, intent(in), optional :: pos4
      type(string_t) :: line
      line = var_str (trim (buf))
      line = replace (line, pos1, "XX")
      line = replace (line, pos2, "XX")
      line = replace (line, pos3, "XX")
      if (present (pos4)) then
         line = replace (line, pos4, "XX")
      end if
      line = replace (line, "4999999999999", "5000000000000")
      buf = char (line)
    end subroutine buffer_blanker

  end subroutine eio_hepmc_1
```

Test also the reading of HepMC events.

⟨*EIO HepMC: execute tests*⟩+≡

```
  call test (eio_hepmc_2, "eio_hepmc_2", &
       "read event contents", &
       u, results)
```

⟨*EIO HepMC: test declarations*⟩+≡

```
  public :: eio_hepmc_2
```

⟨*EIO HepMC: tests*⟩+≡

```
  subroutine eio_hepmc_2 (u)
    integer, intent(in) :: u
    class(model_data_t), pointer :: fallback_model
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(string_t) :: sample
    integer :: u_file, iostat, i_prc

    write (u, "(A)")  "* Test output: eio_hepmc_2"
    write (u, "(A)")  "*   Purpose: read a HepMC event"
    write (u, "(A)")

    write (u, "(A)")  "* Write a HepMC data file"
```

```
write (u, "(A)")

u_file = free_unit ()
sample = "eio_hepmc_2"
open (u_file, file = char (sample // ".hepmc"), &
     status = "replace", action = "readwrite")

write (u_file, "(A)")  "HepMC::Version 2.06.09"
write (u_file, "(A)")  "HepMC::IO_GenEvent-START_EVENT_LISTING"
write (u_file, "(A)")  "E 0 -1 -1.0000000000000000e+00 &
     &-1.0000000000000000e+00 &
     &-1.0000000000000000e+00 42 0 1 10001 10002 0 0"
write (u_file, "(A)")  "U GEV MM"
write (u_file, "(A)")  "V -1 0 0 0 0 0 2 2 0"
write (u_file, "(A)")  "P 10001 25 0 0 4.8412291827592713e+02 &
     &5.0000000000000000e+02 &
     &1.2499999999999989e+02 3 0 0 -1 0"
write (u_file, "(A)")  "P 10002 25 0 0 -4.8412291827592713e+02 &
     &5.0000000000000000e+02 &
     &1.2499999999999989e+02 3 0 0 -1 0"
write (u_file, "(A)")  "P 10003 25 -1.4960220911365536e+02 &
     &-4.6042825611414656e+02 &
     &0 5.0000000000000000e+02 1.2500000000000000e+02 1 0 0 0 0"
write (u_file, "(A)")  "P 10004 25 1.4960220911365536e+02 &
     &4.6042825611414656e+02 &
     &0 5.0000000000000000e+02 1.2500000000000000e+02 1 0 0 0 0"
write (u_file, "(A)")  "HepMC::IO_GenEvent-END_EVENT_LISTING"
close (u_file)

write (u, "(A)")  "* Initialize test process"
write (u, "(A)")

call eio_prepare_fallback_model (fallback_model)
call eio_prepare_test (event, unweighted=.false.)

allocate (eio_hepmc_t :: eio)
select type (eio)
type is (eio_hepmc_t)
   call eio%set_parameters (recover_beams = .false.)
end select
call eio%set_fallback_model (fallback_model)

call data%init (1)
data%n_beam = 2
data%unweighted = .true.
data%norm_mode = NORM_UNIT
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
call data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize"
write (u, "(A)")
```

```
    call eio%init_in (sample, data)
    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Read event"
    write (u, "(A)")

    call eio%input_i_prc (i_prc, iostat)

    select type (eio)
    type is (eio_hepmc_t)
       write (u, "(A,I0,A,I0)")  "Found process #", i_prc, &
            " with ID = ", eio%proc_num_id(i_prc)
    end select

    call eio%input_event (event, iostat)

    call event%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Read closing"
    write (u, "(A)")

    call eio%input_i_prc (i_prc, iostat)
    write (u, "(A,I0)")  "iostat = ", iostat

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio%final ()

    call eio_cleanup_test (event)
    call eio_cleanup_fallback_model (fallback_model)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_hepmc_2"

  end subroutine eio_hepmc_2
```

17.16 LCIO Output

The LCIO event record is standardized for the use with Linear e^+e^- colliders. It is a binary event format. We try our best at using it for both input and output.

⟨eio_lcio.f90⟩≡

```
  ⟨File header⟩

  module eio_lcio

  ⟨Use strings⟩
```

```
  use io_units
  use string_utils
  use diagnostics
  use particles
  use event_base
  use hep_events
  use eio_data
  use eio_base
  use lcio_interface
```

⟨*Standard module head*⟩

⟨*EIO LCIO: public*⟩

⟨*EIO LCIO: types*⟩

```
contains
```

⟨*EIO LCIO: procedures*⟩

```
end module eio_lcio
```

17.16.1 Type

A type `lcio_event` is introduced as container to store LCIO event data, particularly for splitting the reading into read out of the process index and the proper event data.

Note: the `keep_beams` flag is not supported.

⟨*EIO LCIO: public*⟩≡

```
  public :: eio_lcio_t
```

⟨*EIO LCIO: types*⟩≡

```
  type, extends (eio_t) :: eio_lcio_t
     logical :: writing = .false.
     logical :: reading = .false.
     type(event_sample_data_t) :: data
     logical :: recover_beams = .false.
     logical :: use_alpha_s_from_file = .false.
     logical :: use_scale_from_file = .false.
     type(lcio_writer_t) :: lcio_writer
     type(lcio_reader_t) :: lcio_reader
     type(lcio_run_header_t) :: lcio_run_hdr
     type(lcio_event_t) :: lcio_event
     integer, dimension(:), allocatable :: proc_num_id
   contains
   ⟨EIO LCIO: eio lcio: TBP⟩
  end type eio_lcio_t
```

17.16.2 Specific Methods

Set parameters that are specifically used with LCIO.

⟨*EIO LCIO: eio lcio: TBP*⟩≡

```
procedure :: set_parameters => eio_lcio_set_parameters
```

⟨*EIO LCIO: procedures*⟩≡

```
subroutine eio_lcio_set_parameters &
     (eio, recover_beams, use_alpha_s_from_file, use_scale_from_file, &
     extension)
  class(eio_lcio_t), intent(inout) :: eio
  logical, intent(in), optional :: recover_beams
  logical, intent(in), optional :: use_alpha_s_from_file
  logical, intent(in), optional :: use_scale_from_file
  type(string_t), intent(in), optional :: extension
  if (present (recover_beams))  eio%recover_beams = recover_beams
  if (present (use_alpha_s_from_file)) &
       eio%use_alpha_s_from_file = use_alpha_s_from_file
  if (present (use_scale_from_file)) &
       eio%use_scale_from_file = use_scale_from_file
  if (present (extension)) then
     eio%extension = extension
  else
     eio%extension = "slcio"
  end if
end subroutine eio_lcio_set_parameters
```

17.16.3 Common Methods

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO LCIO: eio lcio: TBP*⟩+≡

```
procedure :: write => eio_lcio_write
```

⟨*EIO LCIO: procedures*⟩+≡

```
subroutine eio_lcio_write (object, unit)
  class(eio_lcio_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "LCIO event stream:"
  if (object%writing) then
     write (u, "(3x,A,A)")  "Writing to file   = ", char (object%filename)
  else if (object%reading) then
     write (u, "(3x,A,A)")  "Reading from file = ", char (object%filename)
  else
     write (u, "(3x,A)")  "[closed]"
  end if
  write (u, "(3x,A,L1)")    "Recover beams     = ", object%recover_beams
  write (u, "(3x,A,L1)")    "Alpha_s from file = ", &
       object%use_alpha_s_from_file
  write (u, "(3x,A,L1)")    "Scale from file   = ", &
       object%use_scale_from_file
  write (u, "(3x,A,A,A)")     "File extension    = '", &
       char (object%extension), "'"
  if (allocated (object%proc_num_id)) then
     write (u, "(3x,A)")  "Numerical process IDs:"
```

```
      do i = 1, size (object%proc_num_id)
         write (u, "(5x,I0,': ',I0)")  i, object%proc_num_id(i)
      end do
   end if
end subroutine eio_lcio_write
```

Finalizer: close any open file.

⟨*EIO LCIO: eio lcio: TBP*⟩+≡

```
procedure :: final => eio_lcio_final
```

⟨*EIO LCIO: procedures*⟩+≡

```
subroutine eio_lcio_final (object)
  class(eio_lcio_t), intent(inout) :: object
  if (allocated (object%proc_num_id))  deallocate (object%proc_num_id)
  if (object%writing) then
     write (msg_buffer, "(A,A,A)")  "Events: closing LCIO file '", &
          char (object%filename), "'"
     call msg_message ()
     call lcio_writer_close (object%lcio_writer)
     object%writing = .false.
  else if (object%reading) then
     write (msg_buffer, "(A,A,A)")  "Events: closing LCIO file '", &
          char (object%filename), "'"
     call msg_message ()
     call lcio_reader_close (object%lcio_reader)
     object%reading = .false.
  end if
end subroutine eio_lcio_final
```

Split event file: increment the counter, close the current file, open a new one. If the file needs a header, repeat it for the new file.

⟨*EIO LCIO: eio lcio: TBP*⟩+≡

```
procedure :: split_out => eio_lcio_split_out
```

⟨*EIO LCIO: procedures*⟩+≡

```
subroutine eio_lcio_split_out (eio)
  class(eio_lcio_t), intent(inout) :: eio
  if (eio%split) then
     eio%split_index = eio%split_index + 1
     call eio%set_filename ()
     write (msg_buffer, "(A,A,A)")  "Events: writing to LCIO file '", &
          char (eio%filename), "'"
     call msg_message ()
     call lcio_writer_close (eio%lcio_writer)
     call lcio_writer_open_out (eio%lcio_writer, eio%filename)
  end if
end subroutine eio_lcio_split_out
```

Common initialization for input and output.

⟨*EIO LCIO: eio lcio: TBP*⟩+≡

```
procedure :: common_init => eio_lcio_common_init
```

⟨*EIO LCIO: procedures*⟩+≡

```
subroutine eio_lcio_common_init (eio, sample, data, extension)
  class(eio_lcio_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(string_t), intent(in), optional :: extension
  type(event_sample_data_t), intent(in), optional :: data
  if (.not. present (data)) &
       call msg_bug ("LCIO initialization: missing data")
  eio%data = data
  if (data%n_beam /= 2) &
       call msg_fatal ("LCIO: defined for scattering processes only")
  if (data%unweighted) then
     select case (data%norm_mode)
     case (NORM_UNIT)
     case default; call msg_fatal &
          ("LCIO: normalization for unweighted events must be '1'")
     end select
  else
     call msg_fatal ("LCIO: events must be unweighted")
  end if
  eio%sample = sample
  if (present (extension)) then
     eio%extension = extension
  end if
  call eio%set_filename ()
  allocate (eio%proc_num_id (data%n_proc), source = data%proc_num_id)
end subroutine eio_lcio_common_init
```

Initialize event writing.

⟨*EIO LCIO: eio lcio: TBP*⟩+≡

```
procedure :: init_out => eio_lcio_init_out
```

⟨*EIO LCIO: procedures*⟩+≡

```
subroutine eio_lcio_init_out (eio, sample, data, success, extension)
  class(eio_lcio_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(string_t), intent(in), optional :: extension
  type(event_sample_data_t), intent(in), optional :: data
  logical, intent(out), optional :: success
  call eio%set_splitting (data)
  call eio%common_init (sample, data, extension)
  write (msg_buffer, "(A,A,A)")  "Events: writing to LCIO file '", &
       char (eio%filename), "'"
  call msg_message ()
  eio%writing = .true.
  call lcio_writer_open_out (eio%lcio_writer, eio%filename)
  call lcio_run_header_init (eio%lcio_run_hdr)
  call lcio_run_header_write (eio%lcio_writer, eio%lcio_run_hdr)
  if (present (success))  success = .true.
end subroutine eio_lcio_init_out
```

Initialize event reading. For input, we do not (yet) support split event files.

⟨*EIO LCIO: eio lcio: TBP*⟩+≡

```
procedure :: init_in => eio_lcio_init_in
```

⟨*EIO LCIO: procedures*⟩+≡

```
subroutine eio_lcio_init_in (eio, sample, data, success, extension)
  class(eio_lcio_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(string_t), intent(in), optional :: extension
  type(event_sample_data_t), intent(inout), optional :: data
  logical, intent(out), optional :: success
  logical :: exist
  eio%split = .false.
  call eio%common_init (sample, data, extension)
  write (msg_buffer, "(A,A,A)")  "Events: reading from LCIO file '", &
       char (eio%filename), "'"
  call msg_message ()
  inquire (file = char (eio%filename), exist = exist)
  if (.not. exist)  call msg_fatal ("Events: LCIO file not found.")
  eio%reading = .true.
  call lcio_open_file (eio%lcio_reader, eio%filename)
  if (present (success))  success = .true.
end subroutine eio_lcio_init_in
```

Switch from input to output: reopen the file for reading.

⟨*EIO LCIO: eio lcio: TBP*⟩+≡

```
procedure :: switch_inout => eio_lcio_switch_inout
```

⟨*EIO LCIO: procedures*⟩+≡

```
subroutine eio_lcio_switch_inout (eio, success)
  class(eio_lcio_t), intent(inout) :: eio
  logical, intent(out), optional :: success
  call msg_bug ("LCIO: in-out switch not supported")
  if (present (success))  success = .false.
end subroutine eio_lcio_switch_inout
```

Output an event to the allocated LCIO writer.

⟨*EIO LCIO: eio lcio: TBP*⟩+≡

```
procedure :: output => eio_lcio_output
```

⟨*EIO LCIO: procedures*⟩+≡

```
subroutine eio_lcio_output (eio, event, i_prc, reading, passed, pacify)
  class(eio_lcio_t), intent(inout) :: eio
  class(generic_event_t), intent(in), target :: event
  integer, intent(in) :: i_prc
  logical, intent(in), optional :: reading, passed, pacify
  type(particle_set_t), pointer :: pset_ptr
  if (present (passed)) then
     if (.not. passed)  return
  end if
  if (eio%writing) then
     pset_ptr => event%get_particle_set_ptr ()
     call lcio_event_init (eio%lcio_event, &
           proc_id = eio%proc_num_id (i_prc), &
           event_id = event%get_index ())
     call lcio_event_from_particle_set (eio%lcio_event, pset_ptr)
     call lcio_event_set_sqrts (eio%lcio_event, event%get_sqrts ())
     call lcio_event_set_scale (eio%lcio_event, event%get_fac_scale ())
```

```
      call lcio_event_set_alpha_qcd (eio%lcio_event, event%get_alpha_s ())
      call lcio_event_set_xsec (eio%lcio_event, eio%data%cross_section(i_prc), &
           eio%data%error(i_prc))
      call lcio_event_set_polarization (eio%lcio_event, &
           event%get_polarization ())
      call lcio_event_set_beam_file (eio%lcio_event, &
           event%get_beam_file ())
      call lcio_event_set_process_name (eio%lcio_event, &
           event%get_process_name ())
      call lcio_event_write (eio%lcio_writer, eio%lcio_event)
      call lcio_event_final (eio%lcio_event)
   else
      call eio%write ()
      call msg_fatal ("LCIO file is not open for writing")
   end if
end subroutine eio_lcio_output
```

Input an event.

⟨*EIO LCIO: eio lcio: TBP*⟩+≡

```
procedure :: input_i_prc => eio_lcio_input_i_prc
procedure :: input_event => eio_lcio_input_event
```

⟨*EIO LCIO: procedures*⟩+≡

```
subroutine eio_lcio_input_i_prc (eio, i_prc, iostat)
  class(eio_lcio_t), intent(inout) :: eio
  integer, intent(out) :: i_prc
  integer, intent(out) :: iostat
  logical :: ok
  integer :: i, proc_num_id
  iostat = 0
  call lcio_read_event (eio%lcio_reader, eio%lcio_event, ok)
  if (.not. ok) then
     iostat = -1
     return
  end if
  proc_num_id = lcio_event_get_process_id (eio%lcio_event)
  i_prc = 0
  FIND_I_PRC: do i = 1, size (eio%proc_num_id)
     if (eio%proc_num_id(i) == proc_num_id) then
        i_prc = i
        exit FIND_I_PRC
     end if
  end do FIND_I_PRC
  if (i_prc == 0)  call err_index
contains
  subroutine err_index
    call msg_error ("LCIO: reading events: undefined process ID " &
         // char (str (proc_num_id)) // ", aborting read")
    iostat = 1
  end subroutine err_index
end subroutine eio_lcio_input_i_prc

subroutine eio_lcio_input_event (eio, event, iostat)
  class(eio_lcio_t), intent(inout) :: eio
```

```
      class(generic_event_t), intent(inout), target :: event
      integer, intent(out) :: iostat
      iostat = 0
      call event%reset ()
      call event%select (1, 1, 1)
      call lcio_to_event (event, eio%lcio_event, eio%fallback_model, &
           recover_beams = eio%recover_beams, &
           use_alpha_s = eio%use_alpha_s_from_file, &
           use_scale = eio%use_scale_from_file)
      call lcio_event_final (eio%lcio_event)
    end subroutine eio_lcio_input_event
```

⟨*EIO LCIO: eio lcio: TBP*⟩+≡

```
    procedure :: skip => eio_lcio_skip
```

⟨*EIO LCIO: procedures*⟩+≡

```
    subroutine eio_lcio_skip (eio, iostat)
      class(eio_lcio_t), intent(inout) :: eio
      integer, intent(out) :: iostat
      iostat = 0
    end subroutine eio_lcio_skip
```

17.16.4 Unit tests

Test module, followed by the corresponding implementation module.

⟨eio_lcio_ut.f90⟩≡

```
  ⟨File header⟩

  module eio_lcio_ut
    use unit_tests
    use eio_lcio_uti

  ⟨Standard module head⟩

  ⟨EIO LCIO: public test⟩

  contains

  ⟨EIO LCIO: test driver⟩

  end module eio_lcio_ut
```

⟨eio_lcio_uti.f90⟩≡

```
  ⟨File header⟩

  module eio_lcio_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use io_units
    use model_data
    use particles
    use event_base
```

```
  use eio_data
  use eio_base
  use hep_events
  use lcio_interface

  use eio_lcio

  use eio_base_ut, only: eio_prepare_test, eio_cleanup_test
  use eio_base_ut, only: eio_prepare_fallback_model, eio_cleanup_fallback_model
```

⟨*Standard module head*⟩

⟨*EIO LCIO: test declarations*⟩

```
contains
```

⟨*EIO LCIO: tests*⟩

```
end module eio_lcio_uti
```

API: driver for the unit tests below.

⟨*EIO LCIO: public test*⟩≡

```
  public :: eio_lcio_test
```

⟨*EIO LCIO: test driver*⟩≡

```
  subroutine eio_lcio_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```

⟨*EIO LCIO: execute tests*⟩

```
  end subroutine eio_lcio_test
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO LCIO: execute tests*⟩≡

```
  call test (eio_lcio_1, "eio_lcio_1", &
       "write event contents", &
       u, results)
```

⟨*EIO LCIO: test declarations*⟩≡

```
  public :: eio_lcio_1
```

⟨*EIO LCIO: tests*⟩≡

```
  subroutine eio_lcio_1 (u)
    integer, intent(in) :: u
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
    class(eio_t), allocatable :: eio
    type(particle_set_t), pointer :: pset_ptr
    type(string_t) :: sample
    integer :: u_file, iostat
    character(215) :: buffer

    write (u, "(A)")  "* Test output: eio_lcio_1"
    write (u, "(A)")  "*   Purpose: write a LCIO file"
```

```
write (u, "(A)")

write (u, "(A)")  "* Initialize test process"

call eio_prepare_test (event)

call data%init (1)
data%n_beam = 2
data%unweighted = .true.
data%norm_mode = NORM_UNIT
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_lcio_1"

allocate (eio_lcio_t :: eio)
select type (eio)
type is (eio_lcio_t)
   call eio%set_parameters ()
end select

call eio%init_out (sample, data)

call event%generate (1, [0._default, 0._default])
call event%pacify_particle_set ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()

write (u, "(A)")
write (u, "(A)")  "* Reset data"
write (u, "(A)")

deallocate (eio)
allocate (eio_lcio_t :: eio)

select type (eio)
type is (eio_lcio_t)
   call eio%set_parameters ()
end select
call eio%write (u)

write (u, "(A)")
write (u, "(A)")  "* Write LCIO file contents to ASCII file"
write (u, "(A)")
```

```
    select type (eio)
    type is (eio_lcio_t)
       call lcio_event_init (eio%lcio_event, &
            proc_id = 42, &
            event_id = event%get_index ())
       pset_ptr => event%get_particle_set_ptr ()
       call lcio_event_from_particle_set &
            (eio%lcio_event,  pset_ptr)
       call write_lcio_event (eio%lcio_event, var_str ("test_file.slcio"))
       call lcio_event_final (eio%lcio_event)
    end select

    write (u, "(A)")
    write (u, "(A)")  "* Read in ASCII contents of LCIO file"
    write (u, "(A)")

    u_file = free_unit ()
    open (u_file, file = "test_file.slcio", &
         action = "read", status = "old")
    do
       read (u_file, "(A)", iostat = iostat)  buffer
       if (iostat /= 0)  exit
       if (trim (buffer) == "")  cycle
       if (buffer(1:12) == " - timestamp")  cycle
       if (buffer(1:6) == " date:")  cycle
       write (u, "(A)") trim (buffer)
    end do
    close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio_cleanup_test (event)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_lcio_1"

  end subroutine eio_lcio_1
```

Test also the reading of LCIO events.

⟨*EIO LCIO: execute tests*⟩+≡

```
  call test (eio_lcio_2, "eio_lcio_2", &
       "read event contents", &
       u, results)
```

⟨*EIO LCIO: test declarations*⟩+≡

```
  public :: eio_lcio_2
```

⟨*EIO LCIO: tests*⟩+≡

```
  subroutine eio_lcio_2 (u)
    integer, intent(in) :: u
    class(model_data_t), pointer :: fallback_model
    class(generic_event_t), pointer :: event
    type(event_sample_data_t) :: data
```

```
class(eio_t), allocatable :: eio
type(string_t) :: sample
integer :: iostat, i_prc

write (u, "(A)")  "* Test output: eio_lcio_2"
write (u, "(A)")  "*   Purpose: read a LCIO event"
write (u, "(A)")

write (u, "(A)")  "* Initialize test process"

call eio_prepare_fallback_model (fallback_model)
call eio_prepare_test (event)

call data%init (1)
data%n_beam = 2
data%unweighted = .true.
data%norm_mode = NORM_UNIT
data%pdg_beam = 25
data%energy_beam = 500
data%proc_num_id = [42]
data%cross_section(1) = 100
data%error(1) = 1
data%total_cross_section = sum (data%cross_section)

write (u, "(A)")
write (u, "(A)")  "* Generate and write an event"
write (u, "(A)")

sample = "eio_lcio_2"

allocate (eio_lcio_t :: eio)
select type (eio)
type is (eio_lcio_t)
   call eio%set_parameters (recover_beams = .false.)
end select
call eio%set_fallback_model (fallback_model)

call eio%init_out (sample, data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()
call event%pacify_particle_set ()

call eio%output (event, i_prc = 1)
call eio%write (u)
call eio%final ()
deallocate (eio)

write (u, "(A)")
write (u, "(A)")  "* Initialize"
write (u, "(A)")

allocate (eio_lcio_t :: eio)
select type (eio)
type is (eio_lcio_t)
```

```
      call eio%set_parameters (recover_beams = .false.)
   end select
   call eio%set_fallback_model (fallback_model)

   call data%init (1)
   data%n_beam = 2
   data%unweighted = .true.
   data%norm_mode = NORM_UNIT
   data%pdg_beam = 25
   data%energy_beam = 500
   data%proc_num_id = [42]
   call data%write (u)
   write (u, *)

   write (u, "(A)")  "* Initialize"
   write (u, "(A)")

   call eio%init_in (sample, data)
   call eio%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Read event"
   write (u, "(A)")

   call eio%input_i_prc (i_prc, iostat)

   select type (eio)
   type is (eio_lcio_t)
      write (u, "(A,I0,A,I0)")  "Found process #", i_prc, &
           " with ID = ", eio%proc_num_id(i_prc)
   end select

   call eio%input_event (event, iostat)
   call event%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Read closing"
   write (u, "(A)")

   call eio%input_i_prc (i_prc, iostat)
   write (u, "(A,I0)")  "iostat = ", iostat

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call eio%final ()

   call eio_cleanup_test (event)
   call eio_cleanup_fallback_model (fallback_model)

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: eio_lcio_2"

end subroutine eio_lcio_2
```

Chapter 18

Phase Space

The abstract representation of a type that parameterizes phase space, with methods for construction and evaluation.

phs_base Abstract phase-space representation.

A simple implementation:

phs_single Parameterize the phase space of a single particle, i.e., the solid angle. This is useful only for very restricted problems, but it avoids the complexity of a generic approach in those trivial cases.

The standard implementation is called *wood* phase space. It consists of several auxiliary modules and the actual implementation module.

mappings Generate invariant masses and decay angles from given random numbers (or the inverse operation). Each mapping pertains to a particular node in a phase-space tree. Different mappings account for uniform distributions, resonances, zero-mass behavior, and so on.

phs_trees Phase space parameterizations for scattering processes are defined recursively as if there was an initial particle decaying. This module sets up a representation in terms of abstract trees, where each node gets a unique binary number. Each tree is stored as an array of branches, where integers indicate the connections. This emulates pointers in a transparent way. Real pointers would also be possible, but seem to be less efficient for this particular case.

phs_forests The type defined by this module collects the decay trees corresponding to a given process and the applicable mappings. To set this up, a file is read which is either written by the user or by the **cascades** module functions. The module also contains the routines that evaluate phase space, i.e., generate momenta from random numbers and back.

cascades This module is a pseudo Feynman diagram generator with the particular purpose of finding the phase space parameterizations best suited for a given process. It uses a model file to set up the possible vertices, generates all possible diagrams, identifies resonances and singularities, and simplifies the list by merging equivalent diagrams and dropping irrelevant

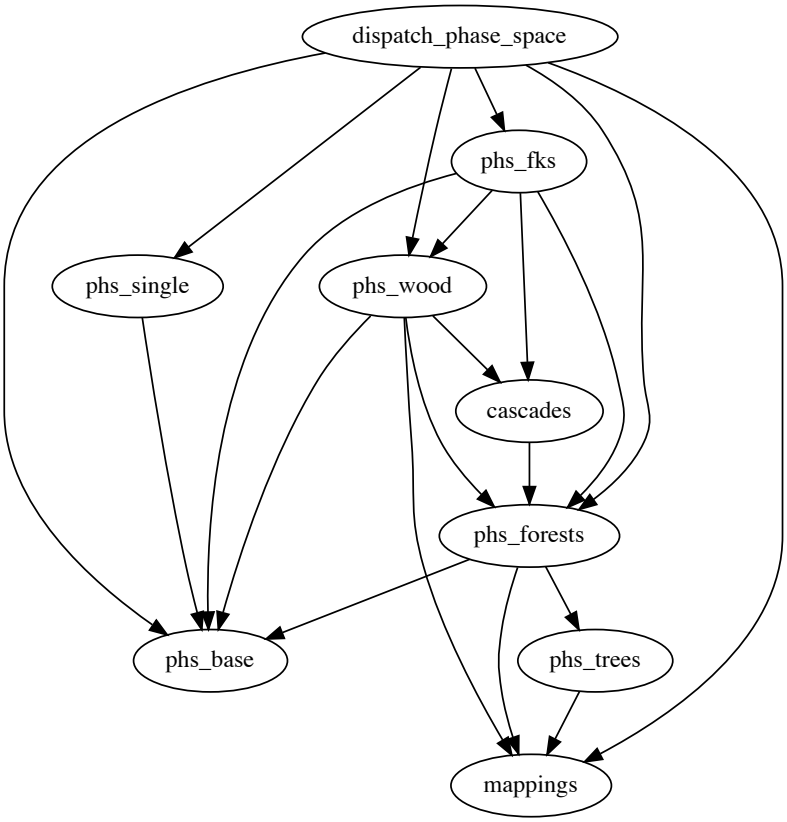

Figure 18.1: Module dependencies in `src/phase_space`.

ones. This process can be controlled at several points by user-defined parameters. Note that it depends on the particular values of particle masses, so it cannot be done before reading the input file.

phs_wood Make the functionality available in form of an implementation of the abstract phase-space type.

phs_fks Phase-space parameterization with modifications for the FKS scheme.

18.1 Abstract phase-space module

In this module we define an abstract base type (and a trivial test implementation) for multi-channel phase-space parameterizations.

⟨phs_base.f90⟩≡
```
⟨File header⟩

module phs_base

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use constants, only: TWOPI, TWOPI4
  use string_utils, only: split_string
  use format_defs, only: FMT_19
  use numeric_utils
  use diagnostics
  use md5
  use physics_defs
  use lorentz
  use model_data
  use flavors
  use process_constants

⟨Standard module head⟩

⟨PHS base: public⟩

⟨PHS base: types⟩

⟨PHS base: interfaces⟩

contains

⟨PHS base: procedures⟩

end module phs_base
```

18.1.1 Phase-space channels

The kinematics configuration may generate multiple parameterizations of phase space. Some of those have specific properties, such as a resonance in the s channel.

Channel properties

This is the abstract type for the channel properties. We need them as a data transfer container, so everything is public and transparent.

⟨PHS base: public⟩≡
```
public :: channel_prop_t
```

⟨PHS base: types⟩≡
```
type, abstract :: channel_prop_t
```

```
 contains
   procedure (channel_prop_to_string), deferred :: to_string
   generic :: operator (==) => is_equal
   procedure (channel_eq), deferred :: is_equal
end type channel_prop_t
```

⟨*PHS base: interfaces*⟩≡

```
abstract interface
   function channel_prop_to_string (object) result (string)
     import
     class(channel_prop_t), intent(in) :: object
     type(string_t) :: string
   end function channel_prop_to_string
end interface
```

⟨*PHS base: interfaces*⟩+≡

```
abstract interface
   function channel_eq (prop1, prop2) result (flag)
     import
     class(channel_prop_t), intent(in) :: prop1, prop2
     logical :: flag
   end function channel_eq
end interface
```

Here is a resonance as a channel property. Mass and width are stored here in physical units.

⟨*PHS base: public*⟩+≡

```
public :: resonance_t
```

⟨*PHS base: types*⟩+≡

```
type, extends (channel_prop_t) :: resonance_t
   real(default) :: mass = 0
   real(default) :: width = 0
 contains
   procedure :: to_string => resonance_to_string
   procedure :: is_equal => resonance_is_equal
end type resonance_t
```

Print mass and width.

⟨*PHS base: procedures*⟩≡

```
function resonance_to_string (object) result (string)
  class(resonance_t), intent(in) :: object
  type(string_t) :: string
  character(32) :: buffer
  string = "resonant: m ="
  write (buffer, "(" // FMT_19 // ")")  object%mass
  string = string // trim (buffer) // " GeV, w ="
  write (buffer, "(" // FMT_19 // ")")  object%width
  string = string // trim (buffer) // " GeV"
end function resonance_to_string
```

Equality.

⟨*PHS base: procedures*⟩+≡

```
function resonance_is_equal (prop1, prop2) result (flag)
  class(resonance_t), intent(in) :: prop1
  class(channel_prop_t), intent(in) :: prop2
  logical :: flag
  select type (prop2)
  type is (resonance_t)
     flag = prop1%mass == prop2%mass .and. prop1%width == prop2%width
  class default
     flag = .false.
  end select
end function resonance_is_equal
```

This is the limiting case of a resonance, namely an on-shell particle. We just store the mass in physical units.

⟨*PHS base: public*⟩+≡

```
public :: on_shell_t
```

⟨*PHS base: types*⟩+≡

```
type, extends (channel_prop_t) :: on_shell_t
   real(default) :: mass = 0
 contains
   procedure :: to_string => on_shell_to_string
   procedure :: is_equal => on_shell_is_equal
end type on_shell_t
```

Print mass and width.

⟨*PHS base: procedures*⟩+≡

```
function on_shell_to_string (object) result (string)
  class(on_shell_t), intent(in) :: object
  type(string_t) :: string
  character(32) :: buffer
  string = "on shell: m ="
  write (buffer, "(" // FMT_19 // ")")  object%mass
  string = string // trim (buffer) // " GeV"
end function on_shell_to_string
```

Equality.

⟨*PHS base: procedures*⟩+≡

```
function on_shell_is_equal (prop1, prop2) result (flag)
  class(on_shell_t), intent(in) :: prop1
  class(channel_prop_t), intent(in) :: prop2
  logical :: flag
  select type (prop2)
  type is (on_shell_t)
     flag = prop1%mass == prop2%mass
  class default
     flag = .false.
  end select
end function on_shell_is_equal
```

Channel equivalences

This type describes an equivalence. The current channel is equivalent to channel `c`. The equivalence involves a permutation `perm` of integration dimensions and, within each integration dimension, a mapping `mode`.

⟨*PHS base: types*⟩+≡
```
type :: phs_equivalence_t
   integer :: c = 0
   integer, dimension(:), allocatable :: perm
   integer, dimension(:), allocatable :: mode
 contains
 ⟨PHS base: phs equivalence: TBP⟩
end type phs_equivalence_t
```

The mapping modes are

⟨*PHS base: types*⟩+≡
```
integer, parameter, public :: &
     EQ_IDENTITY = 0, EQ_INVERT = 1, EQ_SYMMETRIC = 2, EQ_INVARIANT = 3
```

In particular, if a channel is equivalent to itself in the `EQ_SYMMETRIC` mode, the integrand can be assumed to be symmetric w.r.t. a reflection $x \to 1 - x$ of the correponding integration variable.

These are the associated tags, for output:

⟨*PHS base: types*⟩+≡
```
character, dimension(0:3), parameter :: TAG = ["+", "-", ":", "x"]
```

Write an equivalence.

⟨*PHS base: phs equivalence: TBP*⟩≡
```
procedure :: write => phs_equivalence_write
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_equivalence_write (object, unit)
  class(phs_equivalence_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, j
  u = given_output_unit (unit)
  write (u, "(5x,'=',1x,I0,1x)", advance = "no")  object%c
  if (allocated (object%perm)) then
     write (u, "(A)", advance = "no")  "("
     do j = 1, size (object%perm)
        if (j > 1)  write (u, "(1x)", advance = "no")
        write (u, "(I0,A1)", advance = "no") &
             object%perm(j), TAG(object%mode(j))
     end do
     write (u, "(A)")  ")"
  else
     write (u, "(A)")
  end if
end subroutine phs_equivalence_write
```

Initialize an equivalence. This allocates the `perm` and `mode` arrays with equal size.

⟨*PHS base: phs equivalence: TBP*⟩+≡
```
procedure :: init => phs_equivalence_init
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_equivalence_init (eq, n_dim)
  class(phs_equivalence_t), intent(out) :: eq
  integer, intent(in) :: n_dim
  allocate (eq%perm (n_dim), source = 0)
  allocate (eq%mode (n_dim), source = EQ_IDENTITY)
end subroutine phs_equivalence_init
```

Channel objects

The channel entry holds (optionally) specific properties.

`sf_channel` is the structure-function channel that corresponds to this phase-space channel. The structure-function channel may be set up with a specific mapping that depends on the phase-space channel properties. (The default setting is to leave the properties empty.)

⟨*PHS base: public*⟩+≡
```
public :: phs_channel_t
```

⟨*PHS base: types*⟩+≡
```
type :: phs_channel_t
   class(channel_prop_t), allocatable :: prop
   integer :: sf_channel = 1
   type(phs_equivalence_t), dimension(:), allocatable :: eq
 contains
 ⟨PHS base: phs channel: TBP⟩
end type phs_channel_t
```

Output.

⟨*PHS base: phs channel: TBP*⟩≡
```
procedure :: write => phs_channel_write
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_channel_write (object, unit)
  class(phs_channel_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, j
  u = given_output_unit (unit)
  write (u, "(1x,I0)", advance="no") object%sf_channel
  if (allocated (object%prop)) then
     write (u, "(1x,A)")  char (object%prop%to_string ())
  else
     write (u, *)
  end if
  if (allocated (object%eq)) then
     do j = 1, size (object%eq)
        call object%eq(j)%write (u)
     end do
  end if
```

```
end subroutine phs_channel_write
```

Identify the channel with an s-channel resonance.

⟨*PHS base: phs channel: TBP*⟩+≡

```
procedure :: set_resonant => channel_set_resonant
```

⟨*PHS base: procedures*⟩+≡

```
subroutine channel_set_resonant (channel, mass, width)
  class(phs_channel_t), intent(inout) :: channel
  real(default), intent(in) :: mass, width
  allocate (resonance_t :: channel%prop)
  select type (prop => channel%prop)
  type is (resonance_t)
     prop%mass = mass
     prop%width = width
  end select
end subroutine channel_set_resonant
```

Identify the channel with an on-shell particle.

⟨*PHS base: phs channel: TBP*⟩+≡

```
procedure :: set_on_shell => channel_set_on_shell
```

⟨*PHS base: procedures*⟩+≡

```
subroutine channel_set_on_shell (channel, mass)
  class(phs_channel_t), intent(inout) :: channel
  real(default), intent(in) :: mass
  allocate (on_shell_t :: channel%prop)
  select type (prop => channel%prop)
  type is (on_shell_t)
     prop%mass = mass
  end select
end subroutine channel_set_on_shell
```

18.1.2 Property collection

We can set up a list of all distinct channel properties for a given set of channels.

⟨*PHS base: public*⟩+≡

```
public :: phs_channel_collection_t
```

⟨*PHS base: types*⟩+≡

```
type :: prop_entry_t
   integer :: i = 0
   class(channel_prop_t), allocatable :: prop
   type(prop_entry_t), pointer :: next => null ()
end type prop_entry_t

type :: phs_channel_collection_t
   integer :: n = 0
   type(prop_entry_t), pointer :: first => null ()
 contains
 ⟨PHS base: phs channel collection: TBP⟩
end type phs_channel_collection_t
```

Finalizer for the list.

⟨*PHS base: phs channel collection: TBP*⟩≡
```
procedure :: final => phs_channel_collection_final
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_channel_collection_final (object)
  class(phs_channel_collection_t), intent(inout) :: object
  type(prop_entry_t), pointer :: entry
  do while (associated (object%first))
     entry => object%first
     object%first => entry%next
     deallocate (entry)
  end do
end subroutine phs_channel_collection_final
```

Output.

Note: eliminating the `string` auxiliary triggers an ICE in gfortran 4.7.2.

⟨*PHS base: phs channel collection: TBP*⟩+≡
```
procedure :: write => phs_channel_collection_write
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_channel_collection_write (object, unit)
  class(phs_channel_collection_t), intent(in) :: object
  integer, intent(in), optional :: unit
  type(prop_entry_t), pointer :: entry
  type(string_t) :: string
  integer :: u
  u = given_output_unit (unit)
  entry => object%first
  do while (associated (entry))
     if (allocated (entry%prop)) then
        string = entry%prop%to_string ()
        write (u, "(1x,I0,1x,A)")  entry%i, char (string)
     else
        write (u, "(1x,I0)")  entry%i
     end if
     entry => entry%next
  end do
end subroutine phs_channel_collection_write
```

Push a new property to the stack if it is not yet included. Simultaneously, set the `sf_channel` entry in the phase-space channel object to the index of the matching entry, or the new entry if there was no match.

⟨*PHS base: phs channel collection: TBP*⟩+≡
```
procedure :: push => phs_channel_collection_push
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_channel_collection_push (coll, channel)
  class(phs_channel_collection_t), intent(inout) :: coll
  type(phs_channel_t), intent(inout) :: channel
  type(prop_entry_t), pointer :: entry, new
  if (associated (coll%first)) then
     entry => coll%first
     do
```

```
         if (allocated (entry%prop)) then
            if (allocated (channel%prop)) then
               if (entry%prop == channel%prop) then
                  channel%sf_channel = entry%i
                  return
               end if
            end if
         else if (.not. allocated (channel%prop)) then
            channel%sf_channel = entry%i
            return
         end if
         if (associated (entry%next)) then
            entry => entry%next
         else
            exit
         end if
      end do
      allocate (new)
      entry%next => new
   else
      allocate (new)
      coll%first => new
   end if
   coll%n = coll%n + 1
   new%i = coll%n
   channel%sf_channel = new%i
   if (allocated (channel%prop)) then
      allocate (new%prop, source = channel%prop)
   end if
end subroutine phs_channel_collection_push
```

Return the number of collected distinct channels.

⟨*PHS base: phs channel collection: TBP*⟩+≡

```
procedure :: get_n => phs_channel_collection_get_n
```

⟨*PHS base: procedures*⟩+≡

```
function phs_channel_collection_get_n (coll) result (n)
  class(phs_channel_collection_t), intent(in) :: coll
  integer :: n
  n = coll%n
end function phs_channel_collection_get_n
```

Return a specific channel (property object).

⟨*PHS base: phs channel collection: TBP*⟩+≡

```
procedure :: get_entry => phs_channel_collection_get_entry
```

⟨*PHS base: procedures*⟩+≡

```
subroutine phs_channel_collection_get_entry (coll, i, prop)
  class(phs_channel_collection_t), intent(in) :: coll
  integer, intent(in) :: i
  class(channel_prop_t), intent(out), allocatable :: prop
  type(prop_entry_t), pointer :: entry
  integer :: k
  if (i > 0 .and. i <= coll%n) then
```

```
      entry => coll%first
      do k = 2, i
         entry => entry%next
      end do
      if (allocated (entry%prop)) then
         if (allocated (prop))  deallocate (prop)
         allocate (prop, source = entry%prop)
      end if
   else
      call msg_bug ("PHS channel collection: get entry: illegal index")
   end if
end subroutine phs_channel_collection_get_entry
```

18.1.3 Kinematics configuration

Here, we store the universal information that is specifically relevant for phase-space generation. It is a subset of the process data, supplemented by basic information on phase-space parameterization channels.

A concrete implementation will contain more data, that describe the phase space in detail.

MD5 sums: the phase space setup depends on the process, it depends on the model parameters (the masses, that is), and on the configuration parameters. (It doesn't depend on the QCD setup.)

⟨*PHS base: public*⟩+≡

```
public :: phs_config_t
```

⟨*PHS base: types*⟩+≡

```
type, abstract :: phs_config_t
   ! private
   type(string_t) :: id
   integer :: n_in = 0
   integer :: n_out = 0
   integer :: n_tot = 0
   integer :: n_state = 0
   integer :: n_par = 0
   integer :: n_channel = 0
   real(default) :: sqrts = 0
   logical :: sqrts_fixed = .true.
   logical :: cm_frame = .true.
   logical :: azimuthal_dependence = .false.
   integer, dimension(:), allocatable :: dim_flat
   logical :: provides_equivalences = .false.
   logical :: provides_chains = .false.
   logical :: vis_channels = .false.
   integer, dimension(:), allocatable :: chain
   class(model_data_t), pointer :: model => null ()
   type(flavor_t), dimension(:,:), allocatable :: flv
   type(phs_channel_t), dimension(:), allocatable :: channel
   character(32) :: md5sum_process = ""
   character(32) :: md5sum_model_par = ""
   character(32) :: md5sum_phs_config = ""
   integer :: nlo_type
```

```
 contains
 ⟨PHS base: phs config: TBP⟩
end type phs_config_t
```

Finalizer, deferred.

⟨*PHS base: phs config: TBP*⟩≡

```
procedure (phs_config_final), deferred :: final
```

⟨*PHS base: interfaces*⟩+≡

```
abstract interface
   subroutine phs_config_final (object)
     import
     class(phs_config_t), intent(inout) :: object
   end subroutine phs_config_final
end interface
```

Output. We provide an implementation for the output of the base-type contents and an interface for the actual write method.

⟨*PHS base: phs config: TBP*⟩+≡

```
procedure (phs_config_write), deferred :: write
procedure :: base_write => phs_config_write
```

⟨*PHS base: procedures*⟩+≡

```
subroutine phs_config_write (object, unit, include_id)
  class(phs_config_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: include_id
  integer :: u, i, j
  integer :: n_tot_flv
  logical :: use_id
  n_tot_flv = object%n_tot
  u = given_output_unit (unit)
  use_id = .true.; if (present (include_id)) use_id = include_id
  if (use_id) write (u, "(3x,A,A,A)") "ID        = '", char (object%id), "'"
  write (u, "(3x,A,I0)")  "n_in      = ", object%n_in
  write (u, "(3x,A,I0)")  "n_out     = ", object%n_out
  write (u, "(3x,A,I0)")  "n_tot     = ", object%n_tot
  write (u, "(3x,A,I0)")  "n_state   = ", object%n_state
  write (u, "(3x,A,I0)")  "n_par     = ", object%n_par
  write (u, "(3x,A,I0)")  "n_channel = ", object%n_channel
  write (u, "(3x,A," // FMT_19 // ")")  "sqrts     = ", object%sqrts
  write (u, "(3x,A,L1)")  "s_fixed   = ", object%sqrts_fixed
  write (u, "(3x,A,L1)")  "cm_frame  = ", object%cm_frame
  write (u, "(3x,A,L1)")  "azim.dep. = ", object%azimuthal_dependence
  if (allocated (object%dim_flat)) then
     write (u, "(3x,A,I0)")  "flat dim. = ", object%dim_flat
  end if
  write (u, "(1x,A)")  "Flavor combinations:"
  do i = 1, object%n_state
     write (u, "(3x,I0,':')", advance="no")  i
!       do j = 1, object%n_tot
     do j = 1, n_tot_flv
        write (u, "(1x,A)", advance="no")  char (object%flv(j,i)%get_name ())
     end do
```

```
      write (u, "(A)")
   end do
   if (allocated (object%channel)) then
      write (u, "(1x,A)")  "Phase-space / structure-function channels:"
      do i = 1, object%n_channel
         write (u, "(3x,I0,':')", advance="no") i
         call object%channel(i)%write (u)
      end do
   end if
   if (object%md5sum_process /= "") then
      write (u, "(3x,A,A,A)") "MD5 sum (process)    = '", &
           object%md5sum_process, "'"
   end if
   if (object%md5sum_model_par /= "") then
      write (u, "(3x,A,A,A)") "MD5 sum (model par)  = '", &
           object%md5sum_model_par, "'"
   end if
   if (object%md5sum_phs_config /= "") then
      write (u, "(3x,A,A,A)") "MD5 sum (phs config) = '", &
           object%md5sum_phs_config, "'"
   end if
end subroutine phs_config_write
```

Similarly, a basic initializer and an interface. The model pointer is taken as an argument; we may verify that this has the expected model name.

The intent is `inout`. We want to be able to set parameters in advance.

⟨*PHS base: phs config: TBP*⟩+≡

```
procedure :: init => phs_config_init
```

⟨*PHS base: procedures*⟩+≡

```
subroutine phs_config_init (phs_config, data, model)
  class(phs_config_t), intent(inout) :: phs_config
  type(process_constants_t), intent(in) :: data
  class(model_data_t), intent(in), target :: model
  integer :: i, j
  phs_config%id = data%id
  phs_config%n_in  = data%n_in
  phs_config%n_out = data%n_out
  phs_config%n_tot = data%n_in + data%n_out
  phs_config%n_state = data%n_flv
  if (data%model_name == model%get_name ()) then
     phs_config%model => model
  else
     call msg_bug ("phs_config_init: model name mismatch")
  end if
  allocate (phs_config%flv (phs_config%n_tot, phs_config%n_state))
  do i = 1, phs_config%n_state
     do j = 1, phs_config%n_tot
        call phs_config%flv(j,i)%init (data%flv_state(j,i), &
             phs_config%model)
     end do
  end do
  phs_config%md5sum_process = data%md5sum
end subroutine phs_config_init
```

⟨*PHS base: phs config: TBP*⟩+≡
```
procedure :: set_component_index => phs_config_set_component_index
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_config_set_component_index (phs_config, index)
  class(phs_config_t), intent(inout) :: phs_config
  integer, intent(in) :: index
  type(string_t), dimension(:), allocatable :: id
  type(string_t) :: suffix
  integer :: i, n
  suffix = var_str ('i') // int2string (index)
  call split_string (phs_config%id, var_str ('_'), id)
  phs_config%id = var_str ('')
  n = size (id) - 1
  do i = 1, n
     phs_config%id = phs_config%id // id(i) // var_str ('_')
  end do
  phs_config%id = phs_config%id // suffix
end subroutine phs_config_set_component_index
```

This procedure should complete the phase-space configuration. We need the `sqrts` value as overall scale, which is known only after the beams have been defined. The procedure should determine the number of channels, their properties (if any), and allocate and fill the `channel` array accordingly.

⟨*PHS base: phs config: TBP*⟩+≡
```
procedure (phs_config_configure), deferred :: configure
```

⟨*PHS base: interfaces*⟩+≡
```
abstract interface
   subroutine phs_config_configure (phs_config, sqrts, &
        sqrts_fixed, cm_frame, azimuthal_dependence, rebuild, ignore_mismatch, &
        nlo_type)
     import
     class(phs_config_t), intent(inout) :: phs_config
     real(default), intent(in) :: sqrts
     logical, intent(in), optional :: sqrts_fixed
     logical, intent(in), optional :: cm_frame
     logical, intent(in), optional :: azimuthal_dependence
     logical, intent(in), optional :: rebuild
     logical, intent(in), optional :: ignore_mismatch
     integer, intent(in), optional :: nlo_type
   end subroutine phs_config_configure
end interface
```

Manually assign structure-function channel indices to the phase-space channel objects. (Used by a test routine.)

⟨*PHS base: phs config: TBP*⟩+≡
```
procedure :: set_sf_channel => phs_config_set_sf_channel
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_config_set_sf_channel (phs_config, sf_channel)
  class(phs_config_t), intent(inout) :: phs_config
```

```
    integer, dimension(:), intent(in) :: sf_channel
    phs_config%channel%sf_channel = sf_channel
  end subroutine phs_config_set_sf_channel
```

Collect new channels not yet in the collection from this phase-space configuration object. At the same time, assign structure-function channels.

⟨*PHS base: phs config: TBP*⟩+≡

```
  procedure :: collect_channels => phs_config_collect_channels
```

⟨*PHS base: procedures*⟩+≡

```
  subroutine phs_config_collect_channels (phs_config, coll)
    class(phs_config_t), intent(inout) :: phs_config
    type(phs_channel_collection_t), intent(inout) :: coll
    integer :: c
    do c = 1, phs_config%n_channel
       call coll%push (phs_config%channel(c))
    end do
  end subroutine phs_config_collect_channels
```

Compute the MD5 sum. We abuse the `write` method. In type implementations, `write` should only display information that is relevant for the MD5 sum. The data include the process MD5 sum which is taken from the process constants, and the MD5 sum of the model parameters. This may change, so it is computed here.

⟨*PHS base: phs config: TBP*⟩+≡

```
  procedure :: compute_md5sum => phs_config_compute_md5sum
```

⟨*PHS base: procedures*⟩+≡

```
  subroutine phs_config_compute_md5sum (phs_config, include_id)
    class(phs_config_t), intent(inout) :: phs_config
    logical, intent(in), optional :: include_id
    integer :: u
    phs_config%md5sum_model_par = phs_config%model%get_parameters_md5sum ()
    phs_config%md5sum_phs_config = ""
    u = free_unit ()
    open (u, status = "scratch", action = "readwrite")
    call phs_config%write (u, include_id)
    rewind (u)
    phs_config%md5sum_phs_config = md5sum (u)
    close (u)
  end subroutine phs_config_compute_md5sum
```

Print an informative message after phase-space configuration.

⟨*PHS base: phs config: TBP*⟩+≡

```
  procedure (phs_startup_message), deferred :: startup_message
  procedure :: base_startup_message => phs_startup_message
```

⟨*PHS base: procedures*⟩+≡

```
  subroutine phs_startup_message (phs_config, unit)
    class(phs_config_t), intent(in) :: phs_config
    integer, intent(in), optional :: unit
    write (msg_buffer, "(A,3(1x,I0,1x,A))") &
         "Phase space:", &
```

```
        phs_config%n_channel, "channels,", &
        phs_config%n_par, "dimensions"
    call msg_message (unit = unit)
end subroutine phs_startup_message
```

This procedure should be implemented such that the phase-space configuration object allocates a phase-space instance of matching type.

⟨*PHS base: phs config: TBP*⟩+≡

```
procedure (phs_config_allocate_instance), nopass, deferred :: &
     allocate_instance
```

⟨*PHS base: interfaces*⟩+≡

```
abstract interface
   subroutine phs_config_allocate_instance (phs)
     import
     class(phs_t), intent(inout), pointer :: phs
   end subroutine phs_config_allocate_instance
end interface
```

18.1.4 Extract data

Return the number of MC input parameters.

⟨*PHS base: phs config: TBP*⟩+≡

```
procedure :: get_n_par => phs_config_get_n_par
```

⟨*PHS base: procedures*⟩+≡

```
function phs_config_get_n_par (phs_config) result (n)
  class(phs_config_t), intent(in) :: phs_config
  integer :: n
  n = phs_config%n_par
end function phs_config_get_n_par
```

Return dimensions (parameter indices) for which the phase-space dimension is flat, so integration and event generation can be simplified.

⟨*PHS base: phs config: TBP*⟩+≡

```
procedure :: get_flat_dimensions => phs_config_get_flat_dimensions
```

⟨*PHS base: procedures*⟩+≡

```
function phs_config_get_flat_dimensions (phs_config) result (dim_flat)
  class(phs_config_t), intent(in) :: phs_config
  integer, dimension(:), allocatable :: dim_flat
  if (allocated (phs_config%dim_flat)) then
     allocate (dim_flat (size (phs_config%dim_flat)))
     dim_flat = phs_config%dim_flat
  else
     allocate (dim_flat (0))
  end if
end function phs_config_get_flat_dimensions
```

Return the number of phase-space channels.

⟨*PHS base: phs config: TBP*⟩+≡

```
procedure :: get_n_channel => phs_config_get_n_channel
```

⟨*PHS base: procedures*⟩+≡

```
function phs_config_get_n_channel (phs_config) result (n)
  class(phs_config_t), intent(in) :: phs_config
  integer :: n
  n = phs_config%n_channel
end function phs_config_get_n_channel
```

Return the structure-function channel that corresponds to the phase-space channel `c`. If the channel array is not allocated (which happens if there is no structure function), return zero.

⟨*PHS base: phs config: TBP*⟩+≡

```
procedure :: get_sf_channel => phs_config_get_sf_channel
```

⟨*PHS base: procedures*⟩+≡

```
function phs_config_get_sf_channel (phs_config, c) result (c_sf)
  class(phs_config_t), intent(in) :: phs_config
  integer, intent(in) :: c
  integer :: c_sf
  if (allocated (phs_config%channel)) then
     c_sf = phs_config%channel(c)%sf_channel
  else
     c_sf = 0
  end if
end function phs_config_get_sf_channel
```

Return the mass(es) of the incoming particle(s). We take the first flavor combination in the array, assuming that masses must be degenerate among flavors.

⟨*PHS base: phs config: TBP*⟩+≡

```
procedure :: get_masses_in => phs_config_get_masses_in
```

⟨*PHS base: procedures*⟩+≡

```
subroutine phs_config_get_masses_in (phs_config, m)
  class(phs_config_t), intent(in) :: phs_config
  real(default), dimension(:), intent(out) :: m
  integer :: i
  do i = 1, phs_config%n_in
     m(i) = phs_config%flv(i,1)%get_mass ()
  end do
end subroutine phs_config_get_masses_in
```

Return the MD5 sum of the configuration.

⟨*PHS base: phs config: TBP*⟩+≡

```
procedure :: get_md5sum => phs_config_get_md5sum
```

⟨*PHS base: procedures*⟩+≡

```
function phs_config_get_md5sum (phs_config) result (md5sum)
  class(phs_config_t), intent(in) :: phs_config
  character(32) :: md5sum
  md5sum = phs_config%md5sum_phs_config
end function phs_config_get_md5sum
```

18.1.5 Phase-space point instance

The `phs_t` object holds the workspace for phase-space generation. In the base object, we have the MC input parameters `r` and the Jacobian factor `f`, for each channel, and the incoming and outgoing momenta.

Note: The `active_channel` array is not used yet, all elements are initialized with `.true.`. It should be touched by the integrator if it decides to drop irrelevant channels.

⟨*PHS base: public*⟩+≡
```
public :: phs_t
```

⟨*PHS base: types*⟩+≡
```
type, abstract :: phs_t
   class(phs_config_t), pointer :: config => null ()
   logical :: r_defined = .false.
   integer :: selected_channel = 0
   logical, dimension(:), allocatable :: active_channel
   real(default), dimension(:,:), allocatable :: r
   real(default), dimension(:), allocatable :: f
   real(default), dimension(:), allocatable :: m_in
   real(default), dimension(:), allocatable :: m_out
   real(default) :: flux = 0
   real(default) :: volume = 0
   type(lorentz_transformation_t) :: lt_cm_to_lab
   logical :: p_defined = .false.
   real(default) :: sqrts_hat = 0
   type(vector4_t), dimension(:), allocatable :: p
   logical :: q_defined = .false.
   type(vector4_t), dimension(:), allocatable :: q
 contains
 ⟨PHS base: phs: TBP⟩
end type phs_t
```

Output. Since phase space may get complicated, we include a `verbose` option for the abstract `write` procedure.

⟨*PHS base: phs: TBP*⟩≡
```
procedure (phs_write), deferred :: write
```

⟨*PHS base: interfaces*⟩+≡
```
abstract interface
   subroutine phs_write (object, unit, verbose)
     import
     class(phs_t), intent(in) :: object
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: verbose
   end subroutine phs_write
end interface
```

This procedure can be called to print the contents of the base type.

⟨*PHS base: phs: TBP*⟩+≡
```
procedure :: base_write => phs_base_write
```

⟨*PHS base: procedures*⟩+≡

```
subroutine phs_base_write (object, unit)
  class(phs_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, c, i
  u = given_output_unit (unit)
  write (u, "(1x,A)", advance="no")  "Partonic phase space: parameters"
  if (object%r_defined) then
     write (u, *)
  else
     write (u, "(1x,A)")  "[undefined]"
  end if
  write (u, "(3x,A,999(1x," // FMT_19 // "))") "m_in    =", object%m_in
  write (u, "(3x,A,999(1x," // FMT_19 // "))") "m_out   =", object%m_out
  write (u, "(3x,A," // FMT_19 // ")")  "Flux   = ", object%flux
  write (u, "(3x,A," // FMT_19 // ")")  "Volume = ", object%volume
  if (allocated (object%f)) then
     do c = 1, size (object%r, 2)
        write (u, "(1x,A,I0,A)", advance="no")  "Channel #", c, ":"
        if (c == object%selected_channel) then
           write (u, "(1x,A)")  "[selected]"
        else
           write (u, *)
        end if
        write (u, "(3x,A)", advance="no")  "r ="
        do i = 1, size (object%r, 1)
           write (u, "(1x,F9.7)", advance="no")  object%r(i,c)
        end do
        write (u, *)
        write (u, "(3x,A,1x,ES13.7)")  "f =", object%f(c)
     end do
  end if
  write (u, "(1x,A)")  "Partonic phase space: momenta"
  if (object%p_defined) then
     write (u, "(3x,A," // FMT_19 // ")")  "sqrts  = ", object%sqrts_hat
  end if
  write (u, "(1x,A)", advance="no")  "Incoming:"
  if (object%p_defined) then
     write (u, *)
  else
     write (u, "(1x,A)")  "[undefined]"
  end if
  if (allocated (object%p)) then
     do i = 1, size (object%p)
        call vector4_write (object%p(i), u)
     end do
  end if
  write (u, "(1x,A)", advance="no")  "Outgoing:"
  if (object%q_defined) then
     write (u, *)
  else
     write (u, "(1x,A)")  "[undefined]"
  end if
  if (allocated (object%q)) then
```

```
      do i = 1, size (object%q)
         call vector4_write (object%q(i), u)
      end do
   end if
   if (object%p_defined .and. .not. object%config%cm_frame) then
      write (u, "(1x,A)")  "Transformation c.m -> lab frame"
      call lorentz_transformation_write (object%lt_cm_to_lab, u)
   end if
end subroutine phs_base_write
```

Finalizer. The base type does not need it, but extensions may.

⟨*PHS base: phs: TBP*⟩+≡

```
procedure (phs_final), deferred :: final
```

⟨*PHS base: interfaces*⟩+≡

```
abstract interface
   subroutine phs_final (object)
     import
     class(phs_t), intent(inout) :: object
   end subroutine phs_final
end interface
```

Initializer. Everything should be contained in the `process_data` configuration object, so we can require a universal interface.

⟨*PHS base: phs: TBP*⟩+≡

```
procedure (phs_init), deferred :: init
```

⟨*PHS base: interfaces*⟩+≡

```
abstract interface
   subroutine phs_init (phs, phs_config)
     import
     class(phs_t), intent(out) :: phs
     class(phs_config_t), intent(in), target :: phs_config
   end subroutine phs_init
end interface
```

The base version will just allocate the arrays. It should be called at the beginning of the implementation of `phs_init`.

⟨*PHS base: phs: TBP*⟩+≡

```
procedure :: base_init => phs_base_init
```

⟨*PHS base: procedures*⟩+≡

```
subroutine phs_base_init (phs, phs_config)
  class(phs_t), intent(out) :: phs
  class(phs_config_t), intent(in), target :: phs_config
  real(default), dimension(phs_config%n_in) :: m_in
  real(default), dimension(phs_config%n_out) :: m_out
  phs%config => phs_config
  allocate (phs%active_channel (phs%config%n_channel))
  phs%active_channel = .true.
  allocate (phs%r (phs%config%n_par, phs%config%n_channel));  phs%r = 0
  allocate (phs%f (phs%config%n_channel));                    phs%f = 0
  allocate (phs%p (phs%config%n_in))
```

```
    !!! !!! !!! Workaround for gfortran 5.0 ICE
    m_in  = phs_config%flv(:phs_config%n_in, 1)%get_mass ()
    m_out = phs_config%flv(phs_config%n_in+1:, 1)%get_mass ()
    allocate (phs%m_in  (phs%config%n_in), source = m_in)
    !!! allocate (phs%m_in  (phs%config%n_in), &
    !!!      source = phs_config%flv(:phs_config%n_in, 1)%get_mass ())
    allocate (phs%q (phs%config%n_out))
    allocate (phs%m_out (phs%config%n_out), source = m_out)
    !!! allocate (phs%m_out (phs%config%n_out), &
    !!!      source = phs_config%flv(phs_config%n_in+1:, 1)%get_mass ())
    call phs%compute_flux ()
  end subroutine phs_base_init
```

Manually select a channel.

⟨*PHS base: phs: TBP*⟩+≡

```
  procedure :: select_channel => phs_base_select_channel
```

⟨*PHS base: procedures*⟩+≡

```
  subroutine phs_base_select_channel (phs, channel)
    class(phs_t), intent(inout) :: phs
    integer, intent(in), optional :: channel
    if (present (channel)) then
       phs%selected_channel = channel
    else
       phs%selected_channel = 0
    end if
  end subroutine phs_base_select_channel
```

Set incoming momenta. Assume that array shapes match. If requested, compute the Lorentz transformation from the c.m. to the lab frame and apply that transformation to the incoming momenta.

In the c.m. frame, the sum of three-momenta is zero. In a scattering process, the z axis is the direction of the first beam, the second beam is along the negative z axis. The transformation from the c.m. to the lab frame is a rotation from the z axis to the boost axis followed by a boost, such that the c.m. momenta are transformed into the lab-frame momenta. In a decay process, we just boost along the flight direction, without rotation.

⟨*PHS base: phs: TBP*⟩+≡

```
  procedure :: set_incoming_momenta => phs_set_incoming_momenta
```

⟨*PHS base: procedures*⟩+≡

```
  subroutine phs_set_incoming_momenta (phs, p)
    class(phs_t), intent(inout) :: phs
    type(vector4_t), dimension(:), intent(in) :: p
    type(vector4_t) :: p0, p1
    type(lorentz_transformation_t) :: lt0, lt0_inv, lt_inv
    !!! !!! !!! Workaround for standard-semantics ifort 16.0 bug
    integer :: i
    phs%p = p
    if (phs%config%cm_frame) then
       phs%sqrts_hat = phs%config%sqrts
       phs%p = p
       phs%lt_cm_to_lab = identity
```

```
  else
     p0 = sum (p)
     if (phs%config%sqrts_fixed) then
        phs%sqrts_hat = phs%config%sqrts
     else
        phs%sqrts_hat = p0 ** 1
     end if
     lt0 = boost (p0, phs%sqrts_hat)
     select case (phs%config%n_in)
     case (1)
        phs%lt_cm_to_lab = lt0
     case (2)
        !!! !!! !!! Workaround for standard-semantics ifort 16.0 bug
        lt0_inv = inverse (lt0)
        p1 = lt0_inv * p(1)
        !!! p1 = inverse (lt0) * p(1)
        phs%lt_cm_to_lab = lt0 * rotation_to_2nd (3, space_part (p1))
     end select
     !!! !!! !!! Workaround for standard-semantics ifort 16.0 bug
     lt_inv = inverse (phs%lt_cm_to_lab)
     do i = 1, size (p)
        phs%p(i) = lt_inv * p(i)
     end do
     !!! phs%p = inverse (phs%lt_cm_to_lab) * p
  end if
  phs%p_defined = .true.
end subroutine phs_set_incoming_momenta
```

Set outgoing momenta. Assume that array shapes match. The incoming momenta must be known, so can apply the Lorentz transformation from c.m. to lab (inverse) to the momenta.

⟨*PHS base: phs: TBP*⟩+≡

```
procedure :: set_outgoing_momenta => phs_set_outgoing_momenta
```

⟨*PHS base: procedures*⟩+≡

```
subroutine phs_set_outgoing_momenta (phs, q)
  class(phs_t), intent(inout) :: phs
  type(vector4_t), dimension(:), intent(in) :: q
  !!! !!! !!! Workaround for standard-semantics ifort 16.0 bug
  type(lorentz_transformation_t) :: l_inv
  integer :: i
  if (phs%p_defined) then
     if (phs%config%cm_frame) then
        phs%q = q
     else
        l_inv = inverse (phs%lt_cm_to_lab)
        do i = 1, size (q)
           phs%q(i) = l_inv * q(i)
        end do
        !!! phs%q = inverse (phs%lt_cm_to_lab) * q
     end if
     phs%q_defined = .true.
  end if
end subroutine phs_set_outgoing_momenta
```

Return outgoing momenta. Apply the c.m. to lab transformation if necessary.

⟨*PHS base: phs: TBP*⟩+≡
```
procedure :: get_outgoing_momenta => phs_get_outgoing_momenta
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_get_outgoing_momenta (phs, q)
  class(phs_t), intent(in) :: phs
  type(vector4_t), dimension(:), intent(out) :: q
  if (phs%p_defined .and. phs%q_defined) then
     if (phs%config%cm_frame) then
        q = phs%q
     else
        q = phs%lt_cm_to_lab * phs%q
     end if
  else
     q = vector4_null
  end if
end subroutine phs_get_outgoing_momenta
```

⟨*PHS base: phs: TBP*⟩+≡
```
procedure :: is_cm_frame => phs_is_cm_frame
```

⟨*PHS base: procedures*⟩+≡
```
function phs_is_cm_frame (phs) result (cm_frame)
  logical :: cm_frame
  class(phs_t), intent(in) :: phs
  cm_frame = phs%config%cm_frame
end function phs_is_cm_frame
```

⟨*PHS base: phs: TBP*⟩+≡
```
procedure :: get_n_tot => phs_get_n_tot
```

⟨*PHS base: procedures*⟩+≡
```
elemental function phs_get_n_tot (phs) result (n_tot)
  integer :: n_tot
  class(phs_t), intent(in) :: phs
  n_tot = phs%config%n_tot
end function phs_get_n_tot
```

⟨*PHS base: phs: TBP*⟩+≡
```
procedure :: set_lorentz_transformation => phs_set_lorentz_transformation
```

⟨*PHS base: procedures*⟩+≡
```
subroutine phs_set_lorentz_transformation (phs, lt)
  class(phs_t), intent(inout) :: phs
  type(lorentz_transformation_t), intent(in) :: lt
  phs%lt_cm_to_lab = lt
end subroutine phs_set_lorentz_transformation
```

⟨*PHS base: phs: TBP*⟩+≡
```
procedure :: get_lorentz_transformation => phs_get_lorentz_transformation
```

⟨*PHS base: procedures*⟩+≡

```
function phs_get_lorentz_transformation (phs) result (lt)
  type(lorentz_transformation_t) :: lt
  class(phs_t), intent(in) :: phs
  lt = phs%lt_cm_to_lab
end function phs_get_lorentz_transformation
```

Return the input parameter array for a channel.

⟨*PHS base: phs: TBP*⟩+≡

```
procedure :: get_mcpar => phs_get_mcpar
```

⟨*PHS base: procedures*⟩+≡

```
subroutine phs_get_mcpar (phs, c, r)
  class(phs_t), intent(in) :: phs
  integer, intent(in) :: c
  real(default), dimension(:), intent(out) :: r
  if (phs%r_defined) then
     r = phs%r(:,c)
  else
     r = 0
  end if
end subroutine phs_get_mcpar
```

Return the Jacobian factor for a channel.

⟨*PHS base: phs: TBP*⟩+≡

```
procedure :: get_f => phs_get_f
```

⟨*PHS base: procedures*⟩+≡

```
function phs_get_f (phs, c) result (f)
  class(phs_t), intent(in) :: phs
  integer, intent(in) :: c
  real(default) :: f
  if (phs%r_defined) then
     f = phs%f(c)
  else
     f = 0
  end if
end function phs_get_f
```

Return the overall factor, which is the product of the flux factor for the incoming partons and the phase-space volume for the outgoing partons.

⟨*PHS base: phs: TBP*⟩+≡

```
procedure :: get_overall_factor => phs_get_overall_factor
```

⟨*PHS base: procedures*⟩+≡

```
function phs_get_overall_factor (phs) result (f)
  class(phs_t), intent(in) :: phs
  real(default) :: f
  f = phs%flux * phs%volume
end function phs_get_overall_factor
```

Compute flux factor. We do this during initialization (when the incoming momenta `p` are undefined), unless `sqrts` is variable. We do this again once for each phase-space point, but then we skip the calculation if `sqrts` is fixed.

There are three different flux factors.

1. For a decaying massive particle, the factor is

$$f = (2\pi)^4/(2M) \tag{18.1}$$

2. For a $2 \to n$ scattering process with $n > 1$, the factor is

$$f = (2\pi)^4/(2\sqrt{\lambda}) \tag{18.2}$$

   where for massless incoming particles, $\sqrt{\lambda} = s$.

3. For a $2 \to 1$ on-shell production process, the factor includes an extra $1/(2\pi)^3$ factor and a $1/m^2$ factor from the phase-space delta function $\delta(x_1 x_2 - m^2/s)$, which originate from the one-particle phase space that we integrate out.

$$f = 2\pi/(2sm^2) \tag{18.3}$$

   The delta function is handled by the structure-function parameterization.

⟨*PHS base: phs: TBP*⟩+≡

```
procedure :: compute_flux => phs_compute_flux
```

⟨*PHS base: procedures*⟩+≡

```
subroutine phs_compute_flux (phs)
  class(phs_t), intent(inout) :: phs
  real(default) :: s_hat, lda
  select case (phs%config%n_in)
  case (1)
     if (.not. phs%p_defined) then
        phs%flux = twopi4 / (2 * phs%m_in(1))
     end if
  case (2)
     if (phs%p_defined) then
        if (phs%config%sqrts_fixed) then
           return
        else
           s_hat = sum (phs%p) ** 2
        end if
     else
        if (phs%config%sqrts_fixed) then
           s_hat = phs%config%sqrts ** 2
        else
           return
        end if
     end if
     select case (phs%config%n_out)
     case (2:)
        lda = lambda (s_hat, phs%m_in(1) ** 2, phs%m_in(2) ** 2)
        if (lda > 0) then
           phs%flux = conv * twopi4 / (2 * sqrt (lda))
        else
```

```
            phs%flux = 0
         end if
      case (1)
         phs%flux = conv * twopi &
              / (2 * phs%config%sqrts ** 2 * phs%m_out(1) ** 2)
      case default
         phs%flux = 0
      end select
   end select
end subroutine phs_compute_flux
```

Evaluate the phase-space point for a particular channel and compute momenta, Jacobian, and phase-space volume. This is, of course, deferred to the implementation.

⟨*PHS base: phs: TBP*⟩+≡

```
procedure (phs_evaluate_selected_channel), deferred :: &
     evaluate_selected_channel
```

⟨*PHS base: interfaces*⟩+≡

```
abstract interface
   subroutine phs_evaluate_selected_channel (phs, c_in, r_in)
     import
     class(phs_t), intent(inout) :: phs
     integer, intent(in) :: c_in
     real(default), dimension(:), intent(in) :: r_in
   end subroutine phs_evaluate_selected_channel
end interface
```

Compute the inverse mappings to completely fill the `r` and `f` arrays, for the non-selected channels.

⟨*PHS base: phs: TBP*⟩+≡

```
procedure (phs_evaluate_other_channels), deferred :: &
     evaluate_other_channels
```

⟨*PHS base: interfaces*⟩+≡

```
abstract interface
   subroutine phs_evaluate_other_channels (phs, c_in)
     import
     class(phs_t), intent(inout) :: phs
     integer, intent(in) :: c_in
   end subroutine phs_evaluate_other_channels
end interface
```

Inverse evaluation. If all momenta are known, we compute the inverse mappings to fill the `r` and `f` arrays.

⟨*PHS base: phs: TBP*⟩+≡

```
procedure (phs_inverse), deferred :: inverse
```

⟨*PHS base: interfaces*⟩+≡

```
abstract interface
   subroutine phs_inverse (phs)
     import
     class(phs_t), intent(inout) :: phs
```

```
      end subroutine phs_inverse
   end interface
```

⟨*PHS base: phs: TBP*⟩+≡

```
   procedure :: get_sqrts => phs_get_sqrts
```

⟨*PHS base: procedures*⟩+≡

```
   function phs_get_sqrts (phs) result (sqrts)
     real(default) :: sqrts
     class(phs_t), intent(in) :: phs
     sqrts = phs%config%sqrts
   end function phs_get_sqrts
```

Uniform angular distribution

These procedures implement the uniform angular distribution, generated from two parameters x_1 and x_2:

$$\cos\theta = 1 - 2x_1, \qquad \phi = 2\pi x_2 \tag{18.4}$$

We generate a rotation (Lorentz transformation) which rotates the positive z axis into this point on the unit sphere. This rotation is applied to the `p` momenta, which are assumed to be back-to-back, on-shell, and with the correct mass.

We do not compute a Jacobian (constant). The uniform distribution is assumed to be normalized.

⟨*PHS base: public*⟩+≡

```
   public :: compute_kinematics_solid_angle
```

⟨*PHS base: procedures*⟩+≡

```
   subroutine compute_kinematics_solid_angle (p, q, x)
     type(vector4_t), dimension(2), intent(in) :: p
     type(vector4_t), dimension(2), intent(out) :: q
     real(default), dimension(2), intent(in) :: x
     real(default) :: ct, st, phi
     type(lorentz_transformation_t) :: rot
     integer :: i
     ct = 1 - 2*x(1)
     st = sqrt (1 - ct**2)
     phi = twopi * x(2)
     rot = rotation (phi, 3) * rotation (ct, st, 2)
     do i = 1, 2
        q(i) = rot * p(i)
     end do
   end subroutine compute_kinematics_solid_angle
```

This is the inverse transformation. We assume that the outgoing momenta are rotated versions of the incoming momenta, back-to-back. Thus, we determine the angles from $q(1)$ alone. `p` is unused.

⟨*PHS base: public*⟩+≡

```
   public :: inverse_kinematics_solid_angle
```

⟨*PHS base: procedures*⟩+≡

```
subroutine inverse_kinematics_solid_angle (p, q, x)
  type(vector4_t), dimension(:), intent(in) :: p
  type(vector4_t), dimension(2), intent(in) :: q
  real(default), dimension(2), intent(out) :: x
  real(default) :: ct, phi
  ct = polar_angle_ct (q(1))
  phi = azimuthal_angle (q(1))
  x(1) = (1 - ct) / 2
  x(2) = phi / twopi
end subroutine inverse_kinematics_solid_angle
```

18.1.6 Auxiliary stuff

The `pacify` subroutine, which is provided by the Lorentz module, has the purpose of setting numbers to zero which are (by comparing with a `tolerance` parameter) considered equivalent with zero. This is useful for numerical checks.

⟨*PHS base: public*⟩+≡

```
public :: pacify
```

⟨*PHS base: interfaces*⟩+≡

```
interface pacify
   module procedure pacify_phs
end interface pacify
```

⟨*PHS base: procedures*⟩+≡

```
subroutine pacify_phs (phs)
  class(phs_t), intent(inout) :: phs
  if (phs%p_defined) then
     call pacify (phs%p, 30 * epsilon (1._default) * phs%config%sqrts)
     call pacify (phs%lt_cm_to_lab, 30 * epsilon (1._default))
  end if
  if (phs%q_defined) then
     call pacify (phs%q, 30 * epsilon (1._default) * phs%config%sqrts)
  end if
end subroutine pacify_phs
```

18.1.7 Unit tests

Test module, followed by the corresponding implementation module.

⟨`phs_base_ut.f90`⟩≡

```
⟨File header⟩

module phs_base_ut
  use unit_tests
  use phs_base_uti

⟨Standard module head⟩

⟨PHS base: public test⟩
```

```
  ⟨PHS base: public test auxiliary⟩

  contains

  ⟨PHS base: test driver⟩

  end module phs_base_ut
```

⟨phs_base_uti.f90⟩≡

```
  ⟨File header⟩

  module phs_base_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use diagnostics
    use io_units
    use format_defs, only: FMT_19
    use physics_defs, only: BORN
    use lorentz
    use flavors
    use model_data
    use process_constants

    use phs_base

  ⟨Standard module head⟩

  ⟨PHS base: public test auxiliary⟩

  ⟨PHS base: test declarations⟩

  ⟨PHS base: test types⟩

  contains

  ⟨PHS base: tests⟩

  ⟨PHS base: test auxiliary⟩

  end module phs_base_uti
```

API: driver for the unit tests below.

⟨*PHS base: public test*⟩≡

```
    public :: phs_base_test
```

⟨*PHS base: test driver*⟩≡

```
    subroutine phs_base_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨PHS base: execute tests⟩
    end subroutine phs_base_test
```

Test process data

We provide a procedure that initializes a test case for the process constants. This set of process data contains just the minimal contents that we need for the phase space. The rest is left uninitialized.

⟨*PHS base: public test auxiliary*⟩≡
```
public :: init_test_process_data
```

⟨*PHS base: test auxiliary*⟩≡
```
subroutine init_test_process_data (id, data)
  type(process_constants_t), intent(out) :: data
  type(string_t), intent(in), optional :: id
  if (present (id)) then
     data%id = id
  else
     data%id = "testproc"
  end if
  data%model_name = "Test"
  data%n_in = 2
  data%n_out = 2
  data%n_flv = 1
  allocate (data%flv_state (data%n_in + data%n_out, data%n_flv))
  data%flv_state = 25
end subroutine init_test_process_data
```

This is the variant for a decay process.

⟨*PHS base: public test auxiliary*⟩+≡
```
public :: init_test_decay_data
```

⟨*PHS base: test auxiliary*⟩+≡
```
subroutine init_test_decay_data (id, data)
  type(process_constants_t), intent(out) :: data
  type(string_t), intent(in), optional :: id
  if (present (id)) then
     data%id = id
  else
     data%id = "testproc"
  end if
  data%model_name = "Test"
  data%n_in = 1
  data%n_out = 2
  data%n_flv = 1
  allocate (data%flv_state (data%n_in + data%n_out, data%n_flv))
  data%flv_state(:,1) = [25, 6, -6]
end subroutine init_test_decay_data
```

Test kinematics configuration

This is a trivial implementation of the `phs_config_t` configuration object.

⟨*PHS base: public test auxiliary*⟩+≡
```
public :: phs_test_config_t
```

⟨*PHS base: test types*⟩≡

```
type, extends (phs_config_t) :: phs_test_config_t
   logical :: create_equivalences = .false.
 contains
   procedure :: final => phs_test_config_final
   procedure :: write => phs_test_config_write
   procedure :: configure => phs_test_config_configure
   procedure :: startup_message => phs_test_config_startup_message
   procedure, nopass :: allocate_instance => phs_test_config_allocate_instance
end type phs_test_config_t
```

The finalizer is empty.

⟨*PHS base: test auxiliary*⟩+≡

```
subroutine phs_test_config_final (object)
  class(phs_test_config_t), intent(inout) :: object
end subroutine phs_test_config_final
```

The `cm_frame` parameter is not tested here; we defer this to the `phs_single` implementation.

⟨*PHS base: test auxiliary*⟩+≡

```
subroutine phs_test_config_write (object, unit, include_id)
  class(phs_test_config_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: include_id
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Partonic phase-space configuration:"
  call object%base_write (unit)
end subroutine phs_test_config_write

subroutine phs_test_config_configure (phs_config, sqrts, &
     sqrts_fixed, cm_frame, azimuthal_dependence, rebuild, &
     ignore_mismatch, nlo_type)
  class(phs_test_config_t), intent(inout) :: phs_config
  real(default), intent(in) :: sqrts
  logical, intent(in), optional :: sqrts_fixed
  logical, intent(in), optional :: cm_frame
  logical, intent(in), optional :: azimuthal_dependence
  logical, intent(in), optional :: rebuild
  logical, intent(in), optional :: ignore_mismatch
  integer, intent(in), optional :: nlo_type
  phs_config%n_channel = 2
  phs_config%n_par = 2
  phs_config%sqrts = sqrts
  if (.not. present (nlo_type)) &
    phs_config%nlo_type = BORN
  if (present (sqrts_fixed)) then
     phs_config%sqrts_fixed = sqrts_fixed
  end if
  if (present (cm_frame)) then
     phs_config%cm_frame = cm_frame
  end if
  if (present (azimuthal_dependence)) then
```

```
      phs_config%azimuthal_dependence = azimuthal_dependence
    end if
    if (allocated (phs_config%channel))  deallocate (phs_config%channel)
    allocate (phs_config%channel (phs_config%n_channel))
    if (phs_config%create_equivalences) then
       call setup_test_equivalences (phs_config)
       call setup_test_channel_props (phs_config)
    end if
    call phs_config%compute_md5sum ()
  end subroutine phs_test_config_configure
```

If requested, we make up an arbitrary set of equivalences.

⟨*PHS base: test auxiliary*⟩+≡

```
  subroutine setup_test_equivalences (phs_config)
    class(phs_test_config_t), intent(inout) :: phs_config
    integer :: i
    associate (channel => phs_config%channel(1))
      allocate (channel%eq (2))
      do i = 1, size (channel%eq)
         call channel%eq(i)%init (phs_config%n_par)
      end do
      associate (eq => channel%eq(1))
        eq%c = 1;  eq%perm = [1, 2];  eq%mode = [EQ_IDENTITY, EQ_SYMMETRIC]
      end associate
      associate (eq => channel%eq(2))
        eq%c = 2;  eq%perm = [2, 1];  eq%mode = [EQ_INVARIANT, EQ_IDENTITY]
      end associate
    end associate
  end subroutine setup_test_equivalences
```

Ditto, for channel properties.

⟨*PHS base: test auxiliary*⟩+≡

```
  subroutine setup_test_channel_props (phs_config)
    class(phs_test_config_t), intent(inout) :: phs_config
    associate (channel => phs_config%channel(2))
      call channel%set_resonant (140._default, 3.1415_default)
    end associate
  end subroutine setup_test_channel_props
```

Startup message

⟨*PHS base: test auxiliary*⟩+≡

```
  subroutine phs_test_config_startup_message (phs_config, unit)
    class(phs_test_config_t), intent(in) :: phs_config
    integer, intent(in), optional :: unit
    call phs_config%base_startup_message (unit)
    write (msg_buffer, "(A)") "Phase space: Test"
    call msg_message (unit = unit)
  end subroutine phs_test_config_startup_message
```

The instance type that matches `phs_test_config_t` is `phs_test_t`.

⟨*PHS base: test auxiliary*⟩+≡

```
  subroutine phs_test_config_allocate_instance (phs)
```

```
    class(phs_t), intent(inout), pointer :: phs
    allocate (phs_test_t :: phs)
  end subroutine phs_test_config_allocate_instance
```

Test kinematics implementation

This implementation of kinematics generates a simple two-particle configuration from the incoming momenta. The incoming momenta must be in the c.m. system, all masses equal.

There are two channels: one generates $\cos\theta$ and ϕ uniformly, in the other channel we map the r_1 parameter which belongs to $\cos\theta$.

We should store the mass parameter that we need.

⟨*PHS base: public test auxiliary*⟩+≡

```
  public :: phs_test_t
```

⟨*PHS base: test types*⟩+≡

```
  type, extends (phs_t) :: phs_test_t
     real(default) :: m = 0
     real(default), dimension(:), allocatable :: x
   contains
   ⟨PHS base: phs test: TBP⟩
  end type phs_test_t
```

Output. The specific data are displayed only if `verbose` is set.

⟨*PHS base: phs test: TBP*⟩≡

```
  procedure :: write => phs_test_write
```

⟨*PHS base: test auxiliary*⟩+≡

```
  subroutine phs_test_write (object, unit, verbose)
    class(phs_test_t), intent(in) :: object
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    integer :: u
    logical :: verb
    u = given_output_unit (unit)
    verb = .false.;  if (present (verbose))  verb = verbose
    if (verb) then
       write (u, "(1x,A)")  "Partonic phase space: data"
       write (u, "(3x,A," // FMT_19 // ")")  "m = ", object%m
    end if
    call object%base_write (u)
  end subroutine phs_test_write
```

The finalizer is empty.

⟨*PHS base: phs test: TBP*⟩+≡

```
  procedure :: final => phs_test_final
```

⟨*PHS base: test auxiliary*⟩+≡

```
  subroutine phs_test_final (object)
    class(phs_test_t), intent(inout) :: object
  end subroutine phs_test_final
```

Initialization: set the mass value.

⟨*PHS base: phs test: TBP*⟩+≡

```
procedure :: init => phs_test_init
```

⟨*PHS base: test auxiliary*⟩+≡

```
subroutine phs_test_init (phs, phs_config)
  class(phs_test_t), intent(out) :: phs
  class(phs_config_t), intent(in), target :: phs_config
  call phs%base_init (phs_config)
  phs%m = phs%config%flv(1,1)%get_mass ()
  allocate (phs%x (phs_config%n_par), source = 0._default)
end subroutine phs_test_init
```

Evaluation. In channel 1, we uniformly generate $\cos\theta$ and ϕ, with Jacobian normalized to one. In channel 2, we prepend a mapping $r_1 \to r_1^{(}1/3)$ with Jacobian $f = 3r_1^2$.

The component `x` is allocated in the first subroutine, used and deallocated in the second one.

⟨*PHS base: phs test: TBP*⟩+≡

```
procedure :: evaluate_selected_channel => phs_test_evaluate_selected_channel
procedure :: evaluate_other_channels => phs_test_evaluate_other_channels
```

⟨*PHS base: test auxiliary*⟩+≡

```
subroutine phs_test_evaluate_selected_channel (phs, c_in, r_in)
  class(phs_test_t), intent(inout) :: phs
  integer, intent(in) :: c_in
  real(default), intent(in), dimension(:) :: r_in
  if (phs%p_defined) then
     call phs%select_channel (c_in)
     phs%r(:,c_in) = r_in
     select case (c_in)
     case (1)
        phs%x = r_in
     case (2)
        phs%x(1) = r_in(1) ** (1 / 3._default)
        phs%x(2) = r_in(2)
     end select
     call compute_kinematics_solid_angle (phs%p, phs%q, phs%x)
     phs%volume = 1
     phs%q_defined = .true.
  end if
end subroutine phs_test_evaluate_selected_channel

subroutine phs_test_evaluate_other_channels (phs, c_in)
  class(phs_test_t), intent(inout) :: phs
  integer, intent(in) :: c_in
  integer :: c, n_channel
  if (phs%p_defined) then
     n_channel = phs%config%n_channel
     do c = 1, n_channel
        if (c /= c_in) then
           call inverse_kinematics_solid_angle (phs%p, phs%q, phs%x)
           select case (c)
           case (1)
```

```
            phs%r(:,c) = phs%x
         case (2)
            phs%r(1,c) = phs%x(1) ** 3
            phs%r(2,c) = phs%x(2)
         end select
      end if
   end do
   phs%f(1) = 1
   if (phs%r(1,2) /= 0) then
      phs%f(2) = 1 / (3 * phs%r(1,2) ** (2/3._default))
   else
      phs%f(2) = 0
   end if
   phs%r_defined = .true.
 end if
end subroutine phs_test_evaluate_other_channels
```

Inverse evaluation.

⟨*PHS base: phs test: TBP*⟩+≡

```
procedure :: inverse => phs_test_inverse
```

⟨*PHS base: test auxiliary*⟩+≡

```
subroutine phs_test_inverse (phs)
  class(phs_test_t), intent(inout) :: phs
  integer :: c, n_channel
  real(default), dimension(:), allocatable :: x
  if (phs%p_defined .and. phs%q_defined) then
     call phs%select_channel ()
     n_channel = phs%config%n_channel
     allocate (x (phs%config%n_par))
     do c = 1, n_channel
        call inverse_kinematics_solid_angle (phs%p, phs%q, x)
        select case (c)
        case (1)
           phs%r(:,c) = x
        case (2)
           phs%r(1,c) = x(1) ** 3
           phs%r(2,c) = x(2)
        end select
     end do
     phs%f(1) = 1
     if (phs%r(1,2) /= 0) then
        phs%f(2) = 1 / (3 * phs%r(1,2) ** (2/3._default))
     else
        phs%f(2) = 0
     end if
     phs%volume = 1
     phs%r_defined = .true.
  end if
end subroutine phs_test_inverse
```

Phase-space configuration data

Construct and display a test phase-space configuration object.

⟨*PHS base: execute tests*⟩≡
```
call test (phs_base_1, "phs_base_1", &
     "phase-space configuration", &
     u, results)
```

⟨*PHS base: test declarations*⟩≡
```
public :: phs_base_1
```

⟨*PHS base: tests*⟩≡
```
subroutine phs_base_1 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(process_constants_t) :: process_data
  class(phs_config_t), allocatable :: phs_data

  write (u, "(A)")  "* Test output: phs_base_1"
  write (u, "(A)")  "*   Purpose: initialize and display &
       &test phase-space configuration data"
  write (u, "(A)")

  call model%init_test ()

  write (u, "(A)")  "* Initialize a process and a matching &
       &phase-space configuration"
  write (u, "(A)")

  call init_test_process_data (var_str ("phs_base_1"), process_data)

  allocate (phs_test_config_t :: phs_data)
  call phs_data%init (process_data, model)

  call phs_data%write (u)

  call phs_data%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: phs_base_1"

end subroutine phs_base_1
```

Phase space evaluation

Compute kinematics for given parameters, also invert the calculation.

⟨*PHS base: execute tests*⟩+≡
```
call test (phs_base_2, "phs_base_2", &
     "phase-space evaluation", &
     u, results)
```

⟨*PHS base: test declarations*⟩+≡
```
public :: phs_base_2
```

⟨*PHS base: tests*⟩+≡

```
subroutine phs_base_2 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t) :: flv
  type(process_constants_t) :: process_data
  real(default) :: sqrts, E
  class(phs_config_t), allocatable, target :: phs_data
  class(phs_t), pointer :: phs => null ()
  type(vector4_t), dimension(2) :: p, q

  write (u, "(A)")  "* Test output: phs_base_2"
  write (u, "(A)")  "*   Purpose: test simple two-channel phase space"
  write (u, "(A)")

  call model%init_test ()
  call flv%init (25, model)

  write (u, "(A)")  "* Initialize a process and a matching &
       &phase-space configuration"
  write (u, "(A)")

  call init_test_process_data (var_str ("phs_base_2"), process_data)

  allocate (phs_test_config_t :: phs_data)
  call phs_data%init (process_data, model)

  sqrts = 1000._default
  call phs_data%configure (sqrts)

  call phs_data%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize the phase-space instance"
  write (u, "(A)")

  call phs_data%allocate_instance (phs)
  select type (phs)
  type is (phs_test_t)
     call phs%init (phs_data)
  end select

  call phs%write (u, verbose=.true.)

  write (u, "(A)")
  write (u, "(A)")  "* Set incoming momenta"
  write (u, "(A)")

  E = sqrts / 2
  p(1) = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
  p(2) = vector4_moving (E,-sqrt (E**2 - flv%get_mass ()**2), 3)

  call phs%set_incoming_momenta (p)
  call phs%compute_flux ()
```

```
  call phs%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Compute phase-space point in channel 1 &
       &for x = 0.5, 0.125"
  write (u, "(A)")

  call phs%evaluate_selected_channel (1, [0.5_default, 0.125_default])
  call phs%evaluate_other_channels (1)
  call phs%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Compute phase-space point in channel 2 &
       &for x = 0.125, 0.125"
  write (u, "(A)")

  call phs%evaluate_selected_channel (2, [0.125_default, 0.125_default])
  call phs%evaluate_other_channels (2)
  call phs%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Inverse kinematics"
  write (u, "(A)")

  call phs%get_outgoing_momenta (q)
  deallocate (phs)
  call phs_data%allocate_instance (phs)
  call phs%init (phs_data)

  sqrts = 1000._default
  select type (phs_data)
  type is (phs_test_config_t)
     call phs_data%configure (sqrts)
  end select

  call phs%set_incoming_momenta (p)
  call phs%compute_flux ()
  call phs%set_outgoing_momenta (q)

  call phs%inverse ()
  call phs%write (u)

  call phs%final ()
  deallocate (phs)

  call phs_data%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: phs_base_2"

end subroutine phs_base_2
```

Phase-space equivalences

Construct a test phase-space configuration which contains channel equivalences.

⟨*PHS base: execute tests*⟩+≡

```
call test (phs_base_3, "phs_base_3", &
     "channel equivalences", &
     u, results)
```

⟨*PHS base: test declarations*⟩+≡

```
public :: phs_base_3
```

⟨*PHS base: tests*⟩+≡

```
subroutine phs_base_3 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(process_constants_t) :: process_data
  class(phs_config_t), allocatable :: phs_data

  write (u, "(A)")  "* Test output: phs_base_3"
  write (u, "(A)")  "*   Purpose: construct phase-space configuration data &
       &with equivalences"
  write (u, "(A)")

  call model%init_test ()

  write (u, "(A)")  "* Initialize a process and a matching &
       &phase-space configuration"
  write (u, "(A)")

  call init_test_process_data (var_str ("phs_base_3"), process_data)

  allocate (phs_test_config_t :: phs_data)
  call phs_data%init (process_data, model)
  select type (phs_data)
  type is (phs_test_config_t)
     phs_data%create_equivalences = .true.
  end select

  call phs_data%configure (1000._default)
  call phs_data%write (u)

  call phs_data%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: phs_base_3"

end subroutine phs_base_3
```

MD5 sum checks

Construct a test phase-space configuration, compute and compare MD5 sums.

⟨*PHS base: execute tests*⟩+≡

```
call test (phs_base_4, "phs_base_4", &
```

```
        "MD5 sum", &
        u, results)
```

⟨*PHS base: test declarations*⟩+≡

```
    public :: phs_base_4
```

⟨*PHS base: tests*⟩+≡

```
    subroutine phs_base_4 (u)
      integer, intent(in) :: u
      type(model_data_t), target :: model
      type(process_constants_t) :: process_data
      class(phs_config_t), allocatable :: phs_data

      write (u, "(A)")  "* Test output: phs_base_4"
      write (u, "(A)")  "*   Purpose: compute and compare MD5 sums"
      write (u, "(A)")

      call model%init_test ()

      write (u, "(A)")  "* Model parameters"
      write (u, "(A)")

      call model%write (unit = u, &
           show_parameters = .true., &
           show_particles = .false., show_vertices = .false.)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize a process and a matching &
           &phase-space configuration"
      write (u, "(A)")

      call init_test_process_data (var_str ("phs_base_4"), process_data)
      process_data%md5sum = "test_process_data_m6sum_12345678"

      allocate (phs_test_config_t :: phs_data)
      call phs_data%init (process_data, model)

      call phs_data%compute_md5sum ()
      call phs_data%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Modify model parameter"
      write (u, "(A)")

      call model%set_par (var_str ("ms"), 100._default)
      call model%write (show_parameters = .true., &
           show_particles = .false., show_vertices = .false.)

      write (u, "(A)")
      write (u, "(A)")  "* PHS configuration"
      write (u, "(A)")

      call phs_data%compute_md5sum ()
      call phs_data%write (u)
```

```
    call phs_data%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_base_4"

  end subroutine phs_base_4
```

Phase-space channel collection

Set up an array of various phase-space channels and collect them in a list.

⟨*PHS base: execute tests*⟩+≡

```
  call test (phs_base_5, "phs_base_5", &
       "channel collection", &
       u, results)
```

⟨*PHS base: test declarations*⟩+≡

```
  public :: phs_base_5
```

⟨*PHS base: tests*⟩+≡

```
  subroutine phs_base_5 (u)
    integer, intent(in) :: u
    type(phs_channel_t), dimension(:), allocatable :: channel
    type(phs_channel_collection_t) :: coll
    integer :: i, n

    write (u, "(A)")  "* Test output: phs_base_5"
    write (u, "(A)")  "*   Purpose: collect channel properties"
    write (u, "(A)")

    write (u, "(A)")  "* Set up an array of channels"
    write (u, "(A)")

    n = 6

    allocate (channel (n))
    call channel(2)%set_resonant (75._default, 3._default)
    call channel(4)%set_resonant (130._default, 1._default)
    call channel(5)%set_resonant (75._default, 3._default)
    call channel(6)%set_on_shell (33._default)

    do i = 1, n
       write (u, "(1x,I0)", advance="no")  i
       call channel(i)%write (u)
    end do

    write (u, "(A)")
    write (u, "(A)")  "* Collect distinct properties"
    write (u, "(A)")

    do i = 1, n
       call coll%push (channel(i))
    end do
```

```
  write (u, "(1x,A,I0)")  "n = ", coll%get_n ()
  write (u, "(A)")

  call coll%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Channel array with collection index assigned"
  write (u, "(A)")

  do i = 1, n
     write (u, "(1x,I0)", advance="no")  i
     call channel(i)%write (u)
  end do

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call coll%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: phs_base_5"

end subroutine phs_base_5
```

18.2 Single-particle phase space

This module implements the phase space for a single particle, i.e., the solid angle, in a straightforward parameterization with a single channel. The phase-space implementation may be used either for $1 \to 2$ decays or for $2 \to 2$ scattering processes, so the number of incoming particles is the only free parameter in the configuration. In the latter case, we should restrict its use to non-resonant s-channel processes, because there is no mapping of the scattering angle.

(We might extend this later to account for generic $2 \to 2$ situations, e.g., account for a Coulomb singularity or detect an s-channel resonance structure that requires matching structure-function mappings.)

This is derived from the `phs_test` implementation in the `phs_base` module above, even more simplified, but intended for actual use.

⟨`phs_single.f90`⟩≡

```
⟨File header⟩

module phs_single

⟨Use kinds⟩
  use io_units
  use constants
  use numeric_utils
  use diagnostics
  use os_interface
  use lorentz
  use physics_defs
  use model_data
  use flavors
  use process_constants
  use phs_base

⟨Standard module head⟩

⟨PHS single: public⟩

⟨PHS single: types⟩

contains

⟨PHS single: procedures⟩

end module phs_single
```

18.2.1 Configuration

⟨*PHS single: public*⟩≡

```
public :: phs_single_config_t
```

⟨*PHS single: types*⟩≡

```
type, extends (phs_config_t) :: phs_single_config_t
contains
⟨PHS single: phs single config: TBP⟩
end type phs_single_config_t
```

The finalizer is empty.

⟨*PHS single: phs single config: TBP*⟩≡
```
procedure :: final => phs_single_config_final
```

⟨*PHS single: procedures*⟩≡
```
subroutine phs_single_config_final (object)
  class(phs_single_config_t), intent(inout) :: object
end subroutine phs_single_config_final
```

Output.

⟨*PHS single: phs single config: TBP*⟩+≡
```
procedure :: write => phs_single_config_write
```

⟨*PHS single: procedures*⟩+≡
```
subroutine phs_single_config_write (object, unit, include_id)
  class(phs_single_config_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: include_id
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Partonic phase-space configuration (single-particle):"
  call object%base_write (unit)
end subroutine phs_single_config_write
```

Configuration: there is only one channel and two parameters. The second parameter is the azimuthal angle, which may be a flat dimension.

⟨*PHS single: phs single config: TBP*⟩+≡
```
procedure :: configure => phs_single_config_configure
```

⟨*PHS single: procedures*⟩+≡
```
subroutine phs_single_config_configure (phs_config, sqrts, &
     sqrts_fixed, cm_frame, azimuthal_dependence, rebuild, ignore_mismatch, &
     nlo_type)
  class(phs_single_config_t), intent(inout) :: phs_config
  real(default), intent(in) :: sqrts
  logical, intent(in), optional :: sqrts_fixed
  logical, intent(in), optional :: cm_frame
  logical, intent(in), optional :: azimuthal_dependence
  logical, intent(in), optional :: rebuild
  logical, intent(in), optional :: ignore_mismatch
  integer, intent(in), optional :: nlo_type
  if (.not. present (nlo_type)) &
    phs_config%nlo_type = BORN
  if (phs_config%n_out == 2) then
     phs_config%n_channel = 1
     phs_config%n_par = 2
     phs_config%sqrts = sqrts
     if (present (sqrts_fixed))  phs_config%sqrts_fixed = sqrts_fixed
     if (present (cm_frame))  phs_config%cm_frame = cm_frame
     if (present (azimuthal_dependence)) then
        phs_config%azimuthal_dependence = azimuthal_dependence
        if (.not. azimuthal_dependence) then
```

```
            allocate (phs_config%dim_flat (1))
            phs_config%dim_flat(1) = 2
         end if
      end if
      if (allocated (phs_config%channel))  deallocate (phs_config%channel)
      allocate (phs_config%channel (1))
      call phs_config%compute_md5sum ()
   else
      call msg_fatal ("Single-particle phase space requires n_out = 2")
   end if
end subroutine phs_single_config_configure
```

Startup message, after configuration is complete.

⟨*PHS single: phs single config: TBP*⟩+≡

```
procedure :: startup_message => phs_single_config_startup_message
```

⟨*PHS single: procedures*⟩+≡

```
subroutine phs_single_config_startup_message (phs_config, unit)
  class(phs_single_config_t), intent(in) :: phs_config
  integer, intent(in), optional :: unit
  call phs_config%base_startup_message (unit)
  write (msg_buffer, "(A,2(1x,I0,1x,A))") &
       "Phase space: single-particle"
  call msg_message (unit = unit)
end subroutine phs_single_config_startup_message
```

Allocate an instance: the actual phase-space object.

⟨*PHS single: phs single config: TBP*⟩+≡

```
procedure, nopass :: allocate_instance => phs_single_config_allocate_instance
```

⟨*PHS single: procedures*⟩+≡

```
subroutine phs_single_config_allocate_instance (phs)
  class(phs_t), intent(inout), pointer :: phs
  allocate (phs_single_t :: phs)
end subroutine phs_single_config_allocate_instance
```

18.2.2 Kinematics implementation

We generate $\cos\theta$ and ϕ uniformly, covering the solid angle.

Note: The incoming momenta must be in the c.m. system.

⟨*PHS single: public*⟩+≡

```
public :: phs_single_t
```

⟨*PHS single: types*⟩+≡

```
type, extends (phs_t) :: phs_single_t
 contains
 ⟨PHS single: phs single: TBP⟩
end type phs_single_t
```

Output. The `verbose` setting is irrelevant, we just display the contents of the base object.

⟨*PHS single: phs single: TBP*⟩≡

```
procedure :: write => phs_single_write
```

⟨*PHS single: procedures*⟩+≡

```
subroutine phs_single_write (object, unit, verbose)
  class(phs_single_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u
  u = given_output_unit (unit)
  call object%base_write (u)
end subroutine phs_single_write
```

The finalizer is empty.

⟨*PHS single: phs single: TBP*⟩+≡

```
procedure :: final => phs_single_final
```

⟨*PHS single: procedures*⟩+≡

```
subroutine phs_single_final (object)
  class(phs_single_t), intent(inout) :: object
end subroutine phs_single_final
```

Initialization. We allocate arrays (`base_init`) and adjust the phase-space volume. The massless two-particle phase space volume is

$$\Phi_2 = \frac{1}{4(2\pi)^5} = 2.55294034614 \times 10^{-5} \tag{18.5}$$

For a decay with nonvanishing masses (m_3, m_4), there is a correction factor

$$\Phi_2(m)/\Phi_2(0) = \frac{1}{\hat{s}}\lambda^{1/2}(\hat{s}, m_3^2, m_4^2). \tag{18.6}$$

For a scattering process with nonvanishing masses, the correction factor is

$$\Phi_2(m)/\Phi_2(0) = \frac{1}{\hat{s}^2}\lambda^{1/2}(\hat{s}, m_1^2, m_2^2)\,\lambda^{1/2}(\hat{s}, m_3^2, m_4^2). \tag{18.7}$$

If the energy is fixed, this is constant. Otherwise, we have to account for varying $\hat{s}$.

⟨*PHS single: phs single: TBP*⟩+≡

```
procedure :: init => phs_single_init
```

⟨*PHS single: procedures*⟩+≡

```
subroutine phs_single_init (phs, phs_config)
  class(phs_single_t), intent(out) :: phs
  class(phs_config_t), intent(in), target :: phs_config
  call phs%base_init (phs_config)
  phs%volume = 1 / (4 * twopi5)
  call phs%compute_factor ()
end subroutine phs_single_init
```

Compute the correction factor for nonzero masses. We do this during initialization (when the incoming momenta `p` are undefined), unless `sqrts` is variable. We do this again once for each phase-space point, but then we skip the calculation if `sqrts` is fixed.

⟨*PHS single: phs single: TBP*⟩+≡

```
procedure :: compute_factor => phs_single_compute_factor
```

⟨*PHS single: procedures*⟩+≡

```
subroutine phs_single_compute_factor (phs)
  class(phs_single_t), intent(inout) :: phs
  real(default) :: s_hat
  select case (phs%config%n_in)
  case (1)
     if (.not. phs%p_defined) then
        if (sum (phs%m_out) < phs%m_in(1)) then
           s_hat = phs%m_in(1) ** 2
           phs%f(1) = 1 / s_hat &
                * sqrt (lambda (s_hat, phs%m_out(1)**2, phs%m_out(2)**2))
        else
           print *, "m_in  = ", phs%m_in
           print *, "m_out = ", phs%m_out
           call msg_fatal ("Decay is kinematically forbidden")
        end if
     end if
  case (2)
     if (phs%config%sqrts_fixed) then
        if (phs%p_defined)  return
        s_hat = phs%config%sqrts ** 2
     else
        if (.not. phs%p_defined)  return
        s_hat = sum (phs%p) ** 2
     end if
     if (sum (phs%m_in)**2 < s_hat .and. sum (phs%m_out)**2 < s_hat) then
        phs%f(1) = 1 / s_hat * &
              ( lambda (s_hat, phs%m_in (1)**2, phs%m_in (2)**2)   &
              * lambda (s_hat, phs%m_out(1)**2, phs%m_out(2)**2) ) &
              ** 0.25_default
     else
        phs%f(1) = 0
     end if
  end select
end subroutine phs_single_compute_factor
```

Evaluation. We uniformly generate $\cos\theta$ and ϕ, with Jacobian normalized to one.

There is only a single channel, so the second subroutine does nothing.

Note: the current implementation works for elastic scattering only.

⟨*PHS single: phs single: TBP*⟩+≡

```
procedure :: evaluate_selected_channel => phs_single_evaluate_selected_channel
procedure :: evaluate_other_channels => phs_single_evaluate_other_channels
```

⟨*PHS single: procedures*⟩+≡

```
subroutine phs_single_evaluate_selected_channel (phs, c_in, r_in)
  class(phs_single_t), intent(inout) :: phs
  integer, intent(in) :: c_in
  real(default), intent(in), dimension(:) :: r_in
  !!! !!! !!! Catching a gfortran bogus warning
  type(vector4_t), dimension(2) :: p_dum
  if (phs%p_defined) then
     call phs%select_channel (c_in)
     phs%r(:,c_in) = r_in
```

```
      select case (phs%config%n_in)
      case (2)
         if (all (phs%m_in == phs%m_out)) then
            call compute_kinematics_solid_angle (phs%p, phs%q, r_in)
         else
            call msg_bug ("PHS single: inelastic scattering not implemented")
         end if
      case (1)
         !!! !!! !!! Catching a gfortran bogus warning
         !!! call compute_kinematics_solid_angle (phs%decay_p (), phs%q, x)
         p_dum = phs%decay_p ()
         call compute_kinematics_solid_angle (p_dum, phs%q, r_in)
      end select
      call phs%compute_factor ()
      phs%q_defined = .true.
      phs%r_defined = .true.
   end if
end subroutine phs_single_evaluate_selected_channel

subroutine phs_single_evaluate_other_channels (phs, c_in)
   class(phs_single_t), intent(inout) :: phs
   integer, intent(in) :: c_in
end subroutine phs_single_evaluate_other_channels
```

Auxiliary: split a decaying particle at rest into the decay products, aligned along the z axis.

⟨*PHS single: phs single: TBP*⟩+≡

```
procedure :: decay_p => phs_single_decay_p
```

⟨*PHS single: procedures*⟩+≡

```
function phs_single_decay_p (phs) result (p)
   class(phs_single_t), intent(in) :: phs
   type(vector4_t), dimension(2) :: p
   real(default) :: k
   real(default), dimension(2) :: E
   k = sqrt (lambda (phs%m_in(1) ** 2, phs%m_out(1) ** 2, phs%m_out(2) ** 2)) &
        / (2 * phs%m_in(1))
   E = sqrt (phs%m_out ** 2 + k ** 2)
   p(1) = vector4_moving (E(1), k, 3)
   p(2) = vector4_moving (E(2),-k, 3)
end function phs_single_decay_p
```

Inverse evaluation.

⟨*PHS single: phs single: TBP*⟩+≡

```
procedure :: inverse => phs_single_inverse
```

⟨*PHS single: procedures*⟩+≡

```
subroutine phs_single_inverse (phs)
   class(phs_single_t), intent(inout) :: phs
   real(default), dimension(:), allocatable :: x
   if (phs%p_defined .and. phs%q_defined) then
      call phs%select_channel ()
      allocate (x (phs%config%n_par))
      call inverse_kinematics_solid_angle (phs%p, phs%q, x)
```

```
      phs%r(:,1) = x
      call phs%compute_factor ()
      phs%r_defined = .true.
    end if
  end subroutine phs_single_inverse
```

18.2.3 Unit tests

Test module, followed by the corresponding implementation module.

⟨phs_single_ut.f90⟩≡

```
⟨File header⟩

module phs_single_ut
  use unit_tests
  use phs_single_uti

⟨Standard module head⟩

⟨PHS single: public test⟩

contains

⟨PHS single: test driver⟩

end module phs_single_ut
```

⟨phs_single_uti.f90⟩≡

```
⟨File header⟩

module phs_single_uti

⟨Use kinds⟩
⟨Use strings⟩
  use flavors
  use lorentz
  use model_data
  use process_constants
  use phs_base

  use phs_single

  use phs_base_ut, only: init_test_process_data, init_test_decay_data

⟨Standard module head⟩

⟨PHS single: test declarations⟩

contains

⟨PHS single: tests⟩

end module phs_single_uti
```

API: driver for the unit tests below.

⟨*PHS single: public test*⟩≡
```
public :: phs_single_test
```

⟨*PHS single: test driver*⟩≡
```
subroutine phs_single_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨PHS single: execute tests⟩
end subroutine phs_single_test
```

Phase-space configuration data

Construct and display a test phase-space configuration object. Also check the `azimuthal_dependence` flag.

⟨*PHS single: execute tests*⟩≡
```
call test (phs_single_1, "phs_single_1", &
     "phase-space configuration", &
     u, results)
```

⟨*PHS single: test declarations*⟩≡
```
public :: phs_single_1
```

⟨*PHS single: tests*⟩≡
```
subroutine phs_single_1 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(process_constants_t) :: process_data
  class(phs_config_t), allocatable :: phs_data
  real(default) :: sqrts

  write (u, "(A)")  "* Test output: phs_single_1"
  write (u, "(A)")  "*   Purpose: initialize and display &
       &phase-space configuration data"
  write (u, "(A)")

  call model%init_test ()

  write (u, "(A)")  "* Initialize a process and a matching &
       &phase-space configuration"
  write (u, "(A)")

  call init_test_process_data (var_str ("phs_single_1"), process_data)

  allocate (phs_single_config_t :: phs_data)
  call phs_data%init (process_data, model)

  sqrts = 1000._default
  call phs_data%configure (sqrts, azimuthal_dependence=.false.)

  call phs_data%write (u)

  call phs_data%final ()
  call model%final ()
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_single_1"

  end subroutine phs_single_1
```

Phase space evaluation

Compute kinematics for given parameters, also invert the calculation.

⟨*PHS single: execute tests*⟩+≡

```
  call test (phs_single_2, "phs_single_2", &
       "phase-space evaluation", &
       u, results)
```

⟨*PHS single: test declarations*⟩+≡

```
  public :: phs_single_2
```

⟨*PHS single: tests*⟩+≡

```
  subroutine phs_single_2 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t) :: flv
    type(process_constants_t) :: process_data
    real(default) :: sqrts, E
    class(phs_config_t), allocatable, target :: phs_data
    class(phs_t), pointer :: phs => null ()
    type(vector4_t), dimension(2) :: p, q

    write (u, "(A)")  "* Test output: phs_single_2"
    write (u, "(A)")  "*   Purpose: test simple two-channel phase space"
    write (u, "(A)")

    call model%init_test ()
    call flv%init (25, model)

    write (u, "(A)")  "* Initialize a process and a matching &
         &phase-space configuration"
    write (u, "(A)")

    call init_test_process_data (var_str ("phs_single_2"), process_data)

    allocate (phs_single_config_t :: phs_data)
    call phs_data%init (process_data, model)

    sqrts = 1000._default
    call phs_data%configure (sqrts)

    call phs_data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize the phase-space instance"
    write (u, "(A)")

    call phs_data%allocate_instance (phs)
```

```
  call phs%init (phs_data)

  call phs%write (u, verbose=.true.)

  write (u, "(A)")
  write (u, "(A)")  "* Set incoming momenta"
  write (u, "(A)")

  E = sqrts / 2
  p(1) = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
  p(2) = vector4_moving (E,-sqrt (E**2 - flv%get_mass ()**2), 3)

  call phs%set_incoming_momenta (p)
  call phs%compute_flux ()
  call phs%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Compute phase-space point &
       &for x = 0.5, 0.125"
  write (u, "(A)")

  call phs%evaluate_selected_channel (1, [0.5_default, 0.125_default])
  call phs%evaluate_other_channels (1)
  call phs%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Inverse kinematics"
  write (u, "(A)")

  call phs%get_outgoing_momenta (q)
  deallocate (phs)
  call phs_data%allocate_instance (phs)
  call phs%init (phs_data)

  sqrts = 1000._default
  call phs_data%configure (sqrts)

  call phs%set_incoming_momenta (p)
  call phs%compute_flux ()
  call phs%set_outgoing_momenta (q)

  call phs%inverse ()
  call phs%write (u)

  call phs%final ()
  deallocate (phs)

  call phs_data%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: phs_single_2"

end subroutine phs_single_2
```

Phase space for non-c.m. system

Compute kinematics for given parameters, also invert the calculation. Since this will involve cancellations, we call `pacify` to eliminate numerical noise.

⟨*PHS single: execute tests*⟩+≡
```
    call test (phs_single_3, "phs_single_3", &
         "phase-space evaluation in lab frame", &
         u, results)
```

⟨*PHS single: test declarations*⟩+≡
```
    public :: phs_single_3
```

⟨*PHS single: tests*⟩+≡
```
    subroutine phs_single_3 (u)
      integer, intent(in) :: u
      type(model_data_t), target :: model
      type(flavor_t) :: flv
      type(process_constants_t) :: process_data
      real(default) :: sqrts, E
      class(phs_config_t), allocatable, target :: phs_data
      class(phs_t), pointer :: phs => null ()
      type(vector4_t), dimension(2) :: p, q
      type(lorentz_transformation_t) :: lt

      write (u, "(A)")  "* Test output: phs_single_3"
      write (u, "(A)")  "*   Purpose: test simple two-channel phase space"
      write (u, "(A)")  "*            without c.m. kinematics assumption"
      write (u, "(A)")

      call model%init_test ()
      call flv%init (25, model)

      write (u, "(A)")  "* Initialize a process and a matching &
           &phase-space configuration"
      write (u, "(A)")

      call init_test_process_data (var_str ("phs_single_3"), process_data)

      allocate (phs_single_config_t :: phs_data)
      call phs_data%init (process_data, model)

      sqrts = 1000._default
      call phs_data%configure (sqrts, cm_frame=.false., sqrts_fixed=.false.)

      call phs_data%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize the phase-space instance"
      write (u, "(A)")

      call phs_data%allocate_instance (phs)
      call phs%init (phs_data)
```

```
call phs%write (u, verbose=.true.)

write (u, "(A)")
write (u, "(A)")  "* Set incoming momenta in lab system"
write (u, "(A)")

lt = boost (0.1_default, 1) * boost (0.3_default, 3)

E = sqrts / 2
p(1) = lt * vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
p(2) = lt * vector4_moving (E,-sqrt (E**2 - flv%get_mass ()**2), 3)

call vector4_write (p(1), u)
call vector4_write (p(2), u)

write (u, "(A)")
write (u, "(A)")  "* Compute phase-space point &
     &for x = 0.5, 0.125"
write (u, "(A)")

call phs%set_incoming_momenta (p)
call phs%compute_flux ()

call phs%evaluate_selected_channel (1, [0.5_default, 0.125_default])
call phs%evaluate_other_channels (1)
call pacify (phs)
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Extract outgoing momenta in lab system"
write (u, "(A)")

call phs%get_outgoing_momenta (q)
call vector4_write (q(1), u)
call vector4_write (q(2), u)

write (u, "(A)")
write (u, "(A)")  "* Inverse kinematics"
write (u, "(A)")

deallocate (phs)
call phs_data%allocate_instance (phs)
call phs%init (phs_data)

sqrts = 1000._default
call phs_data%configure (sqrts)

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%set_outgoing_momenta (q)

call phs%inverse ()
call pacify (phs)
```

```
    call phs%write (u)

    call phs%final ()
    deallocate (phs)

    call phs_data%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_single_3"

  end subroutine phs_single_3
```

Decay Phase space evaluation

Compute kinematics for given parameters, also invert the calculation. This time, implement a decay process.

⟨*PHS single: execute tests*⟩+≡

```
  call test (phs_single_4, "phs_single_4", &
       "decay phase-space evaluation", &
       u, results)
```

⟨*PHS single: test declarations*⟩+≡

```
  public :: phs_single_4
```

⟨*PHS single: tests*⟩+≡

```
  subroutine phs_single_4 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t) :: flv
    type(process_constants_t) :: process_data
    class(phs_config_t), allocatable, target :: phs_data
    class(phs_t), pointer :: phs => null ()
    type(vector4_t), dimension(1) :: p
    type(vector4_t), dimension(2) :: q

    write (u, "(A)")  "* Test output: phs_single_4"
    write (u, "(A)")  "*   Purpose: test simple two-channel phase space"
    write (u, "(A)")

    call model%init_test ()

    call model%set_par (var_str ("ff"), 0.4_default)
    call model%set_par (var_str ("mf"), &
         model%get_real (var_str ("ff")) * model%get_real (var_str ("ms")))
    call flv%init (25, model)

    write (u, "(A)")  "* Initialize a decay and a matching &
         &phase-space configuration"
    write (u, "(A)")

    call init_test_decay_data (var_str ("phs_single_4"), process_data)
```

```
allocate (phs_single_config_t :: phs_data)
call phs_data%init (process_data, model)

call phs_data%configure (flv%get_mass ())

call phs_data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize the phase-space instance"
write (u, "(A)")

call phs_data%allocate_instance (phs)
call phs%init (phs_data)

call phs%write (u, verbose=.true.)

write (u, "(A)")
write (u, "(A)")  "* Set incoming momenta"
write (u, "(A)")

p(1) = vector4_at_rest (flv%get_mass ())

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Compute phase-space point &
     &for x = 0.5, 0.125"
write (u, "(A)")

call phs%evaluate_selected_channel (1, [0.5_default, 0.125_default])
call phs%evaluate_other_channels (1)
call phs%write (u)

write (u, "(A)")
write (u, "(A)")  "* Inverse kinematics"
write (u, "(A)")

call phs%get_outgoing_momenta (q)
deallocate (phs)
call phs_data%allocate_instance (phs)
call phs%init (phs_data)

call phs_data%configure (flv%get_mass ())

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%set_outgoing_momenta (q)

call phs%inverse ()
call phs%write (u)

call phs%final ()
```

```
  deallocate (phs)

  call phs_data%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: phs_single_4"

end subroutine phs_single_4
```

18.3 Resonance Handler

For various purposes (e.g., shower histories), we should extract the set of resonances and resonant channels from a phase-space tree set. A few methods do kinematics calculations specifically for those resonance data.

⟨resonances.f90⟩≡

```
⟨File header⟩

module resonances

⟨Use kinds⟩
!   use kinds, only: TC
⟨Use strings⟩
  use string_utils, only: str
  use io_units
  use diagnostics
  use lorentz, only: vector4_t, compute_resonance_mass
  use constants, only: one
  use model_data, only: model_data_t
  use flavors, only: flavor_t

⟨Standard module head⟩

⟨Resonances: public⟩

⟨Resonances: parameters⟩

⟨Resonances: types⟩

contains

⟨Resonances: procedures⟩

end module resonances
```

18.3.1 Decay products (contributors)

This stores the indices of the particles that contribute to a resonance, i.e., the decay products.

⟨*Resonances: public*⟩≡

```
public :: resonance_contributors_t
```

⟨*Resonances: types*⟩≡

```
type :: resonance_contributors_t
   integer, dimension(:), allocatable :: c
 contains
 ⟨Resonances: resonance contributors: TBP⟩
end type resonance_contributors_t
```

Equality (comparison)

⟨*Resonances: resonance contributors: TBP*⟩≡

```
procedure, private :: resonance_contributors_equal
generic :: operator(==) => resonance_contributors_equal
```

⟨*Resonances: procedures*⟩≡

```
elemental function resonance_contributors_equal (c1, c2) result (equal)
  logical :: equal
  class(resonance_contributors_t), intent(in) :: c1, c2
  equal = allocated (c1%c) .and. allocated (c2%c)
  if (equal) equal = size (c1%c) == size (c2%c)
  if (equal) equal = all (c1%c == c2%c)
end function resonance_contributors_equal
```

Assignment

⟨*Resonances: resonance contributors: TBP*⟩+≡

```
procedure, private :: resonance_contributors_assign
generic :: assignment(=) => resonance_contributors_assign
```

⟨*Resonances: procedures*⟩+≡

```
pure subroutine resonance_contributors_assign (contributors_out, contributors_in)
  class(resonance_contributors_t), intent(inout) :: contributors_out
  class(resonance_contributors_t), intent(in) :: contributors_in
  if (allocated (contributors_out%c))  deallocate (contributors_out%c)
  if (allocated (contributors_in%c)) then
     allocate (contributors_out%c (size (contributors_in%c)))
     contributors_out%c = contributors_in%c
  end if
end subroutine resonance_contributors_assign
```

18.3.2 Resonance info object

This data structure augments the set of resonance contributors by a flavor object, such that we can perform calculations that take into account the particle properties, including mass and width.

Avoiding nameclash with similar but different `resonance_t` of `phs_base`:

⟨*Resonances: public*⟩+≡

```
public :: resonance_info_t
```

⟨*Resonances: types*⟩+≡

```
type :: resonance_info_t
   type(flavor_t) :: flavor
   type(resonance_contributors_t) :: contributors
contains
 ⟨Resonances: resonance info: TBP⟩
end type resonance_info_t
```

⟨*Resonances: resonance info: TBP*⟩≡

```
procedure :: write => resonance_info_write
```

⟨*Resonances: procedures*⟩+≡

```
subroutine resonance_info_write (resonance, unit)
  class(resonance_info_t), intent(in) :: resonance
  integer, optional, intent(in) :: unit
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, '(A)', advance='no') "Resonance contributors: "
```

```
    if (allocated (resonance%contributors%c)) then
       do i = 1, size(resonance%contributors%c)
          write (u, '(I0,1X)', advance='no') resonance%contributors%c(i)
       end do
    else
       write (u, "(A)", advance="no")  "[not allocated]"
    end if
    if (resonance%flavor%is_defined ()) call resonance%flavor%write (u)
    write (u, '(A)')
  end subroutine resonance_info_write
```

Create a resonance-info object. The particle info may be available in term of a flavor object or as a PDG code; in the latter case we have to require a model data object that provides mass and width information.

⟨*Resonances: resonance info: TBP*⟩+≡

```
  procedure, private :: resonance_info_init_pdg
  procedure, private :: resonance_info_init_flv
  generic :: init => resonance_info_init_pdg, resonance_info_init_flv
```

⟨*Resonances: procedures*⟩+≡

```
  subroutine resonance_info_init_pdg (resonance, mom_id, pdg, model, n_out)
    class(resonance_info_t), intent(out) :: resonance
    integer, intent(in) :: mom_id
    integer, intent(in) :: pdg, n_out
    class(model_data_t), intent(in), target :: model
    type(flavor_t) :: flv
    call msg_debug (D_PHASESPACE, "resonance_info_init_pdg")
    call flv%init (pdg, model)
    call resonance%init (mom_id, flv, n_out)
  end subroutine resonance_info_init_pdg

  subroutine resonance_info_init_flv (resonance, mom_id, flv, n_out)
    class(resonance_info_t), intent(out) :: resonance
    integer, intent(in) :: mom_id
    type(flavor_t), intent(in) :: flv
    integer, intent(in) :: n_out
    integer :: i
    logical, dimension(n_out) :: contrib
    integer, dimension(n_out) :: tmp
    call msg_debug (D_PHASESPACE, "resonance_info_init_flv")
    resonance%flavor = flv
    do i = 1, n_out
       tmp(i) = i
    end do
    contrib = btest (mom_id, tmp - 1)
    allocate (resonance%contributors%c (count (contrib)))
    resonance%contributors%c = pack (tmp, contrib)
  end subroutine resonance_info_init_flv
```

⟨*Resonances: resonance info: TBP*⟩+≡

```
  procedure, private :: resonance_info_equal
  generic :: operator(==) => resonance_info_equal
```

⟨*Resonances: procedures*⟩+≡

```
elemental function resonance_info_equal (r1, r2) result (equal)
  logical :: equal
  class(resonance_info_t), intent(in) :: r1, r2
  equal = r1%flavor == r2%flavor .and. r1%contributors == r2%contributors
end function resonance_info_equal
```

With each resonance region we associate a Breit-Wigner function

$$P = \frac{M_{res}^4}{(s - M_{res}^2)^2 + \Gamma_{res}^2 M_{res}^2},$$

where s denotes the invariant mass of the outgoing momenta originating from this resonance. Note that the M_{res}^4 in the nominator makes the mapping a dimensionless quantity.

⟨*Resonances: resonance info: TBP*⟩+≡

```
procedure :: mapping => resonance_info_mapping
```

⟨*Resonances: procedures*⟩+≡

```
function resonance_info_mapping (resonance, s) result (bw)
  real(default) :: bw
  class(resonance_info_t), intent(in) :: resonance
  real(default), intent(in) :: s
  real(default) :: m, gamma
  if (resonance%flavor%is_defined ()) then
     m = resonance%flavor%get_mass ()
     gamma = resonance%flavor%get_width ()
     bw = m**4 / ((s - m**2)**2 + gamma**2 * m**2)
  else
     bw = one
  end if
end function resonance_info_mapping
```

18.3.3 Resonance history object

This data structure stores a set of resonances, i.e., the resonances that appear in a particular Feynman graph or, in the context of phase space, phase space diagram.

⟨*Resonances: public*⟩+≡

```
public :: resonance_history_t
```

⟨*Resonances: types*⟩+≡

```
type :: resonance_history_t
   type(resonance_info_t), dimension(:), allocatable :: resonances
   integer :: n_resonances = 0
contains
 ⟨Resonances: resonance history: TBP⟩
end type resonance_history_t
```

Clear the resonance history. Assuming that there are no pointer-allocated parts, a straightforward `intent(out)` will do.

⟨*Resonances: resonance history: TBP*⟩≡

```
procedure :: clear => resonance_history_clear
```

⟨*Resonances: procedures*⟩+≡
```
subroutine resonance_history_clear (res_hist)
  class(resonance_history_t), intent(out) :: res_hist
end subroutine resonance_history_clear
```

⟨*Resonances: resonance history: TBP*⟩+≡
```
procedure :: write => resonance_history_write
```

⟨*Resonances: procedures*⟩+≡
```
subroutine resonance_history_write (res_hist, unit)
  class(resonance_history_t), intent(in) :: res_hist
  integer, optional, intent(in) :: unit
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write(u, '(A,I0,A)') "Resonance history with ", &
       res_hist%n_resonances, " resonances:"
  do i = 1, res_hist%n_resonances
     call res_hist%resonances(i)%write (u)
  end do
end subroutine resonance_history_write
```

Equality. If this turns out to slow down the program, we should change the implementation or use hash codes.

⟨*Resonances: resonance history: TBP*⟩+≡
```
procedure, private :: resonance_history_equal
generic :: operator(==) => resonance_history_equal
```

⟨*Resonances: procedures*⟩+≡
```
elemental function resonance_history_equal (rh1, rh2) result (equal)
  logical :: equal
  class(resonance_history_t), intent(in) :: rh1, rh2
  integer :: i
  equal = .false.
  if (rh1%n_resonances == rh2%n_resonances) then
     if (size (rh1%resonances) == size (rh2%resonances)) then
        do i = 1, rh1%n_resonances
           if (.not. rh1%resonances(i) == rh2%resonances(i)) then
              return
           end if
        end do
        equal = .true.
     end if
  end if
end function resonance_history_equal
```

⟨*Resonances: resonance history: TBP*⟩+≡
```
procedure, private :: resonance_history_contains
generic :: operator(.contains.) => resonance_history_contains
```

⟨*Resonances: procedures*⟩+≡
```
elemental function resonance_history_contains (rh1, rh2) result (is_sub_res)
  logical :: is_sub_res
  class(resonance_history_t), intent(in) :: rh1, rh2
  integer :: i_res
```

```
    is_sub_res = .false.
    if (rh1%n_resonances > rh2%n_resonances) then
       do i_res = 1, rh1%n_resonances
          is_sub_res = is_sub_res &
               .or. any (rh1%resonances(i_res) == rh2%resonances)
       end do
    end if
  end function resonance_history_contains
```

Number of entries for dynamically extending the resonance-info array.

⟨*Resonances: parameters*⟩≡

```
  integer, parameter :: n_max_resonances = 10
```

⟨*Resonances: resonance history: TBP*⟩+≡

```
  procedure :: add_resonance => resonance_history_add_resonance
```

⟨*Resonances: procedures*⟩+≡

```
  subroutine resonance_history_add_resonance (res_hist, resonance)
    class(resonance_history_t), intent(inout) :: res_hist
    type(resonance_info_t), intent(in) :: resonance
    type(resonance_info_t), dimension(:), allocatable :: tmp
    integer :: n
    call msg_debug (D_PHASESPACE, "resonance_history_add_resonance")
    if (.not. allocated (res_hist%resonances)) then
       n = 0
       allocate (res_hist%resonances (n_max_resonances))
    else
       n = res_hist%n_resonances
       if (n >= size (res_hist%resonances)) then
          allocate (tmp (n + n_max_resonances))
          tmp(1:n) = res_hist%resonances(1:n)
          call move_alloc (from=tmp, to=res_hist%resonances)
       end if
    end if
    res_hist%resonances(n+1) = resonance
    res_hist%n_resonances = n + 1
    call msg_debug &
         (D_PHASESPACE, "res_hist%n_resonances", res_hist%n_resonances)
  end subroutine resonance_history_add_resonance
```

⟨*Resonances: resonance history: TBP*⟩+≡

```
  procedure :: remove_resonance => resonance_history_remove_resonance
```

⟨*Resonances: procedures*⟩+≡

```
  subroutine resonance_history_remove_resonance (res_hist, i_res)
    class(resonance_history_t), intent(inout) :: res_hist
    integer, intent(in) :: i_res
    integer :: i
    res_hist%n_resonances = res_hist%n_resonances - 1
    if (res_hist%n_resonances == 0) then
       deallocate (res_hist%resonances)
    else
       do i = i_res + 1, size (res_hist%resonances)
          res_hist%resonances (i - 1) = res_hist%resonances (i)
```

```
      end do
    end if
  end subroutine resonance_history_remove_resonance
```

⟨*Resonances: resonance history: TBP*⟩+≡

```
  procedure :: add_offset => resonance_history_add_offset
```

⟨*Resonances: procedures*⟩+≡

```
  subroutine resonance_history_add_offset (res_hist, n)
    class(resonance_history_t), intent(inout) :: res_hist
    integer, intent(in) :: n
    integer :: i_res
    do i_res = 1, res_hist%n_resonances
       associate (contributors => res_hist%resonances(i_res)%contributors%c)
          contributors = contributors + n
       end associate
    end do
  end subroutine resonance_history_add_offset
```

⟨*Resonances: resonance history: TBP*⟩+≡

```
  procedure :: contains_leg => resonance_history_contains_leg
```

⟨*Resonances: procedures*⟩+≡

```
  function resonance_history_contains_leg (res_hist, i_leg) result (val)
    logical :: val
    class(resonance_history_t), intent(in) :: res_hist
    integer, intent(in) :: i_leg
    integer :: i_res
    val = .false.
    do i_res = 1, res_hist%n_resonances
       if (any (res_hist%resonances(i_res)%contributors%c == i_leg)) then
          val = .true.
          exit
       end if
    end do
  end function resonance_history_contains_leg
```

⟨*Resonances: resonance history: TBP*⟩+≡

```
  procedure :: mapping => resonance_history_mapping
```

⟨*Resonances: procedures*⟩+≡

```
  function resonance_history_mapping (res_hist, p, i_gluon) result (p_map)
    real(default) :: p_map
    class(resonance_history_t), intent(in) :: res_hist
    type(vector4_t), intent(in), dimension(:) :: p
    integer, intent(in), optional :: i_gluon
    integer :: i_res
    real(default) :: s
    p_map = one
    do i_res = 1, res_hist%n_resonances
       associate (res => res_hist%resonances(i_res))
          s = compute_resonance_mass (p, res%contributors%c, i_gluon)**2
          p_map = p_map * res%mapping (s)
       end associate
```

```
    end do
  end function resonance_history_mapping
```

This predicate is true if all resonances in the history have exactly `n` contributors. For instance, if $n = 2$, all resonances have a two-particle decay.

⟨*Resonances: resonance history: TBP*⟩+≡

```
  procedure :: only_has_n_contributors => resonance_history_only_has_n_contributors
```

⟨*Resonances: procedures*⟩+≡

```
  function resonance_history_only_has_n_contributors (res_hist, n) result (value)
    logical :: value
    class(resonance_history_t), intent(in) :: res_hist
    integer, intent(in) :: n
    integer :: i_res
    value = .true.
    do i_res = 1, res_hist%n_resonances
       associate (res => res_hist%resonances(i_res))
          value = value .and. size (res%contributors%c) == n
       end associate
    end do
  end function resonance_history_only_has_n_contributors
```

⟨*Resonances: resonance history: TBP*⟩+≡

```
  procedure :: has_flavor => resonance_history_has_flavor
```

⟨*Resonances: procedures*⟩+≡

```
  function resonance_history_has_flavor (res_hist, flv) result (has_flv)
    logical :: has_flv
    class(resonance_history_t), intent(in) :: res_hist
    type(flavor_t), intent(in) :: flv
    integer :: i
    has_flv = .false.
    do i = 1, res_hist%n_resonances
       has_flv = has_flv .or. res_hist%resonances(i)%flavor == flv
    end do
  end function resonance_history_has_flavor
```

18.3.4 O'Mega restriction strings

One application of the resonance module is creating restriction strings that can be fed into process definitions with the O'Mega generator. Since O'Mega counts the incoming particles first, we have to supply `n_in` as an offset.

⟨*Resonances: resonance info: TBP*⟩+≡

```
  procedure :: as_omega_string => resonance_info_as_omega_string
```

⟨*Resonances: resonance history: TBP*⟩+≡

```
  procedure :: as_omega_string => resonance_history_as_omega_string
```

⟨*Resonances: procedures*⟩+≡

```
  function resonance_info_as_omega_string (res_info, n_in) result (string)
    class(resonance_info_t), intent(in) :: res_info
    integer, intent(in) :: n_in
    type(string_t) :: string
```

```
  integer :: i
  string = ""
  if (allocated (res_info%contributors%c)) then
     do i = 1, size (res_info%contributors%c)
        if (i > 1)  string = string // "+"
        string = string // str (res_info%contributors%c(i) + n_in)
     end do
     string = string // "~" // res_info%flavor%get_name ()
  end if
end function resonance_info_as_omega_string

function resonance_history_as_omega_string (res_hist, n_in) result (string)
  class(resonance_history_t), intent(in) :: res_hist
  integer, intent(in) :: n_in
  type(string_t) :: string
  integer :: i
  string = ""
  do i = 1, res_hist%n_resonances
     if (i > 1)  string = string // " && "
     string = string // res_hist%resonances(i)%as_omega_string (n_in)
  end do
end function resonance_history_as_omega_string
```

18.3.5 Resonance history set

This is an array of resonance histories. The elements are supposed to be unique. That is, entering a new element is successful only if the element does not already exist.

The current implementation uses a straightforward linear search for comparison. If this should become an issue, we may change the implementation to a hash table. To keep this freedom, the set should be an opaque object. In fact, we expect to use it as a transient data structure. Once the set is complete, we transform it into a contiguous array.

⟨*Resonances: public*⟩+≡

```
public :: resonance_history_set_t
```

⟨*Resonances: types*⟩+≡

```
type :: resonance_history_set_t
   integer :: n_filter = 0
   type(resonance_history_t), dimension(:), allocatable :: history
   integer :: last = 0
 contains
 ⟨Resonances: resonance history set: TBP⟩
end type resonance_history_set_t
```

Display.

⟨*Resonances: resonance history set: TBP*⟩≡

```
procedure :: write => resonance_history_set_write
```

⟨*Resonances: procedures*⟩+≡

```
subroutine resonance_history_set_write (res_set, unit)
  class(resonance_history_set_t), intent(in) :: res_set
```

```
    integer, intent(in), optional :: unit
    integer :: u, i
    u = given_output_unit (unit)
    write (u, "(A)")  "Resonance history set:"
    do i = 1, res_set%last
       write (u, "('* ')", advance="no")
       call res_set%history(i)%write (u)
    end do
  end subroutine resonance_history_set_write
```

Initialization. The default initial size is 16 elements, to be doubled in size repeatedly as needed.

⟨*Resonances: parameters*⟩+≡

```
  integer, parameter :: resonance_history_set_initial_size = 16
```

⟨*Resonances: resonance history set: TBP*⟩+≡

```
  procedure :: init => resonance_history_set_init
```

⟨*Resonances: procedures*⟩+≡

```
  subroutine resonance_history_set_init (res_set, n_filter, initial_size)
    class(resonance_history_set_t), intent(out) :: res_set
    integer, intent(in), optional :: n_filter
    integer, intent(in), optional :: initial_size
    if (present (n_filter))  res_set%n_filter = n_filter
    if (present (initial_size)) then
       allocate (res_set%history (initial_size))
    else
       allocate (res_set%history (resonance_history_set_initial_size))
    end if
  end subroutine resonance_history_set_init
```

Enter an entry: append to the array if it doesn't yet exist, expand as needed. If a `n_filter` value has been provided, enter the resonance only if it fulfils the requirement.

An empty resonance history is never entered.

⟨*Resonances: resonance history set: TBP*⟩+≡

```
  procedure :: enter => resonance_history_set_enter
```

⟨*Resonances: procedures*⟩+≡

```
  subroutine resonance_history_set_enter (res_set, res_history)
    class(resonance_history_set_t), intent(inout) :: res_set
    type(resonance_history_t), intent(in) :: res_history
    integer :: i, new
    if (res_history%n_resonances == 0)  return
    if (res_set%n_filter > 0) then
       if (.not. res_history%only_has_n_contributors (res_set%n_filter))  return
    end if
    do i = 1, res_set%last
       if (res_set%history(i) == res_history)  return
    end do
    new = res_set%last + 1
    if (new > size (res_set%history))  call res_set%expand ()
    res_set%history(new) = res_history
    res_set%last = new
```

```
    end subroutine resonance_history_set_enter
```

Conversion to a plain array, sized correctly.

⟨*Resonances: resonance history set: TBP*⟩+≡

```
    procedure :: to_array => resonance_history_set_to_array
```

⟨*Resonances: procedures*⟩+≡

```
    subroutine resonance_history_set_to_array (res_set, res_history)
      class(resonance_history_set_t), intent(inout) :: res_set
      type(resonance_history_t), dimension(:), allocatable, intent(out) :: res_history
      allocate (res_history (res_set%last))
      res_history(:) = res_set%history(1:res_set%last)
    end subroutine resonance_history_set_to_array
```

Expand: double the size of the array. We do not need this in the API.

⟨*Resonances: resonance history set: TBP*⟩+≡

```
    procedure, private :: expand => resonance_history_set_expand
```

⟨*Resonances: procedures*⟩+≡

```
    subroutine resonance_history_set_expand (res_set)
      class(resonance_history_set_t), intent(inout) :: res_set
      type(resonance_history_t), dimension(:), allocatable :: history_new
      integer :: s
      s = size (res_set%history)
      allocate (history_new (2 * s))
      history_new(1:s) = res_set%history(1:s)
      call move_alloc (history_new, res_set%history)
    end subroutine resonance_history_set_expand
```

18.3.6 Unit tests

Test module, followed by the corresponding implementation module.

⟨`resonances_ut.f90`⟩≡

```
  ⟨File header⟩

  module resonances_ut
    use unit_tests
    use resonances_uti

  ⟨Standard module head⟩

  ⟨Resonances: public test⟩

  contains

  ⟨Resonances: test driver⟩

  end module resonances_ut
```

⟨`resonances_uti.f90`⟩≡

```
  ⟨File header⟩
```

```
module resonances_uti

!!!⟨Use kinds⟩
⟨Use strings⟩
  use model_data, only: model_data_t

  use resonances, only: resonance_history_t

  use resonances

⟨Standard module head⟩

⟨Resonances: test declarations⟩

contains

⟨Resonances: tests⟩

end module resonances_uti
```

API: driver for the unit tests below.

⟨*Resonances: public test*⟩≡

```
  public :: resonances_test
```

⟨*Resonances: test driver*⟩≡

```
  subroutine resonances_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨Resonances: execute tests⟩
  end subroutine resonances_test
```

Basic operations on a resonance history object.

⟨*Resonances: execute tests*⟩≡

```
  call test (resonances_1, "resonances_1", &
       "check resonance history setup", &
       u, results)
```

⟨*Resonances: test declarations*⟩≡

```
  public :: resonances_1
```

⟨*Resonances: tests*⟩≡

```
  subroutine resonances_1 (u)
    integer, intent(in) :: u
    type(resonance_info_t) :: res_info
    type(resonance_history_t) :: res_history
    type(model_data_t), target :: model

    write (u, "(A)")  "* Test output: resonances_1"
    write (u, "(A)")  "*   Purpose: test resonance history setup"
    write (u, "(A)")

    write (u, "(A)")  "* Read model file"

    call model%init_sm_test ()

    write (u, "(A)")
```

```
    write (u, "(A)")  "* Empty resonance history"
    write (u, "(A)")

    call res_history%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Add resonance"
    write (u, "(A)")

    call res_info%init (3, -24, model, 5)
    call res_history%add_resonance (res_info)
    call res_history%write (u)


    write (u, "(A)")
    write (u, "(A)")  "* Add another resonance"
    write (u, "(A)")

    call res_info%init (7, 23, model, 5)
    call res_history%add_resonance (res_info)
    call res_history%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Remove resonance"
    write (u, "(A)")

    call res_history%remove_resonance (1)
    call res_history%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: resonances_1"

  end subroutine resonances_1
```

Basic operations on a resonance history object.

⟨*Resonances: execute tests*⟩+≡

```
  call test (resonances_2, "resonances_2", &
      "check O'Mega restriction strings", &
      u, results)
```

⟨*Resonances: test declarations*⟩+≡

```
  public :: resonances_2
```

⟨*Resonances: tests*⟩+≡

```
  subroutine resonances_2 (u)
    integer, intent(in) :: u
    type(resonance_info_t) :: res_info
    type(resonance_history_t) :: res_history
    type(model_data_t), target :: model
    type(string_t) :: restrictions
```

```
    write (u, "(A)")  "* Test output: resonances_2"
    write (u, "(A)")  "*   Purpose: test OMega restrictions strings &
         &for resonance history"
    write (u, "(A)")

    write (u, "(A)")  "* Read model file"

    call model%init_sm_test ()

    write (u, "(A)")
    write (u, "(A)")  "* Empty resonance history"
    write (u, "(A)")

    restrictions = res_history%as_omega_string (2)
    write (u, "(A,A,A)")  "restrictions = '", char (restrictions), "'"

    write (u, "(A)")
    write (u, "(A)")  "* Add resonance"
    write (u, "(A)")

    call res_info%init (3, -24, model, 5)
    call res_history%add_resonance (res_info)
    restrictions = res_history%as_omega_string (2)
    write (u, "(A,A,A)")  "restrictions = '", char (restrictions), "'"

    write (u, "(A)")
    write (u, "(A)")  "* Add another resonance"
    write (u, "(A)")

    call res_info%init (7, 23, model, 5)
    call res_history%add_resonance (res_info)
    restrictions = res_history%as_omega_string (2)
    write (u, "(A,A,A)")  "restrictions = '", char (restrictions), "'"

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: resonances_2"

  end subroutine resonances_2
```

Basic operations on a resonance history set.

⟨*Resonances: execute tests*⟩+≡

```
  call test (resonances_3, "resonances_3", &
       "check resonance history set", &
       u, results)
```

⟨*Resonances: test declarations*⟩+≡

```
  public :: resonances_3
```

⟨*Resonances: tests*⟩+≡

```
subroutine resonances_3 (u)
  integer, intent(in) :: u
  type(resonance_info_t) :: res_info
  type(resonance_history_t) :: res_history
  type(resonance_history_t), dimension(:), allocatable :: res_histories
  type(resonance_history_set_t) :: res_set
  type(model_data_t), target :: model
  integer :: i

  write (u, "(A)")  "* Test output: resonances_3"
  write (u, "(A)")  "*   Purpose: test resonance history set"
  write (u, "(A)")

  write (u, "(A)")  "* Read model file"

  call model%init_sm_test ()

  write (u, "(A)")
  write (u, "(A)")  "* Initialize resonance history set"
  write (u, "(A)")

  call res_set%init (initial_size = 2)

  write (u, "(A)")  "* Add resonance histories, one at a time"
  write (u, "(A)")

  call res_history%write (u)
  call res_set%enter (res_history)
  call res_history%clear ()

  write (u, *)

  call res_info%init (3, -24, model, 5)
  call res_history%add_resonance (res_info)
  call res_history%write (u)
  call res_set%enter (res_history)
  call res_history%clear ()

  write (u, *)

  call res_info%init (3, -24, model, 5)
  call res_history%add_resonance (res_info)
  call res_info%init (7, 23, model, 5)
  call res_history%add_resonance (res_info)
  call res_history%write (u)
  call res_set%enter (res_history)
  call res_history%clear ()

  write (u, *)

  call res_info%init (7, 23, model, 5)
  call res_history%add_resonance (res_info)
  call res_history%write (u)
  call res_set%enter (res_history)
```

```
call res_history%clear ()

write (u, *)

call res_info%init (3, -24, model, 5)
call res_history%add_resonance (res_info)
call res_history%write (u)
call res_set%enter (res_history)
call res_history%clear ()

write (u, "(A)")
write (u, "(A)")  "* Result"
write (u, "(A)")

call res_set%write (u)

write (u, "(A)")
write (u, "(A)")  "* Result in array form"

call res_set%to_array (res_histories)
do i = 1, size (res_histories)
   write (u, *)
   call res_histories(i)%write (u)
end do

write (u, "(A)")
write (u, "(A)")  "* Re-initialize resonance history set with filter n=2"
write (u, "(A)")

call res_set%init (n_filter = 2)

write (u, "(A)")  "* Add resonance histories, one at a time"
write (u, "(A)")

call res_info%init (3, -24, model, 5)
call res_history%add_resonance (res_info)
call res_history%write (u)
call res_set%enter (res_history)
call res_history%clear ()

write (u, *)

call res_info%init (3, -24, model, 5)
call res_history%add_resonance (res_info)
call res_info%init (7, 23, model, 5)
call res_history%add_resonance (res_info)
call res_history%write (u)
call res_set%enter (res_history)
call res_history%clear ()

write (u, *)

call res_info%init (7, 23, model, 5)
call res_history%add_resonance (res_info)
```

```
  call res_history%write (u)
  call res_set%enter (res_history)
  call res_history%clear ()

  write (u, *)

  call res_info%init (3, -24, model, 5)
  call res_history%add_resonance (res_info)
  call res_history%write (u)
  call res_set%enter (res_history)
  call res_history%clear ()

  write (u, "(A)")
  write (u, "(A)")  "* Result"
  write (u, "(A)")

  call res_set%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: resonances_3"

end subroutine resonances_3
```

18.4 Mappings

Mappings are objects that encode the transformation of the interval $(0, 1)$ to a physical variable m^2 or $\cos\theta$ (and back), as it is used in the phase space parameterization. The mapping objects contain fixed parameters, the associated methods implement the mapping and inverse mapping operations, including the computation of the Jacobian (phase space factor).

⟨mappings.f90⟩≡
```
⟨File header⟩

module mappings

⟨Use kinds⟩
  use kinds, only: TC
⟨Use strings⟩
  use io_units
  use constants, only: pi
  use format_defs, only: FMT_19
  use diagnostics
  use md5
  use model_data
  use flavors

⟨Standard module head⟩

⟨Mappings: public⟩

⟨Mappings: parameters⟩

⟨Mappings: types⟩

⟨Mappings: interfaces⟩

contains

⟨Mappings: procedures⟩

end module mappings
```

18.4.1 Default parameters

This type holds the default parameters, needed for setting the scale in cases where no mass parameter is available. The contents are public.

⟨*Mappings: public*⟩≡
```
public :: mapping_defaults_t
```

⟨*Mappings: types*⟩≡
```
type :: mapping_defaults_t
   real(default) :: energy_scale = 10
   real(default) :: invariant_mass_scale = 10
   real(default) :: momentum_transfer_scale = 10
   logical :: step_mapping = .true.
   logical :: step_mapping_exp = .true.
```

```
    logical :: enable_s_mapping = .false.
  contains
  ⟨Mappings: mapping defaults: TBP⟩
end type mapping_defaults_t
```

Output.

⟨*Mappings: mapping defaults: TBP*⟩≡

```
procedure :: write => mapping_defaults_write
```

⟨*Mappings: procedures*⟩≡

```
subroutine mapping_defaults_write (object, unit)
  class(mapping_defaults_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(3x,A," // FMT_19 // ")") "energy scale  = ", &
       object%energy_scale
  write (u, "(3x,A," // FMT_19 // ")") "mass scale    = ", &
       object%invariant_mass_scale
  write (u, "(3x,A," // FMT_19 // ")") "q scale       = ", &
       object%momentum_transfer_scale
  write (u, "(3x,A,L1)") "step mapping     = ", &
       object%step_mapping
  write (u, "(3x,A,L1)") "step exp. mode   = ", &
       object%step_mapping_exp
  write (u, "(3x,A,L1)") "allow s mapping  = ", &
       object%enable_s_mapping
end subroutine mapping_defaults_write
```

⟨*Mappings: public*⟩+≡

```
public :: mapping_defaults_md5sum
```

⟨*Mappings: procedures*⟩+≡

```
function mapping_defaults_md5sum (mapping_defaults) result (md5sum_map)
  character(32) :: md5sum_map
  type(mapping_defaults_t), intent(in) :: mapping_defaults
  integer :: u
  u = free_unit ()
  open (u, status = "scratch")
  write (u, *)  mapping_defaults%energy_scale
  write (u, *)  mapping_defaults%invariant_mass_scale
  write (u, *)  mapping_defaults%momentum_transfer_scale
  write (u, *)  mapping_defaults%step_mapping
  write (u, *)  mapping_defaults%step_mapping_exp
  write (u, *)  mapping_defaults%enable_s_mapping
  rewind (u)
  md5sum_map = md5sum (u)
  close (u)
end function mapping_defaults_md5sum
```

18.4.2 The Mapping type

Each mapping has a type (e.g., s-channel, infrared), a binary code (redundant, but useful for debugging), and a reference particle. The flavor code of this particle is stored for bookkeeping reasons, what matters are the mass and width of this particle. Furthermore, depending on the type, various mapping parameters can be set and used.

The parameters `a1` to `a3` (for m^2 mappings) and `b1` to `b3` (for $\cos\theta$ mappings) are values that are stored once to speed up the calculation, if `variable_limits` is false. The exact meaning of these parameters depends on the mapping type. The limits are fixed if there is a fixed c.m. energy.

⟨*Mappings: public*⟩+≡
```
public :: mapping_t
```

⟨*Mappings: types*⟩+≡
```
type :: mapping_t
   private
   integer :: type = NO_MAPPING
   integer(TC) :: bincode
   type(flavor_t) :: flv
   real(default) :: mass = 0
   real(default) :: width = 0
   logical :: a_unknown = .true.
   real(default) :: a1 = 0
   real(default) :: a2 = 0
   real(default) :: a3 = 0
   logical :: b_unknown = .true.
   real(default) :: b1 = 0
   real(default) :: b2 = 0
   real(default) :: b3 = 0
   logical :: variable_limits = .true.
 contains
 ⟨Mappings: mapping: TBP⟩
end type mapping_t
```

The valid mapping types. The extra type `STEP_MAPPING` is used only internally.

⟨*Mappings: parameters*⟩≡
  ⟨*Mapping modes*⟩

18.4.3 Screen output

Do not write empty mappings.

⟨*Mappings: public*⟩+≡
```
public :: mapping_write
```

⟨*Mappings: procedures*⟩+≡
```
subroutine mapping_write (map, unit, verbose)
  type(mapping_t), intent(in) :: map
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u
  character(len=9) :: str
  u = given_output_unit (unit);  if (u < 0)  return
```

```
  select case(map%type)
  case(S_CHANNEL); str = "s_channel"
  case(COLLINEAR); str = "collinear"
  case(INFRARED);  str = "infrared "
  case(RADIATION); str = "radiation"
  case(T_CHANNEL); str = "t_channel"
  case(U_CHANNEL); str = "u_channel"
  case(STEP_MAPPING_E);  str = "step_exp"
  case(STEP_MAPPING_H);  str = "step_hyp"
  case(ON_SHELL);  str = "on_shell"
  case default;    str = "????????"
  end select
  if (map%type /= NO_MAPPING) then
     write (u, '(1x,A,I4,A)') &
          "Branch #", map%bincode, ":  " // &
          "Mapping (" // str // ") for particle " // &
          '"' // char (map%flv%get_name ()) // '"'
     if (present (verbose)) then
        if (verbose) then
           select case (map%type)
           case (S_CHANNEL, RADIATION, STEP_MAPPING_E, STEP_MAPPING_H)
              write (u, "(1x,A,3(" // FMT_19 // "))")  &
                   "  m/w    = ", map%mass, map%width
           case default
              write (u, "(1x,A,3(" // FMT_19 // "))")  &
                   "  m      = ", map%mass
           end select
           select case (map%type)
           case (S_CHANNEL, T_CHANNEL, U_CHANNEL, &
                STEP_MAPPING_E, STEP_MAPPING_H, &
                COLLINEAR, INFRARED, RADIATION)
              write (u, "(1x,A,3(" // FMT_19 // "))")  &
                   "  a1/2/3 = ", map%a1, map%a2, map%a3
           end select
           select case (map%type)
           case (T_CHANNEL, U_CHANNEL, COLLINEAR)
              write (u, "(1x,A,3(" // FMT_19 // "))")  &
                   "  b1/2/3 = ", map%b1, map%b2, map%b3
           end select
        end if
     end if
  end if
end subroutine mapping_write
```

18.4.4 Define a mapping

The initialization routine sets the mapping type and the particle (binary code and flavor code) for which the mapping applies (e.g., a Z resonance in branch #3).

⟨*Mappings: public*⟩+≡

```
public :: mapping_init
```

⟨*Mappings: procedures*⟩+≡

```
subroutine mapping_init (mapping, bincode, type, f, model)
  type(mapping_t), intent(inout) :: mapping
  integer(TC), intent(in) :: bincode
  type(string_t), intent(in) :: type
  integer, intent(in), optional :: f
  class(model_data_t), intent(in), optional, target :: model
  mapping%bincode = bincode
  select case (char (type))
  case ("s_channel");  mapping%type = S_CHANNEL
  case ("collinear");  mapping%type = COLLINEAR
  case ("infrared");   mapping%type = INFRARED
  case ("radiation");  mapping%type = RADIATION
  case ("t_channel");  mapping%type = T_CHANNEL
  case ("u_channel");  mapping%type = U_CHANNEL
  case ("step_exp");  mapping%type = STEP_MAPPING_E
  case ("step_hyp");  mapping%type = STEP_MAPPING_H
  case ("on_shell");  mapping%type = ON_SHELL
  case default
     call msg_bug ("Mappings: encountered undefined mapping key '" &
          // char (type) // "'")
  end select
  if (present (f) .and. present (model))  call mapping%flv%init (f, model)
end subroutine mapping_init
```

This sets the actual mass and width, using a parameter set. Since the auxiliary parameters will only be determined when the mapping is first called, they are marked as unknown.

⟨*Mappings: public*⟩+≡

```
public :: mapping_set_parameters
```

⟨*Mappings: procedures*⟩+≡

```
subroutine mapping_set_parameters (map, mapping_defaults, variable_limits)
  type(mapping_t), intent(inout) :: map
  type(mapping_defaults_t), intent(in) :: mapping_defaults
  logical, intent(in) :: variable_limits
  if (map%type /= NO_MAPPING) then
     map%mass  = map%flv%get_mass ()
     map%width = map%flv%get_width ()
     map%variable_limits = variable_limits
     map%a_unknown = .true.
     map%b_unknown = .true.
     select case (map%type)
     case (S_CHANNEL)
        if (map%mass <= 0) then
           call mapping_write (map)
           call msg_fatal &
                & (" S-channel resonance must have positive mass")
        else if (map%width <= 0) then
           call mapping_write (map)
           call msg_fatal &
                & (" S-channel resonance must have positive width")
        end if
     case (RADIATION)
```

```
            map%width = max (map%width, mapping_defaults%energy_scale)
         case (INFRARED, COLLINEAR)
            map%mass = max (map%mass, mapping_defaults%invariant_mass_scale)
         case (T_CHANNEL, U_CHANNEL)
            map%mass = max (map%mass, mapping_defaults%momentum_transfer_scale)
         end select
      end if
   end subroutine mapping_set_parameters
```

For a step mapping the mass and width are set directly, instead of being determined from the flavor parameter (which is meaningless here). They correspond to the effective upper bound of phase space due to a resonance, as opposed to the absolute upper bound.

⟨*Mappings: public*⟩+≡

```
   public :: mapping_set_step_mapping_parameters
```

⟨*Mappings: procedures*⟩+≡

```
   subroutine mapping_set_step_mapping_parameters (map, &
        mass, width, variable_limits)
      type(mapping_t), intent(inout) :: map
      real(default), intent(in) :: mass, width
      logical, intent(in) :: variable_limits
      select case (map%type)
      case (STEP_MAPPING_E, STEP_MAPPING_H)
         map%variable_limits = variable_limits
         map%a_unknown = .true.
         map%b_unknown = .true.
         map%mass = mass
         map%width = width
      end select
   end subroutine mapping_set_step_mapping_parameters
```

18.4.5 Retrieve contents

Return true if there is any / an s-channel mapping.

⟨*Mappings: public*⟩+≡

```
   public :: mapping_is_set
   public :: mapping_is_s_channel
   public :: mapping_is_on_shell
```

⟨*Mappings: mapping: TBP*⟩≡

```
   procedure :: is_set => mapping_is_set
   procedure :: is_s_channel => mapping_is_s_channel
   procedure :: is_on_shell => mapping_is_on_shell
```

⟨*Mappings: procedures*⟩+≡

```
   function mapping_is_set (mapping) result (flag)
      class(mapping_t), intent(in) :: mapping
      logical :: flag
      flag = mapping%type /= NO_MAPPING
   end function mapping_is_set

   function mapping_is_s_channel (mapping) result (flag)
```

```
    class(mapping_t), intent(in) :: mapping
    logical :: flag
    flag = mapping%type == S_CHANNEL
  end function mapping_is_s_channel

  function mapping_is_on_shell (mapping) result (flag)
    class(mapping_t), intent(in) :: mapping
    logical :: flag
    flag = mapping%type == ON_SHELL
  end function mapping_is_on_shell
```

Return the binary code for the mapped particle.

⟨*Mappings: mapping: TBP*⟩+≡

```
  procedure :: get_bincode => mapping_get_bincode
```

⟨*Mappings: procedures*⟩+≡

```
  function mapping_get_bincode (mapping) result (bincode)
    class(mapping_t), intent(in) :: mapping
    integer(TC) :: bincode
    bincode = mapping%bincode
  end function mapping_get_bincode
```

Return the flavor object for the mapped particle.

⟨*Mappings: mapping: TBP*⟩+≡

```
  procedure :: get_flv => mapping_get_flv
```

⟨*Mappings: procedures*⟩+≡

```
  function mapping_get_flv (mapping) result (flv)
    class(mapping_t), intent(in) :: mapping
    type(flavor_t) :: flv
    flv = mapping%flv
  end function mapping_get_flv
```

Return stored mass and width, respectively.

⟨*Mappings: public*⟩+≡

```
  public :: mapping_get_mass
  public :: mapping_get_width
```

⟨*Mappings: procedures*⟩+≡

```
  function mapping_get_mass (mapping) result (mass)
    real(default) :: mass
    type(mapping_t), intent(in) :: mapping
    mass = mapping%mass
  end function mapping_get_mass

  function mapping_get_width (mapping) result (width)
    real(default) :: width
    type(mapping_t), intent(in) :: mapping
    width = mapping%width
  end function mapping_get_width
```

18.4.6 Compare mappings

Equality for single mappings and arrays

⟨*Mappings: public*⟩+≡
```
public :: operator(==)
```

⟨*Mappings: interfaces*⟩≡
```
interface operator(==)
   module procedure mapping_equal
end interface
```

⟨*Mappings: procedures*⟩+≡
```
function mapping_equal (m1, m2) result (equal)
  type(mapping_t), intent(in) :: m1, m2
  logical :: equal
  if (m1%type == m2%type) then
     select case (m1%type)
     case (NO_MAPPING)
        equal = .true.
     case (S_CHANNEL, RADIATION, STEP_MAPPING_E, STEP_MAPPING_H)
        equal = (m1%mass == m2%mass) .and. (m1%width == m2%width)
     case default
        equal = (m1%mass == m2%mass)
     end select
  else
     equal = .false.
  end if
end function mapping_equal
```

18.4.7 Mappings of the invariant mass

Inserting an x value between 0 and 1, we want to compute the corresponding invariant mass $m^2(x)$ and the jacobian, aka phase space factor $f(x)$. We also need the reverse operation.

In general, the phase space factor f is defined by

$$\frac{1}{s}\int_{m^2_{\min}}^{m^2_{\max}} dm^2\, g(m^2) = \int_0^1 dx\, \frac{1}{s}\,\frac{dm^2}{dx}\, g(m^2(x)) = \int_0^1 dx\, f(x)\, g(x), \tag{18.8}$$

where thus

$$f(x) = \frac{1}{s}\,\frac{dm^2}{dx}. \tag{18.9}$$

With this mapping, a function of the form

$$g(m^2) = c\frac{dx(m^2)}{dm^2} \tag{18.10}$$

is mapped to a constant:

$$\frac{1}{s}\int_{m^2_{\min}}^{m^2_{\max}} dm^2\, g(m^2) = \int_0^1 dx\, f(x)\, g(m^2(x)) = \int_0^1 dx\, \frac{c}{s}. \tag{18.11}$$

Here is the mapping routine. Input are the available energy squared `s`, the limits for m^2, and the x value. Output are the m^2 value and the phase space factor f.

⟨*Mappings: public*⟩+≡
```
public :: mapping_compute_msq_from_x
```

⟨*Mappings: procedures*⟩+≡
```
subroutine mapping_compute_msq_from_x (map, s, msq_min, msq_max, msq, f, x)
  type(mapping_t), intent(inout) :: map
  real(default), intent(in) :: s, msq_min, msq_max
  real(default), intent(out) :: msq, f
  real(default), intent(in) :: x
  real(default) :: z, msq0, msq1, tmp
  integer :: type
  type = map%type
  if (s == 0) &
       call msg_fatal (" Applying msq mapping for zero energy")
⟨Modify mapping type if necessary⟩
  select case(type)
  case (NO_MAPPING)
   ⟨Constants for trivial msq mapping⟩
   ⟨Apply trivial msq mapping⟩
  case (S_CHANNEL)
   ⟨Constants for s-channel resonance mapping⟩
   ⟨Apply s-channel resonance mapping⟩
  case (COLLINEAR, INFRARED, RADIATION)
   ⟨Constants for s-channel pole mapping⟩
   ⟨Apply s-channel pole mapping⟩
  case (T_CHANNEL, U_CHANNEL)
   ⟨Constants for t-channel pole mapping⟩
   ⟨Apply t-channel pole mapping⟩
  case (STEP_MAPPING_E)
   ⟨Constants for exponential step mapping⟩
   ⟨Apply exponential step mapping⟩
  case (STEP_MAPPING_H)
   ⟨Constants for hyperbolic step mapping⟩
   ⟨Apply hyperbolic step mapping⟩
  case default
     call msg_fatal ( " Attempt to apply undefined msq mapping")
  end select
end subroutine mapping_compute_msq_from_x
```

The inverse mapping

⟨*Mappings: public*⟩+≡
```
public :: mapping_compute_x_from_msq
```

⟨*Mappings: procedures*⟩+≡
```
subroutine mapping_compute_x_from_msq (map, s, msq_min, msq_max, msq, f, x)
  type(mapping_t), intent(inout) :: map
  real(default), intent(in) :: s, msq_min, msq_max
  real(default), intent(in) :: msq
  real(default), intent(out) :: f, x
  real(default) :: msq0, msq1, tmp, z
  integer :: type
```

```
  type = map%type
  if (s == 0) &
       call msg_fatal (" Applying inverse msq mapping for zero energy")
⟨Modify mapping type if necessary⟩
  select case (type)
  case (NO_MAPPING)
   ⟨Constants for trivial msq mapping⟩
   ⟨Apply inverse trivial msq mapping⟩
  case (S_CHANNEL)
   ⟨Constants for s-channel resonance mapping⟩
   ⟨Apply inverse s-channel resonance mapping⟩
  case (COLLINEAR, INFRARED, RADIATION)
   ⟨Constants for s-channel pole mapping⟩
   ⟨Apply inverse s-channel pole mapping⟩
  case (T_CHANNEL, U_CHANNEL)
   ⟨Constants for t-channel pole mapping⟩
   ⟨Apply inverse t-channel pole mapping⟩
  case (STEP_MAPPING_E)
   ⟨Constants for exponential step mapping⟩
   ⟨Apply inverse exponential step mapping⟩
  case (STEP_MAPPING_H)
   ⟨Constants for hyperbolic step mapping⟩
   ⟨Apply inverse hyperbolic step mapping⟩
  case default
     call msg_fatal ( " Attempt to apply undefined msq mapping")
  end select
end subroutine mapping_compute_x_from_msq
```

Trivial mapping

We simply map the boundaries of the interval $(m_{\min}, m_{\max})$ to $(0, 1)$:

$$m^2 = (1-x)m_{\min}^2 + x m_{\max}^2; \tag{18.12}$$

the inverse is

$$x = \frac{m^2 - m_{\min}^2}{m_{\max}^2 - m_{\min}^2}. \tag{18.13}$$

Hence

$$f(x) = \frac{m_{\max}^2 - m_{\min}^2}{s}, \tag{18.14}$$

and we have, as required,

$$f(x)\,\frac{dx}{dm^2} = \frac{1}{s}. \tag{18.15}$$

We store the constant parameters the first time the mapping is called – or, if limits vary, recompute them each time.

⟨*Constants for trivial msq mapping*⟩≡

```
if (map%variable_limits .or. map%a_unknown) then
   map%a1 = 0
   map%a2 = msq_max - msq_min
   map%a3 = map%a2 / s
   map%a_unknown = .false.
end if
```

⟨*Apply trivial msq mapping*⟩≡

```
msq = (1-x) * msq_min + x * msq_max
f = map%a3
```

⟨*Apply inverse trivial msq mapping*⟩≡

```
if (map%a2 /= 0) then
   x = (msq - msq_min) / map%a2
else
   x = 0
end if
f = map%a3
```

Resonance or step mapping does not make much sense if the resonance mass is outside the kinematical bounds. If this is the case, revert to `NO_MAPPING`. This is possible even if the kinematical bounds vary from event to event.

⟨*Modify mapping type if necessary*⟩≡

```
select case (type)
case (S_CHANNEL, STEP_MAPPING_E, STEP_MAPPING_H)
   msq0 = map%mass**2
   if (msq0 < msq_min .or. msq0 > msq_max)  type = NO_MAPPING
end select
```

Breit-Wigner mapping

A Breit-Wigner resonance with mass M and width Γ is flattened by the following mapping:

This mapping does not make much sense if the resonance mass is too low. If this is the case, revert to `NO_MAPPING`. There is a tricky point with this if the mass is too high: `msq_max` is not a constant if structure functions are around. However, switching the type depending on the overall energy does not change the integral, it is just another branching point.

$$m^2 = M(M + t\Gamma), \tag{18.16}$$

where

$$t = \tan\left[(1-x)\arctan\frac{m_{\mathrm{min}}^2 - M^2}{M\Gamma} + x\arctan\frac{m_{\mathrm{max}}^2 - M^2}{M\Gamma}\right]. \tag{18.17}$$

The inverse:

$$x = \frac{\arctan\frac{m^2-M^2}{M\Gamma} - \arctan\frac{m_{\mathrm{min}}^2-M^2}{M\Gamma}}{\arctan\frac{m_{\mathrm{max}}^2-M^2}{M\Gamma} - \arctan\frac{m_{\mathrm{min}}^2-M^2}{M\Gamma}} \tag{18.18}$$

The phase-space factor of this transformation is

$$f(x) = \frac{M\Gamma}{s}\left(\arctan\frac{m_{\mathrm{max}}^2 - M^2}{M\Gamma} - \arctan\frac{m_{\mathrm{min}}^2 - M^2}{M\Gamma}\right)(1+t^2). \tag{18.19}$$

This maps any function proportional to

$$g(m^2) = \frac{M\Gamma}{(m^2 - M^2)^2 + M^2\Gamma^2} \tag{18.20}$$

to a constant times $1/s$.

⟨*Constants for s-channel resonance mapping*⟩≡

```
if (map%variable_limits .or. map%a_unknown) then
   msq0 = map%mass ** 2
   map%a1 = atan ((msq_min - msq0) / (map%mass * map%width))
   map%a2 = atan ((msq_max - msq0) / (map%mass * map%width))
   map%a3 = (map%a2 - map%a1) * (map%mass * map%width) / s
   map%a_unknown = .false.
end if
```

⟨*Apply s-channel resonance mapping*⟩≡

```
z = (1-x) * map%a1 + x * map%a2
if (-pi/2 < z .and. z < pi/2) then
   tmp = tan (z)
   msq = map%mass * (map%mass + map%width * tmp)
   f = map%a3 * (1 + tmp**2)
else
   msq = 0
   f = 0
end if
```

⟨*Apply inverse s-channel resonance mapping*⟩≡

```
tmp = (msq - msq0) / (map%mass * map%width)
x = (atan (tmp) - map%a1) / (map%a2 - map%a1)
f = map%a3 * (1 + tmp**2)
```

Mapping for massless splittings

This mapping accounts for approximately scale-invariant behavior where $\ln M^2$ is evenly distributed.

$$m^2 = m_{\min}^2 + M^2 \left(\exp(xL) - 1\right) \tag{18.21}$$

where

$$L = \ln\left(\frac{m_{\max}^2 - m_{\min}^2}{M^2} + 1\right). \tag{18.22}$$

The inverse:

$$x = \frac{1}{L} \ln\left(\frac{m^2 - m_{\min}^2}{M^2} + 1\right) \tag{18.23}$$

The constant M is a characteristic scale. Above this scale ($m^2 - m_{\min}^2 \gg M^2$), this mapping behaves like $x \propto \ln m^2$, while below the scale it reverts to a linear mapping.

The phase-space factor is

$$f(x) = \frac{M^2}{s} \exp(xL)\, L. \tag{18.24}$$

A function proportional to

$$g(m^2) = \frac{1}{(m^2 - m_{\min}^2) + M^2} \tag{18.25}$$

is mapped to a constant, i.e., a simple pole near $m_{\min}$ with a regulator mass M.

This type of mapping is useful for massless collinear and infrared singularities, where the scale is stored as the mass parameter. In the radiation case

(IR radiation off massive particle), the heavy particle width is the characteristic scale.

⟨*Constants for s-channel pole mapping*⟩≡

```
if (map%variable_limits .or. map%a_unknown) then
   if (type == RADIATION) then
      msq0 = map%width**2
   else
      msq0 = map%mass**2
   end if
   map%a1 = msq0
   map%a2 = log ((msq_max - msq_min) / msq0 + 1)
   map%a3 = map%a2 / s
   map%a_unknown = .false.
end if
```

⟨*Apply s-channel pole mapping*⟩≡

```
msq1 = map%a1 * exp (x * map%a2)
msq = msq1 - map%a1 + msq_min
f = map%a3 * msq1
```

⟨*Apply inverse s-channel pole mapping*⟩≡

```
msq1 = msq - msq_min + map%a1
x = log (msq1 / map%a1) / map%a2
f = map%a3 * msq1
```

Mapping for t-channel poles

This is also approximately scale-invariant, and we use the same type of mapping as before. However, we map $1/x$ singularities at both ends of the interval; again, the mapping becomes linear when the distance is less than M^2:

$$m^2 = \begin{cases} m_{\rm min}^2 + M^2\left(\exp(xL) - 1\right) & \text{for } 0 < x < \frac{1}{2} \\ m_{\rm max}^2 - M^2\left(\exp((1-x)L) - 1\right) & \text{for } \frac{1}{2} \leq x < 1 \end{cases} \tag{18.26}$$

where

$$L = 2\ln\left(\frac{m_{\rm max}^2 - m_{\rm min}^2}{2M^2} + 1\right). \tag{18.27}$$

The inverse:

$$x = \begin{cases} \frac{1}{L}\ln\left(\frac{m^2 - m_{\rm min}^2}{M^2} + 1\right) & \text{for } m^2 < (m_{\rm max}^2 - m_{\rm min}^2)/2 \\ 1 - \frac{1}{L}\ln\left(\frac{m_{\rm max} - m^2}{M^2} + 1\right) & \text{for } m^2 \geq (m_{\rm max}^2 - m_{\rm min}^2)/2 \end{cases} \tag{18.28}$$

The phase-space factor is

$$f(x) = \begin{cases} \frac{M^2}{s}\,\exp(xL)\,L. & \text{for } 0 < x < \frac{1}{2} \\ \frac{M^2}{s}\,\exp((1-x)L)\,L. & \text{for } \frac{1}{2} \leq x < 1 \end{cases} \tag{18.29}$$

A (continuous) function proportional to

$$g(m^2) = \begin{cases} 1/(m^2 - m_{\rm min}^2) + M^2) & \text{for } m^2 < (m_{\rm max}^2 - m_{\rm min}^2)/2 \\ 1/((m_{\rm max}^2 - m^2) + M^2) & \text{for } m^2 \leq (m_{\rm max}^2 - m_{\rm min}^2)/2 \end{cases} \tag{18.30}$$

is mapped to a constant by this mapping, i.e., poles near both ends of the interval.

⟨*Constants for t-channel pole mapping*⟩≡

```
if (map%variable_limits .or. map%a_unknown) then
   msq0 = map%mass**2
   map%a1 = msq0
   map%a2 = 2 * log ((msq_max - msq_min)/(2*msq0) + 1)
   map%a3 = map%a2 / s
   map%a_unknown = .false.
end if
```

⟨*Apply t-channel pole mapping*⟩≡

```
if (x < .5_default) then
   msq1 = map%a1 * exp (x * map%a2)
   msq = msq1 - map%a1 + msq_min
else
   msq1 = map%a1 * exp ((1-x) * map%a2)
   msq = -(msq1 - map%a1) + msq_max
end if
f = map%a3 * msq1
```

⟨*Apply inverse t-channel pole mapping*⟩≡

```
if (msq < (msq_max + msq_min)/2) then
   msq1 = msq - msq_min + map%a1
   x = log (msq1/map%a1) / map%a2
else
   msq1 = msq_max - msq + map%a1
   x = 1 - log (msq1/map%a1) / map%a2
end if
f = map%a3 * msq1
```

18.4.8 Step mapping

Step mapping is useful when the allowed range for a squared-mass variable is large, but only a fraction at the lower end is populated because the particle in question is an (off-shell) decay product of a narrow resonance. I.e., if the resonance was forced to be on-shell, the upper end of the range would be the resonance mass, minus the effective (real or resonance) mass of the particle(s) in the sibling branch of the decay.

The edge of this phase space section has a width which is determined by the width of the parent, plus the width of the sibling branch. (The widths might be added in quadrature, but this precision is probably not important.)

Fermi function

A possible mapping is derived from the Fermi function which has precisely this behavior. The Fermi function is given by

$$f(x) = \frac{1}{1 + \exp\frac{x-\mu}{\gamma}} \tag{18.31}$$

where x is taken as the invariant mass squared, μ is the invariant mass squared of the edge, and γ is the effective width which is given by the widths of the

parent and the sibling branch. (Widths might be added in quadrature, but we do not require this level of precision.)

$$x = \frac{m^2 - m_{\rm min}^2}{\Delta m^2} \tag{18.32}$$

$$\mu = \frac{m_{\rm max,eff}^2 - m_{\rm min}^2}{\Delta m^2} \tag{18.33}$$

$$\gamma = \frac{2 m_{\rm max,eff} \Gamma}{\Delta m^2} \tag{18.34}$$

with

$$\Delta m^2 = m_{\rm max}^2 - m_{\rm min}^2 \tag{18.35}$$

m^2 is thus given by

$$m^2(x) = x m_{\rm max}^2 + (1-x) m_{\rm min}^2 \tag{18.36}$$

For the mapping, we compute the integral $g(x)$ of the Fermi function, normalized such that $g(0) = 0$ and $g(1) = 1$. We introduce the abbreviations

$$\alpha = 1 - \gamma \ln \frac{1 + \beta e^{1/\gamma}}{1 + \beta} \tag{18.37}$$

$$\beta = e^{-\mu/\gamma} \tag{18.38}$$

and obtain

$$g(x) = \frac{1}{\alpha} \left(x - \gamma \ln \frac{1 + \beta e^{x/\gamma}}{1 + \beta} \right) \tag{18.39}$$

The actual mapping is the inverse function $h(y) = g^{-1}(y)$,

$$h(y) = -\gamma \ln \left(e^{-\alpha y/\gamma} (1 + \beta) - \beta \right) \tag{18.40}$$

The Jacobian is

$$\frac{dh}{dy} = \alpha \left(1 - e^{\alpha y/\gamma} \frac{\beta}{1 + \beta} \right)^{-1} \tag{18.41}$$

which is equal to $1/(dg/dx)$, namely

$$\frac{dg}{dx} = \frac{1}{\alpha} \frac{1}{1 + \beta e^{x/\gamma}} \tag{18.42}$$

The final result is

$$\int_{m_{\rm min}^2}^{m_{\rm max}^2} dm^2 \, F(m^2) = \Delta m^2 \int_0^1 dx \, F(m^2(x)) \tag{18.43}$$

$$= \Delta m^2 \int_0^1 dy \, F(m^2(h(y))) \, \frac{dh}{dy} \tag{18.44}$$

Here is the implementation. We fill `a1`, `a2`, `a3` with α, β, γ, respectively.

⟨*Constants for exponential step mapping*⟩≡

```
if (map%variable_limits .or. map%a_unknown) then
   map%a3 = max (2 * map%mass * map%width / (msq_max - msq_min), 0.01_default)
   map%a2 = exp (- (map%mass**2 - msq_min) / (msq_max - msq_min) &
                  / map%a3)
   map%a1 = 1 - map%a3 * log ((1 + map%a2 * exp (1 / map%a3)) / (1 + map%a2))
end if
```

⟨*Apply exponential step mapping*⟩≡

```
tmp = exp (- x * map%a1 / map%a3) * (1 + map%a2)
z = - map%a3 * log (tmp - map%a2)
msq  = z * msq_max + (1 - z) * msq_min
f = map%a1 / (1 - map%a2 / tmp) * (msq_max - msq_min) / s
```

⟨*Apply inverse exponential step mapping*⟩≡

```
z = (msq - msq_min) / (msq_max - msq_min)
tmp = 1 + map%a2 * exp (z / map%a3)
x = (z - map%a3 * log (tmp / (1 + map%a2))) &
    / map%a1
f = map%a1 * tmp * (msq_max - msq_min) / s
```

Hyperbolic mapping

The Fermi function has the drawback that it decreases exponentially. It might be preferable to take a function with a power-law decrease, such that the high-mass region is not completely depopulated.

Here, we start with the actual mapping which we take as

$$h(y) = \frac{b}{a-y} - \frac{b}{a} + \mu y \tag{18.45}$$

with the abbreviation

$$a = \frac{1}{2}\left(1 + \sqrt{1 + \frac{4b}{1-\mu}}\right) \tag{18.46}$$

This is a hyperbola in the xy plane. The derivative is

$$\frac{dh}{dy} = \frac{b}{(a-y)^2} + \mu \tag{18.47}$$

The constants correspond to

$$\mu = \frac{m_{\mathrm{max,eff}}^2 - m_{\mathrm{min}}^2}{\Delta m^2} \tag{18.48}$$

$$b = \frac{1}{\mu}\left(\frac{2m_{\mathrm{max,eff}}\Gamma}{\Delta m^2}\right)^2 \tag{18.49}$$

The inverse function is the solution of a quadratic equation,

$$g(x) = \frac{1}{2}\left[\left(a + \frac{x}{\mu} + \frac{b}{a\mu}\right) - \sqrt{\left(a - \frac{x}{\mu}\right)^2 + 2\frac{b}{a\mu}\left(a + \frac{x}{\mu}\right) + \left(\frac{b}{a\mu}\right)^2}\right] \tag{18.50}$$

The constants $a_{1,2,3}$ are identified with a, b, μ.

⟨*Constants for hyperbolic step mapping*⟩≡

```
if (map%variable_limits .or. map%a_unknown) then
   map%a3 = (map%mass**2 - msq_min) / (msq_max - msq_min)
   map%a2 = max ((2 * map%mass * map%width / (msq_max - msq_min))**2 &
                 / map%a3, 1e-6_default)
   map%a1 = (1 + sqrt (1 + 4 * map%a2 / (1 - map%a3))) / 2
end if
```

⟨*Apply hyperbolic step mapping*⟩≡
```
z = map%a2 / (map%a1 - x) - map%a2 / map%a1 + map%a3 * x
msq = z * msq_max + (1 - z) * msq_min
f = (map%a2 / (map%a1 - x)**2 + map%a3) * (msq_max - msq_min) / s
```

⟨*Apply inverse hyperbolic step mapping*⟩≡
```
z = (msq - msq_min) / (msq_max - msq_min)
tmp = map%a2 / (map%a1 * map%a3)
x = ((map%a1 + z / map%a3 + tmp) &
     - sqrt ((map%a1 - z / map%a3)**2 + 2 * tmp * (map%a1 + z / map%a3) &
             + tmp**2)) / 2
f = (map%a2 / (map%a1 - x)**2 + map%a3) * (msq_max - msq_min) / s
```

18.4.9 Mappings of the polar angle

The other type of singularity, a simple pole just outside the integration region, can occur in the integration over $\cos\theta$. This applies to exchange of massless (or light) particles.

Double poles (Coulomb scattering) are also possible, but only in certain cases. These are also handled by the single-pole mapping.

The mapping is analogous to the previous m^2 pole mapping, but with a different normalization and notation of variables:

$$\frac{1}{2}\int_{-1}^{1} d\cos\theta\, g(\theta) = \int_0^1 dx\, \frac{d\cos\theta}{dx}\, g(\theta(x)) = \int_0^1 dx\, f(x)\, g(x), \tag{18.51}$$

where thus

$$f(x) = \frac{1}{2}\,\frac{d\cos\theta}{dx}. \tag{18.52}$$

With this mapping, a function of the form

$$g(\theta) = c\frac{dx(\cos\theta)}{d\cos\theta} \tag{18.53}$$

is mapped to a constant:

$$\int_{-1}^{1} d\cos\theta\, g(\theta) = \int_0^1 dx\, f(x)\, g(\theta(x)) = \int_0^1 dx\, c. \tag{18.54}$$

⟨*Mappings: public*⟩+≡
```
public :: mapping_compute_ct_from_x
```

⟨*Mappings: procedures*⟩+≡
```
subroutine mapping_compute_ct_from_x (map, s, ct, st, f, x)
  type(mapping_t), intent(inout) :: map
  real(default), intent(in) :: s
  real(default), intent(out) :: ct, st, f
  real(default), intent(in) :: x
  real(default) :: tmp, ct1
  select case (map%type)
  case (NO_MAPPING, S_CHANNEL, INFRARED, RADIATION, &
       STEP_MAPPING_E, STEP_MAPPING_H)
   ⟨Apply trivial ct mapping⟩
  case (T_CHANNEL, U_CHANNEL, COLLINEAR)
```

```
    ⟨Constants for ct pole mapping⟩
    ⟨Apply ct pole mapping⟩
  case default
     call msg_fatal (" Attempt to apply undefined ct mapping")
  end select
end subroutine mapping_compute_ct_from_x
```

⟨*Mappings: public*⟩+≡

```
public :: mapping_compute_x_from_ct
```

⟨*Mappings: procedures*⟩+≡

```
subroutine mapping_compute_x_from_ct (map, s, ct, f, x)
  type(mapping_t), intent(inout) :: map
  real(default), intent(in) :: s
  real(default), intent(in) :: ct
  real(default), intent(out) :: f, x
  real(default) :: ct1
  select case (map%type)
  case (NO_MAPPING, S_CHANNEL, INFRARED, RADIATION, &
       STEP_MAPPING_E, STEP_MAPPING_H)
    ⟨Apply inverse trivial ct mapping⟩
  case (T_CHANNEL, U_CHANNEL, COLLINEAR)
    ⟨Constants for ct pole mapping⟩
    ⟨Apply inverse ct pole mapping⟩
  case default
     call msg_fatal (" Attempt to apply undefined inverse ct mapping")
  end select
end subroutine mapping_compute_x_from_ct
```

Trivial mapping

This is just the mapping of the interval $(-1, 1)$ to $(0, 1)$:

$$\cos\theta = -1 + 2x \tag{18.55}$$

and

$$f(x) = 1 \tag{18.56}$$

with the inverse

$$x = \frac{1 + \cos\theta}{2} \tag{18.57}$$

⟨*Apply trivial ct mapping*⟩≡

```
tmp = 2 * (1-x)
ct = 1 - tmp
st = sqrt (tmp * (2-tmp))
f = 1
```

⟨*Apply inverse trivial ct mapping*⟩≡

```
x = (ct + 1) / 2
f = 1
```

Pole mapping

As above for m^2, we simultaneously map poles at both ends of the $\cos\theta$ interval. The formulae are completely analogous:

$$\cos\theta = \begin{cases} \frac{M^2}{s}\left[\exp(xL) - 1\right] - 1 & \text{for } x < \frac{1}{2} \\ -\frac{M^2}{s}\left[\exp((1-x)L) - 1\right] + 1 & \text{for } x \geq \frac{1}{2} \end{cases} \tag{18.58}$$

where

$$L = 2\ln\frac{M^2 + s}{M^2}. \tag{18.59}$$

Inverse:

$$x = \begin{cases} \frac{1}{2L}\ln\frac{1+\cos\theta+M^2/s}{M^2/s} & \text{for } \cos\theta < 0 \\ 1 - \frac{1}{2L}\ln\frac{1-\cos\theta+M^2/s}{M^2/s} & \text{for } \cos\theta \geq 0 \end{cases} \tag{18.60}$$

The phase-space factor:

$$f(x) = \begin{cases} \frac{M^2}{s}\exp(xL)\,L & \text{for } x < \frac{1}{2} \\ \frac{M^2}{s}\exp((1-x)L)\,L & \text{for } x \geq \frac{1}{2} \end{cases} \tag{18.61}$$

⟨*Constants for ct pole mapping*⟩≡

```
if (map%variable_limits .or. map%b_unknown) then
   map%b1 = map%mass**2 / s
   map%b2 = log ((map%b1 + 1) / map%b1)
   map%b3 = 0
   map%b_unknown = .false.
end if
```

⟨*Apply ct pole mapping*⟩≡

```
if (x < .5_default) then
   ct1 = map%b1 * exp (2 * x * map%b2)
   ct = ct1 - map%b1 - 1
else
   ct1 = map%b1 * exp (2 * (1-x) * map%b2)
   ct = -(ct1 - map%b1) + 1
end if
if (ct >= -1 .and. ct <= 1) then
   st = sqrt (1 - ct**2)
   f = ct1 * map%b2
else
   ct = 1;  st = 0;  f = 0
end if
```

⟨*Apply inverse ct pole mapping*⟩≡

```
if (ct < 0) then
   ct1 = ct + map%b1 + 1
   x = log (ct1 / map%b1) / (2 * map%b2)
else
   ct1 = -ct + map%b1 + 1
   x = 1 - log (ct1 / map%b1) / (2 * map%b2)
end if
f = ct1 * map%b2
```

18.5 Phase-space trees

The phase space evaluation is organized in terms of trees, where each branch corresponds to three integrations: m^2, $\cos\theta$, and ϕ. The complete tree thus makes up a specific parameterization of the multidimensional phase-space integral. For the multi-channel integration, the phase-space tree is a single channel.

The trees imply mappings of formal Feynman tree graphs into arrays of integer numbers: Each branch, corresponding to a particular line in the graph, is assigned an integer code c (with kind value `TC` = tree code).

In this integer, each bit determines whether a particular external momentum flows through the line. The external branches therefore have codes $1, 2, 4, 8, \ldots$. An internal branch has those bits ORed corresponding to the momenta flowing through it. For example, a branch with momentum $p_1 + p_4$ has code $2^0 + 2^3 = 1 + 8 = 9$.

There is a two-fold ambiguity: Momentum conservation implies that the branch with code

$$c_0 = \sum_{i=1}^{n(\mathrm{ext})} 2^{i-1} \tag{18.62}$$

i.e. the branch with momentum $p_1 + p_2 + \ldots p_n$ has momentum zero, which is equivalent to tree code 0 by definition. Correspondingly,

$$c \quad \text{and} \quad c_0 - c = c \,\mathrm{XOR}\, c_0 \tag{18.63}$$

are equivalent. E.g., if there are five externals with codes $c = 1, 2, 4, 8, 16$, then $c = 9$ and $\bar{c} = 31 - 9 = 22$ are equivalent.

This ambiguity may be used to assign a direction to the line: If all momenta are understood as outgoing, $c = 9$ in the example above means $p_1 + p_4$, but $c = 22$ means $p_2 + p_3 + p_5 = -(p_1 + p_4)$.

Here we make use of the ambiguity in a slightly different way. First, the initial particles are singled out as those externals with the highest bits, the IN-bits. (Here: 8 and 16 for a $2 \to 3$ scattering process, 16 only for a $1 \to 4$ decay.) Then we invert those codes where all IN-bits are set. For a decay process this maps each tree of an equivalence class onto a unique representative (that one with the smallest integer codes). For a scattering process we proceed further:

The ambiguity remains in all branches where only one IN-bit is set, including the initial particles. If there are only externals with this property, we have an s-channel graph which we leave as it is. In all other cases, an internal with only one IN-bit is a t-channel line, which for phase space integration should be associated with one of the initial momenta as a reference axis. We take that one whose bit is set in the current tree code. (E.g., for branch $c = 9$ we use the initial particle $c = 8$ as reference axis, whereas for the same branch we would take $c = 16$ if it had been assigned $\bar{c} = 31 - 9 = 22$ as tree code.) Thus, different ways of coding the same t-channel graph imply different phase space parameterizations.

s-channel graphs have a unique parameterization. The same sets of parameterizations are used for t-channel graphs, except for the reference frames of their angular parts. We map each t-channel graph onto an s-channel graph as follows:

Working in ascending order, for each t-channel line (whose code has exactly one IN-bit set) the attached initial line is flipped upstream, while the outgoing

line is flipped downstream. (This works only if t-channel graphs are always parameterized beginning at their outer vertices, which we require as a restriction.) After all possible flips have been applied, we have an s-channel graph. We only have to remember the initial particle a vertex was originally attached to.

⟨phs_trees.f90⟩≡
```
  ⟨File header⟩

  module phs_trees

  ⟨Use kinds⟩
    use kinds, only: TC
  ⟨Use strings⟩
    use io_units
    use constants, only: twopi, twopi2, twopi5
    use format_defs, only: FMT_19
    use numeric_utils, only: vanishes
    use diagnostics
    use lorentz
    use permutations, only: permutation_t, permutation_size
    use permutations, only: permutation_init, permutation_find
    use permutations, only: tc_decay_level, tc_permute
    use model_data
    use flavors
    use resonances, only: resonance_history_t, resonance_info_t
    use mappings

  ⟨Standard module head⟩

  ⟨PHS trees: public⟩

  ⟨PHS trees: types⟩

  contains

  ⟨PHS trees: procedures⟩

  end module phs_trees
```

18.5.1 Particles

We define a particle type which contains only four-momentum and invariant mass squared, and a flag that tells whether the momentum is filled or not.

⟨PHS trees: public⟩≡
```
    public :: phs_prt_t
```

⟨PHS trees: types⟩≡
```
    type :: phs_prt_t
       private
       logical :: defined = .false.
       type(vector4_t) :: p
       real(default) :: p2
    end type phs_prt_t
```

Set contents:

⟨*PHS trees: public*⟩+≡

```
public :: phs_prt_set_defined
public :: phs_prt_set_undefined
public :: phs_prt_set_momentum
public :: phs_prt_set_msq
```

⟨*PHS trees: procedures*⟩≡

```
elemental subroutine phs_prt_set_defined (prt)
  type(phs_prt_t), intent(inout) :: prt
  prt%defined = .true.
end subroutine phs_prt_set_defined

elemental subroutine phs_prt_set_undefined (prt)
  type(phs_prt_t), intent(inout) :: prt
  prt%defined = .false.
end subroutine phs_prt_set_undefined

elemental subroutine phs_prt_set_momentum (prt, p)
  type(phs_prt_t), intent(inout) :: prt
  type(vector4_t), intent(in) :: p
  prt%p = p
end subroutine phs_prt_set_momentum

elemental subroutine phs_prt_set_msq (prt, p2)
  type(phs_prt_t), intent(inout) :: prt
  real(default), intent(in) :: p2
  prt%p2 = p2
end subroutine phs_prt_set_msq
```

Access methods:

⟨*PHS trees: public*⟩+≡

```
public :: phs_prt_is_defined
public :: phs_prt_get_momentum
public :: phs_prt_get_msq
```

⟨*PHS trees: procedures*⟩+≡

```
elemental function phs_prt_is_defined (prt) result (defined)
  logical :: defined
  type(phs_prt_t), intent(in) :: prt
  defined = prt%defined
end function phs_prt_is_defined

elemental function phs_prt_get_momentum (prt) result (p)
  type(vector4_t) :: p
  type(phs_prt_t), intent(in) :: prt
  p = prt%p
end function phs_prt_get_momentum

elemental function phs_prt_get_msq (prt) result (p2)
  real(default) :: p2
  type(phs_prt_t), intent(in) :: prt
  p2 = prt%p2
end function phs_prt_get_msq
```

Addition of momenta (invariant mass square is computed).

⟨*PHS trees: public*⟩+≡
```
public :: phs_prt_combine
```

⟨*PHS trees: procedures*⟩+≡
```
elemental subroutine phs_prt_combine (prt, prt1, prt2)
  type(phs_prt_t), intent(inout) :: prt
  type(phs_prt_t), intent(in) :: prt1, prt2
  prt%defined = .true.
  prt%p = prt1%p + prt2%p
  prt%p2 = prt%p ** 2
  call phs_prt_check (prt)
end subroutine phs_prt_combine
```

Output

⟨*PHS trees: public*⟩+≡
```
public :: phs_prt_write
```

⟨*PHS trees: procedures*⟩+≡
```
subroutine phs_prt_write (prt, unit)
  type(phs_prt_t), intent(in) :: prt
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  if (prt%defined) then
     call vector4_write (prt%p, u)
     write (u, "(1x,A,1x," // FMT_19 // ")") "T = ", prt%p2
  else
     write (u, "(3x,A)") "[undefined]"
  end if
end subroutine phs_prt_write
```

⟨*PHS trees: public*⟩+≡
```
public :: phs_prt_check
```

⟨*PHS trees: procedures*⟩+≡
```
elemental subroutine phs_prt_check (prt)
  type(phs_prt_t), intent(inout) :: prt
  if (prt%p2 < 0._default) then
     prt%p2 = 0._default
  end if
end subroutine phs_prt_check
```

18.5.2 The phase-space tree type

Definition

In the concrete implementation, each branch c may have two *daughters* c_1 and c_2 such that $c_1 + c_2 = c$, a *sibling* c_s and a *mother* c_m such that $c + c_s = c_m$, and a *friend* which is kept during flips, such that it can indicate a fixed reference frame. Absent entries are set $c = 0$.

First, declare the branch type. There is some need to have this public. Give initializations for all components, so no `init` routine is necessary. The branch has some information about the associated coordinates and about connections.

⟨*PHS trees: types*⟩+≡

```
type :: phs_branch_t
   private
   logical :: set = .false.
   logical :: inverted_decay = .false.
   logical :: inverted_axis = .false.
   integer(TC) :: mother = 0
   integer(TC) :: sibling = 0
   integer(TC) :: friend = 0
   integer(TC) :: origin = 0
   integer(TC), dimension(2) :: daughter = 0
   integer :: firstborn = 0
   logical :: has_children = .false.
   logical :: has_friend = .false.
   logical :: is_real = .false.
end type phs_branch_t
```

The tree type: No initialization, this is done by `phs_tree_init`. In addition to the branch array which

The branches are collected in an array which holds all possible branches, of which only a few are set. After flips have been applied, the branch $c_M = \sum_{i=1}^{n(\mathrm{fin})} 2^{i-1}$ must be there, indicating the mother of all decay products. In addition, we should check for consistency at the beginning.

`n_branches` is the number of those actually set. `n_externals` defines the number of significant bit, and `mask` is a code where all bits are set. Analogous: `n_in` and `mask_in` for the incoming particles.

The `mapping` array contains the mappings associated to the branches (corresponding indices). The array `mass_sum` contains the sum of the real masses of the external final-state particles associated to the branch. During phase-space evaluation, this determines the boundaries.

⟨*PHS trees: public*⟩+≡

```
public :: phs_tree_t
```

⟨*PHS trees: types*⟩+≡

```
type :: phs_tree_t
   private
   integer :: n_branches, n_externals, n_in, n_msq, n_angles
   integer(TC) :: n_branches_tot, n_branches_out
   integer(TC) :: mask, mask_in, mask_out
   type(phs_branch_t), dimension(:), allocatable :: branch
   type(mapping_t), dimension(:), allocatable :: mapping
   real(default), dimension(:), allocatable :: mass_sum
   real(default), dimension(:), allocatable :: effective_mass
   real(default), dimension(:), allocatable :: effective_width
   logical :: real_phsp = .false.
   integer, dimension(:), allocatable :: momentum_link
 contains
 ⟨PHS trees: phs tree: TBP⟩
end type phs_tree_t
```

The maximum number of external particles that can be represented is related to the bit size of the integer that stores binary codes. With the default integer of 32 bit on common machines, this is more than enough space. If TC is actually the default integer kind, there is no need to keep it separate, but doing so marks this as a special type of integer. So, just state that the maximum number is 32:

⟨*Limits: public parameters*⟩≡
```
integer, parameter, public :: MAX_EXTERNAL = 32
```

Constructor and destructor

Allocate memory for a phase-space tree with given number of externals and incoming. The number of allocated branches can easily become large, but appears manageable for realistic cases, e.g., for n_in=2 and n_out=8 we get $2^{10}-1 = 1023$.

⟨*PHS trees: public*⟩+≡
```
public :: phs_tree_init
public :: phs_tree_final
```

Here we set the masks for incoming and for all externals.

⟨*PHS trees: phs tree: TBP*⟩≡
```
procedure :: init => phs_tree_init
procedure :: final => phs_tree_final
```

⟨*PHS trees: procedures*⟩+≡
```
elemental subroutine phs_tree_init (tree, n_in, n_out, n_masses, n_angles)
  class(phs_tree_t), intent(inout) :: tree
  integer, intent(in) :: n_in, n_out, n_masses, n_angles
  integer(TC) :: i
  tree%n_externals = n_in + n_out
  tree%n_branches_tot = 2**(n_in+n_out) - 1
  tree%n_branches_out = 2**n_out - 1
  tree%mask = 0
  do i = 0, n_in + n_out - 1
     tree%mask = ibset (tree%mask, i)
  end do
  tree%n_in = n_in
  tree%mask_in = 0
  do i = n_out, n_in + n_out - 1
     tree%mask_in = ibset (tree%mask_in, i)
  end do
  tree%mask_out = ieor (tree%mask, tree%mask_in)
  tree%n_msq = n_masses
  tree%n_angles = n_angles
  allocate (tree%branch (tree%n_branches_tot))
  tree%n_branches  = 0
  allocate (tree%mapping (tree%n_branches_out))
  allocate (tree%mass_sum (tree%n_branches_out))
  allocate (tree%effective_mass (tree%n_branches_out))
  allocate (tree%effective_width (tree%n_branches_out))
end subroutine phs_tree_init

elemental subroutine phs_tree_final (tree)
  class(phs_tree_t), intent(inout) :: tree
  deallocate (tree%branch)
  deallocate (tree%mapping)
```

```
    deallocate (tree%mass_sum)
    deallocate (tree%effective_mass)
    deallocate (tree%effective_width)
  end subroutine phs_tree_final
```

Screen output

Write only the branches that are set:

⟨*PHS trees: public*⟩+≡

```
  public :: phs_tree_write
```

⟨*PHS trees: phs tree: TBP*⟩+≡

```
  procedure :: write => phs_tree_write
```

⟨*PHS trees: procedures*⟩+≡

```
  subroutine phs_tree_write (tree, unit)
    class(phs_tree_t), intent(in) :: tree
    integer, intent(in), optional :: unit
    integer :: u
    integer(TC) :: k
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, '(3X,A,1x,I0,5X,A,I3)') &
         'External:', tree%n_externals, 'Mask:', tree%mask
    write (u, '(3X,A,1x,I0,5X,A,I3)') &
         'Incoming:', tree%n_in, 'Mask:', tree%mask_in
    write (u, '(3X,A,1x,I0,5X,A,I3)') &
         'Branches:', tree%n_branches
    do k = size (tree%branch), 1, -1
       if (tree%branch(k)%set) &
            call phs_branch_write (tree%branch(k), unit=unit, kval=k)
    end do
    do k = 1, size (tree%mapping)
       call mapping_write (tree%mapping (k), unit, verbose=.true.)
    end do
    write (u, "(3x,A)") "Arrays: mass_sum, effective_mass, effective_width"
    do k = 1, size (tree%mass_sum)
       if (tree%branch(k)%set) then
          write (u, "(5x,I0,3(2x," // FMT_19 // "))") k, tree%mass_sum(k), &
               tree%effective_mass(k), tree%effective_width(k)
       end if
    end do
  end subroutine phs_tree_write

  subroutine phs_branch_write (b, unit, kval)
    type(phs_branch_t), intent(in) :: b
    integer, intent(in), optional :: unit
    integer(TC), intent(in), optional :: kval
    integer :: u
    integer(TC) :: k
    character(len=6) :: tmp
    character(len=1) :: firstborn(2), sign_decay, sign_axis
    integer :: i
    u = given_output_unit (unit);  if (u < 0)  return
    k = 0;  if (present (kval))  k = kval
```

```
  if (b%origin /= 0) then
     write(tmp, '(A,I4,A)') '(', b%origin, ')'
  else
     tmp = ' '
  end if
  do i=1, 2
     if (b%firstborn == i) then
        firstborn(i) = "*"
     else
        firstborn(i) = " "
     end if
  end do
  if (b%inverted_decay) then
     sign_decay = "-"
  else
     sign_decay = "+"
  end if
  if (b%inverted_axis) then
     sign_axis = "-"
  else
     sign_axis = "+"
  end if
  if (b%has_children) then
     if (b%has_friend) then
        write(u,'(4X,A1,I0,3x,A,1X,A,I0,A1,1x,I0,A1,1X,A1,1X,A,1x,I0)') &
             &   '*', k, tmp, &
             &   'Daughters: ', &
             &   b%daughter(1), firstborn(1), &
             &   b%daughter(2), firstborn(2), sign_decay, &
             &   'Friend:    ', b%friend
     else
        write(u,'(4X,A1,I0,3x,A,1X,A,I0,A1,1x,I0,A1,1X,A1,1X,A)') &
             &   '*', k, tmp, &
             &   'Daughters: ', &
             &   b%daughter(1), firstborn(1), &
             &   b%daughter(2), firstborn(2), sign_decay, &
             &   '(axis '//sign_axis//')'
     end if
  else
     write(u,'(5X,I0)') k
  end if
end subroutine phs_branch_write
```

18.5.3 PHS tree setup

Transformation into an array of branch codes and back

Assume that the tree/array has been created before with the appropriate length and is empty.

⟨*PHS trees: public*⟩+≡

```
public :: phs_tree_from_array
```

⟨*PHS trees: phs tree: TBP*⟩+≡
```
procedure :: from_array => phs_tree_from_array
```

⟨*PHS trees: procedures*⟩+≡
```
subroutine phs_tree_from_array (tree, a)
  class(phs_tree_t), intent(inout) :: tree
  integer(TC), dimension(:), intent(in) :: a
  integer :: i
  integer(TC) :: k
⟨Set branches from array a⟩
⟨Set external branches if necessary⟩
⟨Check number of branches⟩
⟨Determine the connections⟩
contains
⟨Subroutine: set relatives⟩
end subroutine phs_tree_from_array
```

First, set all branches specified by the user. If all IN-bits are set, we invert the branch code.

⟨*Set branches from array* a⟩≡
```
do i=1, size(a)
   k = a(i)
   if (iand(k, tree%mask_in) == tree%mask_in)  k = ieor(tree%mask, k)
   tree%branch(k)%set = .true.
   tree%n_branches = tree%n_branches+1
end do
```

The external branches are understood, so set them now if not yet done. In all cases ensure that the representative with one bit set is used, except for decays where the in-particle is represented by all OUT-bits set instead.

⟨*Set external branches if necessary*⟩≡
```
do i=0, tree%n_externals-1
   k = ibset(0,i)
   if (iand(k, tree%mask_in) == tree%mask_in)  k = ieor(tree%mask, k)
   if (tree%branch(ieor(tree%mask, k))%set) then
      tree%branch(ieor(tree%mask, k))%set = .false.
      tree%branch(k)%set = .true.
   else if (.not.tree%branch(k)%set) then
      tree%branch(k)%set = .true.
      tree%n_branches = tree%n_branches+1
   end if
end do
```

Now the number of branches set can be checked. Here we assume that the tree is binary. For three externals there are three branches in total, and for each additional external branch we get another internal one.

⟨*Check number of branches*⟩≡
```
if (tree%n_branches /= tree%n_externals*2-3) then
   call phs_tree_write (tree)
   call msg_bug &
        & (" Wrong number of branches set in phase space tree")
end if
```

For all branches that are set, except for the externals, we try to find the daughter branches:

⟨*Determine the connections*⟩≡

```
do k=1, size (tree%branch)
   if (tree%branch(k)%set .and. tc_decay_level (k) /= 1) then
      call branch_set_relatives(k)
   end if
end do
```

To this end, we scan all codes less than the current code, whether we can find two branches which are set and which together give the current code. After that, the tree may still not be connected, but at least we know if a branch does not have daughters: This indicates some inconsistency.

The algorithm ensures that, at this stage, the first daughter has a smaller code value than the second one.

⟨*Subroutine: set relatives*⟩≡

```
subroutine branch_set_relatives (k)
  integer(TC), intent(in) :: k
  integer(TC) :: m,n
  do m=1, k-1
     if(iand(k,m)==m) then
        n = ieor(k,m)
        if ( tree%branch(m)%set .and. tree%branch(n)%set ) then
           tree%branch(k)%daughter(1) = m;  tree%branch(k)%daughter(2) = n
           tree%branch(m)%mother      = k;  tree%branch(n)%mother      = k
           tree%branch(m)%sibling     = n;  tree%branch(n)%sibling     = m
           tree%branch(k)%has_children = .true.
           return
        end if
     end if
  end do
  call phs_tree_write (tree)
  call msg_bug &
       & (" Missing daughter branch(es) in phase space tree")
end subroutine branch_set_relatives
```

The inverse: this is trivial, fortunately.

Flip t-channel into s-channel

Flipping the tree is done upwards, beginning from the decay products. First we select a t-channel branch `k`: one which is set, which does have an IN-bit, and which is not an external particle.

Next, we determine the adjacent in-particle (called the 'friend' `f` here, since it will provide the reference axis for the angular integration). In addition, we look for the 'mother' and 'sibling' of this particle. If the latter field is empty, we select the (unique) other out-particle which has no mother, calling the internal subroutine `find_orphan`.

The flip is done as follows: We assume that the first daughter `d` is an s-channel line, which is true if the daughters are sorted. This will stay the first daughter. The second one is a t-channel line; it is exchanged with the 'sibling'

s. The new line which replaces the branch k is just the sum of s and d. In addition, we have to rearrange the relatives of s and d, as well of f.

Finally, we flip 'sibling' and 'friend' and set the new s-channel branch n which replaces the t-channel branch k. After this is complete, we are ready to execute another flip.

[Although the friend is not needed for the final flip, since it would be an initial particle anyway, we need to know whether we have t- or u-channel.]

⟨*PHS trees: public*⟩+≡

```
public :: phs_tree_flip_t_to_s_channel
```

⟨*PHS trees: procedures*⟩+≡

```
subroutine phs_tree_flip_t_to_s_channel (tree)
  type(phs_tree_t), intent(inout) :: tree
  integer(TC) :: k, f, m, n, d, s
  if (tree%n_in == 2) then
     FLIP: do k=3, tree%mask-1
        if (.not. tree%branch(k)%set) cycle FLIP
        f = iand(k,tree%mask_in)
        if (f==0 .or. f==k) cycle FLIP
        m = tree%branch(k)%mother
        s = tree%branch(k)%sibling
        if (s==0) call find_orphan(s)
        d = tree%branch(k)%daughter(1)
        n = ior(d,s)
        tree%branch(k)%set = .false.
        tree%branch(n)%set = .true.
        tree%branch(n)%origin = k
        tree%branch(n)%daughter(1) = d; tree%branch(d)%mother  = n
        tree%branch(n)%daughter(2) = s; tree%branch(s)%mother  = n
        tree%branch(n)%has_children = .true.
        tree%branch(d)%sibling = s;  tree%branch(s)%sibling = d
        tree%branch(n)%sibling = f;  tree%branch(f)%sibling = n
        tree%branch(n)%mother      = m
        tree%branch(f)%mother      = m
        if (m/=0) then
           tree%branch(m)%daughter(1) = n
           tree%branch(m)%daughter(2) = f
        end if
        tree%branch(n)%friend = f
        tree%branch(n)%has_friend = .true.
        tree%branch(n)%firstborn = 2
     end do FLIP
  end if
contains
  subroutine find_orphan(s)
    integer(TC) :: s
    do s=1, tree%mask_out
       if (tree%branch(s)%set .and. tree%branch(s)%mother==0) return
    end do
    call phs_tree_write (tree)
    call msg_bug (" Can't flip phase space tree to channel")
  end subroutine find_orphan
end subroutine phs_tree_flip_t_to_s_channel
```

After the tree has been flipped, one may need to determine what has become of a particular t-channel branch. This function gives the bincode of the flipped tree. If the original bincode does not contain IN-bits, we leave it as it is.

⟨*PHS trees: procedures*⟩+≡

```
function tc_flipped (tree, kt) result (ks)
  type(phs_tree_t), intent(in) :: tree
  integer(TC), intent(in) :: kt
  integer(TC) :: ks
  if (iand (kt, tree%mask_in) == 0) then
     ks = kt
  else
     ks = tree%branch(iand (kt, tree%mask_out))%mother
  end if
end function tc_flipped
```

Scan a tree and make sure that the first daughter has always a smaller code than the second one. Furthermore, delete any `friend` entry in the root branch – this branching has the incoming particle direction as axis anyway. Keep track of reordering by updating `inverted_axis`, `inverted_decay` and `firstborn`.

⟨*PHS trees: public*⟩+≡

```
public :: phs_tree_canonicalize
```

⟨*PHS trees: procedures*⟩+≡

```
subroutine phs_tree_canonicalize (tree)
  type(phs_tree_t), intent(inout) :: tree
  integer :: n_out
  integer(TC) :: k_out
  call branch_canonicalize (tree%branch(tree%mask_out))
  n_out = tree%n_externals - tree%n_in
  k_out = tree%mask_out
  if (tree%branch(k_out)%has_friend &
       & .and. tree%branch(k_out)%friend == ibset (0, n_out)) then
     tree%branch(k_out)%inverted_axis = .not.tree%branch(k_out)%inverted_axis
  end if
  tree%branch(k_out)%has_friend = .false.
  tree%branch(k_out)%friend = 0
contains
  recursive subroutine branch_canonicalize (b)
    type(phs_branch_t), intent(inout) :: b
    integer(TC) :: d1, d2
    if (b%has_children) then
       d1 = b%daughter(1)
       d2 = b%daughter(2)
       if (d1 > d2) then
          b%daughter(1) = d2
          b%daughter(2) = d1
          b%inverted_decay = .not.b%inverted_decay
          if (b%firstborn /= 0)  b%firstborn = 3 - b%firstborn
       end if
       call branch_canonicalize (tree%branch(b%daughter(1)))
       call branch_canonicalize (tree%branch(b%daughter(2)))
    end if
  end subroutine branch_canonicalize
```

```
end subroutine phs_tree_canonicalize
```

Mappings

Initialize a mapping for the current tree. This is done while reading from file, so the mapping parameters are read, but applied to the flipped tree. Thus, the size of the array of mappings is given by the number of outgoing particles only.

⟨*PHS trees: public*⟩+≡
```
public :: phs_tree_init_mapping
```

⟨*PHS trees: phs tree: TBP*⟩+≡
```
procedure :: init_mapping => phs_tree_init_mapping
```

⟨*PHS trees: procedures*⟩+≡
```
subroutine phs_tree_init_mapping (tree, k, type, pdg, model)
  class(phs_tree_t), intent(inout) :: tree
  integer(TC), intent(in) :: k
  type(string_t), intent(in) :: type
  integer, intent(in) :: pdg
  class(model_data_t), intent(in), target :: model
  integer(TC) :: kk
  kk = tc_flipped (tree, k)
  call mapping_init (tree%mapping(kk), kk, type, pdg, model)
end subroutine phs_tree_init_mapping
```

Set the physical parameters for the mapping, using a specific parameter set. Also set the mass sum array.

⟨*PHS trees: public*⟩+≡
```
public :: phs_tree_set_mapping_parameters
```

⟨*PHS trees: phs tree: TBP*⟩+≡
```
procedure :: set_mapping_parameters => phs_tree_set_mapping_parameters
```

⟨*PHS trees: procedures*⟩+≡
```
subroutine phs_tree_set_mapping_parameters &
     (tree, mapping_defaults, variable_limits)
  class(phs_tree_t), intent(inout) :: tree
  type(mapping_defaults_t), intent(in) :: mapping_defaults
  logical, intent(in) :: variable_limits
  integer(TC) :: k
  do k = 1, tree%n_branches_out
     call mapping_set_parameters &
          (tree%mapping(k), mapping_defaults, variable_limits)
  end do
end subroutine phs_tree_set_mapping_parameters
```

Return the mapping for the sum of all outgoing particles. This should either be no mapping or a global s-channel mapping.

⟨*PHS trees: public*⟩+≡
```
public :: phs_tree_assign_s_mapping
```

⟨*PHS trees: procedures*⟩+≡

```
subroutine phs_tree_assign_s_mapping (tree, mapping)
  type(phs_tree_t), intent(in) :: tree
  type(mapping_t), intent(out) :: mapping
  mapping = tree%mapping(tree%mask_out)
end subroutine phs_tree_assign_s_mapping
```

Kinematics

Fill the mass sum array, starting from the external particles and working down to the tree root. For each bincode `k` we scan the bits in `k`; if only one is set, we take the physical mass of the corresponding external particle; if more then one is set, we sum up the two masses (which we know have already been set).

⟨*PHS trees: public*⟩+≡

```
public :: phs_tree_set_mass_sum
```

⟨*PHS trees: phs tree: TBP*⟩+≡

```
procedure :: set_mass_sum => phs_tree_set_mass_sum
```

⟨*PHS trees: procedures*⟩+≡

```
subroutine phs_tree_set_mass_sum (tree, flv)
  class(phs_tree_t), intent(inout) :: tree
  type(flavor_t), dimension(:), intent(in) :: flv
  integer(TC) :: k
  integer :: i
  tree%mass_sum = 0
  do k = 1, tree%n_branches_out
     do i = 0, size (flv) - 1
        if (btest(k,i)) then
           if (ibclr(k,i) == 0) then
              tree%mass_sum(k) = flv(i+1)%get_mass ()
           else
              tree%mass_sum(k) = &
                   tree%mass_sum(ibclr(k,i)) + tree%mass_sum(ibset(0,i))
           end if
           exit
        end if
     end do
  end do
end subroutine phs_tree_set_mass_sum
```

Set the effective masses and widths. For each non-resonant branch in a tree, the effective mass is equal to the sum of the effective masses of the children (and analogous for the width). External particles have their real mass and width zero. For resonant branches, we insert mass and width from the corresponding mapping.

This routine has `phs_tree_set_mass_sum` and `phs_tree_set_mapping_parameters` as prerequisites.

⟨*PHS trees: public*⟩+≡

```
public :: phs_tree_set_effective_masses
```

⟨*PHS trees: phs tree: TBP*⟩+≡

```
procedure :: set_effective_masses => phs_tree_set_effective_masses
```

⟨*PHS trees: procedures*⟩+≡

```
subroutine phs_tree_set_effective_masses (tree)
  class(phs_tree_t), intent(inout) :: tree
  tree%effective_mass = 0
  tree%effective_width = 0
  call set_masses_x (tree%mask_out)
contains
  recursive subroutine set_masses_x (k)
    integer(TC), intent(in) :: k
    integer(TC) :: k1, k2
    if (tree%branch(k)%has_children) then
       k1 = tree%branch(k)%daughter(1)
       k2 = tree%branch(k)%daughter(2)
       call set_masses_x (k1)
       call set_masses_x (k2)
       if (mapping_is_s_channel (tree%mapping(k))) then
          tree%effective_mass(k) = mapping_get_mass (tree%mapping(k))
          tree%effective_width(k) = mapping_get_width (tree%mapping(k))
       else
          tree%effective_mass(k) = &
               tree%effective_mass(k1) + tree%effective_mass(k2)
          tree%effective_width(k) = &
               tree%effective_width(k1) + tree%effective_width(k2)
       end if
    else
       tree%effective_mass(k) = tree%mass_sum(k)
    end if
  end subroutine set_masses_x
end subroutine phs_tree_set_effective_masses
```

Define step mappings, recursively, for the decay products of all intermediate resonances. Step mappings account for the fact that a branch may originate from a resonance, which almost replaces the upper limit on the possible invariant mass. The step mapping implements a smooth cutoff that interpolates between the resonance and the real kinematic limit. The mapping width determines the sharpness of the cutoff.

Step mappings are inserted only for branches that are not mapped otherwise.

At each branch, we record the mass that is effectively available for phase space, by taking the previous limit and subtracting the effective mass of the sibling branch. Widths are added, not subtracted.

If we encounter a resonance decay, we discard the previous limit and replace it by the mass and width of the resonance, also subtracting the sibling branch.

Initially, the limit is zero, so it becomes negative at any branch. Only if there is a resonance, the limit becomes positive. Whenever the limit is positive, and the current branch decays, we activate a step mapping for the current branch.

As a result, step mappings are implemented for all internal lines that originate from an intermediate resonance decay.

The flag `variable_limits` applies to the ultimate limit from the available energy, not to the intermediate resonances whose masses are always fixed.

This routine requires `phs_tree_set_effective_masses`

⟨*PHS trees: public*⟩+≡

```
public :: phs_tree_set_step_mappings
```

⟨*PHS trees: procedures*⟩+≡

```
subroutine phs_tree_set_step_mappings (tree, exp_type, variable_limits)
  type(phs_tree_t), intent(inout) :: tree
  logical, intent(in) :: exp_type
  logical, intent(in) :: variable_limits
  type(string_t) :: map_str
  integer(TC) :: k
  if (exp_type) then
     map_str = "step_exp"
  else
     map_str = "step_hyp"
  end if
  k = tree%mask_out
  call set_step_mappings_x (k, 0._default, 0._default)
contains
  recursive subroutine set_step_mappings_x (k, m_limit, w_limit)
    integer(TC), intent(in) :: k
    real(default), intent(in) :: m_limit, w_limit
    integer(TC), dimension(2) :: kk
    real(default), dimension(2) :: m, w
    if (tree%branch(k)%has_children) then
       if (m_limit > 0) then
          if (.not. mapping_is_set (tree%mapping(k))) then
             call mapping_init (tree%mapping(k), k, map_str)
             call mapping_set_step_mapping_parameters (tree%mapping(k), &
                  m_limit, w_limit, &
                  variable_limits)
          end if
       end if
       kk = tree%branch(k)%daughter
       m = tree%effective_mass(kk)
       w = tree%effective_width(kk)
       if (mapping_is_s_channel (tree%mapping(k))) then
          call set_step_mappings_x (kk(1), &
               mapping_get_mass (tree%mapping(k)) - m(2), &
               mapping_get_width (tree%mapping(k)) + w(2))
          call set_step_mappings_x (kk(2), &
               mapping_get_mass (tree%mapping(k)) - m(1), &
               mapping_get_width (tree%mapping(k)) + w(1))
       else if (m_limit > 0) then
          call set_step_mappings_x (kk(1), &
               m_limit - m(2), &
               w_limit + w(2))
          call set_step_mappings_x (kk(2), &
               m_limit - m(1), &
               w_limit + w(1))
       else
          call set_step_mappings_x (kk(1), &
               - m(2), &
               + w(2))
          call set_step_mappings_x (kk(2), &
               - m(1), &
               + w(1))
       end if
```

```
      end if
    end subroutine set_step_mappings_x
  end subroutine phs_tree_set_step_mappings
```

Resonance structure

We identify the resonances within a tree as the set of s-channel mappings. The `resonance_history_t` type serves as the result container.

⟨*PHS trees: phs tree: TBP*⟩+≡

```
  procedure :: extract_resonance_history => phs_tree_extract_resonance_history
```

⟨*PHS trees: procedures*⟩+≡

```
  subroutine phs_tree_extract_resonance_history (tree, res_history)
    class(phs_tree_t), intent(in) :: tree
    type(resonance_history_t), intent(out) :: res_history
    type(resonance_info_t) :: res_info
    integer :: i
    if (allocated (tree%mapping)) then
       do i = 1, size (tree%mapping)
          associate (mapping => tree%mapping(i))
             if (mapping%is_s_channel ()) then
                call res_info%init (mapping%get_bincode (), mapping%get_flv (), &
                     n_out = tree%n_externals - tree%n_in)
                call res_history%add_resonance (res_info)
             end if
          end associate
       end do
    end if
  end subroutine phs_tree_extract_resonance_history
```

Structural comparison

This function allows to check whether one tree is the permutation of another one. The permutation is applied to the second tree in the argument list. We do not make up a temporary permuted tree, but compare the two trees directly. The branches are scanned recursively, where for each daughter we check the friend and the mapping as well. Once a discrepancy is found, the recursion is exited immediately.

⟨*PHS trees: public*⟩+≡

```
  public :: phs_tree_equivalent
```

⟨*PHS trees: procedures*⟩+≡

```
  function phs_tree_equivalent (t1, t2, perm) result (is_equal)
    type(phs_tree_t), intent(in) :: t1, t2
    type(permutation_t), intent(in) :: perm
    logical :: equal, is_equal
    integer(TC) :: k1, k2, mask_in
    k1 = t1%mask_out
    k2 = t2%mask_out
    mask_in = t1%mask_in
    equal = .true.
```

```
    call check (t1%branch(k1), t2%branch(k2), k1, k2)
    is_equal = equal
  contains
    recursive subroutine check (b1, b2, k1, k2)
      type(phs_branch_t), intent(in) :: b1, b2
      integer(TC), intent(in) :: k1, k2
      integer(TC), dimension(2) :: d1, d2, pd2
      integer :: i
      if (.not.b1%has_friend .and. .not.b2%has_friend) then
         equal = .true.
      else if (b1%has_friend .and. b2%has_friend) then
         equal = (b1%friend == tc_permute (b2%friend, perm, mask_in))
      end if
      if (equal) then
         if (b1%has_children .and. b2%has_children) then
            d1 = b1%daughter
            d2 = b2%daughter
            do i=1, 2
               pd2(i) = tc_permute (d2(i), perm, mask_in)
            end do
            if (d1(1)==pd2(1) .and. d1(2)==pd2(2)) then
               equal = (b1%firstborn == b2%firstborn)
               if (equal) call check &
                    &     (t1%branch(d1(1)), t2%branch(d2(1)), d1(1), d2(1))
               if (equal) call check &
                    &     (t1%branch(d1(2)), t2%branch(d2(2)), d1(2), d2(2))
            else if (d1(1)==pd2(2) .and. d1(2)==pd2(1)) then
               equal = ( (b1%firstborn == 0 .and. b2%firstborn == 0) &
                    &   .or. (b1%firstborn == 3 - b2%firstborn) )
               if (equal) call check &
                    &     (t1%branch(d1(1)), t2%branch(d2(2)), d1(1), d2(2))
               if (equal) call check &
                    &     (t1%branch(d1(2)), t2%branch(d2(1)), d1(2), d2(1))
            else
               equal = .false.
            end if
         end if
      end if
      if (equal) then
         equal = (t1%mapping(k1) == t2%mapping(k2))
      end if
    end subroutine check
  end function phs_tree_equivalent
```

Scan two decay trees and determine the correspondence of mass variables, i.e., the permutation that transfers the ordered list of mass variables belonging to the second tree into the first one. Mass variables are assigned beginning from branches and ending at the root.

⟨*PHS trees: public*⟩+≡

```
  public :: phs_tree_find_msq_permutation
```

⟨*PHS trees: procedures*⟩+≡

```
  subroutine phs_tree_find_msq_permutation (tree1, tree2, perm2, msq_perm)
    type(phs_tree_t), intent(in) :: tree1, tree2
```

```
    type(permutation_t), intent(in) :: perm2
    type(permutation_t), intent(out) :: msq_perm
    type(permutation_t) :: perm1
    integer(TC) :: mask_in, root
    integer(TC), dimension(:), allocatable :: index1, index2
    integer :: i
    allocate (index1 (tree1%n_msq), index2 (tree2%n_msq))
    call permutation_init (perm1, permutation_size (perm2))
    mask_in = tree1%mask_in
    root = tree1%mask_out
    i = 0
    call tree_scan (tree1, root, perm1, index1)
    i = 0
    call tree_scan (tree2, root, perm2, index2)
    call permutation_find (msq_perm, index1, index2)
  contains
    recursive subroutine tree_scan (tree, k, perm, index)
      type(phs_tree_t), intent(in) :: tree
      integer(TC), intent(in) :: k
      type(permutation_t), intent(in) :: perm
      integer, dimension(:), intent(inout) :: index
      if (tree%branch(k)%has_children) then
         call tree_scan (tree, tree%branch(k)%daughter(1), perm, index)
         call tree_scan (tree, tree%branch(k)%daughter(2), perm, index)
         i = i + 1
         if (i <= size (index))  index(i) = tc_permute (k, perm, mask_in)
      end if
    end subroutine tree_scan
  end subroutine phs_tree_find_msq_permutation
```

⟨*PHS trees: public*⟩+≡

```
  public :: phs_tree_find_angle_permutation
```

⟨*PHS trees: procedures*⟩+≡

```
  subroutine phs_tree_find_angle_permutation &
       (tree1, tree2, perm2, angle_perm, sig2)
    type(phs_tree_t), intent(in) :: tree1, tree2
    type(permutation_t), intent(in) :: perm2
    type(permutation_t), intent(out) :: angle_perm
    logical, dimension(:), allocatable, intent(out) :: sig2
    type(permutation_t) :: perm1
    integer(TC) :: mask_in, root
    integer(TC), dimension(:), allocatable :: index1, index2
    logical, dimension(:), allocatable :: sig1
    integer :: i
    allocate (index1 (tree1%n_angles), index2 (tree2%n_angles))
    allocate (sig1 (tree1%n_angles), sig2 (tree2%n_angles))
    call permutation_init (perm1, permutation_size (perm2))
    mask_in = tree1%mask_in
    root = tree1%mask_out
    i = 0
    call tree_scan (tree1, root, perm1, index1, sig1)
    i = 0
    call tree_scan (tree2, root, perm2, index2, sig2)
```

```
  call permutation_find (angle_perm, index1, index2)
contains
  recursive subroutine tree_scan (tree, k, perm, index, sig)
    type(phs_tree_t), intent(in) :: tree
    integer(TC), intent(in) :: k
    type(permutation_t), intent(in) :: perm
    integer, dimension(:), intent(inout) :: index
    logical, dimension(:), intent(inout) :: sig
    integer(TC) :: k1, k2, kp
    logical :: s
    if (tree%branch(k)%has_children) then
       k1 = tree%branch(k)%daughter(1)
       k2 = tree%branch(k)%daughter(2)
       s = (tc_permute(k1, perm, mask_in) < tc_permute(k2, perm, mask_in))
       kp = tc_permute (k, perm, mask_in)
       i = i + 1
       index(i) = kp
       sig(i) = s
       i = i + 1
       index(i) = - kp
       sig(i) = s
       call tree_scan (tree, k1, perm, index, sig)
       call tree_scan (tree, k2, perm, index, sig)
    end if
  end subroutine tree_scan
end subroutine phs_tree_find_angle_permutation
```

18.5.4 Phase-space evaluation

Phase-space volume

We compute the phase-space volume recursively, following the same path as for computing other kinematical variables. However, the volume depends just on $\sqrt{\hat{s}}$, not on the momentum configuration.

Note: counting branches, we may replace this by a simple formula.

⟨*PHS trees: public*⟩+≡

```
public :: phs_tree_compute_volume
```

⟨*PHS trees: procedures*⟩+≡

```
subroutine phs_tree_compute_volume (tree, sqrts, volume)
  type(phs_tree_t), intent(in) :: tree
  real(default), intent(in) :: sqrts
  real(default), intent(out) :: volume
  integer(TC) :: k
  k  = tree%mask_out
  if (tree%branch(k)%has_children) then
     call compute_volume_x (tree%branch(k), k, volume, .true.)
  else
     volume = 1
  end if
contains
  recursive subroutine compute_volume_x (b, k, volume, initial)
    type(phs_branch_t), intent(in) :: b
```

```
    integer(TC), intent(in) :: k
    real(default), intent(out) :: volume
    logical, intent(in) :: initial
    integer(TC) :: k1, k2
    real(default) :: v1, v2
    k1 = b%daughter(1);  k2 = b%daughter(2)
    if (tree%branch(k1)%has_children) then
       call compute_volume_x (tree%branch(k1), k1, v1, .false.)
    else
       v1 = 1
    end if
    if (tree%branch(k2)%has_children) then
       call compute_volume_x (tree%branch(k2), k2, v2, .false.)
    else
       v2 = 1
    end if
    if (initial) then
       volume = v1 * v2 / (4 * twopi5)
    else
       volume = v1 * v2 * sqrts**2 / (4 * twopi2)
    end if
  end subroutine compute_volume_x
end subroutine phs_tree_compute_volume
```

Determine momenta

This is done in two steps: First the masses are determined. This step may fail, in which case `ok` is set to false. If successful, we generate angles and the actual momenta. The array `decay_p` serves for transferring the individual three-momenta of the daughter particles in their mother rest frame from the mass generation to the momentum generation step.

⟨*PHS trees: public*⟩+≡

```
public :: phs_tree_compute_momenta_from_x
```

⟨*PHS trees: procedures*⟩+≡

```
subroutine phs_tree_compute_momenta_from_x &
     (tree, prt, factor, volume, sqrts, x, ok)
  type(phs_tree_t), intent(inout) :: tree
  type(phs_prt_t), dimension(:), intent(inout) :: prt
  real(default), intent(out) :: factor, volume
  real(default), intent(in) :: sqrts
  real(default), dimension(:), intent(in) :: x
  logical, intent(out) :: ok
  real(default), dimension(tree%mask_out) :: decay_p
  integer :: n1, n2
  integer :: n_out
  if (tree%real_phsp) then
    n_out = tree%n_externals - tree%n_in - 1
    n1 = max (n_out-2, 0)
    n2 = n1 + max (2*n_out, 0)
  else
    n1 = tree%n_msq
    n2 = n1 + tree%n_angles
```

```
    end if
    call phs_tree_set_msq &
         (tree, prt, factor, volume, decay_p, sqrts, x(1:n1), ok)
    if (ok) call phs_tree_set_angles &
         (tree, prt, factor, decay_p, sqrts, x(n1+1:n2))
  end subroutine phs_tree_compute_momenta_from_x
```

Mass generation is done recursively. The `ok` flag causes the filled tree to be discarded if set to `.false.`. This happens if a three-momentum turns out to be imaginary, indicating impossible kinematics. The index `ix` tells us how far we have used up the input array `x`.

⟨*PHS trees: procedures*⟩+≡

```
  subroutine phs_tree_set_msq &
       (tree, prt, factor, volume, decay_p, sqrts, x, ok)
    type(phs_tree_t), intent(inout) :: tree
    type(phs_prt_t), dimension(:), intent(inout) :: prt
    real(default), intent(out) :: factor, volume
    real(default), dimension(:), intent(out) :: decay_p
    real(default), intent(in) :: sqrts
    real(default), dimension(:), intent(in) :: x
    logical, intent(out) :: ok
    integer :: ix
    integer(TC) :: k
    real(default) :: m_tot
    ok =.true.
    ix = 1
    k  = tree%mask_out
    m_tot = tree%mass_sum(k)
    decay_p(k) = 0.
    if (m_tot < sqrts .or. k == 1) then
       if (tree%branch(k)%has_children) then
          call set_msq_x (tree%branch(k), k, factor, volume, .true.)
       else
          factor = 1
          volume = 1
       end if
    else
       ok = .false.
    end if
  contains
    recursive subroutine set_msq_x (b, k, factor, volume, initial)
      type(phs_branch_t), intent(in) :: b
      integer(TC), intent(in) :: k
      real(default), intent(out) :: factor, volume
      logical, intent(in) :: initial
      real(default) :: msq, m, m_min, m_max, m1, m2, msq1, msq2, lda, rlda
      integer(TC) :: k1, k2
      real(default) :: f1, f2, v1, v2
      k1 = b%daughter(1);  k2 = b%daughter(2)
      if (tree%branch(k1)%has_children) then
         call set_msq_x (tree%branch(k1), k1, f1, v1, .false.)
         if (.not.ok) return
      else
```

```
      f1 = 1;  v1 = 1
   end if
   if (tree%branch(k2)%has_children) then
      call set_msq_x (tree%branch(k2), k2, f2, v2, .false.)
      if (.not.ok) return
   else
      f2 = 1;  v2 = 1
   end if
   m_min = tree%mass_sum(k)
   if (initial) then
      msq = sqrts**2
      m = sqrts
      m_max = sqrts
      factor = f1 * f2
      volume = v1 * v2 / (4 * twopi5)
   else
      m_max = sqrts - m_tot + m_min
      call mapping_compute_msq_from_x &
           (tree%mapping(k), sqrts**2, m_min**2, m_max**2, msq, factor, &
            x(ix)); ix = ix + 1
      if (msq >= 0) then
         m = sqrt (msq)
         factor = f1 * f2 * factor
         volume = v1 * v2 * sqrts**2 / (4 * twopi2)
         call phs_prt_set_msq (prt(k), msq)
         call phs_prt_set_defined (prt(k))
      else
         ok = .false.
      end if
   end if
   if (ok) then
      msq1 = phs_prt_get_msq (prt(k1));  m1 = sqrt (msq1)
      msq2 = phs_prt_get_msq (prt(k2));  m2 = sqrt (msq2)
      lda = lambda (msq, msq1, msq2)
      if (lda > 0 .and. m > m1 + m2 .and. m <= m_max) then
         rlda = sqrt (lda)
         decay_p(k1) = rlda / (2*m)
         decay_p(k2) = - decay_p(k1)
         factor = rlda / msq * factor
      else
         ok = .false.
      end if
   end if
 end subroutine set_msq_x

end subroutine phs_tree_set_msq
```

The heart of phase space generation: Now we have the invariant masses, let us generate angles. At each branch, we take a Lorentz transformation and augment it by a boost to the current particle rest frame, and by rotations ϕ and θ around the z and y axis, respectively. This transformation is passed down to the daughter particles, if present.

⟨*PHS trees: procedures*⟩+≡

```
subroutine phs_tree_set_angles (tree, prt, factor, decay_p, sqrts, x)
  type(phs_tree_t), intent(inout) :: tree
  type(phs_prt_t), dimension(:), intent(inout) :: prt
  real(default), intent(inout) :: factor
  real(default), dimension(:), intent(in) :: decay_p
  real(default), intent(in) :: sqrts
  real(default), dimension(:), intent(in) :: x
  integer :: ix
  integer(TC) :: k
  ix = 1
  k  = tree%mask_out
  call set_angles_x (tree%branch(k), k)
contains
  recursive subroutine set_angles_x (b, k, L0)
    type(phs_branch_t), intent(in) :: b
    integer(TC), intent(in) :: k
    type(lorentz_transformation_t), intent(in), optional :: L0
    real(default) :: m, msq, ct, st, phi, f, E, p, bg
    type(lorentz_transformation_t) :: L, LL
    integer(TC) :: k1, k2
    type(vector3_t) :: axis
    p = decay_p(k)
    msq = phs_prt_get_msq (prt(k));  m = sqrt (msq)
    E = sqrt (msq + p**2)
    if (present (L0)) then
       call phs_prt_set_momentum (prt(k), L0 * vector4_moving (E,p,3))
    else
       call phs_prt_set_momentum (prt(k), vector4_moving (E,p,3))
    end if
    call phs_prt_set_defined (prt(k))
    if (b%has_children) then
       k1 = b%daughter(1)
       k2 = b%daughter(2)
       if (m > 0) then
          bg = p / m
       else
          bg = 0
       end if
       phi = x(ix) * twopi;  ix = ix + 1
       call mapping_compute_ct_from_x &
            (tree%mapping(k), sqrts**2, ct, st, f, x(ix));  ix = ix + 1
       factor = factor * f
       if (.not. b%has_friend) then
          L = LT_compose_r2_r3_b3 (ct, st, cos(phi), sin(phi), bg)
          !!! The function above is equivalent to:
          ! L = boost (bg,3) * rotation (phi,3) * rotation (ct,st,2)
       else
          LL = boost (-bg,3);  if (present (L0))  LL = LL * inverse(L0)
          axis = space_part ( &
               LL * phs_prt_get_momentum (prt(tree%branch(k)%friend)) )
          L = boost(bg,3) * rotation_to_2nd (vector3_canonical(3), axis) &
               * LT_compose_r2_r3_b3 (ct, st, cos(phi), sin(phi), 0._default)
       end if
       if (present (L0))  L = L0 * L
```

```
        call set_angles_x (tree%branch(k1), k1, L)
        call set_angles_x (tree%branch(k2), k2, L)
      end if
    end subroutine set_angles_x

  end subroutine phs_tree_set_angles
```

Recover random numbers

For the other channels we want to compute the random numbers that would have generated the momenta that we already know.

⟨*PHS trees: public*⟩+≡

```
  public :: phs_tree_compute_x_from_momenta
```

⟨*PHS trees: procedures*⟩+≡

```
  subroutine phs_tree_compute_x_from_momenta (tree, prt, factor, sqrts, x)
    type(phs_tree_t), intent(inout) :: tree
    type(phs_prt_t), dimension(:), intent(in) :: prt
    real(default), intent(out) :: factor
    real(default), intent(in) :: sqrts
    real(default), dimension(:), intent(inout) :: x
    real(default), dimension(tree%mask_out) :: decay_p
    integer :: n1, n2
    n1 = tree%n_msq
    n2 = n1 + tree%n_angles
    call phs_tree_get_msq &
         (tree, prt, factor, decay_p, sqrts, x(1:n1))
    call phs_tree_get_angles &
         (tree, prt, factor, decay_p, sqrts, x(n1+1:n2))
  end subroutine phs_tree_compute_x_from_momenta
```

The inverse operation follows exactly the same steps. The tree is `inout` because it contains mappings whose parameters can be reset when the mapping is applied.

⟨*PHS trees: procedures*⟩+≡

```
  subroutine phs_tree_get_msq (tree, prt, factor, decay_p, sqrts, x)
    type(phs_tree_t), intent(inout) :: tree
    type(phs_prt_t), dimension(:), intent(in) :: prt
    real(default), intent(out) :: factor
    real(default), dimension(:), intent(out) :: decay_p
    real(default), intent(in) :: sqrts
    real(default), dimension(:), intent(inout) :: x
    integer :: ix
    integer(TC) :: k
    real(default) :: m_tot
    ix = 1
    k  = tree%mask_out
    m_tot = tree%mass_sum(k)
    decay_p(k) = 0.
    if (tree%branch(k)%has_children) then
       call get_msq_x (tree%branch(k), k, factor, .true.)
    else
```

```
      factor = 1
    end if
  contains
    recursive subroutine get_msq_x (b, k, factor, initial)
      type(phs_branch_t), intent(in) :: b
      integer(TC), intent(in) :: k
      real(default), intent(out) :: factor
      logical, intent(in) :: initial
      real(default) :: msq, m, m_min, m_max, msq1, msq2, lda, rlda
      integer(TC) :: k1, k2
      real(default) :: f1, f2
      k1 = b%daughter(1);  k2 = b%daughter(2)
      if (tree%branch(k1)%has_children) then
         call get_msq_x (tree%branch(k1), k1, f1, .false.)
      else
         f1 = 1
      end if
      if (tree%branch(k2)%has_children) then
         call get_msq_x (tree%branch(k2), k2, f2, .false.)
      else
         f2 = 1
      end if
      m_min = tree%mass_sum(k)
      m_max = sqrts - m_tot + m_min
      msq = phs_prt_get_msq (prt(k));  m = sqrt (msq)
      if (initial) then
         factor = f1 * f2
      else
         call mapping_compute_x_from_msq &
              (tree%mapping(k), sqrts**2, m_min**2, m_max**2, msq, factor, &
               x(ix));  ix = ix + 1
         factor = f1 * f2 * factor
      end if
      msq1 = phs_prt_get_msq (prt(k1))
      msq2 = phs_prt_get_msq (prt(k2))
      lda = lambda (msq, msq1, msq2)
      if (lda > 0) then
         rlda = sqrt (lda)
         decay_p(k1) = rlda / (2 * m)
         decay_p(k2) = - decay_p(k1)
         factor = rlda / msq * factor
      else
         decay_p(k1) = 0
         decay_p(k2) = 0
         factor = 0
      end if
    end subroutine get_msq_x

  end subroutine phs_tree_get_msq
```

This subroutine is the most time-critical part of the whole program. Therefore, we do not exactly parallel the angle generation routine above but make sure that things get evaluated only if they are really needed, at the expense of

readability. Particularly important is to have as few multiplications of Lorentz transformations as possible.

⟨*PHS trees: procedures*⟩+≡

```
subroutine phs_tree_get_angles (tree, prt, factor, decay_p, sqrts, x)
  type(phs_tree_t), intent(inout) :: tree
  type(phs_prt_t), dimension(:), intent(in) :: prt
  real(default), intent(inout) :: factor
  real(default), dimension(:), intent(in) :: decay_p
  real(default), intent(in) :: sqrts
  real(default), dimension(:), intent(out) :: x
  integer :: ix
  integer(TC) :: k
  ix = 1
  k  = tree%mask_out
  if (tree%branch(k)%has_children) then
     call get_angles_x (tree%branch(k), k)
  end if
contains
  recursive subroutine get_angles_x (b, k, ct0, st0, phi0, L0)
    type(phs_branch_t), intent(in) :: b
    integer(TC), intent(in) :: k
    real(default), intent(in), optional :: ct0, st0, phi0
    type(lorentz_transformation_t), intent(in), optional :: L0
    real(default) :: cp0, sp0, m, msq, ct, st, phi, bg, f
    type(lorentz_transformation_t) :: L, LL
    type(vector4_t) :: p1, pf
    type(vector3_t) :: n, axis
    integer(TC) :: k1, k2, kf
    logical :: has_friend, need_L
    k1 = b%daughter(1)
    k2 = b%daughter(2)
    kf = b%friend
    has_friend = b%has_friend
    if (present(L0)) then
       p1 = L0 * phs_prt_get_momentum (prt(k1))
       if (has_friend)  pf = L0 * phs_prt_get_momentum (prt(kf))
    else
       p1 = phs_prt_get_momentum (prt(k1))
       if (has_friend)  pf = phs_prt_get_momentum (prt(kf))
    end if
    if (present(phi0)) then
       cp0 = cos (phi0)
       sp0 = sin (phi0)
    end if
    msq = phs_prt_get_msq (prt(k));  m = sqrt (msq)
    if (m > 0) then
       bg = decay_p(k) / m
    else
       bg = 0
    end if
    if (has_friend) then
       if (present (phi0)) then
          axis = axis_from_p_r3_r2_b3 (pf, cp0, -sp0, ct0, -st0, -bg)
          LL = rotation_to_2nd (axis, vector3_canonical (3)) &
```

```
            * LT_compose_r3_r2_b3 (cp0, -sp0, ct0, -st0, -bg)
      else
         axis = axis_from_p_b3 (pf, -bg)
         LL = rotation_to_2nd (axis, vector3_canonical(3))
         if (.not. vanishes (bg))  LL = LL * boost(-bg, 3)
      end if
      n = space_part (LL * p1)
   else if (present (phi0)) then
      n = axis_from_p_r3_r2_b3 (p1, cp0, -sp0, ct0, -st0, -bg)
   else
      n = axis_from_p_b3 (p1, -bg)
   end if
   phi = azimuthal_angle (n)
   x(ix) = phi / twopi;  ix = ix + 1
   ct = polar_angle_ct (n)
   st = sqrt (1 - ct**2)
   call mapping_compute_x_from_ct (tree%mapping(k), sqrts**2, ct, f, &
        x(ix));  ix = ix + 1
   factor = factor * f
   if (tree%branch(k1)%has_children .or. tree%branch(k2)%has_children) then
      need_L = .true.
      if (has_friend) then
         if (present (L0)) then
            L = LL * L0
         else
            L = LL
         end if
      else if (present (L0)) then
         L = LT_compose_r3_r2_b3 (cp0, -sp0, ct0, -st0, -bg) * L0
      else if (present (phi0)) then
         L = LT_compose_r3_r2_b3 (cp0, -sp0, ct0, -st0, -bg)
      else if (bg /= 0) then
         L = boost(-bg, 3)
      else
         need_L = .false.
      end if
      if (need_L) then
         if (tree%branch(k1)%has_children) &
              call get_angles_x (tree%branch(k1), k1, ct, st, phi, L)
         if (tree%branch(k2)%has_children) &
              call get_angles_x (tree%branch(k2), k2, ct, st, phi, L)
      else
         if (tree%branch(k1)%has_children) &
              call get_angles_x (tree%branch(k1), k1, ct, st, phi)
         if (tree%branch(k2)%has_children) &
              call get_angles_x (tree%branch(k2), k2, ct, st, phi)
      end if
   end if
  end subroutine get_angles_x
end subroutine phs_tree_get_angles
```

Auxiliary stuff

This calculates all momenta that are not yet known by summing up daughter particle momenta. The external particles must be known. Only composite particles not yet known are calculated.

⟨*PHS trees: public*⟩+≡
```
public :: phs_tree_combine_particles
```

⟨*PHS trees: procedures*⟩+≡
```
subroutine phs_tree_combine_particles (tree, prt)
  type(phs_tree_t), intent(in) :: tree
  type(phs_prt_t), dimension(:), intent(inout) :: prt
  call combine_particles_x (tree%mask_out)
contains
  recursive subroutine combine_particles_x (k)
    integer(TC), intent(in) :: k
    integer :: k1, k2
    if (tree%branch(k)%has_children) then
       k1 = tree%branch(k)%daughter(1);  k2 = tree%branch(k)%daughter(2)
       call combine_particles_x (k1)
       call combine_particles_x (k2)
       if (.not. prt(k)%defined) then
          call phs_prt_combine (prt(k), prt(k1), prt(k2))
       end if
    end if
  end subroutine combine_particles_x
end subroutine phs_tree_combine_particles
```

The previous routine is to be evaluated at runtime. Instead of scanning trees, we can as well set up a multiplication table. This is generated here. Note that the table is `intent(out)`.

⟨*PHS trees: public*⟩+≡
```
public :: phs_tree_setup_prt_combinations
```

⟨*PHS trees: procedures*⟩+≡
```
subroutine phs_tree_setup_prt_combinations (tree, comb)
  type(phs_tree_t), intent(in) :: tree
  integer, dimension(:,:), intent(out) :: comb
  comb = 0
  call setup_prt_combinations_x (tree%mask_out)
contains
  recursive subroutine setup_prt_combinations_x (k)
    integer(TC), intent(in) :: k
    integer, dimension(2) :: kk
    if (tree%branch(k)%has_children) then
       kk = tree%branch(k)%daughter
       call setup_prt_combinations_x (kk(1))
       call setup_prt_combinations_x (kk(2))
       comb(:,k) = kk
    end if
  end subroutine setup_prt_combinations_x
end subroutine phs_tree_setup_prt_combinations
```

⟨*PHS trees: public*⟩+≡
```
public :: phs_tree_reshuffle_mappings
```
⟨*PHS trees: procedures*⟩+≡
```
subroutine phs_tree_reshuffle_mappings (tree)
 type(phs_tree_t), intent(inout) :: tree
 integer(TC) :: k0, k_old, k_new, k2
 integer :: i
 type(mapping_t) :: mapping_tmp
 real(default) :: mass_tmp
 do i = 1, size (tree%momentum_link)
   if (i /= tree%momentum_link (i)) then
     k_old = 2**(i-tree%n_in-1)
     k_new = 2**(tree%momentum_link(i)-tree%n_in-1)
     k0 = tree%branch(k_old)%mother
     k2 = k_new + tree%branch(k_old)%sibling
     mapping_tmp = tree%mapping(k0)
     mass_tmp = tree%mass_sum(k0)
     tree%mapping(k0) = tree%mapping(k2)
     tree%mapping(k2) = mapping_tmp
     tree%mass_sum(k0) = tree%mass_sum(k2)
     tree%mass_sum(k2) = mass_tmp
   end if
 end do
end subroutine phs_tree_reshuffle_mappings
```

⟨*PHS trees: public*⟩+≡
```
public :: phs_tree_set_momentum_links
```
⟨*PHS trees: procedures*⟩+≡
```
subroutine phs_tree_set_momentum_links (tree, list)
  type(phs_tree_t), intent(inout) :: tree
  integer, dimension(:), allocatable :: list
  tree%momentum_link = list
end subroutine phs_tree_set_momentum_links
```

18.5.5 Unit tests

Test module, followed by the corresponding implementation module.

⟨phs_trees_ut.f90⟩≡
```
⟨File header⟩

module phs_trees_ut
  use unit_tests
  use phs_trees_uti

⟨Standard module head⟩

⟨PHS trees: public test⟩

contains

⟨PHS trees: test driver⟩
```

```
  end module phs_trees_ut
```

⟨phs_trees_uti.f90⟩≡

```
  ⟨File header⟩

  module phs_trees_uti

  !!!⟨Use kinds⟩
    use kinds, only: TC
  ⟨Use strings⟩
    use flavors, only: flavor_t
    use model_data, only: model_data_t

    use resonances, only: resonance_history_t
    use mappings, only: mapping_defaults_t

    use phs_trees

  ⟨Standard module head⟩

  ⟨PHS trees: test declarations⟩

  contains

  ⟨PHS trees: tests⟩

  end module phs_trees_uti
```

API: driver for the unit tests below.

⟨PHS trees: public test⟩≡

```
    public :: phs_trees_test
```

⟨PHS trees: test driver⟩≡

```
    subroutine phs_trees_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨PHS trees: execute tests⟩
    end subroutine phs_trees_test
```

Create a simple $2 \to 3$ PHS tree and display it.

⟨PHS trees: execute tests⟩≡

```
    call test (phs_tree_1, "phs_tree_1", &
         "check phs tree setup", &
         u, results)
```

⟨PHS trees: test declarations⟩≡

```
    public :: phs_tree_1
```

⟨PHS trees: tests⟩≡

```
    subroutine phs_tree_1 (u)
      integer, intent(in) :: u
      type(phs_tree_t) :: tree
      type(model_data_t), target :: model
      type(flavor_t), dimension(5) :: flv
      integer :: i
```

```
    write (u, "(A)")  "* Test output: phs_tree_1"
    write (u, "(A)")  "*   Purpose: test PHS tree routines"
    write (u, "(A)")

    write (u, "(A)")  "* Read model file"

    call model%init_sm_test ()

    write (u, "(A)")
    write (u, "(A)")  "* Set up flavors"
    write (u, "(A)")

    call flv%init ([1, -2, 24, 5, -5], model)
    do i = 1, 5
       write (u, "(1x)", advance="no")
       call flv(i)%write (u)
    end do
    write (u, *)

    write (u, "(A)")
    write (u, "(A)")  "* Create tree"
    write (u, "(A)")

    call tree%init (2, 3, 0, 0)
    call tree%from_array ([integer(TC) :: 1, 2, 3, 4, 7, 8, 16])
    call tree%set_mass_sum (flv)
    call tree%set_effective_masses ()

    call tree%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call tree%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_tree_1"

  end subroutine phs_tree_1
```

The analogous tree with resonance (s-channel) mappings.

⟨*PHS trees: execute tests*⟩+≡

```
  call test (phs_tree_2, "phs_tree_2", &
       "check phs tree with resonances", &
       u, results)
```

⟨*PHS trees: test declarations*⟩+≡

```
  public :: phs_tree_2
```

⟨*PHS trees: tests*⟩+≡

```
  subroutine phs_tree_2 (u)
    integer, intent(in) :: u
    type(phs_tree_t) :: tree
```

```
type(model_data_t), target :: model
type(mapping_defaults_t) :: mapping_defaults
type(flavor_t), dimension(5) :: flv
type(resonance_history_t) :: res_history
integer :: i

write (u, "(A)")  "* Test output: phs_tree_2"
write (u, "(A)")  "*   Purpose: test PHS tree with resonances"
write (u, "(A)")

write (u, "(A)")  "* Read model file"

call model%init_sm_test ()

write (u, "(A)")
write (u, "(A)")  "* Set up flavors"
write (u, "(A)")

call flv%init ([1, -2, 24, 5, -5], model)
do i = 1, 5
   write (u, "(1x)", advance="no")
   call flv(i)%write (u)
end do
write (u, *)

write (u, "(A)")
write (u, "(A)")  "* Create tree with mappings"
write (u, "(A)")

call tree%init (2, 3, 0, 0)
call tree%from_array ([integer(TC) :: 1, 2, 3, 4, 7, 8, 16])
call tree%set_mass_sum (flv)

call tree%init_mapping (3_TC, var_str ("s_channel"), -24, model)
call tree%init_mapping (7_TC, var_str ("s_channel"), 23, model)

call tree%set_mapping_parameters (mapping_defaults, variable_limits=.false.)
call tree%set_effective_masses ()

call tree%write (u)

write (u, "(A)")
write (u, "(A)")  "* Extract resonances from mappings"
write (u, "(A)")

call tree%extract_resonance_history (res_history)
call res_history%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call tree%final ()
call model%final ()
```

```
  write (u, "(A)")
  write (u, "(A)")  "* Test output end: phs_tree_2"

end subroutine phs_tree_2
```

18.6 The phase-space forest

Simply stated, a phase-space forest is a collection of phase-space trees. More precisely, a `phs_forest` object contains all parameterizations of phase space that `WHIZARD` will use for a single hard process, prepared in the form of `phs_tree` objects. This is suitable for evaluation by the `VAMP` integration package: each parameterization (tree) is a valid channel in the multi-channel adaptive integration, and each variable in a tree corresponds to an integration dimension, defined by an appropriate mapping of the $(0, 1)$ interval to the allowed range of the integration variable.

The trees are grouped in groves. The trees (integration channels) within a grove share a common weight, assuming that they are related by some approximate symmetry.

Trees/channels that are related by an exact symmetry are connected by an array of equivalences; each equivalence object holds the data that relate one channel to another.

The phase-space setup, i.e., the detailed structure of trees and forest, are read from a file. Therefore, this module also contains the syntax definition and the parser needed for interpreting this file.

⟨`phs_forests.f90`⟩≡

```
  ⟨File header⟩

  module phs_forests

  ⟨Use kinds⟩
    use kinds, only: TC
  ⟨Use strings⟩
    use io_units
    use format_defs, only: FMT_19
    use diagnostics
    use lorentz
    use numeric_utils
    use permutations
    use ifiles
    use syntax_rules
    use lexers
    use parser
    use model_data
    use model_data
    use flavors
    use interactions

    use phs_base
    use resonances, only: resonance_history_t
    use resonances, only: resonance_history_set_t
    use mappings
    use phs_trees

  ⟨Standard module head⟩

  ⟨PHS forests: public⟩
```

⟨*PHS forests: types*⟩

⟨*PHS forests: interfaces*⟩

⟨*PHS forests: variables*⟩

```
contains
```

⟨*PHS forests: procedures*⟩

```
end module phs_forests
```

18.6.1 Phase-space setup parameters

This transparent container holds the parameters that the algorithm needs for phase-space setup, with reasonable defaults.

The threshold mass (for considering a particle as effectively massless) is specified separately for s- and t-channel. The default is to treat W and Z bosons as massive in the s-channel, but as massless in the t-channel. The b-quark is treated always massless, the t-quark always massive.

⟨*PHS forests: public*⟩≡
```
public :: phs_parameters_t
```

⟨*PHS forests: types*⟩≡
```
type :: phs_parameters_t
   real(default) :: sqrts = 0
   real(default) :: m_threshold_s = 50._default
   real(default) :: m_threshold_t = 100._default
   integer :: off_shell = 1
   integer :: t_channel = 2
   logical :: keep_nonresonant = .true.
end type phs_parameters_t
```

Write phase-space parameters to file.

⟨*PHS forests: public*⟩+≡
```
public :: phs_parameters_write
```

⟨*PHS forests: procedures*⟩≡
```
subroutine phs_parameters_write (phs_par, unit)
  type(phs_parameters_t), intent(in) :: phs_par
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(3x,A," // FMT_19 // ")") "sqrts         = ", phs_par%sqrts
  write (u, "(3x,A," // FMT_19 // ")") "m_threshold_s = ", phs_par%m_threshold_s
  write (u, "(3x,A," // FMT_19 // ")") "m_threshold_t = ", phs_par%m_threshold_t
  write (u, "(3x,A,I0)") "off_shell = ", phs_par%off_shell
  write (u, "(3x,A,I0)") "t_channel = ", phs_par%t_channel
  write (u, "(3x,A,L1)") "keep_nonresonant = ", phs_par%keep_nonresonant
end subroutine phs_parameters_write
```

Read phase-space parameters from file.

⟨*PHS forests: public*⟩+≡

```
public :: phs_parameters_read
```

⟨*PHS forests: procedures*⟩+≡

```
subroutine phs_parameters_read (phs_par, unit)
  type(phs_parameters_t), intent(out) :: phs_par
  integer, intent(in) :: unit
  character(20) :: dummy
  character :: equals
  read (unit, *)  dummy, equals, phs_par%sqrts
  read (unit, *)  dummy, equals, phs_par%m_threshold_s
  read (unit, *)  dummy, equals, phs_par%m_threshold_t
  read (unit, *)  dummy, equals, phs_par%off_shell
  read (unit, *)  dummy, equals, phs_par%t_channel
  read (unit, *)  dummy, equals, phs_par%keep_nonresonant
end subroutine phs_parameters_read
```

Comparison.

⟨*PHS forests: interfaces*⟩≡

```
interface operator(==)
   module procedure phs_parameters_eq
end interface
interface operator(/=)
   module procedure phs_parameters_ne
end interface
```

⟨*PHS forests: procedures*⟩+≡

```
function phs_parameters_eq (phs_par1, phs_par2) result (equal)
  logical :: equal
  type(phs_parameters_t), intent(in) :: phs_par1, phs_par2
  equal = phs_par1%sqrts == phs_par2%sqrts &
       .and. phs_par1%m_threshold_s == phs_par2%m_threshold_s &
       .and. phs_par1%m_threshold_t == phs_par2%m_threshold_t &
       .and. phs_par1%off_shell == phs_par2%off_shell &
       .and. phs_par1%t_channel == phs_par2%t_channel &
       .and.(phs_par1%keep_nonresonant .eqv. phs_par2%keep_nonresonant)
end function phs_parameters_eq

function phs_parameters_ne (phs_par1, phs_par2) result (ne)
  logical :: ne
  type(phs_parameters_t), intent(in) :: phs_par1, phs_par2
  ne = phs_par1%sqrts /= phs_par2%sqrts &
       .or. phs_par1%m_threshold_s /= phs_par2%m_threshold_s &
       .or. phs_par1%m_threshold_t /= phs_par2%m_threshold_t &
       .or. phs_par1%off_shell /= phs_par2%off_shell &
       .or. phs_par1%t_channel /= phs_par2%t_channel &
       .or.(phs_par1%keep_nonresonant .neqv. phs_par2%keep_nonresonant)
end function phs_parameters_ne
```

18.6.2 Equivalences

This type holds information about equivalences between phase-space trees. We make a linked list, where each node contains the two trees which are equivalent and the corresponding permutation of external particles. Two more arrays are to be filled: The permutation of mass variables and the permutation of angular variables, where the signature indicates a necessary exchange of daughter branches.

⟨*PHS forests: types*⟩+≡

```
type :: equivalence_t
   private
   integer :: left, right
   type(permutation_t) :: perm
   type(permutation_t) :: msq_perm, angle_perm
   logical, dimension(:), allocatable :: angle_sig
   type(equivalence_t), pointer :: next => null ()
end type equivalence_t
```

⟨*PHS forests: types*⟩+≡

```
type :: equivalence_list_t
   private
   integer :: length = 0
   type(equivalence_t), pointer :: first => null ()
   type(equivalence_t), pointer :: last => null ()
end type equivalence_list_t
```

Append an equivalence to the list

⟨*PHS forests: procedures*⟩+≡

```
subroutine equivalence_list_add (eql, left, right, perm)
  type(equivalence_list_t), intent(inout) :: eql
  integer, intent(in) :: left, right
  type(permutation_t), intent(in) :: perm
  type(equivalence_t), pointer :: eq
  allocate (eq)
  eq%left = left
  eq%right = right
  eq%perm = perm
  if (associated (eql%last)) then
     eql%last%next => eq
  else
     eql%first => eq
  end if
  eql%last => eq
  eql%length = eql%length + 1
end subroutine equivalence_list_add
```

Delete the list contents. Has to be pure because it is called from an elemental subroutine.

⟨*PHS forests: procedures*⟩+≡

```
pure subroutine equivalence_list_final (eql)
  type(equivalence_list_t), intent(inout) :: eql
  type(equivalence_t), pointer :: eq
```

```
    do while (associated (eql%first))
       eq => eql%first
       eql%first => eql%first%next
       deallocate (eq)
    end do
    eql%last => null ()
    eql%length = 0
  end subroutine equivalence_list_final
```

Make a deep copy of the equivalence list. This allows for deep copies of groves and forests.

⟨*PHS forests: interfaces*⟩+≡

```
  interface assignment(=)
     module procedure equivalence_list_assign
  end interface
```

⟨*PHS forests: procedures*⟩+≡

```
  subroutine equivalence_list_assign (eql_out, eql_in)
    type(equivalence_list_t), intent(out) :: eql_out
    type(equivalence_list_t), intent(in) :: eql_in
    type(equivalence_t), pointer :: eq, eq_copy
    eq => eql_in%first
    do while (associated (eq))
       allocate (eq_copy)
       eq_copy = eq
       eq_copy%next => null ()
       if (associated (eql_out%first)) then
          eql_out%last%next => eq_copy
       else
          eql_out%first => eq_copy
       end if
       eql_out%last => eq_copy
       eq => eq%next
    end do
  end subroutine equivalence_list_assign
```

The number of list entries

⟨*PHS forests: procedures*⟩+≡

```
  elemental function equivalence_list_length (eql) result (length)
    integer :: length
    type(equivalence_list_t), intent(in) :: eql
    length = eql%length
  end function equivalence_list_length
```

Recursively write the equivalences list

⟨*PHS forests: procedures*⟩+≡

```
  subroutine equivalence_list_write (eql, unit)
    type(equivalence_list_t), intent(in) :: eql
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    if (associated (eql%first)) then
```

```
      call equivalence_write_rec (eql%first, u)
   else
      write (u, *) " [empty]"
   end if
contains
   recursive subroutine equivalence_write_rec (eq, u)
      type(equivalence_t), intent(in) :: eq
      integer, intent(in) :: u
      integer :: i
      write (u, "(3x,A,1x,I0,1x,I0,2x,A)", advance="no") &
           "Equivalence:", eq%left, eq%right, "Final state permutation:"
      call permutation_write (eq%perm, u)
      write (u, "(1x,12x,1x,A,1x)", advance="no") &
           "       msq permutation:  "
      call permutation_write (eq%msq_perm, u)
      write (u, "(1x,12x,1x,A,1x)", advance="no") &
           "       angle permutation:"
      call permutation_write (eq%angle_perm, u)
      write (u, "(1x,12x,1x,26x)", advance="no")
      do i = 1, size (eq%angle_sig)
         if (eq%angle_sig(i)) then
            write (u, "(1x,A)", advance="no") "+"
         else
            write (u, "(1x,A)", advance="no") "-"
         end if
      end do
      write (u, *)
      if (associated (eq%next))  call equivalence_write_rec (eq%next, u)
   end subroutine equivalence_write_rec
end subroutine equivalence_list_write
```

18.6.3 Groves

A grove is a group of trees (phase-space channels) that share a common weight in the integration. Within a grove, channels can be declared equivalent, so they also share their integration grids (up to symmetries). The grove contains a list of equivalences. The `tree_count_offset` is the total number of trees of the preceding groves; when the trees are counted per forest (integration channels), the offset has to be added to all tree indices.

⟨*PHS forests: types*⟩+≡

```
type :: phs_grove_t
   private
   integer :: tree_count_offset
   type(phs_tree_t), dimension(:), allocatable :: tree
   type(equivalence_list_t) :: equivalence_list
end type phs_grove_t
```

Call `phs_tree_init` which is also elemental:

⟨*PHS forests: procedures*⟩+≡

```
elemental subroutine phs_grove_init &
     (grove, n_trees, n_in, n_out, n_masses, n_angles)
```

```
    type(phs_grove_t), intent(inout) :: grove
    integer, intent(in) :: n_trees, n_in, n_out, n_masses, n_angles
    grove%tree_count_offset = 0
    allocate (grove%tree (n_trees))
    call phs_tree_init (grove%tree, n_in, n_out, n_masses, n_angles)
  end subroutine phs_grove_init
```

The trees do not have pointer components, thus no call to `phs_tree_final`:

⟨*PHS forests: procedures*⟩+≡

```
  elemental subroutine phs_grove_final (grove)
    type(phs_grove_t), intent(inout) :: grove
    deallocate (grove%tree)
    call equivalence_list_final (grove%equivalence_list)
  end subroutine phs_grove_final
```

Deep copy.

⟨*PHS forests: interfaces*⟩+≡

```
  interface assignment(=)
     module procedure phs_grove_assign0
     module procedure phs_grove_assign1
  end interface
```

⟨*PHS forests: procedures*⟩+≡

```
  subroutine phs_grove_assign0 (grove_out, grove_in)
    type(phs_grove_t), intent(out) :: grove_out
    type(phs_grove_t), intent(in) :: grove_in
    grove_out%tree_count_offset = grove_in%tree_count_offset
    if (allocated (grove_in%tree)) then
       allocate (grove_out%tree (size (grove_in%tree)))
       grove_out%tree = grove_in%tree
    end if
    grove_out%equivalence_list = grove_in%equivalence_list
  end subroutine phs_grove_assign0

  subroutine phs_grove_assign1 (grove_out, grove_in)
    type(phs_grove_t), dimension(:), intent(out) :: grove_out
    type(phs_grove_t), dimension(:), intent(in) :: grove_in
    integer :: i
    do i = 1, size (grove_in)
       call phs_grove_assign0 (grove_out(i), grove_in(i))
    end do
  end subroutine phs_grove_assign1
```

Get the global (s-channel) mappings. Implemented as a subroutine which returns an array (slice).

⟨*PHS forests: procedures*⟩+≡

```
  subroutine phs_grove_assign_s_mappings (grove, mapping)
    type(phs_grove_t), intent(in) :: grove
    type(mapping_t), dimension(:), intent(out) :: mapping
    integer :: i
    if (size (mapping) == size (grove%tree)) then
       do i = 1, size (mapping)
```

```
        call phs_tree_assign_s_mapping (grove%tree(i), mapping(i))
      end do
    else
      call msg_bug ("phs_grove_assign_s_mappings: array size mismatch")
    end if
  end subroutine phs_grove_assign_s_mappings
```

18.6.4 The forest type

This is a collection of trees and associated particles. In a given tree, each branch code corresponds to a particle in the `prt` array. Furthermore, we have an array of mass sums which is independent of the decay tree and of the particular event. The mappings directly correspond to the decay trees, and the decay groves collect the trees in classes. The permutation list consists of all permutations of outgoing particles that map the decay forest onto itself.

The particle codes `flv` (one for each external particle) are needed for determining masses and such. The trees and associated information are collected in the `grove` array, together with a lookup table that associates tree indices to groves. Finally, the `prt` array serves as workspace for phase-space evaluation.

The `prt_combination` is a list of index pairs, namely the particle momenta pairs that need to be combined in order to provide all momentum combinations that the phase-space trees need to know.

⟨*PHS forests: public*⟩+≡

```
  public :: phs_forest_t
```

⟨*PHS forests: types*⟩+≡

```
  type :: phs_forest_t
    private
    integer :: n_in, n_out, n_tot
    integer :: n_masses, n_angles, n_dimensions
    integer :: n_trees, n_equivalences
    type(flavor_t), dimension(:), allocatable :: flv
    type(phs_grove_t), dimension(:), allocatable :: grove
    integer, dimension(:), allocatable :: grove_lookup
    type(phs_prt_t), dimension(:), allocatable :: prt_in
    type(phs_prt_t), dimension(:), allocatable :: prt_out
    type(phs_prt_t), dimension(:), allocatable :: prt
    integer(TC), dimension(:,:), allocatable :: prt_combination
    type(mapping_t), dimension(:), allocatable :: s_mapping
  contains
  ⟨PHS forests: phs forest: TBP⟩
  end type phs_forest_t
```

The initialization merely allocates memory. We have to know how many trees there are in each grove, so we can initialize everything. The number of groves is the size of the `n_tree` array.

In the `grove_lookup` table we store the grove index that belongs to each absolute tree index. The difference between the absolute index and the relative (to the grove) index is stored, for each grove, as `tree_count_offset`.

The particle array is allocated according to the total number of branches each tree has, but not filled.

⟨*PHS forests: public*⟩+≡
```
public :: phs_forest_init
```

⟨*PHS forests: procedures*⟩+≡
```
subroutine phs_forest_init (forest, n_tree, n_in, n_out)
  type(phs_forest_t), intent(inout) :: forest
  integer, dimension(:), intent(in) :: n_tree
  integer, intent(in) :: n_in, n_out
  integer :: g, count, k_root
  forest%n_in = n_in
  forest%n_out = n_out
  forest%n_tot = n_in + n_out
  forest%n_masses = max (n_out - 2, 0)
  forest%n_angles = max (2*n_out - 2, 0)
  forest%n_dimensions = forest%n_masses + forest%n_angles
  forest%n_trees = sum (n_tree)
  forest%n_equivalences = 0
  allocate (forest%grove (size (n_tree)))
  call phs_grove_init &
       (forest%grove, n_tree, n_in, n_out, forest%n_masses, &
        forest%n_angles)
  allocate (forest%grove_lookup (forest%n_trees))
  count = 0
  do g = 1, size (forest%grove)
     forest%grove(g)%tree_count_offset = count
     forest%grove_lookup (count+1:count+n_tree(g)) = g
     count = count + n_tree(g)
  end do
  allocate (forest%prt_in  (n_in))
  allocate (forest%prt_out (forest%n_out))
  k_root = 2**forest%n_tot - 1
  allocate (forest%prt (k_root))
  allocate (forest%prt_combination (2, k_root))
  allocate (forest%s_mapping (forest%n_trees))
end subroutine phs_forest_init
```

Assign the global (s-channel) mappings.

⟨*PHS forests: public*⟩+≡
```
public :: phs_forest_set_s_mappings
```

⟨*PHS forests: procedures*⟩+≡
```
subroutine phs_forest_set_s_mappings (forest)
  type(phs_forest_t), intent(inout) :: forest
  integer :: g, i0, i1, n
  do g = 1, size (forest%grove)
     call phs_forest_get_grove_bounds (forest, g, i0, i1, n)
     call phs_grove_assign_s_mappings &
          (forest%grove(g), forest%s_mapping(i0:i1))
  end do
end subroutine phs_forest_set_s_mappings
```

The grove finalizer is called because it contains the equivalence list:

⟨*PHS forests: public*⟩+≡
```
public :: phs_forest_final
```

⟨*PHS forests: procedures*⟩+≡
```
subroutine phs_forest_final (forest)
  type(phs_forest_t), intent(inout) :: forest
  if (allocated (forest%grove)) then
     call phs_grove_final (forest%grove)
     deallocate (forest%grove)
  end if
  if (allocated (forest%grove_lookup))  deallocate (forest%grove_lookup)
  if (allocated (forest%prt))  deallocate (forest%prt)
  if (allocated (forest%s_mapping))  deallocate (forest%s_mapping)
end subroutine phs_forest_final
```

18.6.5 Screen output

Write the particles that are non-null, then the trees which point to them:

⟨*PHS forests: public*⟩+≡
```
public :: phs_forest_write
```

⟨*PHS forests: phs forest: TBP*⟩≡
```
procedure :: write => phs_forest_write
```

⟨*PHS forests: procedures*⟩+≡
```
subroutine phs_forest_write (forest, unit)
  class(phs_forest_t), intent(in) :: forest
  integer, intent(in), optional :: unit
  integer :: u
  integer :: i, g, k
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)") "Phase space forest:"
  write (u, "(3x,A,I0)") "n_in  = ", forest%n_in
  write (u, "(3x,A,I0)") "n_out = ", forest%n_out
  write (u, "(3x,A,I0)") "n_tot = ", forest%n_tot
  write (u, "(3x,A,I0)") "n_masses = ", forest%n_masses
  write (u, "(3x,A,I0)") "n_angles = ", forest%n_angles
  write (u, "(3x,A,I0)") "n_dim    = ", forest%n_dimensions
  write (u, "(3x,A,I0)") "n_trees  = ", forest%n_trees
  write (u, "(3x,A,I0)") "n_equiv  = ", forest%n_equivalences
  write (u, "(3x,A)", advance="no") "flavors  ="
  if (allocated (forest%flv)) then
     do i = 1, size (forest%flv)
        write (u, "(1x,I0)", advance="no")  forest%flv(i)%get_pdg ()
     end do
     write (u, "(A)")
  else
     write (u, "(1x,A)") "[empty]"
  end if
  write (u, "(1x,A)") "Particle combinations:"
  if (allocated (forest%prt_combination)) then
     do k = 1, size (forest%prt_combination, 2)
        if (forest%prt_combination(1, k) /= 0) then
```

```
         write (u, "(3x,I0,1x,'<=',1x,I0,1x,'+',1x,I0)") &
              k, forest%prt_combination(:,k)
      end if
   end do
else
   write (u, "(3x,A)") "  [empty]"
end if
write (u, "(1x,A)") "Groves and trees:"
if (allocated (forest%grove)) then
   do g = 1, size (forest%grove)
      write (u, "(3x,A,1x,I0)") "Grove    ", g
      call phs_grove_write (forest%grove(g), unit)
   end do
else
   write (u, "(3x,A)") "  [empty]"
end if
write (u, "(1x,A,I0)") "Total number of equivalences: ", &
     forest%n_equivalences
write (u, "(A)")
write (u, "(1x,A)") "Global s-channel mappings:"
if (allocated (forest%s_mapping)) then
   do i = 1, size (forest%s_mapping)
      associate (mapping => forest%s_mapping(i))
        if (mapping_is_s_channel (mapping) &
             .or. mapping_is_on_shell (mapping)) then
           write (u, "(1x,I0,':',1x)", advance="no")  i
           call mapping_write (forest%s_mapping(i), unit)
        end if
      end associate
   end do
else
   write (u, "(3x,A)") "  [empty]"
end if
write (u, "(A)")
write (u, "(1x,A)") "Incoming particles:"
if (allocated (forest%prt_in)) then
   if (any (phs_prt_is_defined (forest%prt_in))) then
      do i = 1, size (forest%prt_in)
         if (phs_prt_is_defined (forest%prt_in(i))) then
            write (u, "(1x,A,1x,I0)")  "Particle", i
            call phs_prt_write (forest%prt_in(i), u)
         end if
      end do
   else
      write (u, "(3x,A)")  "[all undefined]"
   end if
else
   write (u, "(3x,A)")  "  [empty]"
end if
write (u, "(A)")
write (u, "(1x,A)") "Outgoing particles:"
if (allocated (forest%prt_out)) then
   if (any (phs_prt_is_defined (forest%prt_out))) then
      do i = 1, size (forest%prt_out)
```

```
            if (phs_prt_is_defined (forest%prt_out(i))) then
               write (u, "(1x,A,1x,I0)")  "Particle", i
               call phs_prt_write (forest%prt_out(i), u)
            end if
         end do
      else
         write (u, "(3x,A)")  "[all undefined]"
      end if
   else
      write (u, "(1x,A)")  "  [empty]"
   end if
   write (u, "(A)")
   write (u, "(1x,A)") "Tree particles:"
   if (allocated (forest%prt)) then
      if (any (phs_prt_is_defined (forest%prt))) then
         do i = 1, size (forest%prt)
            if (phs_prt_is_defined (forest%prt(i))) then
               write (u, "(1x,A,1x,I0)")  "Particle", i
               call phs_prt_write (forest%prt(i), u)
            end if
         end do
      else
         write (u, "(3x,A)")  "[all undefined]"
      end if
   else
      write (u, "(3x,A)")  "  [empty]"
   end if
end subroutine phs_forest_write

subroutine phs_grove_write (grove, unit)
   type(phs_grove_t), intent(in) :: grove
   integer, intent(in), optional :: unit
   integer :: u
   integer :: t
   u = given_output_unit (unit);  if (u < 0)  return
   do t = 1, size (grove%tree)
      write (u, "(3x,A,I0)") "Tree      ", t
      call phs_tree_write (grove%tree(t), unit)
   end do
   write (u, "(1x,A)") "Equivalence list:"
   call equivalence_list_write (grove%equivalence_list, unit)
end subroutine phs_grove_write
```

Deep copy.

⟨*PHS forests: public*⟩+≡

```
public :: assignment(=)
```

⟨*PHS forests: interfaces*⟩+≡

```
interface assignment(=)
   module procedure phs_forest_assign
end interface
```

⟨*PHS forests: procedures*⟩+≡

```
subroutine phs_forest_assign (forest_out, forest_in)
```

```
    type(phs_forest_t), intent(out) :: forest_out
    type(phs_forest_t), intent(in) :: forest_in
    forest_out%n_in  = forest_in%n_in
    forest_out%n_out = forest_in%n_out
    forest_out%n_tot = forest_in%n_tot
    forest_out%n_masses = forest_in%n_masses
    forest_out%n_angles = forest_in%n_angles
    forest_out%n_dimensions  = forest_in%n_dimensions
    forest_out%n_trees  = forest_in%n_trees
    forest_out%n_equivalences  = forest_in%n_equivalences
    if (allocated (forest_in%flv)) then
       allocate (forest_out%flv (size (forest_in%flv)))
       forest_out%flv = forest_in%flv
    end if
    if (allocated (forest_in%grove)) then
       allocate (forest_out%grove (size (forest_in%grove)))
       forest_out%grove = forest_in%grove
    end if
    if (allocated (forest_in%grove_lookup)) then
       allocate (forest_out%grove_lookup (size (forest_in%grove_lookup)))
       forest_out%grove_lookup = forest_in%grove_lookup
    end if
    if (allocated (forest_in%prt_in)) then
       allocate (forest_out%prt_in (size (forest_in%prt_in)))
       forest_out%prt_in = forest_in%prt_in
    end if
    if (allocated (forest_in%prt_out)) then
       allocate (forest_out%prt_out (size (forest_in%prt_out)))
       forest_out%prt_out = forest_in%prt_out
    end if
    if (allocated (forest_in%prt)) then
       allocate (forest_out%prt (size (forest_in%prt)))
       forest_out%prt = forest_in%prt
    end if
    if (allocated (forest_in%s_mapping)) then
       allocate (forest_out%s_mapping (size (forest_in%s_mapping)))
       forest_out%s_mapping = forest_in%s_mapping
    end if
    if (allocated (forest_in%prt_combination)) then
       allocate (forest_out%prt_combination &
            (2, size (forest_in%prt_combination, 2)))
       forest_out%prt_combination = forest_in%prt_combination
    end if
  end subroutine phs_forest_assign
```

18.6.6 Accessing contents

Get the number of integration parameters

⟨*PHS forests: public*⟩+≡

```
  public :: phs_forest_get_n_parameters
```

⟨*PHS forests: procedures*⟩+≡

```
  function phs_forest_get_n_parameters (forest) result (n)
```

```
    integer :: n
    type(phs_forest_t), intent(in) :: forest
    n = forest%n_dimensions
  end function phs_forest_get_n_parameters
```

Get the number of integration channels

⟨*PHS forests: public*⟩+≡

```
  public :: phs_forest_get_n_channels
```

⟨*PHS forests: procedures*⟩+≡

```
  function phs_forest_get_n_channels (forest) result (n)
    integer :: n
    type(phs_forest_t), intent(in) :: forest
    n = forest%n_trees
  end function phs_forest_get_n_channels
```

Get the number of groves

⟨*PHS forests: public*⟩+≡

```
  public :: phs_forest_get_n_groves
```

⟨*PHS forests: procedures*⟩+≡

```
  function phs_forest_get_n_groves (forest) result (n)
    integer :: n
    type(phs_forest_t), intent(in) :: forest
    n = size (forest%grove)
  end function phs_forest_get_n_groves
```

Get the index bounds for a specific grove.

⟨*PHS forests: public*⟩+≡

```
  public :: phs_forest_get_grove_bounds
```

⟨*PHS forests: procedures*⟩+≡

```
  subroutine phs_forest_get_grove_bounds (forest, g, i0, i1, n)
    type(phs_forest_t), intent(in) :: forest
    integer, intent(in) :: g
    integer, intent(out) :: i0, i1, n
    n = size (forest%grove(g)%tree)
    i0 = forest%grove(g)%tree_count_offset + 1
    i1 = forest%grove(g)%tree_count_offset + n
  end subroutine phs_forest_get_grove_bounds
```

Get the number of equivalences

⟨*PHS forests: public*⟩+≡

```
  public :: phs_forest_get_n_equivalences
```

⟨*PHS forests: procedures*⟩+≡

```
  function phs_forest_get_n_equivalences (forest) result (n)
    integer :: n
    type(phs_forest_t), intent(in) :: forest
    n = forest%n_equivalences
  end function phs_forest_get_n_equivalences
```

Return true if a particular channel has a global (s-channel) mapping; also return the resonance mass and width for this mapping.

⟨*PHS forests: public*⟩+≡

```
public :: phs_forest_get_s_mapping
public :: phs_forest_get_on_shell
```

⟨*PHS forests: procedures*⟩+≡

```
subroutine phs_forest_get_s_mapping (forest, channel, flag, mass, width)
  type(phs_forest_t), intent(in) :: forest
  integer, intent(in) :: channel
  logical, intent(out) :: flag
  real(default), intent(out) :: mass, width
  flag = mapping_is_s_channel (forest%s_mapping(channel))
  if (flag) then
     mass = mapping_get_mass (forest%s_mapping(channel))
     width = mapping_get_width (forest%s_mapping(channel))
  else
     mass = 0
     width = 0
  end if
end subroutine phs_forest_get_s_mapping

subroutine phs_forest_get_on_shell (forest, channel, flag, mass)
  type(phs_forest_t), intent(in) :: forest
  integer, intent(in) :: channel
  logical, intent(out) :: flag
  real(default), intent(out) :: mass
  flag = mapping_is_on_shell (forest%s_mapping(channel))
  if (flag) then
     mass = mapping_get_mass (forest%s_mapping(channel))
  else
     mass = 0
  end if
end subroutine phs_forest_get_on_shell
```

Extract the set of unique resonance histories, in form of an array.

⟨*PHS forests: phs forest: TBP*⟩+≡

```
procedure :: extract_resonance_histories => phs_forest_extract_resonance_histories
```

⟨*PHS forests: procedures*⟩+≡

```
subroutine phs_forest_extract_resonance_histories (forest, res_hist)
  class(phs_forest_t), intent(in) :: forest
  type(resonance_history_t), dimension(:), allocatable, intent(out) :: res_hist
  type(resonance_history_t) :: rh
  type(resonance_history_set_t) :: res_hist_set
  integer :: g, t
  call res_hist_set%init ()
  do g = 1, size (forest%grove)
     associate (grove => forest%grove(g))
        do t = 1, size (grove%tree)
           call grove%tree(t)%extract_resonance_history (rh)
           call res_hist_set%enter (rh)
        end do
     end associate
```

```
    end do
    call res_hist_set%to_array (res_hist)
  end subroutine phs_forest_extract_resonance_histories
```

18.6.7 Read the phase space setup from file

The phase space setup is stored in a file. The file may be generated by the `cascades` module below, or by other means. This file has to be read and parsed to create the PHS forest as the internal phase-space representation.

Create lexer and syntax:

⟨*PHS forests: procedures*⟩+≡

```
  subroutine define_phs_forest_syntax (ifile)
    type(ifile_t) :: ifile
    call ifile_append (ifile, "SEQ phase_space_list = process_phase_space*")
    call ifile_append (ifile, "SEQ process_phase_space = " &
         // "process_def process_header phase_space")
    call ifile_append (ifile, "SEQ process_def = process process_list")
    call ifile_append (ifile, "KEY process")
    call ifile_append (ifile, "LIS process_list = process_tag*")
    call ifile_append (ifile, "IDE process_tag")
    call ifile_append (ifile, "SEQ process_header = " &
         // "md5sum_process = md5sum " &
         // "md5sum_model_par = md5sum " &
         // "md5sum_phs_config = md5sum " &
         // "sqrts = real " &
         // "m_threshold_s = real " &
         // "m_threshold_t = real " &
         // "off_shell = integer " &
         // "t_channel = integer " &
         // "keep_nonresonant = logical")
    call ifile_append (ifile, "KEY '='")
    call ifile_append (ifile, "KEY '-'")
    call ifile_append (ifile, "KEY md5sum_process")
    call ifile_append (ifile, "KEY md5sum_model_par")
    call ifile_append (ifile, "KEY md5sum_phs_config")
    call ifile_append (ifile, "KEY sqrts")
    call ifile_append (ifile, "KEY m_threshold_s")
    call ifile_append (ifile, "KEY m_threshold_t")
    call ifile_append (ifile, "KEY off_shell")
    call ifile_append (ifile, "KEY t_channel")
    call ifile_append (ifile, "KEY keep_nonresonant")
    call ifile_append (ifile, "QUO md5sum = '"" ... ""'")
    call ifile_append (ifile, "REA real")
    call ifile_append (ifile, "INT integer")
    call ifile_append (ifile, "IDE logical")
    call ifile_append (ifile, "SEQ phase_space = grove_def+")
    call ifile_append (ifile, "SEQ grove_def = grove tree_def+")
    call ifile_append (ifile, "KEY grove")
    call ifile_append (ifile, "SEQ tree_def = tree bincodes mapping*")
    call ifile_append (ifile, "KEY tree")
    call ifile_append (ifile, "SEQ bincodes = bincode*")
    call ifile_append (ifile, "INT bincode")
```

```
    call ifile_append (ifile, "SEQ mapping = map bincode channel signed_pdg")
    call ifile_append (ifile, "KEY map")
    call ifile_append (ifile, "ALT channel = &
         &s_channel | t_channel | u_channel | &
         &collinear | infrared | radiation | on_shell")
    call ifile_append (ifile, "KEY s_channel")
    ! call ifile_append (ifile, "KEY t_channel")   !!! Key already exists
    call ifile_append (ifile, "KEY u_channel")
    call ifile_append (ifile, "KEY collinear")
    call ifile_append (ifile, "KEY infrared")
    call ifile_append (ifile, "KEY radiation")
    call ifile_append (ifile, "KEY on_shell")
    call ifile_append (ifile, "ALT signed_pdg = &
         &pdg | negative_pdg")
    call ifile_append (ifile, "SEQ negative_pdg = '-' pdg")
    call ifile_append (ifile, "INT pdg")
  end subroutine define_phs_forest_syntax
```

The model-file syntax and lexer are fixed, therefore stored as module variables:

⟨*PHS forests: variables*⟩≡

```
  type(syntax_t), target, save :: syntax_phs_forest
```

⟨*PHS forests: public*⟩+≡

```
  public :: syntax_phs_forest_init
```

⟨*PHS forests: procedures*⟩+≡

```
  subroutine syntax_phs_forest_init ()
    type(ifile_t) :: ifile
    call define_phs_forest_syntax (ifile)
    call syntax_init (syntax_phs_forest, ifile)
    call ifile_final (ifile)
  end subroutine syntax_phs_forest_init
```

⟨*PHS forests: procedures*⟩+≡

```
  subroutine lexer_init_phs_forest (lexer)
    type(lexer_t), intent(out) :: lexer
    call lexer_init (lexer, &
         comment_chars = "#!", &
         quote_chars = '"', &
         quote_match = '"', &
         single_chars = "-", &
         special_class = ["="] , &
         keyword_list = syntax_get_keyword_list_ptr (syntax_phs_forest))
  end subroutine lexer_init_phs_forest
```

⟨*PHS forests: public*⟩+≡

```
  public :: syntax_phs_forest_final
```

⟨*PHS forests: procedures*⟩+≡

```
  subroutine syntax_phs_forest_final ()
    call syntax_final (syntax_phs_forest)
  end subroutine syntax_phs_forest_final
```

⟨*PHS forests: public*⟩+≡
```
public :: syntax_phs_forest_write
```
⟨*PHS forests: procedures*⟩+≡
```
subroutine syntax_phs_forest_write (unit)
  integer, intent(in), optional :: unit
  call syntax_write (syntax_phs_forest, unit)
end subroutine syntax_phs_forest_write
```

The concrete parser and interpreter. Generate an input stream for the external `unit`, read the parse tree (with given `syntax` and `lexer`) from this stream, and transfer the contents of the parse tree to the PHS `forest`.

We look for the matching `process` tag, count groves and trees for initializing the `forest`, and fill the trees.

If the optional parameters are set, compare the parameters stored in the file to those. Set `match` true if everything agrees.

⟨*PHS forests: public*⟩+≡
```
public :: phs_forest_read
```
⟨*PHS forests: interfaces*⟩+≡
```
interface phs_forest_read
   module procedure phs_forest_read_file
   module procedure phs_forest_read_unit
   module procedure phs_forest_read_parse_tree
end interface
```

⟨*PHS forests: procedures*⟩+≡
```
subroutine phs_forest_read_file &
     (forest, filename, process_id, n_in, n_out, model, found, &
      md5sum_process, md5sum_model_par, &
      md5sum_phs_config, phs_par, match)
  type(phs_forest_t), intent(out) :: forest
  type(string_t), intent(in) :: filename
  type(string_t), intent(in) :: process_id
  integer, intent(in) :: n_in, n_out
  class(model_data_t), intent(in), target :: model
  logical, intent(out) :: found
  character(32), intent(in), optional :: &
       md5sum_process, md5sum_model_par, md5sum_phs_config
  type(phs_parameters_t), intent(in), optional :: phs_par
  logical, intent(out), optional :: match
  type(parse_tree_t), target :: parse_tree
  type(stream_t), target :: stream
  type(lexer_t) :: lexer
  call lexer_init_phs_forest (lexer)
  call stream_init (stream, char (filename))
  call lexer_assign_stream (lexer, stream)
  call parse_tree_init (parse_tree, syntax_phs_forest, lexer)
  call phs_forest_read (forest, parse_tree, &
       process_id, n_in, n_out, model, found, &
       md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
  call stream_final (stream)
  call lexer_final (lexer)
  call parse_tree_final (parse_tree)
```

```
end subroutine phs_forest_read_file

subroutine phs_forest_read_unit &
     (forest, unit, process_id, n_in, n_out, model, found, &
      md5sum_process, md5sum_model_par, md5sum_phs_config, &
      phs_par, match)
  type(phs_forest_t), intent(out) :: forest
  integer, intent(in) :: unit
  type(string_t), intent(in) :: process_id
  integer, intent(in) :: n_in, n_out
  class(model_data_t), intent(in), target :: model
  logical, intent(out) :: found
  character(32), intent(in), optional :: &
       md5sum_process, md5sum_model_par, md5sum_phs_config
  type(phs_parameters_t), intent(in), optional :: phs_par
  logical, intent(out), optional :: match
  type(parse_tree_t), target :: parse_tree
  type(stream_t), target :: stream
  type(lexer_t) :: lexer
  call lexer_init_phs_forest (lexer)
  call stream_init (stream, unit)
  call lexer_assign_stream (lexer, stream)
  call parse_tree_init (parse_tree, syntax_phs_forest, lexer)
  call phs_forest_read (forest, parse_tree, &
       process_id, n_in, n_out, model, found, &
       md5sum_process, md5sum_model_par, md5sum_phs_config, &
       phs_par, match)
  call stream_final (stream)
  call lexer_final (lexer)
  call parse_tree_final (parse_tree)
end subroutine phs_forest_read_unit

subroutine phs_forest_read_parse_tree &
     (forest, parse_tree, process_id, n_in, n_out, model, found, &
      md5sum_process, md5sum_model_par, md5sum_phs_config, &
      phs_par, match)
  type(phs_forest_t), intent(out) :: forest
  type(parse_tree_t), intent(in), target :: parse_tree
  type(string_t), intent(in) :: process_id
  integer, intent(in) :: n_in, n_out
  class(model_data_t), intent(in), target :: model
  logical, intent(out) :: found
  character(32), intent(in), optional :: &
       md5sum_process, md5sum_model_par, md5sum_phs_config
  type(phs_parameters_t), intent(in), optional :: phs_par
  logical, intent(out), optional :: match
  type(parse_node_t), pointer :: node_header, node_phs, node_grove
  integer :: n_grove, g
  integer, dimension(:), allocatable :: n_tree
  integer :: t
  node_header => parse_tree_get_process_ptr (parse_tree, process_id)
  found = associated (node_header);  if (.not. found)  return
  if (present (match)) then
     call phs_forest_check_input (node_header, &
```

```
            md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
         if (.not. match)  return
      end if
      node_phs => parse_node_get_next_ptr (node_header)
      n_grove = parse_node_get_n_sub (node_phs)
      allocate (n_tree (n_grove))
      do g = 1, n_grove
         node_grove => parse_node_get_sub_ptr (node_phs, g)
         n_tree(g) = parse_node_get_n_sub (node_grove) - 1
      end do
      call phs_forest_init (forest, n_tree, n_in, n_out)
      do g = 1, n_grove
         node_grove => parse_node_get_sub_ptr (node_phs, g)
         do t = 1, n_tree(g)
            call phs_tree_set (forest%grove(g)%tree(t), &
                 parse_node_get_sub_ptr (node_grove, t+1), model)
         end do
      end do
   end subroutine phs_forest_read_parse_tree
```

Check the input for consistency. If any MD5 sum or phase-space parameter disagrees, the phase-space file cannot be used. The MD5 sum checks are skipped if the stored MD5 sum is empty.

⟨*PHS forests: procedures*⟩+≡

```
   subroutine phs_forest_check_input (pn_header, &
       md5sum_process, md5sum_model_par, md5sum_phs_config, phs_par, match)
      type(parse_node_t), intent(in), target :: pn_header
      character(32), intent(in) :: &
           md5sum_process, md5sum_model_par, md5sum_phs_config
      type(phs_parameters_t), intent(in), optional :: phs_par
      logical, intent(out) :: match
      type(parse_node_t), pointer :: pn_md5sum, pn_rval, pn_ival, pn_lval
      character(32) :: md5sum
      type(phs_parameters_t) :: phs_par_old
      character(1) :: lstr
      pn_md5sum => parse_node_get_sub_ptr (pn_header, 3)
      md5sum = parse_node_get_string (pn_md5sum)
      if (md5sum /= "" .and. md5sum /= md5sum_process) then
         call msg_message ("Phase space: discarding old configuration &
              &(process changed)")
         match = .false.;  return
      end if
      pn_md5sum => parse_node_get_next_ptr (pn_md5sum, 3)
      md5sum = parse_node_get_string (pn_md5sum)
      if (md5sum /= "" .and. md5sum /= md5sum_model_par) then
         call msg_message ("Phase space: discarding old configuration &
              &(model parameters changed)")
         match = .false.;  return
      end if
      pn_md5sum => parse_node_get_next_ptr (pn_md5sum, 3)
      md5sum = parse_node_get_string (pn_md5sum)
      if (md5sum /= "" .and. md5sum /= md5sum_phs_config) then
         call msg_message ("Phase space: discarding old configuration &
```

```
            &(configuration parameters changed)")
         match = .false.;  return
      end if
      if (present (phs_par)) then
         pn_rval => parse_node_get_next_ptr (pn_md5sum, 3)
         phs_par_old%sqrts = parse_node_get_real (pn_rval)
         pn_rval => parse_node_get_next_ptr (pn_rval, 3)
         phs_par_old%m_threshold_s = parse_node_get_real (pn_rval)
         pn_rval => parse_node_get_next_ptr (pn_rval, 3)
         phs_par_old%m_threshold_t = parse_node_get_real (pn_rval)
         pn_ival => parse_node_get_next_ptr (pn_rval, 3)
         phs_par_old%off_shell = parse_node_get_integer (pn_ival)
         pn_ival => parse_node_get_next_ptr (pn_ival, 3)
         phs_par_old%t_channel = parse_node_get_integer (pn_ival)
         pn_lval => parse_node_get_next_ptr (pn_ival, 3)
         lstr = parse_node_get_string (pn_lval)
         read (lstr, "(L1)")  phs_par_old%keep_nonresonant
         if (phs_par_old /= phs_par) then
            call msg_message &
                 ("Phase space: discarding old configuration &
                 &(configuration parameters changed)")
            match = .false.;  return
         end if
      end if
      match = .true.
   end subroutine phs_forest_check_input
```

Initialize a specific tree in the forest, using the contents of the 'tree' node. First, count the bincodes, allocate an array and read them in, and make the tree. Each t-channel tree is flipped to s-channel. Then, find mappings and initialize them.

⟨*PHS forests: procedures*⟩+≡

```
   subroutine phs_tree_set (tree, node, model)
      type(phs_tree_t), intent(inout) :: tree
      type(parse_node_t), intent(in), target :: node
      class(model_data_t), intent(in), target :: model
      type(parse_node_t), pointer :: node_bincodes, node_mapping, pn_pdg
      integer :: n_bincodes, offset
      integer(TC), dimension(:), allocatable :: bincode
      integer :: b, n_mappings, m
      integer(TC) :: k
      type(string_t) :: type
      integer :: pdg
      node_bincodes => parse_node_get_sub_ptr (node, 2)
      if (associated (node_bincodes)) then
         select case (char (parse_node_get_rule_key (node_bincodes)))
         case ("bincodes")
            n_bincodes = parse_node_get_n_sub (node_bincodes)
            offset = 2
         case default
            n_bincodes = 0
            offset = 1
         end select
      else
```

```
      n_bincodes = 0
      offset = 2
   end if
   allocate (bincode (n_bincodes))
   do b = 1, n_bincodes
      bincode(b) = parse_node_get_integer &
           (parse_node_get_sub_ptr (node_bincodes, b))
   end do
   call phs_tree_from_array (tree, bincode)
   call phs_tree_flip_t_to_s_channel (tree)
   call phs_tree_canonicalize (tree)
   n_mappings = parse_node_get_n_sub (node) - offset
   do m = 1, n_mappings
      node_mapping => parse_node_get_sub_ptr (node, m + offset)
      k = parse_node_get_integer &
           (parse_node_get_sub_ptr (node_mapping, 2))
      type = parse_node_get_key &
           (parse_node_get_sub_ptr (node_mapping, 3))
      pn_pdg => parse_node_get_sub_ptr (node_mapping, 4)
      select case (char (pn_pdg%get_rule_key ()))
      case ("pdg")
         pdg = pn_pdg%get_integer ()
      case ("negative_pdg")
         pdg = - parse_node_get_integer (pn_pdg%get_sub_ptr (2))
      end select
      call phs_tree_init_mapping (tree, k, type, pdg, model)
   end do
end subroutine phs_tree_set
```

18.6.8 Preparation

The trees that we read from file do not carry flavor information. This is set separately:

The flavor list must be unique for a unique set of masses; if a given particle can have different flavor, the mass must be degenerate, so we can choose one of the possible flavor combinations.

⟨*PHS forests: public*⟩+≡

```
public :: phs_forest_set_flavors
```

⟨*PHS forests: procedures*⟩+≡

```
subroutine phs_forest_set_flavors (forest, flv, reshuffle, flv_extra)
  type(phs_forest_t), intent(inout) :: forest
  type(flavor_t), dimension(:), intent(in) :: flv
  integer, intent(in), dimension(:), allocatable, optional :: reshuffle
  type(flavor_t), intent(in), optional :: flv_extra
  integer :: i, n_flv0
  if (present (reshuffle) .and. present (flv_extra)) then
     n_flv0 = size (flv)
     do i = 1, n_flv0
        if (reshuffle(i) <= n_flv0) then
           forest%flv(i) = flv (reshuffle(i))
        else
           forest%flv(i) = flv_extra
```

```
          end if
       end do
    else
       allocate (forest%flv (size (flv)))
       forest%flv = flv
    end if
  end subroutine phs_forest_set_flavors
```

⟨*PHS forests: public*⟩+≡

```
  public :: phs_forest_set_momentum_links
```

⟨*PHS forests: procedures*⟩+≡

```
  subroutine phs_forest_set_momentum_links (forest, list)
    type(phs_forest_t), intent(inout) :: forest
    integer, intent(in), dimension(:), allocatable :: list
    integer :: g, t
    do g = 1, size (forest%grove)
      do t = 1, size (forest%grove(g)%tree)
        associate (tree => forest%grove(g)%tree(t))
          call phs_tree_set_momentum_links (tree, list)
!!!          call phs_tree_reshuffle_mappings (tree)
        end associate
      end do
    end do
  end subroutine phs_forest_set_momentum_links
```

Once the parameter set is fixed, the masses and the widths of the particles are known and the `mass_sum` arrays as well as the mapping parameters can be computed. Note that order is important: we first compute the mass sums, then the ordinary mappings. The resonances obtained here determine the effective masses, which in turn are used to implement step mappings for resonance decay products that are not mapped otherwise.

⟨*PHS forests: public*⟩+≡

```
  public :: phs_forest_set_parameters
```

⟨*PHS forests: procedures*⟩+≡

```
  subroutine phs_forest_set_parameters &
       (forest, mapping_defaults, variable_limits)
    type(phs_forest_t), intent(inout) :: forest
    type(mapping_defaults_t), intent(in) :: mapping_defaults
    logical, intent(in) :: variable_limits
    integer :: g, t
    do g = 1, size (forest%grove)
       do t = 1, size (forest%grove(g)%tree)
          call phs_tree_set_mass_sum &
               (forest%grove(g)%tree(t), forest%flv(forest%n_in+1:))
          call phs_tree_set_mapping_parameters (forest%grove(g)%tree(t), &
               mapping_defaults, variable_limits)
          call phs_tree_set_effective_masses (forest%grove(g)%tree(t))
          if (mapping_defaults%step_mapping) then
             call phs_tree_set_step_mappings (forest%grove(g)%tree(t), &
                  mapping_defaults%step_mapping_exp, variable_limits)
          end if
```

```
      end do
    end do
  end subroutine phs_forest_set_parameters
```

Generate the particle combination table. Scan all trees and merge their individual combination tables. At the end, valid entries are non-zero, and they indicate the indices of a pair of particles to be combined to a new particle. If a particle is accessible by more than one tree (this is usual), only keep the first possibility.

⟨*PHS forests: public*⟩+≡

```
  public :: phs_forest_setup_prt_combinations
```

⟨*PHS forests: procedures*⟩+≡

```
  subroutine phs_forest_setup_prt_combinations (forest)
    type(phs_forest_t), intent(inout) :: forest
    integer :: g, t
    integer, dimension(:,:), allocatable :: tree_prt_combination
    forest%prt_combination = 0
    allocate (tree_prt_combination (2, size (forest%prt_combination, 2)))
    do g = 1, size (forest%grove)
       do t = 1, size (forest%grove(g)%tree)
          call phs_tree_setup_prt_combinations &
               (forest%grove(g)%tree(t), tree_prt_combination)
          where (tree_prt_combination /= 0 .and. forest%prt_combination == 0)
             forest%prt_combination = tree_prt_combination
          end where
       end do
    end do
  end subroutine phs_forest_setup_prt_combinations
```

18.6.9 Accessing the particle arrays

Set the incoming particles from the contents of an interaction.

⟨*PHS forests: public*⟩+≡

```
  public :: phs_forest_set_prt_in
```

⟨*PHS forests: interfaces*⟩+≡

```
  interface phs_forest_set_prt_in
     module procedure phs_forest_set_prt_in_int, phs_forest_set_prt_in_mom
  end interface phs_forest_set_prt_in
```

⟨*PHS forests: procedures*⟩+≡

```
  subroutine phs_forest_set_prt_in_int (forest, int, lt_cm_to_lab)
    type(phs_forest_t), intent(inout) :: forest
    type(interaction_t), intent(in) :: int
    type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
    if (present (lt_cm_to_lab)) then
       call phs_prt_set_momentum (forest%prt_in, &
            inverse (lt_cm_to_lab) * &
            int%get_momenta (outgoing=.false.))
    else
       call phs_prt_set_momentum (forest%prt_in, &
            int%get_momenta (outgoing=.false.))
    end if
```

```
    associate (m_in => forest%flv(:forest%n_in)%get_mass ())
      call phs_prt_set_msq (forest%prt_in, m_in ** 2)
    end associate
    call phs_prt_set_defined (forest%prt_in)
  end subroutine phs_forest_set_prt_in_int

  subroutine phs_forest_set_prt_in_mom (forest, mom, lt_cm_to_lab)
    type(phs_forest_t), intent(inout) :: forest
    type(vector4_t), dimension(size (forest%prt_in)), intent(in) :: mom
    type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
    if (present (lt_cm_to_lab)) then
       call phs_prt_set_momentum (forest%prt_in, &
            inverse (lt_cm_to_lab) * mom)
    else
       call phs_prt_set_momentum (forest%prt_in, mom)
    end if
    associate (m_in => forest%flv(:forest%n_in)%get_mass ())
      call phs_prt_set_msq (forest%prt_in, m_in ** 2)
    end associate
    call phs_prt_set_defined (forest%prt_in)
  end subroutine phs_forest_set_prt_in_mom
```

Set the outgoing particles from the contents of an interaction.

⟨*PHS forests: public*⟩+≡

```
  public :: phs_forest_set_prt_out
```

⟨*PHS forests: interfaces*⟩+≡

```
  interface phs_forest_set_prt_out
     module procedure phs_forest_set_prt_out_int, phs_forest_set_prt_out_mom
  end interface phs_forest_set_prt_out
```

⟨*PHS forests: procedures*⟩+≡

```
  subroutine phs_forest_set_prt_out_int (forest, int, lt_cm_to_lab)
    type(phs_forest_t), intent(inout) :: forest
    type(interaction_t), intent(in) :: int
    type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
    if (present (lt_cm_to_lab)) then
       call phs_prt_set_momentum (forest%prt_out, &
            inverse (lt_cm_to_lab) * &
            int%get_momenta (outgoing=.true.))
    else
       call phs_prt_set_momentum (forest%prt_out, &
            int%get_momenta (outgoing=.true.))
    end if
    associate (m_out => forest%flv(forest%n_in+1:)%get_mass ())
      call phs_prt_set_msq (forest%prt_out, m_out ** 2)
    end associate
    call phs_prt_set_defined (forest%prt_out)
  end subroutine phs_forest_set_prt_out_int

  subroutine phs_forest_set_prt_out_mom (forest, mom, lt_cm_to_lab)
    type(phs_forest_t), intent(inout) :: forest
    type(vector4_t), dimension(size (forest%prt_out)), intent(in) :: mom
    type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
    if (present (lt_cm_to_lab)) then
```

```
      call phs_prt_set_momentum (forest%prt_out, &
           inverse (lt_cm_to_lab) * mom)
   else
      call phs_prt_set_momentum (forest%prt_out, mom)
   end if
   associate (m_out => forest%flv(forest%n_in+1:)%get_mass ())
     call phs_prt_set_msq (forest%prt_out, m_out ** 2)
   end associate
   call phs_prt_set_defined (forest%prt_out)
end subroutine phs_forest_set_prt_out_mom
```

Combine particles as described by the particle combination table. Particle momentum sums will be calculated only if the resulting particle is contained in at least one of the trees in the current forest. The others are kept undefined.

⟨*PHS forests: public*⟩+≡

```
public :: phs_forest_combine_particles
```

⟨*PHS forests: procedures*⟩+≡

```
subroutine phs_forest_combine_particles (forest)
  type(phs_forest_t), intent(inout) :: forest
  integer :: k
  integer, dimension(2) :: kk
  do k = 1, size (forest%prt_combination, 2)
     kk = forest%prt_combination(:,k)
     if (kk(1) /= 0) then
        call phs_prt_combine (forest%prt(k), &
             forest%prt(kk(1)), forest%prt(kk(2)))
     end if
  end do
end subroutine phs_forest_combine_particles
```

Extract the outgoing particles and insert into an interaction.

⟨*PHS forests: public*⟩+≡

```
public :: phs_forest_get_prt_out
```

⟨*PHS forests: procedures*⟩+≡

```
subroutine phs_forest_get_prt_out (forest, int, lt_cm_to_lab)
  type(phs_forest_t), intent(in) :: forest
  type(interaction_t), intent(inout) :: int
  type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
  if (present (lt_cm_to_lab)) then
     call int%set_momenta (lt_cm_to_lab * &
          phs_prt_get_momentum (forest%prt_out), outgoing=.true.)
  else
     call int%set_momenta (phs_prt_get_momentum (forest%prt_out), &
          outgoing=.true.)
  end if
end subroutine phs_forest_get_prt_out
```

Extract the outgoing particle momenta

⟨*PHS forests: public*⟩+≡

```
public :: phs_forest_get_momenta_out
```

⟨*PHS forests: procedures*⟩+≡

```
function phs_forest_get_momenta_out (forest, lt_cm_to_lab) result (p)
  type(phs_forest_t), intent(in) :: forest
  type(lorentz_transformation_t), intent(in), optional :: lt_cm_to_lab
  type(vector4_t), dimension(size (forest%prt_out)) :: p
  p = phs_prt_get_momentum (forest%prt_out)
  if (present (lt_cm_to_lab)) p = p * lt_cm_to_lab
end function phs_forest_get_momenta_out
```

18.6.10 Find equivalences among phase-space trees

Scan phase space for equivalences. We generate the complete set of unique permutations for the given list of outgoing particles, and use this for scanning equivalences within each grove. We scan all pairs of trees, using all permutations. This implies that trivial equivalences are included, and equivalences between different trees are recorded twice. This is intentional.

⟨*PHS forests: procedures*⟩+≡

```
subroutine phs_grove_set_equivalences (grove, perm_array)
  type(phs_grove_t), intent(inout) :: grove
  type(permutation_t), dimension(:), intent(in) :: perm_array
  type(equivalence_t), pointer :: eq
  integer :: t1, t2, i
  do t1 = 1, size (grove%tree)
     do t2 = 1, size (grove%tree)
        SCAN_PERM: do i = 1, size (perm_array)
           if (phs_tree_equivalent &
                (grove%tree(t1), grove%tree(t2), perm_array(i))) then
              call equivalence_list_add &
                   (grove%equivalence_list, t1, t2, perm_array(i))
              eq => grove%equivalence_list%last
              call phs_tree_find_msq_permutation &
                   (grove%tree(t1), grove%tree(t2), eq%perm, &
                    eq%msq_perm)
              call phs_tree_find_angle_permutation &
                   (grove%tree(t1), grove%tree(t2), eq%perm, &
                    eq%angle_perm, eq%angle_sig)
           end if
        end do SCAN_PERM
     end do
  end do
end subroutine phs_grove_set_equivalences
```

⟨*PHS forests: public*⟩+≡

```
public :: phs_forest_set_equivalences
```

⟨*PHS forests: procedures*⟩+≡

```
subroutine phs_forest_set_equivalences (forest)
  type(phs_forest_t), intent(inout) :: forest
  type(permutation_t), dimension(:), allocatable :: perm_array
  integer :: i
  call permutation_array_make &
       (perm_array, forest%flv(forest%n_in+1:)%get_pdg ())
```

```
  do i = 1, size (forest%grove)
     call phs_grove_set_equivalences (forest%grove(i), perm_array)
  end do
  forest%n_equivalences = sum (forest%grove%equivalence_list%length)
end subroutine phs_forest_set_equivalences
```

18.6.11 Interface for channel equivalences

Here, we store the equivalence list in the appropriate containers that the phs_base module provides. There is one separate list for each channel.

⟨*PHS forests: public*⟩+≡

```
public :: phs_forest_get_equivalences
```

⟨*PHS forests: procedures*⟩+≡

```
subroutine phs_forest_get_equivalences (forest, channel, azimuthal_dependence)
  type(phs_forest_t), intent(in) :: forest
  type(phs_channel_t), dimension(:), intent(out) :: channel
  logical, intent(in) :: azimuthal_dependence
  integer :: n_masses, n_angles
  integer :: mode_azimuthal_angle
  integer, dimension(:), allocatable :: n_eq
  type(equivalence_t), pointer :: eq
  integer, dimension(:), allocatable :: perm, mode
  integer :: g, c, j, left, right
  n_masses = forest%n_masses
  n_angles = forest%n_angles
  allocate (n_eq (forest%n_trees), source = 0)
  allocate (perm (forest%n_dimensions))
  allocate (mode (forest%n_dimensions), source = EQ_IDENTITY)
  do g = 1, size (forest%grove)
     eq => forest%grove(g)%equivalence_list%first
     do while (associated (eq))
        left = eq%left + forest%grove(g)%tree_count_offset
        n_eq(left) = n_eq(left) + 1
        eq => eq%next
     end do
  end do
  do c = 1, size (channel)
     allocate (channel(c)%eq (n_eq(c)))
     do j = 1, n_eq(c)
        call channel(c)%eq(j)%init (forest%n_dimensions)
     end do
  end do
  n_eq = 0
  if (azimuthal_dependence) then
     mode_azimuthal_angle = EQ_IDENTITY
  else
     mode_azimuthal_angle = EQ_INVARIANT
  end if
  do g = 1, size (forest%grove)
     eq => forest%grove(g)%equivalence_list%first
     do while (associated (eq))
        left = eq%left + forest%grove(g)%tree_count_offset
```

```
          right = eq%right + forest%grove(g)%tree_count_offset
          do j = 1, n_masses
             perm(j) = permute (j, eq%msq_perm)
             mode(j) = EQ_IDENTITY
          end do
          do j = 1, n_angles
             perm(n_masses+j) = n_masses + permute (j, eq%angle_perm)
             if (j == 1) then
                mode(n_masses+j) = mode_azimuthal_angle   ! first az. angle
             else if (mod(j,2) == 1) then
                mode(n_masses+j) = EQ_SYMMETRIC          ! other az. angles
             else if (eq%angle_sig(j)) then
                mode(n_masses+j) = EQ_IDENTITY           ! polar angle +
             else
                mode(n_masses+j) = EQ_INVERT             ! polar angle -
             end if
          end do
          n_eq(left) = n_eq(left) + 1
          associate (eq_cur => channel(left)%eq(n_eq(left)))
            eq_cur%c = right
            eq_cur%perm = perm
            eq_cur%mode = mode
          end associate
          eq => eq%next
       end do
    end do
  end subroutine phs_forest_get_equivalences
```

18.6.12 Phase-space evaluation

Given one row of the `x` parameter array and the corresponding channel index, compute first all relevant momenta and then recover the remainder of the `x` array, the Jacobians `phs_factor`, and the phase-space `volume`.

The output argument `ok` indicates whether this was successful.

⟨*PHS forests: public*⟩+≡

```
  public :: phs_forest_evaluate_selected_channel
```

⟨*PHS forests: procedures*⟩+≡

```
  subroutine phs_forest_evaluate_selected_channel &
       (forest, channel, active, sqrts, x, phs_factor, volume, ok)
    type(phs_forest_t), intent(inout) :: forest
    integer, intent(in) :: channel
    logical, dimension(:), intent(in) :: active
    real(default), intent(in) :: sqrts
    real(default), dimension(:,:), intent(inout) :: x
    real(default), dimension(:), intent(out) :: phs_factor
    real(default), intent(out) :: volume
    logical, intent(out) :: ok
    integer :: g, t
    integer(TC) :: k, k_root, k_in

    g = forest%grove_lookup (channel)
    t = channel - forest%grove(g)%tree_count_offset
```

```
    call phs_prt_set_undefined (forest%prt)
    call phs_prt_set_undefined (forest%prt_out)
    k_in = forest%n_tot

    do k = 1,forest%n_in
       forest%prt(ibset(0,k_in-k)) = forest%prt_in(k)
    end do

    do k = 1, forest%n_out
       call phs_prt_set_msq (forest%prt(ibset(0,k-1)), &
            forest%flv(forest%n_in+k)%get_mass () ** 2)
    end do


    k_root = 2**forest%n_out - 1
    select case (forest%n_in)
    case (1)
       forest%prt(k_root) = forest%prt_in(1)
    case (2)
       call phs_prt_combine &
            (forest%prt(k_root), forest%prt_in(1), forest%prt_in(2))
    end select
    call phs_tree_compute_momenta_from_x (forest%grove(g)%tree(t), &
         forest%prt, phs_factor(channel), volume, sqrts, x(:,channel), ok)
    if (ok) then
       do k = 1, forest%n_out
          forest%prt_out(k) = forest%prt(ibset(0,k-1))
       end do
    end if
  end subroutine phs_forest_evaluate_selected_channel
```

The remainder: recover x values for all channels except for the current channel.

NOTE: OpenMP not used for the first loop. `combine_particles` is not a channel-local operation.

⟨*PHS forests: public*⟩+≡

```
  public :: phs_forest_evaluate_other_channels
```

⟨*PHS forests: procedures*⟩+≡

```
  subroutine phs_forest_evaluate_other_channels &
       (forest, channel, active, sqrts, x, phs_factor, combine)
    type(phs_forest_t), intent(inout) :: forest
    integer, intent(in) :: channel
    logical, dimension(:), intent(in) :: active
    real(default), intent(in) :: sqrts
    real(default), dimension(:,:), intent(inout) :: x
    real(default), dimension(:), intent(inout) :: phs_factor
    logical, intent(in) :: combine
    integer :: g, t, ch, n_channel

    g = forest%grove_lookup (channel)
    t = channel - forest%grove(g)%tree_count_offset

    n_channel = forest%n_trees
    if (combine) then
```

```
      do ch = 1, n_channel
         if (ch == channel)  cycle
         if (active(ch)) then
            g = forest%grove_lookup(ch)
            t = ch - forest%grove(g)%tree_count_offset
            call phs_tree_combine_particles &
                 (forest%grove(g)%tree(t), forest%prt)
         end if
      end do
   end if

   !OMP PARALLEL PRIVATE (g,t,ch) SHARED(active,forest,sqrts,x,channel)
   !OMP DO SCHEDULE(STATIC)
   do ch = 1, n_channel
      if (ch == channel)  cycle
      if (active(ch)) then
         g = forest%grove_lookup(ch)
         t = ch - forest%grove(g)%tree_count_offset
         call phs_tree_compute_x_from_momenta &
              (forest%grove(g)%tree(t), &
              forest%prt, phs_factor(ch), sqrts, x(:,ch))
      end if
   end do
   !OMP END DO
   !OMP END PARALLEL

end subroutine phs_forest_evaluate_other_channels
```

The complement: recover one row of the `x` array and the associated Jacobian entry, corresponding to `channel`, from incoming and outgoing momenta. Also compute the phase-space volume.

⟨*PHS forests: public*⟩+≡

```
public :: phs_forest_recover_channel
```

⟨*PHS forests: procedures*⟩+≡

```
subroutine phs_forest_recover_channel &
     (forest, channel, sqrts, x, phs_factor, volume)
  type(phs_forest_t), intent(inout) :: forest
  integer, intent(in) :: channel
  real(default), intent(in) :: sqrts
  real(default), dimension(:,:), intent(inout) :: x
  real(default), dimension(:), intent(inout) :: phs_factor
  real(default), intent(out) :: volume
  integer :: g, t
  integer(TC) :: k, k_in
  g = forest%grove_lookup (channel)
  t = channel - forest%grove(g)%tree_count_offset
  call phs_prt_set_undefined (forest%prt)
  k_in = forest%n_tot
  forall (k = 1:forest%n_in)
     forest%prt(ibset(0,k_in-k)) = forest%prt_in(k)
  end forall
  forall (k = 1:forest%n_out)
     forest%prt(ibset(0,k-1)) = forest%prt_out(k)
```

```
    end forall
    call phs_forest_combine_particles (forest)
    call phs_tree_compute_volume &
         (forest%grove(g)%tree(t), sqrts, volume)
    call phs_tree_compute_x_from_momenta &
         (forest%grove(g)%tree(t), &
         forest%prt, phs_factor(channel), sqrts, x(:,channel))
  end subroutine phs_forest_recover_channel
```

18.6.13 Unit tests

Test module, followed by the corresponding implementation module.

⟨phs_forests_ut.f90⟩≡

```
⟨File header⟩

module phs_forests_ut
  use unit_tests
  use phs_forests_uti

⟨Standard module head⟩

⟨PHS forests: public test⟩

contains

⟨PHS forests: test driver⟩

end module phs_forests_ut
```

⟨phs_forests_uti.f90⟩≡

```
⟨File header⟩

module phs_forests_uti

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_defs, only: FMT_12
  use lorentz
  use flavors
  use interactions
  use model_data
  use mappings
  use phs_base
  use resonances, only: resonance_history_t

  use phs_forests

⟨Standard module head⟩

⟨PHS forests: test declarations⟩

contains
```

⟨*PHS forests: tests*⟩

```
end module phs_forests_uti
```

API: driver for the unit tests below.

⟨*PHS forests: public test*⟩≡

```
public :: phs_forests_test
```

⟨*PHS forests: test driver*⟩≡

```
subroutine phs_forests_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨PHS forests: execute tests⟩
end subroutine phs_forests_test
```

Basic universal test

Write a possible phase-space file for a $2 \to 3$ process and make the corresponding forest, print the forest. Choose some in-particle momenta and a random-number array and evaluate out-particles and phase-space factors.

⟨*PHS forests: execute tests*⟩≡

```
call test (phs_forest_1, "phs_forest_1", &
     "check phs forest setup", &
     u, results)
```

⟨*PHS forests: test declarations*⟩≡

```
public :: phs_forest_1
```

⟨*PHS forests: tests*⟩≡

```
subroutine phs_forest_1 (u)
  use os_interface
  integer, intent(in) :: u
  type(phs_forest_t) :: forest
  type(phs_channel_t), dimension(:), allocatable :: channel
  type(model_data_t), target :: model
  type(string_t) :: process_id
  type(flavor_t), dimension(5) :: flv
  type(string_t) :: filename
  type(interaction_t) :: int
  integer :: unit_fix
  type(mapping_defaults_t) :: mapping_defaults
  logical :: found_process, ok
  integer :: n_channel, ch, i
  logical, dimension(4) :: active = .true.
  real(default) :: sqrts = 1000
  real(default), dimension(5,4) :: x
  real(default), dimension(4) :: factor
  real(default) :: volume

  write (u, "(A)")  "* Test output: PHS forest"
  write (u, "(A)")  "*   Purpose: test PHS forest routines"
  write (u, "(A)")
```

```
write (u, "(A)")  "* Reading model file"

call model%init_sm_test ()

write (u, "(A)")
write (u, "(A)")  "* Create phase-space file 'phs_forest_test.phs'"
write (u, "(A)")

call flv%init ([11, -11, 11, -11, 22], model)
unit_fix = free_unit ()
open (file="phs_forest_test.phs", unit=unit_fix, action="write")
write (unit_fix, *) "process foo"
write (unit_fix, *) 'md5sum_process    = "6ABA33BC2927925D0F073B1C1170780A"'
write (unit_fix, *) 'md5sum_model_par  = "1A0B151EE6E2DEB92D880320355A3EAB"'
write (unit_fix, *) 'md5sum_phs_config = "B6A8877058809A8BDD54753CDAB83ACE"'
write (unit_fix, *) "sqrts         =    100.00000000000000"
write (unit_fix, *) "m_threshold_s =    50.000000000000000"
write (unit_fix, *) "m_threshold_t =    100.00000000000000"
write (unit_fix, *) "off_shell =            2"
write (unit_fix, *) "t_channel =            6"
write (unit_fix, *) "keep_nonresonant =  F"
write (unit_fix, *) ""
write (unit_fix, *) "  grove"
write (unit_fix, *) "    tree 3 7"
write (unit_fix, *) "      map 3 s_channel 23"
write (unit_fix, *) "    tree 5 7"
write (unit_fix, *) "    tree 6 7"
write (unit_fix, *) "  grove"
write (unit_fix, *) "    tree 9 11"
write (unit_fix, *) "      map 9 t_channel 22"
close (unit_fix)

write (u, "(A)")
write (u, "(A)")  "* Read phase-space file 'phs_forest_test.phs'"

call syntax_phs_forest_init ()
process_id = "foo"
filename = "phs_forest_test.phs"
call phs_forest_read &
     (forest, filename, process_id, 2, 3, model, found_process)

write (u, "(A)")
write (u, "(A)")  "* Set parameters, flavors, equiv, momenta"
write (u, "(A)")

call phs_forest_set_flavors (forest, flv)
call phs_forest_set_parameters (forest, mapping_defaults, .false.)
call phs_forest_setup_prt_combinations (forest)
call phs_forest_set_equivalences (forest)
call int%basic_init (2, 0, 3)
call int%set_momentum &
     (vector4_moving (500._default, 500._default, 3), 1)
call int%set_momentum &
```

```
      (vector4_moving (500._default,-500._default, 3), 2)
   call phs_forest_set_prt_in (forest, int)
   n_channel = 2
   x = 0
   x(:,n_channel) = [0.3, 0.4, 0.1, 0.9, 0.6]
   write (u, "(A)")  "   Input values:"
   write (u, "(3x,5(1x," // FMT_12 // "))")  x(:,n_channel)

   write (u, "(A)")
   write (u, "(A)")  "* Evaluating phase space"

   call phs_forest_evaluate_selected_channel (forest, &
      n_channel, active, sqrts, x, factor, volume, ok)
   call phs_forest_evaluate_other_channels (forest, &
      n_channel, active, sqrts, x, factor, combine=.true.)
   call phs_forest_get_prt_out (forest, int)
   write (u, "(A)")  "   Output values:"
   do ch = 1, 4
      write (u, "(3x,5(1x," // FMT_12 // "))")  x(:,ch)
   end do
   call int%basic_write (u)
   write (u, "(A)")  "   Factors:"
   write (u, "(3x,5(1x," // FMT_12 // "))")  factor
   write (u, "(A)")  "   Volume:"
   write (u, "(3x,5(1x," // FMT_12 // "))")  volume
   call phs_forest_write (forest, u)

   write (u, "(A)")
   write (u, "(A)")  "* Compute equivalences"

   n_channel = 4
   allocate (channel (n_channel))
   call phs_forest_get_equivalences (forest, &
      channel, .true.)
   do i = 1, n_channel
      write (u, "(1x,I0,':')", advance = "no")  ch
      call channel(i)%write (u)
   end do

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call model%final ()
   call phs_forest_final (forest)
   call syntax_phs_forest_final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: phs_forest_1"

end subroutine phs_forest_1
```

Resonance histories

Read a suitably nontrivial forest from file and recover the set of resonance histories.

⟨*PHS forests: execute tests*⟩+≡
```
call test (phs_forest_2, "phs_forest_2", &
     "handle phs forest resonance content", &
     u, results)
```

⟨*PHS forests: test declarations*⟩+≡
```
public :: phs_forest_2
```

⟨*PHS forests: tests*⟩+≡
```
subroutine phs_forest_2 (u)
  use os_interface
  integer, intent(in) :: u
  integer :: unit_fix
  type(phs_forest_t) :: forest
  type(model_data_t), target :: model
  type(string_t) :: process_id
  type(string_t) :: filename
  logical :: found_process
  type(resonance_history_t), dimension(:), allocatable :: res_history
  integer :: i

  write (u, "(A)")  "* Test output: phs_forest_2"
  write (u, "(A)")  "*   Purpose: test PHS forest routines"
  write (u, "(A)")

  write (u, "(A)")  "* Reading model file"

  call model%init_sm_test ()

  write (u, "(A)")
  write (u, "(A)")  "* Create phase-space file 'phs_forest_2.phs'"
  write (u, "(A)")

  unit_fix = free_unit ()
  open (file="phs_forest_2.phs", unit=unit_fix, action="write")
  write (unit_fix, *) "process foo"
  write (unit_fix, *) 'md5sum_process    = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"'
  write (unit_fix, *) 'md5sum_model_par  = "1A0B151EE6E2DEB92D880320355A3EAB"'
  write (unit_fix, *) 'md5sum_phs_config = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"'
  write (unit_fix, *) "sqrts         =    100.00000000000000"
  write (unit_fix, *) "m_threshold_s =    50.000000000000000"
  write (unit_fix, *) "m_threshold_t =    100.00000000000000"
  write (unit_fix, *) "off_shell =            2"
  write (unit_fix, *) "t_channel =            6"
  write (unit_fix, *) "keep_nonresonant =  F"
  write (unit_fix, *) ""
  write (unit_fix, *) "  grove"
  write (unit_fix, *) "    tree 3 7"
  write (unit_fix, *) "    tree 3 7"
  write (unit_fix, *) "      map 3 s_channel -24"
  write (unit_fix, *) "    tree 5 7"
```

```
  write (unit_fix, *) "    tree 3 7"
  write (unit_fix, *) "      map 3 s_channel -24"
  write (unit_fix, *) "      map 7 s_channel 23"
  write (unit_fix, *) "    tree 5 7"
  write (unit_fix, *) "      map 7 s_channel 25"
  write (unit_fix, *) "    tree 3 11"
  write (unit_fix, *) "      map 3 s_channel -24"
  close (unit_fix)

  write (u, "(A)")  "* Read phase-space file 'phs_forest_2.phs'"

  call syntax_phs_forest_init ()
  process_id = "foo"
  filename = "phs_forest_2.phs"
  call phs_forest_read &
       (forest, filename, process_id, 2, 3, model, found_process)

  write (u, "(A)")
  write (u, "(A)")  "* Extract resonance history set"

  call forest%extract_resonance_histories (res_history)
  do i = 1, size (res_history)
     write (u, *)
     call res_history(i)%write (u)
  end do

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call model%final ()
  call phs_forest_final (forest)
  call syntax_phs_forest_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: phs_forest_2"

end subroutine phs_forest_2
```

18.7 Finding phase space parameterizations

If the phase space configuration is not found in the appropriate file, we should generate one.

The idea is to construct all Feynman diagrams subject to certain constraints which eliminate everything that is probably irrelevant for the integration. These Feynman diagrams (cascades) are grouped in groves by finding equivalence classes related by symmetry and ordered with respect to their importance (resonances). Finally, the result (or part of it) is written to file and used for the integration.

This module may eventually disappear and be replaced by CAML code. In particular, we need here a set of Feynman rules (vertices with particle codes, but not the factors). Thus, the module works for the Standard Model only.

Note that this module is stand-alone, it communicates to the main program only via the generated ASCII phase-space configuration file.

⟨cascades.f90⟩≡

```
⟨File header⟩

module cascades

⟨Use kinds⟩
  use kinds, only: TC, i8, i32
⟨Use strings⟩
  use io_units
  use constants, only: one
  use format_defs, only: FMT_12, FMT_19
  use numeric_utils
  use diagnostics
  use hashes
  use sorting
  use physics_defs, only: SCALAR, SPINOR, VECTOR, VECTORSPINOR, TENSOR
  use physics_defs, only: UNDEFINED
  use model_data
  use flavors
  use lorentz

  use resonances, only: resonance_info_t
  use resonances, only: resonance_history_t
  use resonances, only: resonance_history_set_t
  use phs_forests

⟨Standard module head⟩

⟨Cascades: public⟩

⟨Cascades: parameters⟩

⟨Cascades: types⟩

⟨Cascades: interfaces⟩

contains

⟨Cascades: procedures⟩

end module cascades
```

18.7.1 The mapping modes

The valid mapping modes, to be used below. We will make use of the convention that mappings of internal particles have a positive value. Only for positive values, the flavor code is propagated when combining cascades.

⟨Mapping modes⟩≡

```
integer, parameter :: &
     & EXTERNAL_PRT = -1, &
     & NO_MAPPING = 0, S_CHANNEL = 1, T_CHANNEL =  2, U_CHANNEL = 3, &
```

```
& RADIATION = 4, COLLINEAR = 5, INFRARED = 6, &
& STEP_MAPPING_E = 11, STEP_MAPPING_H = 12, &
& ON_SHELL = 99
```

⟨*Cascades: parameters*⟩≡
  ⟨*Mapping modes*⟩

18.7.2 The cascade type

A cascade is essentially the same as a decay tree (both definitions may be merged in a later version). It contains a linked tree of nodes, each of which representing an internal particle. In contrast to decay trees, each node has a definite particle code. These nodes need not be modified, therefore we can use pointers and do not have to copy them. Thus, physically each cascades has only a single node, the mother particle. However, to be able to compare trees quickly, we store in addition an array of binary codes which is always sorted in ascending order. This is accompanied by a corresponding list of particle codes. The index is the location of the corresponding cascade in the cascade set, this may be used to access the daughters directly.

The real mass is the particle mass belonging to the particle code. The minimal mass is the sum of the real masses of all its daughters; this is the kinematical cutoff. The effective mass may be zero if the particle mass is below a certain threshold; it may be the real mass if the particle is resonant; or it may be some other value.

The logical `t_channel` is set if this a t-channel line, while `initial` is true only for an initial particle. Note that both initial particles are also `t_channel` by definition, and that they are distinguished by the direction of the tree: One of them decays and is the root of the tree, while the other one is one of the leaves.

The cascade is a list of nodes (particles) which are linked via the `daughter` entries. The node is the mother particle of the decay cascade. Much of the information in the nodes is repeated in arrays, to be accessible more easily. The arrays will be kept sorted by binary codes.

The counter `n_off_shell` is increased for each internal line that is neither resonant nor log-enhanced. It is set to zero if the current line is resonant, since this implies on-shell particle production and subsequent decay.

The counter `n_t_channel` is non-negative once an initial particle is included in the tree: then, it counts the number of t-channel lines.

The `multiplicity` is the number of branchings to follow until all daughters are on-shell. A resonant or non-decaying particle has multiplicity one. Merging nodes, the multiplicities add unless the mother is a resonance. An initial or final node has multiplicity zero.

The arrays correspond to the subnode tree `tree` of the current cascade. PDG codes are stored only for those positions which are resonant, with the exception of the last entry, i.e., the current node. Other positions, in particular external legs, are assigned undefined PDG code.

A cascade is uniquely identified by its tree, the tree of PDG codes, and the tree of mappings. The tree of resonances is kept only to mask the PDG tree as described above.

⟨*Cascades: types*⟩≡

```
type :: cascade_t
   private
   ! counters
   integer :: index = 0
   integer :: grove = 0
   ! status
   logical :: active = .false.
   logical :: complete = .false.
   logical :: incoming = .false.
   ! this node
   integer(TC) :: bincode = 0
   type(flavor_t) :: flv
   integer :: pdg = UNDEFINED
   logical :: is_vector = .false.
   real(default) :: m_min = 0
   real(default) :: m_rea = 0
   real(default) :: m_eff = 0
   integer :: mapping = NO_MAPPING
   logical :: on_shell = .false.
   logical :: resonant = .false.
   logical :: log_enhanced = .false.
   logical :: t_channel = .false.
   ! global tree properties
   integer :: multiplicity = 0
   integer :: internal = 0
   integer :: n_off_shell = 0
   integer :: n_resonances = 0
   integer :: n_log_enhanced = 0
   integer :: n_t_channel = 0
   integer :: res_hash = 0
   ! the sub-node tree
   integer :: depth = 0
   integer(TC), dimension(:), allocatable :: tree
   integer, dimension(:), allocatable :: tree_pdg
   integer, dimension(:), allocatable :: tree_mapping
   logical, dimension(:), allocatable :: tree_resonant
   ! branch connections
   logical :: has_children = .false.
   type(cascade_t), pointer :: daughter1 => null ()
   type(cascade_t), pointer :: daughter2 => null ()
   type(cascade_t), pointer :: mother => null ()
   ! next in list
   type(cascade_t), pointer :: next => null ()
 contains
 ⟨Cascades: cascade: TBP⟩
end type cascade_t
```

⟨*Cascades: procedures*⟩≡

```
subroutine cascade_init (cascade, depth)
  type(cascade_t), intent(out) :: cascade
  integer, intent(in) :: depth
  integer, save :: index = 0
  index = cascade_index ()
  cascade%index = index
```

```
    cascade%depth = depth
    cascade%active = .true.
    allocate (cascade%tree (depth))
    allocate (cascade%tree_pdg (depth))
    allocate (cascade%tree_mapping (depth))
    allocate (cascade%tree_resonant (depth))
  end subroutine cascade_init
```

Keep and increment a global index

⟨*Cascades: procedures*⟩+≡

```
  function cascade_index (seed) result (index)
    integer :: index
    integer, intent(in), optional :: seed
    integer, save :: i = 0
    if (present (seed))  i = seed
    i = i + 1
    index = i
  end function cascade_index
```

We need three versions of writing cascades. This goes to the phase-space file.

For t/u channel mappings, we use the absolute value of the PDG code.

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_write_file_format (cascade, model, unit)
    type(cascade_t), intent(in) :: cascade
    class(model_data_t), intent(in), target :: model
    integer, intent(in), optional :: unit
    type(flavor_t) :: flv
    integer :: u, i
2   format(3x,A,1x,I3,1x,A,1x,I7,1x,'!',1x,A)
    u = given_output_unit (unit);  if (u < 0)  return
    call write_reduced (cascade%tree, u)
    write (u, "(A)")
    do i = 1, cascade%depth
       call flv%init (cascade%tree_pdg(i), model)
       select case (cascade%tree_mapping(i))
       case (NO_MAPPING, EXTERNAL_PRT)
       case (S_CHANNEL)
          write(u,2) 'map', &
               cascade%tree(i), 's_channel', cascade%tree_pdg(i), &
               char (flv%get_name ())
       case (T_CHANNEL)
          write(u,2) 'map', &
               cascade%tree(i), 't_channel', abs (cascade%tree_pdg(i)), &
               char (flv%get_name ())
       case (U_CHANNEL)
          write(u,2) 'map', &
               cascade%tree(i), 'u_channel', abs (cascade%tree_pdg(i)), &
               char (flv%get_name ())
       case (RADIATION)
          write(u,2) 'map', &
               cascade%tree(i), 'radiation', cascade%tree_pdg(i), &
               char (flv%get_name ())
       case (COLLINEAR)
          write(u,2) 'map', &
```

```
            cascade%tree(i), 'collinear', cascade%tree_pdg(i), &
            char (flv%get_name ())
      case (INFRARED)
         write(u,2) 'map', &
            cascade%tree(i), 'infrared ', cascade%tree_pdg(i), &
            char (flv%get_name ())
      case (ON_SHELL)
         write(u,2) 'map', &
            cascade%tree(i), 'on_shell ', cascade%tree_pdg(i), &
            char (flv%get_name ())
      case default
         call msg_bug (" Impossible mapping mode encountered")
      end select
   end do
contains
   subroutine write_reduced (array, unit)
      integer(TC), dimension(:), intent(in) :: array
      integer, intent(in) :: unit
      integer :: i
      write (u, "(3x,A,1x)", advance="no")  "tree"
      do i = 1, size (array)
         if (decay_level (array(i)) > 1) then
            write (u, "(1x,I0)", advance="no")  array(i)
         end if
      end do
   end subroutine write_reduced

   elemental function decay_level (k) result (l)
      integer(TC), intent(in) :: k
      integer :: l
      integer :: i
      l = 0
      do i = 0, bit_size(k) - 1
         if (btest(k,i)) l = l + 1
      end do
   end function decay_level
   subroutine start_comment (u)
      integer, intent(in) :: u
      write(u, '(1x,A)', advance='no') '!'
   end subroutine start_comment
end subroutine cascade_write_file_format
```

This creates metapost source for graphical display:

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_write_graph_format (cascade, count, unit)
   type(cascade_t), intent(in) :: cascade
   integer, intent(in) :: count
   integer, intent(in), optional :: unit
   integer :: u
   integer(TC) :: mask
   type(string_t) :: left_str, right_str
   u = given_output_unit (unit);  if (u < 0)  return
   mask = 2**((cascade%depth+3)/2) - 1
   left_str = ""
```

```
  right_str = ""
  write (u, '(A)') "\begin{minipage}{105pt}"
  write (u, '(A)') "\vspace{30pt}"
  write (u, '(A)') "\begin{center}"
  write (u, '(A)') "\begin{fmfgraph*}(55,55)"
  call graph_write (cascade, mask)
  write (u, '(A)') "\fmfleft{" // char (extract (left_str, 2)) // "}"
  write (u, '(A)') "\fmfright{" // char (extract (right_str, 2)) // "}"
  write (u, '(A)') "\end{fmfgraph*}\\"
  write (u, '(A,I5,A)') "\fbox{$", count, "$}"
  write (u, '(A)') "\end{center}"
  write (u, '(A)') "\end{minipage}"
  write (u, '(A)') "%"
contains
  recursive subroutine graph_write (cascade, mask, reverse)
    type(cascade_t), intent(in) :: cascade
    integer(TC), intent(in) :: mask
    logical, intent(in), optional :: reverse
    type(flavor_t) :: anti
    logical :: rev
    rev = .false.;  if (present(reverse))  rev = reverse
    if (cascade%has_children) then
       if (.not.rev) then
          call vertex_write (cascade, cascade%daughter1, mask)
          call vertex_write (cascade, cascade%daughter2, mask)
       else
          call vertex_write (cascade, cascade%daughter2, mask, .true.)
          call vertex_write (cascade, cascade%daughter1, mask, .true.)
       end if
       if (cascade%complete) then
          call vertex_write (cascade, cascade%mother, mask, .true.)
          write (u, '(A,I0,A)') "\fmfv{d.shape=square}{v0}"
       end if
    else
       if (cascade%incoming) then
          anti = cascade%flv%anti ()
          call external_write (cascade%bincode, anti%get_tex_name (), &
               left_str)
       else
          call external_write (cascade%bincode, cascade%flv%get_tex_name (), &
               right_str)
       end if
    end if
  end subroutine graph_write
  recursive subroutine vertex_write (cascade, daughter, mask, reverse)
    type(cascade_t), intent(in) :: cascade, daughter
    integer(TC), intent(in) :: mask
    logical, intent(in), optional :: reverse
    integer :: bincode
    if (cascade%complete) then
       bincode = 0
    else
       bincode = cascade%bincode
    end if
```

```
  call graph_write (daughter, mask, reverse)
  if (daughter%has_children) then
     call line_write (bincode, daughter%bincode, daughter%flv, &
          mapping=daughter%mapping)
  else
     call line_write (bincode, daughter%bincode, daughter%flv)
  end if
end subroutine vertex_write
subroutine line_write (i1, i2, flv, mapping)
  integer(TC), intent(in) :: i1, i2
  type(flavor_t), intent(in) :: flv
  integer, intent(in), optional :: mapping
  integer :: k1, k2
  type(string_t) :: prt_type
  select case (flv%get_spin_type ())
  case (SCALAR);       prt_type = "plain"
  case (SPINOR);       prt_type = "fermion"
  case (VECTOR);       prt_type = "boson"
  case (VECTORSPINOR); prt_type = "fermion"
  case (TENSOR);       prt_type = "dbl_wiggly"
  case default;        prt_type = "dashes"
  end select
  if (flv%is_antiparticle ()) then
     k1 = i2;  k2 = i1
  else
     k1 = i1;  k2 = i2
  end if
  if (present (mapping)) then
     select case (mapping)
     case (S_CHANNEL)
        write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
             & ",f=blue,lab=\sm\blue$" // &
             & char (flv%get_tex_name ()) // "$}" // &
             & "{v", k1, ",v", k2, "}"
     case (T_CHANNEL, U_CHANNEL)
        write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
             & ",f=cyan,lab=\sm\cyan$" // &
             & char (flv%get_tex_name ()) // "$}" // &
             & "{v", k1, ",v", k2, "}"
     case (RADIATION)
        write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
             & ",f=green,lab=\sm\green$" // &
             & char (flv%get_tex_name ()) // "$}" // &
             & "{v", k1, ",v", k2, "}"
     case (COLLINEAR)
        write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
             & ",f=magenta,lab=\sm\magenta$" // &
             & char (flv%get_tex_name ()) // "$}" // &
             & "{v", k1, ",v", k2, "}"
     case (INFRARED)
        write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
             & ",f=red,lab=\sm\red$" // &
             & char (flv%get_tex_name ()) // "$}" // &
             & "{v", k1, ",v", k2, "}"
```

```
        case default
           write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
                & ",f=black}" // &
                & "{v", k1, ",v", k2, "}"
        end select
      else
        write (u, '(A,I0,A,I0,A)') "\fmf{" // char (prt_type) // &
                & "}" // &
                & "{v", k1, ",v", k2, "}"
      end if
    end subroutine line_write
    subroutine external_write (bincode, name, ext_str)
      integer(TC), intent(in) :: bincode
      type(string_t), intent(in) :: name
      type(string_t), intent(inout) :: ext_str
      character(len=20) :: str
      write (str, '(A2,I0)') ",v", bincode
      ext_str = ext_str // trim (str)
      write (u, '(A,I0,A,I0,A)') "\fmflabel{\sm$" &
        // char (name) &
        // "\,(", bincode, ")" &
        // "$}{v", bincode, "}"
    end subroutine external_write
  end subroutine cascade_write_graph_format
```

This is for screen/debugging output:

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_write (cascade, unit)
    type(cascade_t), intent(in) :: cascade
    integer, intent(in), optional :: unit
    integer :: u
    character(9) :: depth
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(A,(1x,I7))") 'Cascade #', cascade%index
    write (u, "(A,(1x,I7))") '  Grove:       #', cascade%grove
    write (u, "(A,3(1x,L1))") '  act/cmp/inc:  ', &
         cascade%active, cascade%complete, cascade%incoming
    write (u, "(A,I0)") '  Bincode:      ', cascade%bincode
    write (u, "(A)", advance="no") '  Flavor:       '
    call cascade%flv%write (unit)
    write (u, "(A,I9)") '  Active flavor:', cascade%pdg
    write (u, "(A,L1)") '  Is vector:    ', cascade%is_vector
    write (u, "(A,3(1x," // FMT_19 // "))") '  Mass (m/r/e): ', &
         cascade%m_min, cascade%m_rea, cascade%m_eff
    write (u, "(A,I1)") '  Mapping:      ', cascade%mapping
    write (u, "(A,3(1x,L1))") '  res/log/tch:  ', &
         cascade%resonant, cascade%log_enhanced, cascade%t_channel
    write (u, "(A,(1x,I7))") '  Multiplicity: ', cascade%multiplicity
    write (u, "(A,2(1x,I7))") '  n intern/off: ', &
         cascade%internal, cascade%n_off_shell
    write (u, "(A,3(1x,I7))") '  n res/log/tch:', &
         cascade%n_resonances, cascade%n_log_enhanced, cascade%n_t_channel
    write (u, "(A,I7)") '  Depth:        ', cascade%depth
    write (depth, "(I7)") cascade%depth
```

```
    write (u, "(A," // depth // "(1x,I7))") &
       '  Tree:         ', cascade%tree
    write (u, "(A," // depth // "(1x,I7))") &
       '  Tree(PDG):    ', cascade%tree_pdg
    write (u, "(A," // depth // "(1x,I7))") &
       '  Tree(mapping):', cascade%tree_mapping
    write (u, "(A," // depth // "(1x,L1))") &
       '  Tree(res):    ', cascade%tree_resonant
    if (cascade%has_children) then
       write (u, "(A,I7,1x,I7)") '  Daughter1/2:  ', &
            cascade%daughter1%index, cascade%daughter2%index
    end if
    if (associated (cascade%mother)) then
       write (u, "(A,I7)") '  Mother:       ', cascade%mother%index
    end if
  end subroutine cascade_write
```

18.7.3 Creating new cascades

This initializes a single-particle cascade (external, final state). The PDG entry in the tree is set undefined because the cascade is not resonant. However, the flavor entry is set, so the cascade flavor is identified nevertheless.

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_init_outgoing (cascade, flv, pos, m_thr)
    type(cascade_t), intent(out) :: cascade
    type(flavor_t), intent(in) :: flv
    integer, intent(in) :: pos
    real(default), intent(in) :: m_thr
    call cascade_init (cascade, 1)
    cascade%bincode = ibset (0_TC, pos-1)
    cascade%flv = flv
    cascade%pdg = cascade%flv%get_pdg ()
    cascade%is_vector = flv%get_spin_type () == VECTOR
    cascade%m_min = flv%get_mass ()
    cascade%m_rea = cascade%m_min
    if (cascade%m_rea >= m_thr) then
       cascade%m_eff = cascade%m_rea
    end if
    cascade%on_shell = .true.
    cascade%multiplicity = 1
    cascade%tree(1) = cascade%bincode
    cascade%tree_pdg(1) = cascade%pdg
    cascade%tree_mapping(1) = EXTERNAL_PRT
    cascade%tree_resonant(1) = .false.
  end subroutine cascade_init_outgoing
```

The same for an incoming line:

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_init_incoming (cascade, flv, pos, m_thr)
    type(cascade_t), intent(out) :: cascade
    type(flavor_t), intent(in) :: flv
    integer, intent(in) :: pos
```

```
    real(default), intent(in) :: m_thr
    call cascade_init (cascade, 1)
    cascade%incoming = .true.
    cascade%bincode = ibset (0_TC, pos-1)
    cascade%flv = flv%anti ()
    cascade%pdg = cascade%flv%get_pdg ()
    cascade%is_vector = flv%get_spin_type () == VECTOR
    cascade%m_min = flv%get_mass ()
    cascade%m_rea = cascade%m_min
    if (cascade%m_rea >= m_thr) then
       cascade%m_eff = cascade%m_rea
    end if
    cascade%on_shell = .true.
    cascade%n_t_channel = 0
    cascade%n_off_shell = 0
    cascade%tree(1) = cascade%bincode
    cascade%tree_pdg(1) = cascade%pdg
    cascade%tree_mapping(1) = EXTERNAL_PRT
    cascade%tree_resonant(1) = .false.
  end subroutine cascade_init_incoming
```

18.7.4 Tools

This function returns true if the two cascades share no common external particle. This is a requirement for joining them.

⟨*Cascades: interfaces*⟩≡

```
  interface operator(.disjunct.)
     module procedure cascade_disjunct
  end interface
```

⟨*Cascades: procedures*⟩+≡

```
  function cascade_disjunct (cascade1, cascade2) result (flag)
    logical :: flag
    type(cascade_t), intent(in) :: cascade1, cascade2
    flag = iand (cascade1%bincode, cascade2%bincode) == 0
  end function cascade_disjunct
```

Compute a hash code for the resonance pattern of a cascade. We count the number of times each particle appears as a resonance.

We pack the PDG codes of the resonances in two arrays (s-channel and t-channel), sort them both, concatenate the results, transfer to `i8` integers, and compute the hash code from this byte stream.

For t/u-channel, we remove the sign for antiparticles since this is not well-defined.

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_assign_resonance_hash (cascade)
    type(cascade_t), intent(inout) :: cascade
    integer(i8), dimension(1) :: mold
    cascade%res_hash = hash (transfer &
         (concat (sort (pack (cascade%tree_pdg, &
                              cascade%tree_resonant)), &
```

```
                 sort (pack (abs (cascade%tree_pdg), &
                             cascade%tree_mapping == T_CHANNEL .or. &
                             cascade%tree_mapping == U_CHANNEL))), &
          mold))
end subroutine cascade_assign_resonance_hash
```

18.7.5 Hash entries for cascades

We will set up a hash array which contains keys of and pointers to cascades. We hold a list of cascade (pointers) within each bucket. This is not for collision resolution, but for keeping similar, but unequal cascades together.

⟨*Cascades: types*⟩+≡

```
type :: cascade_p
   type(cascade_t), pointer :: cascade => null ()
   type(cascade_p), pointer :: next => null ()
end type cascade_p
```

Here is the bucket or hash entry type:

⟨*Cascades: types*⟩+≡

```
type :: hash_entry_t
   integer(i32) :: hashval = 0
   integer(i8), dimension(:), allocatable :: key
   type(cascade_p), pointer :: first => null ()
   type(cascade_p), pointer :: last => null ()
end type hash_entry_t
```

⟨*Cascades: public*⟩≡

```
public :: hash_entry_init
```

⟨*Cascades: procedures*⟩+≡

```
subroutine hash_entry_init (entry, entry_in)
  type(hash_entry_t), intent(out) :: entry
  type(hash_entry_t), intent(in) :: entry_in
  type(cascade_p), pointer :: casc_iter, casc_copy
  entry%hashval = entry_in%hashval
  entry%key = entry_in%key
  casc_iter => entry_in%first
  do while (associated (casc_iter))
     allocate (casc_copy)
     casc_copy = casc_iter
     casc_copy%next => null ()
     if (associated (entry%first)) then
        entry%last%next => casc_copy
     else
        entry%first => casc_copy
     end if
     entry%last => casc_copy
     casc_iter => casc_iter%next
  end do
end subroutine hash_entry_init
```

Finalize: just deallocate the list; the contents are just pointers.

⟨*Cascades: procedures*⟩+≡

```
subroutine hash_entry_final (hash_entry)
  type(hash_entry_t), intent(inout) :: hash_entry
  type(cascade_p), pointer :: current
  do while (associated (hash_entry%first))
     current => hash_entry%first
     hash_entry%first => current%next
     deallocate (current)
  end do
end subroutine hash_entry_final
```

Output: concise format for debugging, just list cascade indices.

⟨*Cascades: procedures*⟩+≡

```
subroutine hash_entry_write (hash_entry, unit)
  type(hash_entry_t), intent(in) :: hash_entry
  integer, intent(in), optional :: unit
  type(cascade_p), pointer :: current
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)", advance="no")  "Entry:"
  do i = 1, size (hash_entry%key)
     write (u, "(1x,I0)", advance="no")  hash_entry%key(i)
  end do
  write (u, "(1x,A)", advance="no")  "->"
  current => hash_entry%first
  do while (associated (current))
     write (u, "(1x,I7)", advance="no") current%cascade%index
     current => current%next
  end do
  write (u, *)
end subroutine hash_entry_write
```

This function adds a cascade pointer to the bucket. If `ok` is present, check first if it is already there and return failure if yes. If `cascade_ptr` is also present, set it to the current cascade if successful. If not, set it to the cascade that is already there.

⟨*Cascades: procedures*⟩+≡

```
subroutine hash_entry_add_cascade_ptr (hash_entry, cascade, ok, cascade_ptr)
  type(hash_entry_t), intent(inout) :: hash_entry
  type(cascade_t), intent(in), target :: cascade
  logical, intent(out), optional :: ok
  type(cascade_t), optional, pointer :: cascade_ptr
  type(cascade_p), pointer :: current
  if (present (ok)) then
     call hash_entry_check_cascade (hash_entry, cascade, ok, cascade_ptr)
     if (.not. ok)  return
  end if
  allocate (current)
  current%cascade => cascade
  if (associated (hash_entry%last)) then
     hash_entry%last%next => current
```

```
    else
       hash_entry%first => current
    end if
    hash_entry%last => current
  end subroutine hash_entry_add_cascade_ptr
```

This function checks whether a cascade is already in the bucket. For incomplete cascades, we look for an exact match. It should suffice to verify the tree, the PDG codes, and the mapping modes. This is the information that is written to the phase space file.

For complete cascades, we ignore the PDG code at positions with mappings infrared, collinear, or t/u-channel. Thus a cascade which is distinguished only by PDG code at such places, is flagged existent. If the convention is followed that light particles come before heavier ones (in the model definition), this ensures that the lightest particle is kept in the appropriate place, corresponding to the strongest peak.

For external cascades (incoming/outgoing) we take the PDG code into account even though it is zeroed in the PDG-code tree.

⟨*Cascades: procedures*⟩+≡

```
  subroutine hash_entry_check_cascade (hash_entry, cascade, ok, cascade_ptr)
    type(hash_entry_t), intent(in), target :: hash_entry
    type(cascade_t), intent(in), target :: cascade
    logical, intent(out) :: ok
    type(cascade_t), optional, pointer :: cascade_ptr
    type(cascade_p), pointer :: current
    integer, dimension(:), allocatable :: tree_pdg
    ok = .true.
    allocate (tree_pdg (size (cascade%tree_pdg)))
    if (cascade%complete) then
       where (cascade%tree_mapping == INFRARED .or. &
            cascade%tree_mapping == COLLINEAR .or. &
            cascade%tree_mapping == T_CHANNEL .or. &
            cascade%tree_mapping == U_CHANNEL)
          tree_pdg = 0
       elsewhere
          tree_pdg = cascade%tree_pdg
       end where
    else
       tree_pdg = cascade%tree_pdg
    end if
    current => hash_entry%first
    do while (associated (current))
       if (current%cascade%depth == cascade%depth) then
          if (all (current%cascade%tree == cascade%tree)) then
             if (all (current%cascade%tree_mapping == cascade%tree_mapping)) &
                  then
                if (all (current%cascade%tree_pdg .match. tree_pdg)) then
                   if (present (cascade_ptr))  cascade_ptr => current%cascade
                   ok = .false.;  return
                end if
             end if
          end if
       end if
```

```
      current => current%next
    end do
    if (present (cascade_ptr))  cascade_ptr => cascade
  end subroutine hash_entry_check_cascade
```

For PDG codes, we specify that the undefined code matches any code. This is already defined for flavor objects, but here we need it for the codes themselves.

⟨*Cascades: interfaces*⟩+≡

```
  interface operator(.match.)
     module procedure pdg_match
  end interface
```

⟨*Cascades: procedures*⟩+≡

```
  elemental function pdg_match (pdg1, pdg2) result (flag)
    logical :: flag
    integer(TC), intent(in) :: pdg1, pdg2
    select case (pdg1)
    case (0)
       flag = .true.
    case default
       select case (pdg2)
       case (0)
          flag = .true.
       case default
          flag = pdg1 == pdg2
       end select
    end select
  end function pdg_match
```

18.7.6 The cascade set

The cascade set will later be transformed into the decay forest. It is set up as a linked list. In addition to the usual `first` and `last` pointers, there is a `first_t` pointer which points to the first t-channel cascade (after all s-channel cascades), and a `first_k` pointer which points to the first final cascade (with a keystone).

As an auxiliary device, the object contains a hash array with associated parameters where an additional pointer is stored for each cascade. The keys are made from the relevant cascade data. This hash is used for fast detection (and thus avoidance) of double entries in the cascade list.

⟨*Cascades: public*⟩+≡

```
  public :: cascade_set_t
```

⟨*Cascades: types*⟩+≡

```
  type :: cascade_set_t
     private
     class(model_data_t), pointer :: model
     integer :: n_in, n_out, n_tot
     type(flavor_t), dimension(:,:), allocatable :: flv
     integer :: depth_out, depth_tot
     real(default) :: sqrts = 0
     real(default) :: m_threshold_s = 0
     real(default) :: m_threshold_t = 0
```

```
      integer :: off_shell = 0
      integer :: t_channel = 0
      logical :: keep_nonresonant
      integer :: n_groves = 0
      ! The cascade list
      type(cascade_t), pointer :: first => null ()
      type(cascade_t), pointer :: last => null ()
      type(cascade_t), pointer :: first_t => null ()
      type(cascade_t), pointer :: first_k => null ()
      ! The hashtable
      integer :: n_entries = 0
      real :: fill_ratio = 0
      integer :: n_entries_max = 0
      integer(i32) :: mask = 0
      logical :: fatal_beam_decay = .true.
      type(hash_entry_t), dimension(:), allocatable :: entry
   end type cascade_set_t
```

⟨*Cascades: public*⟩+≡

```
   interface cascade_set_init
      module procedure cascade_set_init_base
      module procedure cascade_set_init_from_cascade
   end interface
```

This might be broken. Test before using.

⟨*Cascades: procedures*⟩+≡

```
   subroutine cascade_set_init_from_cascade (cascade_set, cascade_set_in)
     type(cascade_set_t), intent(out) :: cascade_set
     type(cascade_set_t), intent(in), target :: cascade_set_in
     type(cascade_t), pointer :: casc_iter, casc_copy
     cascade_set%model => cascade_set_in%model
     cascade_set%n_in = cascade_set_in%n_in
     cascade_set%n_out = cascade_set_in%n_out
     cascade_set%n_tot = cascade_set_in%n_tot
     cascade_set%flv = cascade_set_in%flv
     cascade_set%depth_out = cascade_set_in%depth_out
     cascade_set%depth_tot = cascade_set_in%depth_tot
     cascade_set%sqrts = cascade_set_in%sqrts
     cascade_set%m_threshold_s = cascade_set_in%m_threshold_s
     cascade_set%m_threshold_t = cascade_set_in%m_threshold_t
     cascade_set%off_shell = cascade_set_in%off_shell
     cascade_set%t_channel = cascade_set_in%t_channel
     cascade_set%keep_nonresonant = cascade_set_in%keep_nonresonant
     cascade_set%n_groves = cascade_set_in%n_groves

     casc_iter => cascade_set_in%first
     do while (associated (casc_iter))
        allocate (casc_copy)
        casc_copy = casc_iter
        casc_copy%next => null ()
        if (associated (cascade_set%first)) then
           cascade_set%last%next => casc_copy
        else
           cascade_set%first => casc_copy
```

```
      end if
      cascade_set%last => casc_copy
      casc_iter => casc_iter%next
   end do

   cascade_set%n_entries = cascade_set_in%n_entries
   cascade_set%fill_ratio = cascade_set_in%fill_ratio
   cascade_set%n_entries_max = cascade_set_in%n_entries_max
   cascade_set%mask = cascade_set_in%mask
   cascade_set%fatal_beam_decay = cascade_set_in%fatal_beam_decay
   allocate (cascade_set%entry (0:cascade_set%mask))
   cascade_set%entry = cascade_set_in%entry
end subroutine cascade_set_init_from_cascade
```

Return true if there are cascades which are active and complete, so the phase space file would be nonempty.

⟨*Cascades: public*⟩+≡

```
public :: cascade_set_is_valid
```

⟨*Cascades: procedures*⟩+≡

```
function cascade_set_is_valid (cascade_set) result (flag)
  logical :: flag
  type(cascade_set_t), intent(in) :: cascade_set
  type(cascade_t), pointer :: cascade
  flag = .false.
  cascade => cascade_set%first_k
  do while (associated (cascade))
     if (cascade%active .and. cascade%complete) then
        flag = .true.
        return
     end if
     cascade => cascade%next
  end do
end function cascade_set_is_valid
```

The initializer sets up the hash table with some initial size guessed by looking at the number of external particles. We choose 256 for 3 external particles and a factor of 4 for each additional particle, limited at 2^{30}=1G.

Note: the explicit initialization loop might be avoided (ELEMENTAL), but a bug in nagfor 5.3.2 prevents this.

⟨*Cascades: parameters*⟩+≡

```
real, parameter, public :: CASCADE_SET_FILL_RATIO = 0.1
```

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_init_base (cascade_set, model, n_in, n_out, phs_par, &
      fatal_beam_decay, flv)
  type(cascade_set_t), intent(out) :: cascade_set
  class(model_data_t), intent(in), target :: model
  integer, intent(in) :: n_in, n_out
  type(phs_parameters_t), intent(in) :: phs_par
  logical, intent(in) :: fatal_beam_decay
  type(flavor_t), dimension(:,:), intent(in), optional :: flv
  integer :: size_guess
```

```
    integer :: i, j
    cascade_set%model => model
    cascade_set%n_in = n_in
    cascade_set%n_out = n_out
    cascade_set%n_tot = n_in + n_out
    if (present (flv)) then
       allocate (cascade_set%flv (size (flv, 1), size (flv, 2)))
       do i = 1, size (flv, 2)
          do j = 1, size (flv, 1)
             call cascade_set%flv(j,i)%init (flv(j,i)%get_pdg (), model)
          end do
       end do
    end if
    select case (n_in)
    case (1);  cascade_set%depth_out = 2 * n_out - 3
    case (2);  cascade_set%depth_out = 2 * n_out - 1
    end select
    cascade_set%depth_tot = 2 * cascade_set%n_tot - 3
    cascade_set%sqrts = phs_par%sqrts
    cascade_set%m_threshold_s = phs_par%m_threshold_s
    cascade_set%m_threshold_t = phs_par%m_threshold_t
    cascade_set%off_shell = phs_par%off_shell
    cascade_set%t_channel = phs_par%t_channel
    cascade_set%keep_nonresonant = phs_par%keep_nonresonant
    cascade_set%fill_ratio = CASCADE_SET_FILL_RATIO
    size_guess = ishft (256, min (2 * (cascade_set%n_tot - 3), 22))
    cascade_set%n_entries_max = size_guess * cascade_set%fill_ratio
    cascade_set%mask = size_guess - 1
    allocate (cascade_set%entry (0:cascade_set%mask))
    cascade_set%fatal_beam_decay = fatal_beam_decay
  end subroutine cascade_set_init_base
```

The finalizer has to delete both the hash and the list.

⟨*Cascades: public*⟩+≡

```
  public :: cascade_set_final
```

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_set_final (cascade_set)
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(cascade_t), pointer :: current
    integer :: i
    if (allocated (cascade_set%entry)) then
       do i = 0, cascade_set%mask
          call hash_entry_final (cascade_set%entry(i))
       end do
       deallocate (cascade_set%entry)
    end if
    do while (associated (cascade_set%first))
       current => cascade_set%first
       cascade_set%first => cascade_set%first%next
       deallocate (current)
    end do
  end subroutine cascade_set_final
```

Write the process in ASCII format, in columns that are headed by the corresponding bincode.

⟨*Cascades: public*⟩+≡

```
public :: cascade_set_write_process_bincode_format
```

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_write_process_bincode_format (cascade_set, unit)
  type(cascade_set_t), intent(in), target :: cascade_set
  integer, intent(in), optional :: unit
  integer, dimension(:), allocatable :: bincode, field_width
  integer :: n_in, n_out, n_tot, n_flv
  integer :: u, f, i, bc
  character(20) :: str
  type(string_t) :: fmt_head
  type(string_t), dimension(:), allocatable :: fmt_proc
  u = given_output_unit (unit);  if (u < 0)  return
  if (.not. allocated (cascade_set%flv)) return
  write (u, "('!',1x,A)")  "List of subprocesses with particle bincodes:"
  n_in  = cascade_set%n_in
  n_out = cascade_set%n_out
  n_tot = cascade_set%n_tot
  n_flv = size (cascade_set%flv, 2)
  allocate (bincode (n_tot), field_width (n_tot), fmt_proc (n_tot))
  bc = 1
  do i = 1, n_out
     bincode(n_in + i) = bc
     bc = 2 * bc
  end do
  do i = n_in, 1, -1
     bincode(i) = bc
     bc = 2 * bc
  end do
  do i = 1, n_tot
     write (str, "(I0)")  bincode(i)
     field_width(i) = len_trim (str)
     do f = 1, n_flv
        field_width(i) = max (field_width(i), &
             len (cascade_set%flv(i,f)%get_name ()))
     end do
  end do
  fmt_head = "('!'"
  do i = 1, n_tot
     fmt_head = fmt_head // ",1x,"
     fmt_proc(i) = "(1x,"
     write (str, "(I0)")  field_width(i)
     fmt_head = fmt_head // "I" // trim(str)
     fmt_proc(i) = fmt_proc(i) // "A" // trim(str)
     if (i == n_in) then
        fmt_head = fmt_head // ",1x,'  '"
     end if
  end do
  !!! !!! !!! Workaround for standard-semantics ifort 16.0 bug
  do i = 1, n_tot
     fmt_proc(i) = fmt_proc(i) // ")"
```

```
    end do
    fmt_head = fmt_head // ")"
    write (u, char (fmt_head))  bincode
    do f = 1, n_flv
       write (u, "('!')", advance="no")
       do i = 1, n_tot
          write (u, char (fmt_proc(i)), advance="no") &
               char (cascade_set%flv(i,f)%get_name ())
          if (i == n_in)  write (u, "(1x,'=>')", advance="no")
       end do
       write (u, *)
    end do
    write (u, char (fmt_head))  bincode
  end subroutine cascade_set_write_process_bincode_format
```

Write the process as a LaTeX expression.

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_set_write_process_tex_format (cascade_set, unit)
    type(cascade_set_t), intent(in), target :: cascade_set
    integer, intent(in), optional :: unit
    integer :: u, f, i
    u = given_output_unit (unit);  if (u < 0)  return
    if (.not. allocated (cascade_set%flv)) return
    write (u, "(A)")  "\begin{align*}"
    do f = 1, size (cascade_set%flv, 2)
       do i = 1, cascade_set%n_in
          if (i > 1)  write (u, "(A)", advance="no") "\quad "
          write (u, "(A)", advance="no") &
               char (cascade_set%flv(i,f)%get_tex_name ())
       end do
       write (u, "(A)", advance="no")  "\quad &\to\quad "
       do i = cascade_set%n_in + 1, cascade_set%n_tot
          if (i > cascade_set%n_in + 1)  write (u, "(A)", advance="no") "\quad "
          write (u, "(A)", advance="no") &
               char (cascade_set%flv(i,f)%get_tex_name ())
       end do
       if (f < size (cascade_set%flv, 2)) then
          write (u, "(A)")  "\\"
       else
          write (u, "(A)")  ""
       end if
    end do
    write (u, "(A)")  "\end{align*}"
  end subroutine cascade_set_write_process_tex_format
```

Three output routines: phase-space file, graph source code, and screen output.

This version generates the phase space file. It deals only with complete cascades.

⟨*Cascades: public*⟩+≡

```
  public :: cascade_set_write_file_format
```

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_set_write_file_format (cascade_set, unit)
```

```
    type(cascade_set_t), intent(in), target :: cascade_set
    integer, intent(in), optional :: unit
    type(cascade_t), pointer :: cascade
    integer :: u, grove, count
    logical :: first_in_grove
    u = given_output_unit (unit);  if (u < 0)  return
    count = 0
    do grove = 1, cascade_set%n_groves
       first_in_grove = .true.
       cascade => cascade_set%first_k
       do while (associated (cascade))
          if (cascade%active .and. cascade%complete) then
             if (cascade%grove == grove) then
                if (first_in_grove) then
                   first_in_grove = .false.
                   write (u, "(A)")
                   write (u, "(1x,'!',1x,A,1x,I0,A)", advance='no') &
                      'Multiplicity =', cascade%multiplicity, ","
                   select case (cascade%n_resonances)
                   case (0)
                      write (u, '(1x,A)', advance='no') 'no resonances, '
                   case (1)
                      write (u, '(1x,A)', advance='no') '1 resonance,  '
                   case default
                      write (u, '(1x,I0,1x,A)', advance='no') &
                           cascade%n_resonances, 'resonances, '
                   end select
                   write (u, '(1x,I0,1x,A)', advance='no') &
                        cascade%n_log_enhanced, 'logs, '
                   write (u, '(1x,I0,1x,A)', advance='no') &
                        cascade%n_off_shell, 'off-shell, '
                   select case (cascade%n_t_channel)
                   case (0);  write (u, '(1x,A)') 's-channel graph'
                   case (1);  write (u, '(1x,A)') '1 t-channel line'
                   case default
                      write(u,'(1x,I0,1x,A)') &
                           cascade%n_t_channel, 't-channel lines'
                   end select
                   write (u, '(1x,A,I0)') 'grove #', grove
                end if
                count = count + 1
                write (u, "(1x,'!',1x,A,I0)")  "Channel #", count
                call cascade_write_file_format (cascade, cascade_set%model, u)
             end if
          end if
          cascade => cascade%next
       end do
    end do
  end subroutine cascade_set_write_file_format
```

This is the graph output format, the driver-file

⟨*Cascades: public*⟩+≡

```
  public :: cascade_set_write_graph_format
```

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_write_graph_format &
    (cascade_set, filename, process_id, unit)
  type(cascade_set_t), intent(in), target :: cascade_set
  type(string_t), intent(in) :: filename, process_id
  integer, intent(in), optional :: unit
  type(cascade_t), pointer :: cascade
  integer :: u, grove, count, pgcount
  logical :: first_in_grove
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, '(A)') "\documentclass[10pt]{article}"
  write (u, '(A)') "\usepackage{amsmath}"
  write (u, '(A)') "\usepackage{feynmp}"
  write (u, '(A)') "\usepackage{url}"
  write (u, '(A)') "\usepackage{color}"
  write (u, *)
  write (u, '(A)') "\textwidth 18.5cm"
  write (u, '(A)') "\evensidemargin -1.5cm"
  write (u, '(A)') "\oddsidemargin -1.5cm"
  write (u, *)
  write (u, '(A)') "\newcommand{\blue}{\color{blue}}"
  write (u, '(A)') "\newcommand{\green}{\color{green}}"
  write (u, '(A)') "\newcommand{\red}{\color{red}}"
  write (u, '(A)') "\newcommand{\magenta}{\color{magenta}}"
  write (u, '(A)') "\newcommand{\cyan}{\color{cyan}}"
  write (u, '(A)') "\newcommand{\sm}{\footnotesize}"
  write (u, '(A)') "\setlength{\parindent}{0pt}"
  write (u, '(A)') "\setlength{\parsep}{20pt}"
  write (u, *)
  write (u, '(A)') "\begin{document}"
  write (u, '(A)') "\begin{fmffile}{" // char (filename) // "}"
  write (u, '(A)') "\fmfcmd{color magenta; magenta = red + blue;}"
  write (u, '(A)') "\fmfcmd{color cyan; cyan = green + blue;}"
  write (u, '(A)') "\begin{fmfshrink}{0.5}"
  write (u, '(A)') "\begin{flushleft}"
  write (u, *)
  write (u, '(A)') "\noindent" // &
       & "\textbf{\large\texttt{WHIZARD} phase space channels}" // &
       & "\hfill\today"
  write (u, *)
  write (u, '(A)') "\vspace{10pt}"
  write (u, '(A)') "\noindent" // &
       & "\textbf{Process:} \url{" // char (process_id) // "}"
  call cascade_set_write_process_tex_format (cascade_set, u)
  write (u, *)
  write (u, '(A)') "\noindent" // &
       & "\textbf{Note:} These are pseudo Feynman graphs that "
  write (u, '(A)') "visualize phase-space parameterizations " // &
       & "(``integration channels'').  "
  write (u, '(A)') "They do \emph{not} indicate Feynman graphs used for the " // &
       & "matrix element."
  write (u, *)
  write (u, '(A)') "\textbf{Color code:} " // &
       & "{\blue resonance,} " // &
```

```
         & "{\cyan t-channel,} " // &
         & "{\green radiation,} "
      write (u, '(A)') "{\red infrared,} " // &
         & "{\magenta collinear,} " // &
         & "external/off-shell"
      write (u, *)
      write (u, '(A)') "\noindent" // &
         & "\textbf{Black square:} Keystone, indicates ordering of " // &
         & "phase space parameters."
      write (u, *)
      write (u, '(A)') "\vspace{-20pt}"
      count = 0
      pgcount = 0
      do grove = 1, cascade_set%n_groves
         first_in_grove = .true.
         cascade => cascade_set%first
         do while (associated (cascade))
            if (cascade%active .and. cascade%complete) then
               if (cascade%grove == grove) then
                  if (first_in_grove) then
                     first_in_grove = .false.
                     write (u, *)
                     write (u, '(A)') "\vspace{20pt}"
                     write (u, '(A)') "\begin{tabular}{l}"
                     write (u, '(A,I5,A)') &
                          & "\fbox{\bf Grove \boldmath$", grove, "$} \\[10pt]"
                     write (u, '(A,I1,A)') "Multiplicity: ", &
                          cascade%multiplicity, "\\"
                     write (u, '(A,I1,A)') "Resonances:   ", &
                          cascade%n_resonances, "\\"
                     write (u, '(A,I1,A)') "Log-enhanced: ", &
                          cascade%n_log_enhanced, "\\"
                     write (u, '(A,I1,A)') "Off-shell:    ", &
                          cascade%n_off_shell, "\\"
                     write (u, '(A,I1,A)') "t-channel:    ", &
                          cascade%n_t_channel, ""
                     write (u, '(A)') "\end{tabular}"
                  end if
                  count = count + 1
                  call cascade_write_graph_format (cascade, count, unit)
                  if (pgcount >= 250) then
                     write (u, '(A)') "\clearpage"
                     pgcount = 0
                  end if
               end if
            end if
            cascade => cascade%next
         end do
      end do
      write (u, '(A)') "\end{flushleft}"
      write (u, '(A)') "\end{fmfshrink}"
      write (u, '(A)') "\end{fmffile}"
      write (u, '(A)') "\end{document}"
   end subroutine cascade_set_write_graph_format
```

This is for screen output and debugging:

⟨*Cascades: public*⟩+≡

```
public :: cascade_set_write
```

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_write (cascade_set, unit, active_only, complete_only)
  type(cascade_set_t), intent(in), target :: cascade_set
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: active_only, complete_only
  logical :: active, complete
  type(cascade_t), pointer :: cascade
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  active = .true.;  if (present (active_only))  active = active_only
  complete = .false.;  if (present (complete_only))  complete = complete_only
  write (u, "(A)") "Cascade set:"
  write (u, "(3x,A)", advance="no")  "Model:"
  if (associated (cascade_set%model)) then
     write (u, "(1x,A)") char (cascade_set%model%get_name ())
  else
     write (u, "(1x,A)") "[none]"
  end if
  write (u, "(3x,A)", advance="no")  "n_in/out/tot  ="
  write (u, "(3(1x,I7))")  &
       cascade_set%n_in, cascade_set%n_out, cascade_set%n_tot
  write (u, "(3x,A)", advance="no")  "depth_out/tot ="
  write (u, "(2(1x,I7))")  cascade_set%depth_out, cascade_set%depth_tot
  write (u, "(3x,A)", advance="no")  "mass thr(s/t) ="
  write (u, "(2(1x," // FMT_19 // "))")  &
       cascade_set%m_threshold_s, cascade_set%m_threshold_t
  write (u, "(3x,A)", advance="no")  "off shell     ="
  write (u, "(1x,I7)")  cascade_set%off_shell
  write (u, "(3x,A)", advance="no")  "keep_nonreson ="
  write (u, "(1x,L1)")  cascade_set%keep_nonresonant
  write (u, "(3x,A)", advance="no")  "n_groves      ="
  write (u, "(1x,I7)")  cascade_set%n_groves
  write (u, "(A)")
  write (u, "(A)") "Cascade list:"
  if (associated (cascade_set%first)) then
     cascade => cascade_set%first
     do while (associated (cascade))
        if (active .and. .not. cascade%active)  cycle
        if (complete .and. .not. cascade%complete)  cycle
        call cascade_write (cascade, unit)
        cascade => cascade%next
     end do
  else
     write (u, "(A)") "[empty]"
  end if
  write (u, "(A)") "Hash array"
  write (u, "(3x,A)", advance="no")  "n_entries     ="
  write (u, "(1x,I7)")  cascade_set%n_entries
  write (u, "(3x,A)", advance="no")  "fill_ratio    ="
```

```
  write (u, "(1x," // FMT_12 // ")")  cascade_set%fill_ratio
  write (u, "(3x,A)", advance="no")  "n_entries_max ="
  write (u, "(1x,I7)")  cascade_set%n_entries_max
  write (u, "(3x,A)", advance="no")  "mask          ="
  write (u, "(1x,I0)")  cascade_set%mask
  do i = 0, ubound (cascade_set%entry, 1)
     if (allocated (cascade_set%entry(i)%key)) then
        write (u, "(1x,I7)") i
        call hash_entry_write (cascade_set%entry(i), u)
     end if
  end do
end subroutine cascade_set_write
```

18.7.7 Adding cascades

Add a deep copy of a cascade to the set. The copy has all content of the original, but the pointers are nullified. We do not care whether insertion was successful or not. The pointer argument, if present, is assigned to the input cascade, or to the hash entry if it is already present.

The procedure is recursive: any daughter or mother entries are also deep-copied and added to the cascade set before the current copy is added.

⟨*Cascades: procedures*⟩+≡

```
recursive subroutine cascade_set_add_copy &
     (cascade_set, cascade_in, cascade_ptr)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), intent(in) :: cascade_in
  type(cascade_t), optional, pointer :: cascade_ptr
  type(cascade_t), pointer :: cascade
  logical :: ok
  allocate (cascade)
  cascade = cascade_in
  if (associated (cascade_in%daughter1))  call cascade_set_add_copy &
       (cascade_set, cascade_in%daughter1, cascade%daughter1)
  if (associated (cascade_in%daughter2))  call cascade_set_add_copy &
       (cascade_set, cascade_in%daughter2, cascade%daughter2)
  if (associated (cascade_in%mother))  call cascade_set_add_copy &
       (cascade_set, cascade_in%mother, cascade%mother)
  cascade%next => null ()
  call cascade_set_add (cascade_set, cascade, ok, cascade_ptr)
  if (.not. ok)  deallocate (cascade)
end subroutine cascade_set_add_copy
```

Add a cascade to the set. This does not deep-copy. We first try to insert it in the hash array. If successful, add it to the list. Failure indicates that it is already present, and we drop it.

The hash key is built solely from the tree array, so neither particle codes nor resonances count, just topology.

Technically, hash and list receive only pointers, so the cascade can be considered as being in either of both. We treat it as part of the list.

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_add (cascade_set, cascade, ok, cascade_ptr)
```

```
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(cascade_t), intent(in), target :: cascade
    logical, intent(out) :: ok
    type(cascade_t), optional, pointer :: cascade_ptr
    integer(i8), dimension(1) :: mold
    call cascade_set_hash_insert &
         (cascade_set, transfer (cascade%tree, mold), cascade, ok, cascade_ptr)
    if (ok)  call cascade_set_list_add (cascade_set, cascade)
  end subroutine cascade_set_add
```

Add a new cascade to the list:

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_set_list_add (cascade_set, cascade)
    type(cascade_set_t), intent(inout) :: cascade_set
    type(cascade_t), intent(in), target :: cascade
    if (associated (cascade_set%last)) then
       cascade_set%last%next => cascade
    else
       cascade_set%first => cascade
    end if
    cascade_set%last => cascade
  end subroutine cascade_set_list_add
```

Add a cascade entry to the hash array:

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_set_hash_insert &
       (cascade_set, key, cascade, ok, cascade_ptr)
    type(cascade_set_t), intent(inout), target :: cascade_set
    integer(i8), dimension(:), intent(in) :: key
    type(cascade_t), intent(in), target :: cascade
    logical, intent(out) :: ok
    type(cascade_t), optional, pointer :: cascade_ptr
    integer(i32) :: h
    if (cascade_set%n_entries >= cascade_set%n_entries_max) &
         call cascade_set_hash_expand (cascade_set)
    h = hash (key)
    call cascade_set_hash_insert_rec &
         (cascade_set, h, h, key, cascade, ok, cascade_ptr)
  end subroutine cascade_set_hash_insert
```

Double the hashtable size when necesssary:

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_set_hash_expand (cascade_set)
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(hash_entry_t), dimension(:), allocatable, target :: table_tmp
    type(cascade_p), pointer :: current
    integer :: i, s
    allocate (table_tmp (0:cascade_set%mask))
    table_tmp = cascade_set%entry
    deallocate (cascade_set%entry)
    s = 2 * size (table_tmp)
    cascade_set%n_entries = 0
```

```
    cascade_set%n_entries_max = s * cascade_set%fill_ratio
    cascade_set%mask = s - 1
    allocate (cascade_set%entry (0:cascade_set%mask))
    do i = 0, ubound (table_tmp, 1)
       current => table_tmp(i)%first
       do while (associated (current))
          call cascade_set_hash_insert_rec &
               (cascade_set, table_tmp(i)%hashval, table_tmp(i)%hashval, &
                table_tmp(i)%key, current%cascade)
          current => current%next
       end do
    end do
  end subroutine cascade_set_hash_expand
```

Insert the cascade at the bucket determined by the hash value. If the bucket is filled, check first for a collision (unequal keys). In that case, choose the following bucket and repeat. Otherwise, add the cascade to the bucket.

If the bucket is empty, record the hash value, allocate and store the key, and then add the cascade to the bucket.

If `ok` is present, before insertion we check whether the cascade is already stored, and return failure if yes.

⟨*Cascades: procedures*⟩+≡

```
  recursive subroutine cascade_set_hash_insert_rec &
       (cascade_set, h, hashval, key, cascade, ok, cascade_ptr)
    type(cascade_set_t), intent(inout) :: cascade_set
    integer(i32), intent(in) :: h, hashval
    integer(i8), dimension(:), intent(in) :: key
    type(cascade_t), intent(in), target :: cascade
    logical, intent(out), optional :: ok
    type(cascade_t), optional, pointer :: cascade_ptr
    integer(i32) :: i
    i = iand (h, cascade_set%mask)
    if (allocated (cascade_set%entry(i)%key)) then
       if (size (cascade_set%entry(i)%key) /= size (key)) then
          call cascade_set_hash_insert_rec &
               (cascade_set, h + 1, hashval, key, cascade, ok, cascade_ptr)
       else if (any (cascade_set%entry(i)%key /= key)) then
          call cascade_set_hash_insert_rec &
               (cascade_set, h + 1, hashval, key, cascade, ok, cascade_ptr)
       else
          call hash_entry_add_cascade_ptr &
               (cascade_set%entry(i), cascade, ok, cascade_ptr)
       end if
    else
       cascade_set%entry(i)%hashval = hashval
       allocate (cascade_set%entry(i)%key (size (key)))
       cascade_set%entry(i)%key = key
       call hash_entry_add_cascade_ptr &
            (cascade_set%entry(i), cascade, ok, cascade_ptr)
       cascade_set%n_entries = cascade_set%n_entries + 1
    end if
  end subroutine cascade_set_hash_insert_rec
```

18.7.8 External particles

We want to initialize the cascade set with the outgoing particles. In case of multiple processes, initial cascades are prepared for all of them. The hash array check ensures that no particle appears more than once at the same place.

⟨*Cascades: interfaces*⟩+≡

```
interface cascade_set_add_outgoing
   module procedure cascade_set_add_outgoing1
   module procedure cascade_set_add_outgoing2
end interface
```

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_add_outgoing2 (cascade_set, flv)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(flavor_t), dimension(:,:), intent(in) :: flv
  integer :: pos, prc, n_out, n_prc
  type(cascade_t), pointer :: cascade
  logical :: ok
  n_out = size (flv, dim=1)
  n_prc = size (flv, dim=2)
  do prc = 1, n_prc
     do pos = 1, n_out
        allocate (cascade)
        call cascade_init_outgoing &
             (cascade, flv(pos,prc), pos, cascade_set%m_threshold_s)
        call cascade_set_add (cascade_set, cascade, ok)
        if (.not. ok) then
           deallocate (cascade)
        end if
     end do
  end do
end subroutine cascade_set_add_outgoing2

subroutine cascade_set_add_outgoing1 (cascade_set, flv)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(flavor_t), dimension(:), intent(in) :: flv
  integer :: pos, n_out
  type(cascade_t), pointer :: cascade
  logical :: ok
  n_out = size (flv, dim=1)
  do pos = 1, n_out
     allocate (cascade)
     call cascade_init_outgoing &
          (cascade, flv(pos), pos, cascade_set%m_threshold_s)
     call cascade_set_add (cascade_set, cascade, ok)
     if (.not. ok) then
        deallocate (cascade)
     end if
  end do
end subroutine cascade_set_add_outgoing1
```

The incoming particles are added one at a time. Nevertheless, we may have several processes which are looped over. At the first opportunity, we set the

pointer `first_t` in the cascade set which should point to the first t-channel cascade.

Return the indices of the first and last cascade generated.

⟨*Cascades: interfaces*⟩+≡

```
interface cascade_set_add_incoming
   module procedure cascade_set_add_incoming0
   module procedure cascade_set_add_incoming1
end interface
```

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_add_incoming1 (cascade_set, n1, n2, pos, flv)
  type(cascade_set_t), intent(inout), target :: cascade_set
  integer, intent(out) :: n1, n2
  integer, intent(in) :: pos
  type(flavor_t), dimension(:), intent(in) :: flv
  integer :: prc, n_prc
  type(cascade_t), pointer :: cascade
  logical :: ok
  n1 = 0
  n2 = 0
  n_prc = size (flv)
  do prc = 1, n_prc
     allocate (cascade)
     call cascade_init_incoming &
          (cascade, flv(prc), pos, cascade_set%m_threshold_t)
     call cascade_set_add (cascade_set, cascade, ok)
     if (ok) then
        if (n1 == 0)  n1 = cascade%index
        n2 = cascade%index
        if (.not. associated (cascade_set%first_t)) then
           cascade_set%first_t => cascade
        end if
     else
        deallocate (cascade)
     end if
  end do
end subroutine cascade_set_add_incoming1

subroutine cascade_set_add_incoming0 (cascade_set, n1, n2, pos, flv)
  type(cascade_set_t), intent(inout), target :: cascade_set
  integer, intent(out) :: n1, n2
  integer, intent(in) :: pos
  type(flavor_t), intent(in) :: flv
  type(cascade_t), pointer :: cascade
  logical :: ok
  n1 = 0
  n2 = 0
  allocate (cascade)
  call cascade_init_incoming &
       (cascade, flv, pos, cascade_set%m_threshold_t)
  call cascade_set_add (cascade_set, cascade, ok)
  if (ok) then
     if (n1 == 0)  n1 = cascade%index
```

```
      n2 = cascade%index
      if (.not. associated (cascade_set%first_t)) then
         cascade_set%first_t => cascade
      end if
   else
      deallocate (cascade)
   end if
end subroutine cascade_set_add_incoming0
```

18.7.9 Cascade combination I: flavor assignment

We have two disjunct cascades, now use the vertex table to determine the possible flavors of the combination cascade. For each possibility, try to generate a new cascade. The total cascade depth has to be one less than the limit, because this is reached by setting the keystone.

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_match_pair (cascade_set, cascade1, cascade2, s_channel)
   type(cascade_set_t), intent(inout), target :: cascade_set
   type(cascade_t), intent(in), target :: cascade1, cascade2
   logical, intent(in) :: s_channel
   integer, dimension(:), allocatable :: pdg3
   integer :: i, depth_max
   type(flavor_t) :: flv
   if (s_channel) then
      depth_max = cascade_set%depth_out
   else
      depth_max = cascade_set%depth_tot
   end if
   if (cascade1%depth + cascade2%depth < depth_max) then
      call cascade_set%model%match_vertex ( &
           cascade1%flv%get_pdg (), &
           cascade2%flv%get_pdg (), &
           pdg3)
      do i = 1, size (pdg3)
         call flv%init (pdg3(i), cascade_set%model)
         if (s_channel) then
            call cascade_combine_s (cascade_set, cascade1, cascade2, flv)
         else
            call cascade_combine_t (cascade_set, cascade1, cascade2, flv)
         end if
      end do
      deallocate (pdg3)
   end if
end subroutine cascade_match_pair
```

The triplet version takes a third cascade, and we check whether this triplet has a matching vertex in the database. If yes, we make a keystone cascade.

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_match_triplet &
     (cascade_set, cascade1, cascade2, cascade3, s_channel)
   type(cascade_set_t), intent(inout), target :: cascade_set
```

```
    type(cascade_t), intent(in), target :: cascade1, cascade2, cascade3
    logical, intent(in) :: s_channel
    integer :: depth_max
    depth_max = cascade_set%depth_tot
    if (cascade1%depth + cascade2%depth + cascade3%depth == depth_max) then
       if (cascade_set%model%check_vertex ( &
            cascade1%flv%get_pdg (), &
            cascade2%flv%get_pdg (), &
            cascade3%flv%get_pdg ())) then
          call cascade_combine_keystone &
               (cascade_set, cascade1, cascade2, cascade3, s_channel)
       end if
    end if
  end subroutine cascade_match_triplet
```

18.7.10 Cascade combination II: kinematics setup and check

Having three matching flavors, we start constructing the combination cascade. We look at the mass hierarchies and determine whether the cascade is to be kept. In passing we set mapping modes, resonance properties and such.

If successful, the cascade is finalized. For a resonant cascade, we prepare in addition a copy without the resonance.

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_combine_s (cascade_set, cascade1, cascade2, flv)
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(cascade_t), intent(in), target :: cascade1, cascade2
    type(flavor_t), intent(in) :: flv
    type(cascade_t), pointer :: cascade3, cascade4
    logical :: keep
    keep = .false.
    allocate (cascade3)
    call cascade_init (cascade3, cascade1%depth + cascade2%depth + 1)
    cascade3%bincode = ior (cascade1%bincode, cascade2%bincode)
    cascade3%flv = flv%anti ()
    cascade3%pdg = cascade3%flv%get_pdg ()
    cascade3%is_vector = flv%get_spin_type () == VECTOR
    cascade3%m_min = cascade1%m_min + cascade2%m_min
    cascade3%m_rea = flv%get_mass ()
    if (cascade3%m_rea > cascade_set%m_threshold_s) then
       cascade3%m_eff = cascade3%m_rea
    end if
    ! Potentially resonant cases [sqrts = m_rea for on-shell decay]
    if (cascade3%m_rea > cascade3%m_min &
         .and. cascade3%m_rea <= cascade_set%sqrts) then
       if (flv%get_width () /= 0) then
          if (cascade1%on_shell .or. cascade2%on_shell) then
             keep = .true.
             cascade3%mapping = S_CHANNEL
             cascade3%resonant = .true.
          end if
       else
          call warn_decay (flv)
```

```
   end if
! Collinear and IR singular cases
else if (cascade3%m_rea < cascade_set%sqrts) then
   ! Massless splitting
   if (cascade1%m_eff == 0 .and. cascade2%m_eff == 0 &
         .and. cascade3%depth <= 3) then
      keep = .true.
      cascade3%log_enhanced = .true.
      if (cascade3%is_vector) then
         if (cascade1%is_vector .and. cascade2%is_vector) then
            cascade3%mapping = COLLINEAR   ! three-vector-vertex
         else
            cascade3%mapping = INFRARED    ! vector splitting into matter
         end if
      else
         if (cascade1%is_vector .or. cascade2%is_vector) then
            cascade3%mapping = COLLINEAR   ! vector radiation off matter
         else
            cascade3%mapping = INFRARED    ! scalar radiation/splitting
         end if
      end if
   ! IR radiation off massive particle
   else if (cascade3%m_eff > 0 .and. cascade1%m_eff > 0 &
         .and. cascade2%m_eff == 0 &
         .and. (cascade1%on_shell .or. cascade1%mapping == RADIATION) &
         .and. abs (cascade3%m_eff - cascade1%m_eff) &
                   < cascade_set%m_threshold_s) &
         then
      keep = .true.
      cascade3%log_enhanced = .true.
      cascade3%mapping = RADIATION
   else if (cascade3%m_eff > 0 .and. cascade2%m_eff > 0 &
         .and. cascade1%m_eff == 0 &
         .and. (cascade2%on_shell .or. cascade2%mapping == RADIATION) &
         .and. abs (cascade3%m_eff - cascade2%m_eff) &
               < cascade_set%m_threshold_s) &
         then
      keep = .true.
      cascade3%log_enhanced = .true.
      cascade3%mapping = RADIATION
   end if
end if
! Non-singular cases, including failed resonances
if (.not. keep) then
   ! Two on-shell particles from a virtual mother
   if (cascade1%on_shell .or. cascade2%on_shell) then
      keep = .true.
      cascade3%m_eff = max (cascade3%m_min, &
                            cascade1%m_eff + cascade2%m_eff)
      if (cascade3%m_eff < cascade_set%m_threshold_s) then
         cascade3%m_eff = 0
      end if
   end if
end if
```

```
  ! Complete and register the cascade (two in case of resonance)
  if (keep) then
     cascade3%on_shell = cascade3%resonant .or. cascade3%log_enhanced
     if (cascade3%resonant) then
        cascade3%pdg = cascade3%flv%get_pdg ()
        if (cascade_set%keep_nonresonant) then
           allocate (cascade4)
           cascade4 = cascade3
           cascade4%index = cascade_index ()
           cascade4%pdg = UNDEFINED
           cascade4%mapping = NO_MAPPING
           cascade4%resonant = .false.
           cascade4%on_shell = .false.
        end if
        cascade3%m_min = cascade3%m_rea
        call cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
        if (cascade_set%keep_nonresonant) then
           call cascade_fusion (cascade_set, cascade1, cascade2, cascade4)
        end if
     else
        call cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
     end if
  else
     deallocate (cascade3)
  end if
contains
  subroutine warn_decay (flv)
    type(flavor_t), intent(in) :: flv
    integer :: i
    integer, dimension(MAX_WARN_RESONANCE), save :: warned_code = 0
    LOOP_WARNED: do i = 1, MAX_WARN_RESONANCE
       if (warned_code(i) == 0) then
          warned_code(i) = flv%get_pdg ()
          write (msg_buffer, "(A)") &
               & " Intermediate decay of zero-width particle " &
               & // char (flv%get_name ()) &
               & // " may be possible."
          call msg_warning
          exit LOOP_WARNED
       else if (warned_code(i) == flv%get_pdg ()) then
          exit LOOP_WARNED
       end if
    end do LOOP_WARNED
  end subroutine warn_decay
end subroutine cascade_combine_s
```

⟨*Cascades: parameters*⟩+≡

```
integer, parameter, public :: MAX_WARN_RESONANCE = 50
```

This is the t-channel version. `cascade1` is t-channel and contains the seed, `cascade2` is s-channel. We check for kinematically allowed beam decay (which is a fatal error), or massless splitting / soft radiation. The cascade is kept in all remaining cases and submitted for registration.

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_combine_t (cascade_set, cascade1, cascade2, flv)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), intent(in), target :: cascade1, cascade2
  type(flavor_t), intent(in) :: flv
  type(cascade_t), pointer :: cascade3
  allocate (cascade3)
  call cascade_init (cascade3, cascade1%depth + cascade2%depth + 1)
  cascade3%bincode = ior (cascade1%bincode, cascade2%bincode)
  cascade3%flv = flv%anti ()
  cascade3%pdg = abs (cascade3%flv%get_pdg ())
  cascade3%is_vector = flv%get_spin_type () == VECTOR
  if (cascade1%incoming) then
     cascade3%m_min = cascade2%m_min
  else
     cascade3%m_min = cascade1%m_min + cascade2%m_min
  end if
  cascade3%m_rea = flv%get_mass ()
  if (cascade3%m_rea > cascade_set%m_threshold_t) then
     cascade3%m_eff = max (cascade3%m_rea, cascade2%m_eff)
  else if (cascade2%m_eff > cascade_set%m_threshold_t) then
     cascade3%m_eff = cascade2%m_eff
  else
     cascade3%m_eff = 0
  end if
  ! Allowed decay of beam particle
  if (cascade1%incoming &
       .and. cascade1%m_rea > cascade2%m_rea + cascade3%m_rea) then
       call beam_decay (cascade_set%fatal_beam_decay)
  ! Massless splitting
  else if (cascade1%m_eff == 0 &
       .and. cascade2%m_eff < cascade_set%m_threshold_t &
       .and. cascade3%m_eff == 0) then
     cascade3%mapping = U_CHANNEL
     cascade3%log_enhanced = .true.
  ! IR radiation off massive particle
  else if (cascade1%m_eff /= 0 .and. cascade2%m_eff == 0 &
       .and. cascade3%m_eff /= 0 &
       .and. (cascade1%on_shell .or. cascade1%mapping == RADIATION) &
       .and. abs (cascade1%m_eff - cascade3%m_eff) &
             < cascade_set%m_threshold_t) &
       then
     cascade3%pdg = flv%get_pdg ()
     cascade3%log_enhanced = .true.
     cascade3%mapping = RADIATION
  end if
  cascade3%t_channel = .true.
  call cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
contains
  subroutine beam_decay (fatal_beam_decay)
    logical, intent(in) :: fatal_beam_decay
    write (msg_buffer, "(1x,A,1x,'->',1x,A,1x,A)") &
         char (cascade1%flv%get_name ()), &
         char (cascade3%flv%get_name ()), &
         char (cascade2%flv%get_name ())
```

```
      call msg_message
      write (msg_buffer, "(1x,'mass(',A,') =',1x,E17.10)") &
           char (cascade1%flv%get_name ()), cascade1%m_rea
      call msg_message
      write (msg_buffer, "(1x,'mass(',A,') =',1x,E17.10)") &
           char (cascade3%flv%get_name ()), cascade3%m_rea
      call msg_message
      write (msg_buffer, "(1x,'mass(',A,') =',1x,E17.10)") &
           char (cascade2%flv%get_name ()), cascade2%m_rea
      call msg_message
      if (fatal_beam_decay) then
         call msg_fatal (" Phase space: Initial beam particle can decay")
      else
         call msg_warning (" Phase space: Initial beam particle can decay")
      end if
    end subroutine beam_decay
  end subroutine cascade_combine_t
```

Here we complete a decay cascade. The third input is the single-particle cascade for the initial particle. There is no resonance or mapping assignment. The only condition for keeping the cascade is the mass sum of the final state, which must be less than the available energy.

Two modifications are necessary for scattering cascades: a pure s-channel diagram (cascade1 is the incoming particle) do not have a logarithmic mapping at top-level. And in a t-channel diagram, the last line exchanged is mapped t-channel, not u-channel. Finally, we can encounter the case of a $2 \to 1$ process, where cascade1 is incoming, and cascade2 is the outgoing particle. In all three cases we register a new cascade with the modified mapping.

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_combine_keystone &
       (cascade_set, cascade1, cascade2, cascade3, s_channel)
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(cascade_t), intent(in), target :: cascade1, cascade2, cascade3
    logical, intent(in) :: s_channel
    type(cascade_t), pointer :: cascade4, cascade0
    logical :: keep, ok
    keep = .false.
    allocate (cascade4)
    call cascade_init &
         (cascade4, cascade1%depth + cascade2%depth + cascade3%depth)
    cascade4%complete = .true.
    if (s_channel) then
       cascade4%bincode = ior (cascade1%bincode, cascade2%bincode)
    else
       cascade4%bincode = cascade3%bincode
    end if
    cascade4%flv = cascade3%flv
    cascade4%pdg = cascade3%pdg
    cascade4%mapping = EXTERNAL_PRT
    cascade4%is_vector = cascade3%is_vector
    cascade4%m_min = cascade1%m_min + cascade2%m_min
    cascade4%m_rea = cascade3%m_rea
    cascade4%m_eff = cascade3%m_rea
```

```
if (cascade4%m_min < cascade_set%sqrts) then
   keep = .true.
end if
if (keep) then
   if (cascade1%incoming .and. cascade2%log_enhanced) then
      allocate (cascade0)
      cascade0 = cascade2
      cascade0%next => null ()
      cascade0%index = cascade_index ()
      cascade0%mapping = NO_MAPPING
      cascade0%log_enhanced = .false.
      cascade0%n_log_enhanced = cascade0%n_log_enhanced - 1
      cascade0%tree_mapping(cascade0%depth) = NO_MAPPING
      call cascade_keystone &
           (cascade_set, cascade1, cascade0, cascade3, cascade4, ok)
      if (ok) then
         call cascade_set_add (cascade_set, cascade0, ok)
      else
         deallocate (cascade0)
      end if
   else if (cascade1%t_channel .and. cascade1%mapping == U_CHANNEL) then
      allocate (cascade0)
      cascade0 = cascade1
      cascade0%next => null ()
      cascade0%index = cascade_index ()
      cascade0%mapping = T_CHANNEL
      cascade0%tree_mapping(cascade0%depth) = T_CHANNEL
      call cascade_keystone &
           (cascade_set, cascade0, cascade2, cascade3, cascade4, ok)
      if (ok) then
         call cascade_set_add (cascade_set, cascade0, ok)
      else
         deallocate (cascade0)
      end if
   else if (cascade1%incoming .and. cascade2%depth == 1) then
      allocate (cascade0)
      cascade0 = cascade2
      cascade0%next => null ()
      cascade0%index = cascade_index ()
      cascade0%mapping = ON_SHELL
      cascade0%tree_mapping(cascade0%depth) = ON_SHELL
      call cascade_keystone &
           (cascade_set, cascade1, cascade0, cascade3, cascade4, ok)
      if (ok) then
         call cascade_set_add (cascade_set, cascade0, ok)
      else
         deallocate (cascade0)
      end if
   else
      call cascade_keystone &
           (cascade_set, cascade1, cascade2, cascade3, cascade4, ok)
   end if
else
   deallocate (cascade4)
```

```
    end if
  end subroutine cascade_combine_keystone
```

18.7.11 Cascade combination III: node connections and tree fusion

Here we assign global tree properties. If the allowed number of off-shell lines is exceeded, discard the new cascade. Otherwise, assign the trees, sort them, and assign connections. Finally, append the cascade to the list. This may fail (because in the hash array there is already an equivalent cascade). On failure, discard the cascade.

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_fusion (cascade_set, cascade1, cascade2, cascade3)
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(cascade_t), intent(in), target :: cascade1, cascade2
    type(cascade_t), pointer :: cascade3
    integer :: i1, i2, i3, i4
    logical :: ok
    cascade3%internal = (cascade3%depth - 3) / 2
    if (cascade3%resonant) then
       cascade3%multiplicity = 1
       cascade3%n_resonances = &
            cascade1%n_resonances + cascade2%n_resonances + 1
    else
       cascade3%multiplicity = cascade1%multiplicity + cascade2%multiplicity
       cascade3%n_resonances = cascade1%n_resonances + cascade2%n_resonances
    end if
    if (cascade3%log_enhanced) then
       cascade3%n_log_enhanced = &
            cascade1%n_log_enhanced + cascade2%n_log_enhanced + 1
    else
       cascade3%n_log_enhanced = &
            cascade1%n_log_enhanced + cascade2%n_log_enhanced
    end if
    if (cascade3%resonant) then
       cascade3%n_off_shell = 0
    else if (cascade3%log_enhanced) then
       cascade3%n_off_shell = cascade1%n_off_shell + cascade2%n_off_shell
    else
       cascade3%n_off_shell = cascade1%n_off_shell + cascade2%n_off_shell + 1
    end if
    if (cascade3%t_channel) then
       cascade3%n_t_channel = cascade1%n_t_channel + 1
    end if
    if (cascade3%n_off_shell > cascade_set%off_shell) then
       deallocate (cascade3)
    else if (cascade3%n_t_channel > cascade_set%t_channel) then
       deallocate (cascade3)
    else
       i1 = cascade1%depth
       i2 = i1 + 1
       i3 = i1 + cascade2%depth
```

```
      i4 = cascade3%depth
      cascade3%tree(:i1) = cascade1%tree
      where (cascade1%tree_mapping > NO_MAPPING)
         cascade3%tree_pdg(:i1) = cascade1%tree_pdg
      elsewhere
         cascade3%tree_pdg(:i1) = UNDEFINED
      end where
      cascade3%tree_mapping(:i1) = cascade1%tree_mapping
      cascade3%tree_resonant(:i1) = cascade1%tree_resonant
      cascade3%tree(i2:i3) = cascade2%tree
      where (cascade2%tree_mapping > NO_MAPPING)
         cascade3%tree_pdg(i2:i3) = cascade2%tree_pdg
      elsewhere
         cascade3%tree_pdg(i2:i3) = UNDEFINED
      end where
      cascade3%tree_mapping(i2:i3) = cascade2%tree_mapping
      cascade3%tree_resonant(i2:i3) = cascade2%tree_resonant
      cascade3%tree(i4) = cascade3%bincode
      cascade3%tree_pdg(i4) = cascade3%pdg
      cascade3%tree_mapping(i4) = cascade3%mapping
      cascade3%tree_resonant(i4) = cascade3%resonant
      call tree_sort (cascade3%tree, &
           cascade3%tree_pdg, cascade3%tree_mapping, cascade3%tree_resonant)
      cascade3%has_children = .true.
      cascade3%daughter1 => cascade1
      cascade3%daughter2 => cascade2
      call cascade_set_add (cascade_set, cascade3, ok)
      if (.not. ok)  deallocate (cascade3)
   end if
end subroutine cascade_fusion
```

Here we combine a cascade pair with an incoming particle, i.e., we set a keystone. Otherwise, this is similar. On the first opportunity, we set the `first_k` pointer in the cascade set.

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_keystone &
     (cascade_set, cascade1, cascade2, cascade3, cascade4, ok)
   type(cascade_set_t), intent(inout), target :: cascade_set
   type(cascade_t), intent(in), target :: cascade1, cascade2, cascade3
   type(cascade_t), pointer :: cascade4
   logical, intent(out) :: ok
   integer :: i1, i2, i3, i4
   cascade4%internal = (cascade4%depth - 3) / 2
   cascade4%multiplicity = cascade1%multiplicity + cascade2%multiplicity
   cascade4%n_resonances = cascade1%n_resonances + cascade2%n_resonances
   cascade4%n_off_shell = cascade1%n_off_shell + cascade2%n_off_shell
   cascade4%n_log_enhanced = &
           cascade1%n_log_enhanced + cascade2%n_log_enhanced
   cascade4%n_t_channel = cascade1%n_t_channel + cascade2%n_t_channel
   if (cascade4%n_off_shell > cascade_set%off_shell) then
      deallocate (cascade4)
      ok = .false.
   else if (cascade4%n_t_channel > cascade_set%t_channel) then
```

```
      deallocate (cascade4)
      ok = .false.
   else
      i1 = cascade1%depth
      i2 = i1 + 1
      i3 = i1 + cascade2%depth
      i4 = cascade4%depth
      cascade4%tree(:i1) = cascade1%tree
      where (cascade1%tree_mapping > NO_MAPPING)
         cascade4%tree_pdg(:i1) = cascade1%tree_pdg
      elsewhere
         cascade4%tree_pdg(:i1) = UNDEFINED
      end where
      cascade4%tree_mapping(:i1) = cascade1%tree_mapping
      cascade4%tree_resonant(:i1) = cascade1%tree_resonant
      cascade4%tree(i2:i3) = cascade2%tree
      where (cascade2%tree_mapping > NO_MAPPING)
         cascade4%tree_pdg(i2:i3) = cascade2%tree_pdg
      elsewhere
         cascade4%tree_pdg(i2:i3) = UNDEFINED
      end where
      cascade4%tree_mapping(i2:i3) = cascade2%tree_mapping
      cascade4%tree_resonant(i2:i3) = cascade2%tree_resonant
      cascade4%tree(i4) = cascade4%bincode
      cascade4%tree_pdg(i4) = UNDEFINED
      cascade4%tree_mapping(i4) = cascade4%mapping
      cascade4%tree_resonant(i4) = .false.
      call tree_sort (cascade4%tree, &
           cascade4%tree_pdg, cascade4%tree_mapping, cascade4%tree_resonant)
      cascade4%has_children = .true.
      cascade4%daughter1 => cascade1
      cascade4%daughter2 => cascade2
      cascade4%mother => cascade3
      call cascade_set_add (cascade_set, cascade4, ok)
      if (ok) then
         if (.not. associated (cascade_set%first_k)) then
            cascade_set%first_k => cascade4
         end if
      else
         deallocate (cascade4)
      end if
   end if
end subroutine cascade_keystone
```

Sort a tree (array of binary codes) and particle code array simultaneously, by ascending binary codes. A convenient method is to use the `maxloc` function iteratively, to find and remove the largest entry in the tree array one by one.

⟨*Cascades: procedures*⟩+≡

```
subroutine tree_sort (tree, pdg, mapping, resonant)
  integer(TC), dimension(:), intent(inout) :: tree
  integer, dimension(:), intent(inout) :: pdg, mapping
  logical, dimension(:), intent(inout) :: resonant
  integer(TC), dimension(size(tree)) :: tree_tmp
```

```
    integer, dimension(size(pdg)) :: pdg_tmp, mapping_tmp
    logical, dimension(size(resonant)) :: resonant_tmp
    integer, dimension(1) :: pos
    integer :: i
    tree_tmp = tree
    pdg_tmp = pdg
    mapping_tmp = mapping
    resonant_tmp = resonant
    do i = size(tree),1,-1
       pos = maxloc (tree_tmp)
       tree(i) = tree_tmp (pos(1))
       pdg(i) = pdg_tmp (pos(1))
       mapping(i) = mapping_tmp (pos(1))
       resonant(i) = resonant_tmp (pos(1))
       tree_tmp(pos(1)) = 0
    end do
  end subroutine tree_sort
```

18.7.12 Cascade set generation

These procedures loop over cascades and build up the cascade set. After each iteration of the innermost loop, we set a breakpoint.

s-channel: We use a nested scan to combine all cascades with all other cascades.

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_set_generate_s (cascade_set)
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(cascade_t), pointer :: cascade1, cascade2
    cascade1 => cascade_set%first
    LOOP1: do while (associated (cascade1))
       cascade2 => cascade_set%first
       LOOP2: do while (associated (cascade2))
          if (cascade2%index >= cascade1%index)  exit LOOP2
          if (cascade1 .disjunct. cascade2) then
             call cascade_match_pair (cascade_set, cascade1, cascade2, .true.)
          end if
          call terminate_now_if_signal ()
          cascade2 => cascade2%next
       end do LOOP2
       cascade1 => cascade1%next
    end do LOOP1
  end subroutine cascade_set_generate_s
```

The t-channel cascades are directed and have a seed (one of the incoming particles) and a target (the other one). We loop over all possible seeds and targets. Inside this, we loop over all t-channel cascades (`cascade1`) and s-channel cascades (`cascade2`) and try to combine them.

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_set_generate_t (cascade_set, pos_seed, pos_target)
    type(cascade_set_t), intent(inout), target :: cascade_set
    integer, intent(in) :: pos_seed, pos_target
```

```
  type(cascade_t), pointer :: cascade_seed, cascade_target
  type(cascade_t), pointer :: cascade1, cascade2
  integer(TC) :: bc_seed, bc_target
  bc_seed = ibset (0_TC, pos_seed-1)
  bc_target = ibset (0_TC, pos_target-1)
  cascade_seed => cascade_set%first_t
  LOOP_SEED: do while (associated (cascade_seed))
     if (cascade_seed%bincode == bc_seed) then
        cascade_target => cascade_set%first_t
        LOOP_TARGET: do while (associated (cascade_target))
           if (cascade_target%bincode == bc_target) then
              cascade1 => cascade_set%first_t
              LOOP_T: do while (associated (cascade1))
                 if ((cascade1 .disjunct. cascade_target) &
                       .and. .not. (cascade1 .disjunct. cascade_seed)) then
                    cascade2 => cascade_set%first
                    LOOP_S: do while (associated (cascade2))
                       if ((cascade2 .disjunct. cascade_target) &
                             .and. (cascade2 .disjunct. cascade1)) then
                          call cascade_match_pair &
                                (cascade_set, cascade1, cascade2, .false.)
                       end if
                       call terminate_now_if_signal ()
                       cascade2 => cascade2%next
                    end do LOOP_S
                 end if
                 call terminate_now_if_signal ()
                 cascade1 => cascade1%next
              end do LOOP_T
           end if
           call terminate_now_if_signal ()
           cascade_target => cascade_target%next
        end do LOOP_TARGET
     end if
     call terminate_now_if_signal ()
     cascade_seed => cascade_seed%next
  end do LOOP_SEED
end subroutine cascade_set_generate_t
```

This part completes the phase space for decay processes. It is similar to s-channel cascade generation, but combines two cascade with the particular cascade of the incoming particle. This particular cascade is expected to be pointed at by `first_t`.

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_generate_decay (cascade_set)
  type(cascade_set_t), intent(inout), target :: cascade_set
  type(cascade_t), pointer :: cascade1, cascade2
  type(cascade_t), pointer :: cascade_in
  cascade_in => cascade_set%first_t
  cascade1 => cascade_set%first
  do while (associated (cascade1))
     if (cascade1 .disjunct. cascade_in) then
        cascade2 => cascade1%next
```

```
      do while (associated (cascade2))
         if ((cascade2 .disjunct. cascade1) &
              .and. (cascade2 .disjunct. cascade_in)) then
            call cascade_match_triplet (cascade_set, &
                 cascade1, cascade2, cascade_in, .true.)
         end if
         call terminate_now_if_signal ()
         cascade2 => cascade2%next
      end do
   end if
   call terminate_now_if_signal ()
   cascade1 => cascade1%next
  end do
end subroutine cascade_set_generate_decay
```

This part completes the phase space for scattering processes. We combine a t-channel cascade (containing the seed) with a s-channel cascade and the target.

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_generate_scattering &
     (cascade_set, ns1, ns2, nt1, nt2, pos_seed, pos_target)
  type(cascade_set_t), intent(inout), target :: cascade_set
  integer, intent(in) :: pos_seed, pos_target
  integer, intent(in) :: ns1, ns2, nt1, nt2
  type(cascade_t), pointer :: cascade_seed, cascade_target
  type(cascade_t), pointer :: cascade1, cascade2
  integer(TC) :: bc_seed, bc_target
  bc_seed = ibset (0_TC, pos_seed-1)
  bc_target = ibset (0_TC, pos_target-1)
  cascade_seed => cascade_set%first_t
  LOOP_SEED: do while (associated (cascade_seed))
     if (cascade_seed%index < ns1) then
        cascade_seed => cascade_seed%next
        cycle LOOP_SEED
     else if (cascade_seed%index > ns2) then
        exit LOOP_SEED
     else if (cascade_seed%bincode == bc_seed) then
        cascade_target => cascade_set%first_t
        LOOP_TARGET: do while (associated (cascade_target))
           if (cascade_target%index < nt1) then
              cascade_target => cascade_target%next
              cycle LOOP_TARGET
           else if (cascade_target%index > nt2) then
              exit LOOP_TARGET
           else if (cascade_target%bincode == bc_target) then
              cascade1 => cascade_set%first_t
              LOOP_T: do while (associated (cascade1))
                 if ((cascade1 .disjunct. cascade_target) &
                       .and. .not. (cascade1 .disjunct. cascade_seed)) then
                    cascade2 => cascade_set%first
                    LOOP_S: do while (associated (cascade2))
                       if ((cascade2 .disjunct. cascade_target) &
                            .and. (cascade2 .disjunct. cascade1)) then
                          call cascade_match_triplet (cascade_set, &
```

```
                                cascade1, cascade2, cascade_target, .false.)
                         end if
                         call terminate_now_if_signal ()
                         cascade2 => cascade2%next
                      end do LOOP_S
                   end if
                   call terminate_now_if_signal ()
                   cascade1 => cascade1%next
                end do LOOP_T
             end if
             call terminate_now_if_signal ()
             cascade_target => cascade_target%next
          end do LOOP_TARGET
       end if
       call terminate_now_if_signal ()
       cascade_seed => cascade_seed%next
    end do LOOP_SEED
  end subroutine cascade_set_generate_scattering
```

18.7.13 Groves

Before assigning groves, assign hashcodes to the resonance patterns, so they can easily be compared.

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_set_assign_resonance_hash (cascade_set)
    type(cascade_set_t), intent(inout) :: cascade_set
    type(cascade_t), pointer :: cascade
    cascade => cascade_set%first_k
    do while (associated (cascade))
       call cascade_assign_resonance_hash (cascade)
       cascade => cascade%next
    end do
  end subroutine cascade_set_assign_resonance_hash
```

After all cascades are recorded, we group the complete cascades in groves. A grove consists of cascades with identical multiplicity, number of resonances, log-enhanced, t-channel lines, and resonance flavors.

⟨*Cascades: procedures*⟩+≡

```
  subroutine cascade_set_assign_groves (cascade_set)
    type(cascade_set_t), intent(inout), target :: cascade_set
    type(cascade_t), pointer :: cascade1, cascade2
    integer :: multiplicity
    integer :: n_resonances, n_log_enhanced, n_t_channel, n_off_shell
    integer :: res_hash
    integer :: grove
    grove = 0
    cascade1 => cascade_set%first_k
    do while (associated (cascade1))
       if (cascade1%active .and. cascade1%complete &
            .and. cascade1%grove == 0) then
          grove = grove + 1
          cascade1%grove = grove
```

```
            multiplicity = cascade1%multiplicity
            n_resonances = cascade1%n_resonances
            n_log_enhanced = cascade1%n_log_enhanced
            n_off_shell = cascade1%n_off_shell
            n_t_channel = cascade1%n_t_channel
            res_hash = cascade1%res_hash
            cascade2 => cascade1%next
            do while (associated (cascade2))
               if (cascade2%grove == 0) then
                  if (cascade2%multiplicity == multiplicity &
                       .and. cascade2%n_resonances == n_resonances &
                       .and. cascade2%n_log_enhanced == n_log_enhanced &
                       .and. cascade2%n_off_shell == n_off_shell &
                       .and. cascade2%n_t_channel == n_t_channel &
                       .and. cascade2%res_hash == res_hash) then
                     cascade2%grove = grove
                  end if
               end if
               call terminate_now_if_signal ()
               cascade2 => cascade2%next
            end do
         end if
         call terminate_now_if_signal ()
         cascade1 => cascade1%next
      end do
      cascade_set%n_groves = grove
   end subroutine cascade_set_assign_groves
```

18.7.14 Generate the phase space file

Generate a complete phase space configuration.

For each flavor assignment: First, all s-channel graphs that can be built up from the outgoing particles. Then we distinguish (1) decay, where we complete the s-channel graphs by connecting to the input line, and (2) scattering, where we now generate t-channel graphs by introducing an incoming particle, and complete this by connecting to the other incoming particle.

After all cascade sets have been generated, merge them into a common set. This eliminates redunancies between flavor assignments.

⟨*Cascades: public*⟩+≡

```
   public :: cascade_set_generate
```

⟨*Cascades: procedures*⟩+≡

```
   subroutine cascade_set_generate &
        (cascade_set, model, n_in, n_out, flv, phs_par, fatal_beam_decay)
     type(cascade_set_t), intent(out) :: cascade_set
     class(model_data_t), intent(in), target :: model
     integer, intent(in) :: n_in, n_out
     type(flavor_t), dimension(:,:), intent(in) :: flv
     type(phs_parameters_t), intent(in) :: phs_par
     logical, intent(in) :: fatal_beam_decay
     type(cascade_set_t), dimension(:), allocatable :: cset
     type(cascade_t), pointer :: cascade
```

```
  integer :: i
  if (phase_space_vanishes (phs_par%sqrts, n_in, flv))  return
  call cascade_set_init (cascade_set, model, n_in, n_out, phs_par, &
     fatal_beam_decay, flv)
  allocate (cset (size (flv, 2)))
  do i = 1, size (cset)
     call cascade_set_generate_single (cset(i), &
          model, n_in, n_out, flv(:,i), phs_par, fatal_beam_decay)
     cascade => cset(i)%first_k
     do while (associated (cascade))
        if (cascade%active .and. cascade%complete) then
           call cascade_set_add_copy (cascade_set, cascade)
        end if
        cascade => cascade%next
     end do
     call cascade_set_final (cset(i))
  end do
  cascade_set%first_k => cascade_set%first
  call cascade_set_assign_resonance_hash (cascade_set)
  call cascade_set_assign_groves (cascade_set)
end subroutine cascade_set_generate
```

This generates phase space for a single channel, without assigning groves.

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_set_generate_single (cascade_set, &
    model, n_in, n_out, flv, phs_par, fatal_beam_decay)
  type(cascade_set_t), intent(out) :: cascade_set
  class(model_data_t), intent(in), target :: model
  integer, intent(in) :: n_in, n_out
  type(flavor_t), dimension(:), intent(in) :: flv
  type(phs_parameters_t), intent(in) :: phs_par
  logical, intent(in) :: fatal_beam_decay
  integer :: n11, n12, n21, n22
  call cascade_set_init (cascade_set, model, n_in, n_out, phs_par, &
     fatal_beam_decay)
  call cascade_set_add_outgoing (cascade_set, flv(n_in+1:))
  call cascade_set_generate_s (cascade_set)
  select case (n_in)
  case(1)
     call cascade_set_add_incoming &
          (cascade_set, n11, n12, n_out + 1, flv(1))
     call cascade_set_generate_decay (cascade_set)
  case(2)
     call cascade_set_add_incoming &
          (cascade_set, n11, n12, n_out + 1, flv(2))
     call cascade_set_add_incoming &
          (cascade_set, n21, n22, n_out + 2, flv(1))
     call cascade_set_generate_t (cascade_set, n_out + 1, n_out + 2)
     call cascade_set_generate_t (cascade_set, n_out + 2, n_out + 1)
     call cascade_set_generate_scattering &
          (cascade_set, n11, n12, n21, n22, n_out + 1, n_out + 2)
     call cascade_set_generate_scattering &
          (cascade_set, n21, n22, n11, n12, n_out + 2, n_out + 1)
```

```
    end select
  end subroutine cascade_set_generate_single
```

Sanity check: Before anything else is done, check if there could possibly be any phase space.

⟨*Cascades: procedures*⟩+≡

```
  function phase_space_vanishes (sqrts, n_in, flv) result (flag)
    logical :: flag
    real(default), intent(in) :: sqrts
    integer, intent(in) :: n_in
    type(flavor_t), dimension(:,:), intent(in) :: flv
    real(default), dimension(:,:), allocatable :: mass
    real(default), dimension(:), allocatable :: mass_in, mass_out
    integer :: n_prt, n_flv, i, j
    flag = .false.
    if (sqrts <= 0) then
       call msg_error ("Phase space vanishes (sqrts must be positive)")
       flag = .true.;  return
    end if
    n_prt = size (flv, 1)
    n_flv = size (flv, 2)
    allocate (mass (n_prt, n_flv), mass_in (n_flv), mass_out (n_flv))
    !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
    !!! mass = flv%get_mass ()
    do i = 1, n_prt
       do j = 1, n_flv
          mass(i,j) = flv(i,j)%get_mass ()
       end do
    end do
    mass_in = sum (mass(:n_in,:), 1)
    mass_out = sum (mass(n_in+1:,:), 1)
    if (any (mass_in > sqrts)) then
       call msg_error ("Mass sum of incoming particles " &
            // "is more than available energy")
       flag = .true.;  return
    end if
    if (any (mass_out > sqrts)) then
       call msg_error ("Mass sum of outgoing particles " &
            // "is more than available energy")
       flag = .true.;  return
    end if
  end function phase_space_vanishes
```

18.7.15 Return the resonance histories for subtraction

This appears to be essential (re-export of some imported assignment?)!

⟨*Cascades: public*⟩+≡

```
  public :: assignment(=)
```

Extract the resonance set from a complete cascade.

⟨*Cascades: cascade: TBP*⟩≡

```
  procedure :: extract_resonance_history => cascade_extract_resonance_history
```

⟨*Cascades: procedures*⟩+≡

```
subroutine cascade_extract_resonance_history &
     (cascade, res_hist, model, n_out)
  class(cascade_t), intent(in), target :: cascade
  type(resonance_history_t), intent(out) :: res_hist
  class(model_data_t), intent(in), target :: model
  integer, intent(in) :: n_out
  type(resonance_info_t) :: resonance
  integer :: i, mom_id, pdg
  call msg_debug2 (D_PHASESPACE, "cascade_extract_resonance_history")
  if (cascade%n_resonances > 0) then
     if (cascade%has_children) then
        call msg_debug2 (D_PHASESPACE, "cascade has resonances and children")
        do i = 1, size(cascade%tree_resonant)
           if (cascade%tree_resonant (i)) then
              mom_id = cascade%tree (i)
              pdg = cascade%tree_pdg (i)
              call resonance%init (mom_id, pdg, model, n_out)
              if (debug2_active (D_PHASESPACE)) then
                 print *, 'D: Adding resonance'
                 call resonance%write ()
              end if
              call res_hist%add_resonance (resonance)
           end if
        end do
     end if
  end if
end subroutine cascade_extract_resonance_history
```

⟨*Cascades: public*⟩+≡

```
public :: cascade_set_get_n_trees
```

⟨*Cascades: procedures*⟩+≡

```
function cascade_set_get_n_trees (cascade_set) result (n)
  type(cascade_set_t), intent(in), target :: cascade_set
  integer :: n
  type(cascade_t), pointer :: cascade
  integer :: grove
  call msg_debug (D_PHASESPACE, "cascade_set_get_n_trees")
  n = 0
  do grove = 1, cascade_set%n_groves
     cascade => cascade_set%first_k
     do while (associated (cascade))
        if (cascade%active .and. cascade%complete) then
           if (cascade%grove == grove) then
              n = n + 1
           end if
        end if
        cascade => cascade%next
     end do
  end do
  call msg_debug (D_PHASESPACE, "n", n)
end function cascade_set_get_n_trees
```

Distill the set of resonance histories from the cascade set. The result is an array which contains each valid history exactly once.

⟨*Cascades: public*⟩+≡
```
public :: cascade_set_get_resonance_histories
```

⟨*Cascades: procedures*⟩+≡
```
subroutine cascade_set_get_resonance_histories (cascade_set, n_filter, res_hists)
  type(cascade_set_t), intent(in), target :: cascade_set
  integer, intent(in), optional :: n_filter
  type(resonance_history_t), dimension(:), allocatable, intent(out) :: res_hists
  type(resonance_history_t), dimension(:), allocatable :: tmp
  type(cascade_t), pointer :: cascade
  type(resonance_history_t) :: res_hist
  type(resonance_history_set_t) :: res_hist_set
  integer :: grove, i, n_hists
  logical :: included, add_to_list
  call msg_debug (D_PHASESPACE, "cascade_set_get_resonance_histories")
  call res_hist_set%init (n_filter = n_filter)
  do grove = 1, cascade_set%n_groves
     cascade => cascade_set%first_k
     do while (associated (cascade))
        if (cascade%active .and. cascade%complete) then
           if (cascade%grove == grove) then
              call msg_debug2 (D_PHASESPACE, "grove", grove)
              call cascade%extract_resonance_history &
                   (res_hist, cascade_set%model, cascade_set%n_out)
              call res_hist_set%enter (res_hist)
           end if
        end if
        cascade => cascade%next
     end do
  end do
  call res_hist_set%to_array (res_hists)
end subroutine cascade_set_get_resonance_histories
```

18.7.16 Unit tests

Test module, followed by the corresponding implementation module.

⟨`cascades_ut.f90`⟩≡
```
⟨File header⟩

module cascades_ut
  use unit_tests
  use cascades_uti

⟨Standard module head⟩

⟨Cascades: public test⟩

contains
```

```
  ⟨Cascades: test driver⟩

  end module cascades_ut
```

⟨cascades_uti.f90⟩≡

```
  ⟨File header⟩

  module cascades_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use numeric_utils
    use flavors
    use model_data
    use phs_forests, only: phs_parameters_t
    use resonances, only: resonance_history_t

    use cascades

  ⟨Standard module head⟩

  ⟨Cascades: test declarations⟩

  contains

  ⟨Cascades: tests⟩

  end module cascades_uti
```

API: driver for the unit tests below.

⟨*Cascades: public test*⟩≡

```
    public :: cascades_test
```

⟨*Cascades: test driver*⟩≡

```
    subroutine cascades_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Cascades: execute tests⟩
    end subroutine cascades_test
```

Check cascade setup

Checking the basic setup up of the phase space cascade parameterizations.

⟨*Cascades: execute tests*⟩≡

```
    call test (cascades_1, "cascades_1", &
         "check cascade setup", &
         u, results)
```

⟨*Cascades: test declarations*⟩≡

```
    public :: cascades_1
```

⟨*Cascades: tests*⟩≡

```
    subroutine cascades_1 (u)
      integer, intent(in) :: u
      type(model_data_t), target :: model
```

```
    type(flavor_t), dimension(5,2) :: flv
    type(cascade_set_t) :: cascade_set
    type(phs_parameters_t) :: phs_par

    write (u, "(A)")  "* Test output: cascades_1"
    write (u, "(A)")  "*   Purpose: test cascade phase space functions"
    write (u, "(A)")

    write (u, "(A)")  "* Initializing"
    write (u, "(A)")

    call model%init_sm_test ()

    call flv(1,1)%init ( 2, model)
    call flv(2,1)%init (-2, model)
    call flv(3,1)%init ( 1, model)
    call flv(4,1)%init (-1, model)
    call flv(5,1)%init (21, model)
    call flv(1,2)%init ( 2, model)
    call flv(2,2)%init (-2, model)
    call flv(3,2)%init ( 2, model)
    call flv(4,2)%init (-2, model)
    call flv(5,2)%init (21, model)
    phs_par%sqrts = 1000._default
    phs_par%off_shell = 2

    write (u, "(A)")
    write (u, "(A)")  "* Generating the cascades"
    write (u, "(A)")

    call cascade_set_generate (cascade_set, model, 2, 3, flv, phs_par,.true.)
    call cascade_set_write (cascade_set, u)
    call cascade_set_write_file_format (cascade_set, u)

    write (u, "(A)")  "* Cleanup"
    write (u, "(A)")

    call cascade_set_final (cascade_set)
    call model%final ()

    write (u, *)
    write (u, "(A)")  "* Test output end: cascades_1"

  end subroutine cascades_1
```

Check resonance history

⟨*Cascades: execute tests*⟩+≡

```
  call test(cascades_2, "cascades_2", &
            "Check resonance history", u, results)
```

⟨*Cascades: test declarations*⟩+≡

```
  public :: cascades_2
```

⟨*Cascades: tests*⟩+≡

```
subroutine cascades_2 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(flavor_t), dimension(5,1) :: flv
  type(cascade_set_t) :: cascade_set
  type(phs_parameters_t) :: phs_par
  type(resonance_history_t), dimension(:), allocatable :: res_hists
  integer :: n, i
  write (u, "(A)")  "* Test output: cascades_2"
  write (u, "(A)")  "*   Purpose: Check resonance history"
  write (u, "(A)")

  write (u, "(A)")  "* Initializing"
  write (u, "(A)")

  call model%init_sm_test ()

  call flv(1,1)%init ( 2, model)
  call flv(2,1)%init (-2, model)
  call flv(3,1)%init ( 1, model)
  call flv(4,1)%init (-1, model)
  call flv(5,1)%init (22, model)
  phs_par%sqrts = 1000._default
  phs_par%off_shell = 2

  write (u, "(A)")
  write (u, "(A)")  "* Generating the cascades"
  write (u, "(A)")

  call cascade_set_generate (cascade_set, model, 2, 3, flv, phs_par,.true.)
  call cascade_set_get_resonance_histories (cascade_set, res_hists = res_hists)
  n = cascade_set_get_n_trees (cascade_set)
  call assert_equal (u, n, 24, "Number of trees")
  do i = 1, size(res_hists)
     call res_hists(i)%write (u)
     write (u, "(A)")
  end do

  write (u, "(A)")  "* Cleanup"
  write (u, "(A)")

  call cascade_set_final (cascade_set)
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: cascades_2"
end subroutine cascades_2
```

18.8 WOOD phase space

This is the module that interfaces the `phs_forests` phase-space treatment and the `cascades` module for generating phase-space channels. As an extension of the `phs_base` abstract type, the phase-space configuration and instance implement the standard API.

(Currently, this is the only generic phase-space implementation of `WHIZARD`. For trivial two-particle phase space, there is `phs_wood` as an alternative.)

⟨`phs_wood.f90`⟩≡

```
⟨File header⟩

module phs_wood

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use constants
  use numeric_utils
  use diagnostics
  use os_interface
  use md5
  use physics_defs
  use lorentz
  use model_data
  use flavors
  use process_constants
  use sf_mappings
  use sf_base
  use phs_base
  use mappings
  use resonances, only: resonance_history_t
  use phs_forests
  use cascades

⟨Standard module head⟩

⟨PHS wood: public⟩

⟨PHS wood: parameters⟩

⟨PHS wood: types⟩

contains

⟨PHS wood: procedures⟩

end module phs_wood
```

18.8.1 Configuration

⟨*PHS wood: parameters*⟩≡

```
  integer, parameter, public :: EXTENSION_NONE = 0
  integer, parameter, public :: EXTENSION_DEFAULT = 1
```

```
integer, parameter, public :: EXTENSION_DGLAP = 2
```

⟨*PHS wood: public*⟩≡

```
public :: phs_wood_config_t
```

⟨*PHS wood: types*⟩≡

```
type, extends (phs_config_t) :: phs_wood_config_t
   character(32) :: md5sum_forest = ""
   integer :: io_unit = 0
   logical :: io_unit_keep_open = .false.
   logical :: use_equivalences = .false.
   logical :: fatal_beam_decay = .true.
   type(mapping_defaults_t) :: mapping_defaults
   type(phs_parameters_t) :: par
   type(string_t) :: run_id
   type(cascade_set_t), allocatable :: cascade_set
   type(phs_forest_t) :: forest
   type(os_data_t) :: os_data
   integer :: extension_mode = EXTENSION_NONE
 contains
 ⟨PHS wood: phs wood config: TBP⟩
end type phs_wood_config_t
```

Finalizer. We should delete the cascade set and the forest subobject.

Also close the I/O unit, just in case. (We assume that `io_unit` is not standard input/output.)

⟨*PHS wood: phs wood config: TBP*⟩≡

```
procedure :: final => phs_wood_config_final
```

⟨*PHS wood: procedures*⟩≡

```
subroutine phs_wood_config_final (object)
  class(phs_wood_config_t), intent(inout) :: object
  logical :: opened
  if (object%io_unit /= 0) then
     inquire (unit = object%io_unit, opened = opened)
     if (opened)  close (object%io_unit)
  end if
  call object%clear_phase_space ()
  call phs_forest_final (object%forest)
end subroutine phs_wood_config_final
```

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
procedure :: increase_n_par => phs_wood_config_increase_n_par
```

⟨*PHS wood: procedures*⟩+≡

```
subroutine phs_wood_config_increase_n_par (phs_config)
  class(phs_wood_config_t), intent(inout) :: phs_config
  select case (phs_config%extension_mode)
  case (EXTENSION_DEFAULT)
     phs_config%n_par = phs_config%n_par + 3
  case (EXTENSION_DGLAP)
     phs_config%n_par = phs_config%n_par + 4
  end select
end subroutine phs_wood_config_increase_n_par
```

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
procedure :: set_extension_mode => phs_wood_config_set_extension_mode
```
⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_config_set_extension_mode (phs_config, mode)
  class(phs_wood_config_t), intent(inout) :: phs_config
  integer, intent(in) :: mode
  phs_config%extension_mode = mode
end subroutine phs_wood_config_set_extension_mode
```

Output. The contents of the PHS forest are not printed explicitly.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
procedure :: write => phs_wood_config_write
```
⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_config_write (object, unit, include_id)
  class(phs_wood_config_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: include_id
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)") &
       "Partonic phase-space configuration (phase-space forest):"
  call object%base_write (unit)
  write (u, "(1x,A)")    "Phase-space configuration parameters:"
  call phs_parameters_write (object%par, u)
  call object%mapping_defaults%write (u)
  write (u, "(3x,A,A,A)")  "Run ID: '", char (object%run_id), "'"
end subroutine phs_wood_config_write
```

Print the PHS forest contents.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
procedure :: write_forest => phs_wood_config_write_forest
```
⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_config_write_forest (object, unit)
  class(phs_wood_config_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  call phs_forest_write (object%forest, u)
end subroutine phs_wood_config_write_forest
```

Set the phase-space parameters that the configuration generator requests.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
procedure :: set_parameters => phs_wood_config_set_parameters
```
⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_config_set_parameters (phs_config, par)
  class(phs_wood_config_t), intent(inout) :: phs_config
  type(phs_parameters_t), intent(in) :: par
  phs_config%par = par
end subroutine phs_wood_config_set_parameters
```

Enable the generation of channel equivalences (when calling `configure`).

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
procedure :: enable_equivalences => phs_wood_config_enable_equivalences
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_config_enable_equivalences (phs_config)
  class(phs_wood_config_t), intent(inout) :: phs_config
  phs_config%use_equivalences = .true.
end subroutine phs_wood_config_enable_equivalences
```

Set the phase-space mapping parameters that the configuration generator requests.g

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
procedure :: set_mapping_defaults => phs_wood_config_set_mapping_defaults
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_config_set_mapping_defaults (phs_config, mapping_defaults)
  class(phs_wood_config_t), intent(inout) :: phs_config
  type(mapping_defaults_t), intent(in) :: mapping_defaults
  phs_config%mapping_defaults = mapping_defaults
end subroutine phs_wood_config_set_mapping_defaults
```

Define the input stream for the phase-space file as an open logical unit. The unit must be connected.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
procedure :: set_input => phs_wood_config_set_input
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_config_set_input (phs_config, unit)
  class(phs_wood_config_t), intent(inout) :: phs_config
  integer, intent(in) :: unit
  phs_config%io_unit = unit
  rewind (unit)
end subroutine phs_wood_config_set_input
```

18.8.2 Phase-space generation

This subroutine generates a phase space configuration using the `cascades` module. Note that this may take time, and the `cascade_set` subobject may consume a large amount of memory.

⟨*PHS wood: phs wood config: TBP*⟩+≡
```
procedure :: generate_phase_space => phs_wood_config_generate_phase_space
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_config_generate_phase_space (phs_config)
  class(phs_wood_config_t), intent(inout) :: phs_config
  integer :: off_shell, extra_off_shell
  call msg_message ("Phase space: generating configuration ...")
  off_shell = phs_config%par%off_shell
  allocate (phs_config%cascade_set)
  do extra_off_shell = 0, max (phs_config%n_tot - 3, 0)
     phs_config%par%off_shell = off_shell + extra_off_shell
```

```
      call cascade_set_generate (phs_config%cascade_set, &
           phs_config%model, phs_config%n_in, phs_config%n_out, &
           phs_config%flv, &
           phs_config%par, phs_config%fatal_beam_decay)
      if (cascade_set_is_valid (phs_config%cascade_set)) then
         exit
      else
         call msg_message ("Phase space: ... failed.  &
              &Increasing phs_off_shell ...")
      end if
   end do
   if (cascade_set_is_valid (phs_config%cascade_set)) then
      call msg_message ("Phase space: ... success.")
   else
      call msg_fatal ("Phase-space: generation failed")
   end if
end subroutine phs_wood_config_generate_phase_space
```

Using the generated phase-space configuration, write an appropriate phase-space file to the stored (or explicitly specified) I/O unit.

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
procedure :: write_phase_space => phs_wood_config_write_phase_space
```

⟨*PHS wood: procedures*⟩+≡

```
subroutine phs_wood_config_write_phase_space (phs_config, &
     filename_vis, unit)
   class(phs_wood_config_t), intent(in) :: phs_config
   integer, intent(in), optional :: unit
   type(string_t), intent(in), optional :: filename_vis
   type(string_t) :: setenv_tex, setenv_mp, pipe, pipe_dvi
   integer :: u, unit_tex, unit_dev, status
   if (allocated (phs_config%cascade_set)) then
      if (present (unit)) then
         u = unit
      else
         u = phs_config%io_unit
      end if
      write (u, "(1x,A,A)") "process ", char (phs_config%id)
      write (u, "(A)")
      call cascade_set_write_process_bincode_format (phs_config%cascade_set, u)
      write (u, "(A)")
      write (u, "(3x,A,A,A32,A)") "md5sum_process    = ", &
           '"', phs_config%md5sum_process, '"'
      write (u, "(3x,A,A,A32,A)") "md5sum_model_par  = ", &
           '"', phs_config%md5sum_model_par, '"'
      write (u, "(3x,A,A,A32,A)") "md5sum_phs_config = ", &
           '"', phs_config%md5sum_phs_config, '"'
      call phs_parameters_write (phs_config%par, u)
      call cascade_set_write_file_format (phs_config%cascade_set, u)
      if (phs_config%vis_channels) then
         unit_tex = free_unit ()
         open (unit=unit_tex, file=char(filename_vis // ".tex"), &
              action="write", status="replace")
         call cascade_set_write_graph_format (phs_config%cascade_set, &
```

```
     filename_vis // "-graphs", phs_config%id, unit_tex)
close (unit_tex)
call msg_message ("Phase space: visualizing channels in file " &
     // char(trim(filename_vis)) // "...")
if (phs_config%os_data%event_analysis_ps) then
   BLOCK: do
      unit_dev = free_unit ()
      open (file = "/dev/null", unit = unit_dev, &
           action = "write", iostat = status)
      if (status /= 0) then
         pipe = ""
         pipe_dvi = ""
      else
         pipe = " > /dev/null"
         pipe_dvi = " 2>/dev/null 1>/dev/null"
      end if
      close (unit_dev)
      if (phs_config%os_data%whizard_texpath /= "") then
         setenv_tex = "TEXINPUTS=" // &
              phs_config%os_data%whizard_texpath // ":$TEXINPUTS "
         setenv_mp = "MPINPUTS=" // &
              phs_config%os_data%whizard_texpath // ":$MPINPUTS "
      else
         setenv_tex = ""
         setenv_mp = ""
      end if
      call os_system_call (setenv_tex // &
           phs_config%os_data%latex // " " // &
           filename_vis // ".tex " // pipe, status)
      if (status /= 0)  exit BLOCK
      if (phs_config%os_data%mpost /= "") then
         call os_system_call (setenv_mp // &
              phs_config%os_data%mpost // " " // &
              filename_vis // "-graphs.mp" // pipe, status)
      else
         call msg_fatal ("Could not use MetaPOST.")
      end if
      if (status /= 0)  exit BLOCK
      call os_system_call (setenv_tex // &
           phs_config%os_data%latex // " " // &
           filename_vis // ".tex" // pipe, status)
      if (status /= 0)  exit BLOCK
      call os_system_call &
           (phs_config%os_data%dvips // " -o " // filename_vis &
           // ".ps " // filename_vis // ".dvi" // pipe_dvi, status)
      if (status /= 0)  exit BLOCK
      if (phs_config%os_data%event_analysis_pdf) then
         call os_system_call (phs_config%os_data%ps2pdf // " " // &
              filename_vis // ".ps", status)
         if (status /= 0)  exit BLOCK
      end if
      exit BLOCK
   end do BLOCK
   if (status /= 0) then
```

```
            call msg_error ("Unable to compile analysis output file")
         end if
      end if
   end if
else
   call msg_fatal ("Phase-space configuration: &
        &no phase space object generated")
end if
end subroutine phs_wood_config_write_phase_space
```

Clear the phase-space configuration. This is useful since the object may become *really* large.

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
procedure :: clear_phase_space => phs_wood_config_clear_phase_space
```

⟨*PHS wood: procedures*⟩+≡

```
subroutine phs_wood_config_clear_phase_space (phs_config)
  class(phs_wood_config_t), intent(inout) :: phs_config
  if (allocated (phs_config%cascade_set)) then
     call cascade_set_final (phs_config%cascade_set)
     deallocate (phs_config%cascade_set)
  end if
end subroutine phs_wood_config_clear_phase_space
```

Extract the set of resonance histories

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
procedure :: extract_resonance_histories &
     => phs_wood_config_extract_resonance_histories
```

⟨*PHS wood: procedures*⟩+≡

```
subroutine phs_wood_config_extract_resonance_histories (phs_config, res_hist)
  class(phs_wood_config_t), intent(in) :: phs_config
  type(resonance_history_t), dimension(:), allocatable, intent(out) :: res_hist
  call phs_config%forest%extract_resonance_histories (res_hist)
end subroutine phs_wood_config_extract_resonance_histories
```

18.8.3 Phase-space configuration

We read the phase-space configuration from the stored I/O unit. If this is not set, we assume that we have to generate a phase space configuration. When done, we open a scratch file and write the configuration.

If `rebuild` is set, we should trash any existing phase space file and build a new one. Otherwise, we try to use an old one, which we check for existence and integrity. If `ignore_mismatch` is set, we reuse an existing file even if it does not match the current setup.

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
procedure :: configure => phs_wood_config_configure
```

⟨*PHS wood: procedures*⟩+≡

```
subroutine phs_wood_config_configure (phs_config, sqrts, &
     sqrts_fixed, cm_frame, azimuthal_dependence, rebuild, ignore_mismatch, &
     nlo_type)
```

```
class(phs_wood_config_t), intent(inout) :: phs_config
real(default), intent(in) :: sqrts
logical, intent(in), optional :: sqrts_fixed
logical, intent(in), optional :: cm_frame
logical, intent(in), optional :: azimuthal_dependence
logical, intent(in), optional :: rebuild
logical, intent(in), optional :: ignore_mismatch
integer, intent(in), optional :: nlo_type
type(string_t) :: filename, filename_vis
logical :: variable_limits
logical :: ok, exist, found, check, match, rebuild_phs
integer :: g, c0, c1, n
if (present (nlo_type)) then
  phs_config%nlo_type = nlo_type
else
  phs_config%nlo_type = BORN
end if
phs_config%sqrts = sqrts
phs_config%par%sqrts = sqrts
if (present (sqrts_fixed)) &
     phs_config%sqrts_fixed = sqrts_fixed
if (present (cm_frame)) &
     phs_config%cm_frame = cm_frame
if (present (azimuthal_dependence)) &
     phs_config%azimuthal_dependence = azimuthal_dependence
if (present (rebuild)) then
   rebuild_phs = rebuild
else
   rebuild_phs = .true.
end if
if (present (ignore_mismatch)) then
   check = .not. ignore_mismatch
   if (ignore_mismatch) &
        call msg_warning ("Reading phs file: MD5 sum check disabled")
else
   check = .true.
end if
phs_config%md5sum_forest = ""
call phs_config%compute_md5sum (include_id = .false.)
if (phs_config%io_unit == 0) then
   if (phs_config%run_id /= "") then
      filename = phs_config%id // "." // phs_config%run_id // ".phs"
      filename_vis = phs_config%id // "." // phs_config%run_id // "_phs"
   else
      filename = phs_config%id // ".phs"
      filename_vis = phs_config%id // "_phs"
   end if
   if (.not. rebuild_phs) then
      if (check) then
         call phs_config%read_phs_file (exist, found, match)
         rebuild_phs = .not. (exist .and. found .and. match)
      else
         call phs_config%read_phs_file (exist, found)
         rebuild_phs = .not. (exist .and. found)
```

```
      end if
   end if
   if (rebuild_phs) then
      call phs_config%generate_phase_space ()
      phs_config%io_unit = free_unit ()
      if (phs_config%id /= "") then
         call msg_message ("Phase space: writing configuration file '" &
              // char (filename) // "'")
         open (phs_config%io_unit, file = char (filename), &
              status = "replace", action = "readwrite")
      else
         open (phs_config%io_unit, status = "scratch", action = "readwrite")
      end if
      call phs_config%write_phase_space (filename_vis)
      rewind (phs_config%io_unit)
   else
      call msg_message ("Phase space: keeping configuration file '" &
           // char (filename) // "'")
   end if
end if
if (phs_config%io_unit == 0) then
   ok = .true.
else
  call phs_forest_read (phs_config%forest, phs_config%io_unit, &
        phs_config%id, phs_config%n_in, phs_config%n_out, &
        phs_config%model, ok)
   if (.not. phs_config%io_unit_keep_open) then
      close (phs_config%io_unit)
      phs_config%io_unit = 0
   end if
end if
if (ok) then
   call phs_forest_set_flavors (phs_config%forest, phs_config%flv(:,1))
   variable_limits = .not. phs_config%cm_frame
   call phs_forest_set_parameters &
        (phs_config%forest, phs_config%mapping_defaults, variable_limits)
   call phs_forest_setup_prt_combinations (phs_config%forest)
   phs_config%n_channel = phs_forest_get_n_channels (phs_config%forest)
   phs_config%n_par = phs_forest_get_n_parameters (phs_config%forest)
   allocate (phs_config%channel (phs_config%n_channel))
   if (phs_config%use_equivalences) then
      call phs_forest_set_equivalences (phs_config%forest)
      call phs_forest_get_equivalences (phs_config%forest, &
           phs_config%channel, phs_config%azimuthal_dependence)
      phs_config%provides_equivalences = .true.
   end if
   call phs_forest_set_s_mappings (phs_config%forest)
   call phs_config%record_on_shell ()
   if (phs_config%mapping_defaults%enable_s_mapping) then
      call phs_config%record_s_mappings ()
   end if
   allocate (phs_config%chain (phs_config%n_channel), source = 0)
   do g = 1, phs_forest_get_n_groves (phs_config%forest)
      call phs_forest_get_grove_bounds (phs_config%forest, g, c0, c1, n)
```

```
         phs_config%chain (c0:c1) = g
      end do
      phs_config%provides_chains = .true.
      call phs_config%compute_md5sum_forest ()
   else
      write (msg_buffer, "(A,A,A)") &
           "Phase space: process '", &
           char (phs_config%id), "' not found in configuration file"
      call msg_fatal ()
   end if
end subroutine phs_wood_config_configure
```

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
procedure :: reshuffle_flavors => phs_wood_config_reshuffle_flavors
```

⟨*PHS wood: procedures*⟩+≡

```
subroutine phs_wood_config_reshuffle_flavors (phs_config, reshuffle, flv_extra)
  class(phs_wood_config_t), intent(inout) :: phs_config
  integer, intent(in), dimension(:), allocatable :: reshuffle
  type(flavor_t), intent(in) :: flv_extra
  call phs_forest_set_flavors (phs_config%forest, phs_config%flv(:,1), reshuffle, flv_extra)
end subroutine phs_wood_config_reshuffle_flavors
```

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
procedure :: set_momentum_links => phs_wood_config_set_momentum_links
```

⟨*PHS wood: procedures*⟩+≡

```
subroutine phs_wood_config_set_momentum_links (phs_config, reshuffle)
  class(phs_wood_config_t), intent(inout) :: phs_config
  integer, intent(in), dimension(:), allocatable :: reshuffle
  call phs_forest_set_momentum_links (phs_config%forest, reshuffle)
end subroutine phs_wood_config_set_momentum_links
```

Identify resonances which are marked by s-channel mappings for the whole phase space and report them to the channel array.

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
procedure :: record_s_mappings => phs_wood_config_record_s_mappings
```

⟨*PHS wood: procedures*⟩+≡

```
subroutine phs_wood_config_record_s_mappings (phs_config)
  class(phs_wood_config_t), intent(inout) :: phs_config
  logical :: flag
  real(default) :: mass, width
  integer :: c
  do c = 1, phs_config%n_channel
     call phs_forest_get_s_mapping (phs_config%forest, c, flag, mass, width)
     if (flag) then
        if (mass == 0) then
           call msg_fatal ("Phase space: s-channel resonance " &
                // " has zero mass")
        end if
        if (width == 0) then
           call msg_fatal ("Phase space: s-channel resonance " &
                // " has zero width")
```

```
        end if
        call phs_config%channel(c)%set_resonant (mass, width)
     end if
  end do
end subroutine phs_wood_config_record_s_mappings
```

Identify on-shell mappings for the whole phase space and report them to the channel array.

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
procedure :: record_on_shell => phs_wood_config_record_on_shell
```

⟨*PHS wood: procedures*⟩+≡

```
subroutine phs_wood_config_record_on_shell (phs_config)
  class(phs_wood_config_t), intent(inout) :: phs_config
  logical :: flag
  real(default) :: mass
  integer :: c
  do c = 1, phs_config%n_channel
     call phs_forest_get_on_shell (phs_config%forest, c, flag, mass)
     if (flag) then
        call phs_config%channel(c)%set_on_shell (mass)
     end if
  end do
end subroutine phs_wood_config_record_on_shell
```

The MD5 sum of the forest is computed in addition to the MD5 sum of the configuration. The reason is that the forest may depend on a user-provided external file. On the other hand, this MD5 sum encodes all information that is relevant for further processing. Therefore, the `get_md5sum` method returns this result, once it is available.

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
procedure :: compute_md5sum_forest => phs_wood_config_compute_md5sum_forest
```

⟨*PHS wood: procedures*⟩+≡

```
subroutine phs_wood_config_compute_md5sum_forest (phs_config)
  class(phs_wood_config_t), intent(inout) :: phs_config
  integer :: u
  u = free_unit ()
  open (u, status = "scratch", action = "readwrite")
  call phs_config%write_forest (u)
  rewind (u)
  phs_config%md5sum_forest = md5sum (u)
  close (u)
end subroutine phs_wood_config_compute_md5sum_forest
```

Return the most relevant MD5 sum. This overrides the method of the base type.

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
procedure :: get_md5sum => phs_wood_config_get_md5sum
```

⟨*PHS wood: procedures*⟩+≡

```
function phs_wood_config_get_md5sum (phs_config) result (md5sum)
  class(phs_wood_config_t), intent(in) :: phs_config
```

```
    character(32) :: md5sum
    if (phs_config%md5sum_forest /= "") then
       md5sum = phs_config%md5sum_forest
    else
       md5sum = phs_config%md5sum_phs_config
    end if
  end function phs_wood_config_get_md5sum
```

Check whether a phase-space configuration for the current process exists. We look for the phase-space file that should correspond to the current process. If we find it, we check the MD5 sums stored in the file against the MD5 sums in the current configuration (if required).

If successful, read the PHS file.

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
  procedure :: read_phs_file => phs_wood_read_phs_file
```

⟨*PHS wood: procedures*⟩+≡

```
  subroutine phs_wood_read_phs_file (phs_config, exist, found, match)
    class(phs_wood_config_t), intent(inout) :: phs_config
    logical, intent(out) :: exist
    logical, intent(out) :: found
    logical, intent(out), optional :: match
    type(string_t) :: filename
    integer :: u
    filename = phs_config%id // ".phs"
    inquire (file = char (filename), exist = exist)
    if (exist) then
       u = free_unit ()
       open (u, file = char (filename), action = "read", status = "old")
       call phs_forest_read (phs_config%forest, u, &
            phs_config%id, phs_config%n_in, phs_config%n_out, &
            phs_config%model, found, &
            phs_config%md5sum_process, &
            phs_config%md5sum_model_par, &
            phs_config%md5sum_phs_config, &
            match = match)
       close (u)
    else
       found = .false.
       if (present (match))  match = .false.
    end if
  end subroutine phs_wood_read_phs_file
```

Startup message, after configuration is complete.

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
  procedure :: startup_message => phs_wood_config_startup_message
```

⟨*PHS wood: procedures*⟩+≡

```
  subroutine phs_wood_config_startup_message (phs_config, unit)
    class(phs_wood_config_t), intent(in) :: phs_config
    integer, intent(in), optional :: unit
    integer :: n_groves, n_eq
    n_groves = phs_forest_get_n_groves (phs_config%forest)
```

```
    n_eq = phs_forest_get_n_equivalences (phs_config%forest)
    call phs_config%base_startup_message (unit)
    if (phs_config%n_channel == 1) then
       write (msg_buffer, "(A,2(I0,A))") &
            "Phase space: found ", phs_config%n_channel, &
            " channel, collected in ", n_groves, &
            " grove."
    else if (n_groves == 1) then
       write (msg_buffer, "(A,2(I0,A))") &
            "Phase space: found ", phs_config%n_channel, &
            " channels, collected in ", n_groves, &
            " grove."
       else
       write (msg_buffer, "(A,2(I0,A))") &
            "Phase space: found ", phs_config%n_channel, &
            " channels, collected in ", &
            phs_forest_get_n_groves (phs_config%forest), &
            " groves."
    end if
    call msg_message (unit = unit)
    if (phs_config%use_equivalences) then
       if (n_eq == 1) then
          write (msg_buffer, "(A,I0,A)") &
               "Phase space: Using ", n_eq, &
               " equivalence between channels."
       else
          write (msg_buffer, "(A,I0,A)") &
               "Phase space: Using ", n_eq, &
               " equivalences between channels."
       end if
    else
       write (msg_buffer, "(A)") &
            "Phase space: no equivalences between channels used."
    end if
    call msg_message (unit = unit)
    write (msg_buffer, "(A,2(1x,I0,1x,A))") &
         "Phase space: wood"
    call msg_message (unit = unit)
  end subroutine phs_wood_config_startup_message
```

Allocate an instance: the actual phase-space object.

⟨*PHS wood: phs wood config: TBP*⟩+≡

```
  procedure, nopass :: allocate_instance => phs_wood_config_allocate_instance
```

⟨*PHS wood: procedures*⟩+≡

```
  subroutine phs_wood_config_allocate_instance (phs)
    class(phs_t), intent(inout), pointer :: phs
    allocate (phs_wood_t :: phs)
  end subroutine phs_wood_config_allocate_instance
```

18.8.4 Kinematics implementation

We generate $\cos\theta$ and ϕ uniformly, covering the solid angle.

⟨*PHS wood: public*⟩+≡
```
public :: phs_wood_t
```

⟨*PHS wood: types*⟩+≡
```
type, extends (phs_t) :: phs_wood_t
   real(default) :: sqrts = 0
   type(phs_forest_t) :: forest
   real(default), dimension(3) :: r_real
   integer :: n_r_born = 0
 contains
 ⟨PHS wood: phs wood: TBP⟩
end type phs_wood_t
```

Output. The `verbose` setting is irrelevant, we just display the contents of the base object.

⟨*PHS wood: phs wood: TBP*⟩≡
```
procedure :: write => phs_wood_write
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_write (object, unit, verbose)
  class(phs_wood_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u
  u = given_output_unit (unit)
  call object%base_write (u)
end subroutine phs_wood_write
```

Write the forest separately.

⟨*PHS wood: phs wood: TBP*⟩+≡
```
procedure :: write_forest => phs_wood_write_forest
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_write_forest (object, unit)
  class(phs_wood_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  call phs_forest_write (object%forest, u)
end subroutine phs_wood_write_forest
```

Finalizer.

⟨*PHS wood: phs wood: TBP*⟩+≡
```
procedure :: final => phs_wood_final
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_final (object)
  class(phs_wood_t), intent(inout) :: object
  call phs_forest_final (object%forest)
end subroutine phs_wood_final
```

Initialization. We allocate arrays (base_init) and adjust the phase-space volume. The two-particle phase space volume is

$$\Phi_2 = \frac{1}{4(2\pi)^5} = 2.55294034614 \times 10^{-5} \tag{18.64}$$

independent of the particle masses.

⟨*PHS wood: phs wood: TBP*⟩+≡
```
procedure :: init => phs_wood_init
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_init (phs, phs_config)
  class(phs_wood_t), intent(out) :: phs
  class(phs_config_t), intent(in), target :: phs_config
  call phs%base_init (phs_config)
  select type (phs_config)
  type is (phs_wood_config_t)
     phs%forest = phs_config%forest
     select case (phs_config%extension_mode)
     case (EXTENSION_DEFAULT)
        phs%n_r_born = phs_config%n_par - 3
     case (EXTENSION_DGLAP)
        phs%n_r_born = phs_config%n_par - 4
     end select
  end select
end subroutine phs_wood_init
```

18.8.5 Evaluation

We compute the outgoing momenta from the incoming momenta and the input parameter set r_in in channel r_in. We also compute the r parameters and Jacobians f for all other channels.

We do *not* need to a apply a transformation from/to the c.m. frame, because in phs_base the momenta are already boosted to the c.m. frame before assigning them in the phs object, and inversely boosted when extracting them.

⟨*PHS wood: phs wood: TBP*⟩+≡
```
procedure :: evaluate_selected_channel => phs_wood_evaluate_selected_channel
procedure :: evaluate_other_channels => phs_wood_evaluate_other_channels
```

⟨*PHS wood: procedures*⟩+≡
```
subroutine phs_wood_evaluate_selected_channel (phs, c_in, r_in)
  class(phs_wood_t), intent(inout) :: phs
  integer, intent(in) :: c_in
  real(default), intent(in), dimension(:) :: r_in
  logical :: ok
  phs%q_defined = .false.
  if (phs%p_defined) then
     call phs_forest_set_prt_in (phs%forest, phs%p)
     phs%r(:,c_in) = r_in
     call phs_forest_evaluate_selected_channel (phs%forest, &
          c_in, phs%active_channel, &
          phs%sqrts_hat, phs%r, phs%f, phs%volume, ok)
     select type (config => phs%config)
```

```
      type is (phs_wood_config_t)
         if (config%extension_mode > EXTENSION_NONE) then
            if (phs%n_r_born > 0) then
               phs%r_real = r_in (phs%n_r_born + 1 : phs%n_r_born + 3)
            else
               call msg_fatal ("n_r_born should be larger than 0!")
            end if
         end if
      end select
      if (ok) then
         phs%q = phs_forest_get_momenta_out (phs%forest)
         phs%q_defined = .true.
      end if
   end if
end subroutine phs_wood_evaluate_selected_channel

subroutine phs_wood_evaluate_other_channels (phs, c_in)
   class(phs_wood_t), intent(inout) :: phs
   integer, intent(in) :: c_in
   integer :: c
   if (phs%q_defined) then
      call phs_forest_evaluate_other_channels (phs%forest, &
           c_in, phs%active_channel, &
           phs%sqrts_hat, phs%r, phs%f, combine=.true.)
      select type (config => phs%config)
      type is (phs_wood_config_t)
         if (config%extension_mode > EXTENSION_NONE) then
            if (phs%n_r_born > 0) then
               do c = 1, size (phs%r, 2)
                  phs%r(phs%n_r_born + 1 : phs%n_r_born + 3, c) = phs%r_real
               end do
            else
               phs%r_defined = .false.
            end if
         end if
      end select
      phs%r_defined = .true.
   end if
end subroutine phs_wood_evaluate_other_channels
```

Inverse evaluation.

⟨*PHS wood: phs wood: TBP*⟩+≡

```
procedure :: inverse => phs_wood_inverse
```

⟨*PHS wood: procedures*⟩+≡

```
subroutine phs_wood_inverse (phs)
   class(phs_wood_t), intent(inout) :: phs
   if (phs%p_defined .and. phs%q_defined) then
      call phs_forest_set_prt_in (phs%forest, phs%p)
      call phs_forest_set_prt_out (phs%forest, phs%q)
      call phs_forest_recover_channel (phs%forest, &
           1, &
           phs%sqrts_hat, phs%r, phs%f, phs%volume)
      call phs_forest_evaluate_other_channels (phs%forest, &
```

```
            1, phs%active_channel, &
            phs%sqrts_hat, phs%r, phs%f, combine=.false.)
        phs%r_defined = .true.
      end if
    end subroutine phs_wood_inverse
```

18.8.6 Unit tests

Test module, followed by the corresponding implementation module.

⟨phs_wood_ut.f90⟩≡

```
  ⟨File header⟩

  module phs_wood_ut
    use unit_tests
    use phs_wood_uti

  ⟨Standard module head⟩

  ⟨PHS wood: public test⟩

  ⟨PHS wood: public test auxiliary⟩

  contains

  ⟨PHS wood: test driver⟩

  end module phs_wood_ut
```

⟨phs_wood_uti.f90⟩≡

```
  ⟨File header⟩

  module phs_wood_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use io_units
    use os_interface
    use lorentz
    use flavors
    use model_data
    use process_constants
    use mappings
    use phs_base
    use phs_forests

    use phs_wood

    use phs_base_ut, only: init_test_process_data, init_test_decay_data

  ⟨Standard module head⟩

  ⟨PHS wood: public test auxiliary⟩
```

⟨*PHS wood: test declarations*⟩

```
contains
```

⟨*PHS wood: tests*⟩

⟨*PHS wood: test auxiliary*⟩

```
end module phs_wood_uti
```

API: driver for the unit tests below.

⟨*PHS wood: public test*⟩≡

```
public :: phs_wood_test
```

⟨*PHS wood: test driver*⟩≡

```
subroutine phs_wood_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```
⟨*PHS wood: execute tests*⟩
```
end subroutine phs_wood_test
```

⟨*PHS wood: public test*⟩+≡

```
public :: phs_wood_vis_test
```

⟨*PHS wood: test driver*⟩+≡

```
subroutine phs_wood_vis_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
```
⟨*PHS wood: execute vis tests*⟩
```
end subroutine phs_wood_vis_test
```

Phase-space configuration data

Construct and display a test phase-space configuration object. Also check the `azimuthal_dependence` flag.

This auxiliary routine writes a phase-space configuration file to unit `u_phs`.

⟨*PHS wood: public test auxiliary*⟩≡

```
public :: write_test_phs_file
```

⟨*PHS wood: test auxiliary*⟩≡

```
subroutine write_test_phs_file (u_phs, procname)
  integer, intent(in) :: u_phs
  type(string_t), intent(in), optional :: procname
  if (present (procname)) then
     write (u_phs, "(A,A)")  "process ", char (procname)
  else
     write (u_phs, "(A)")  "process testproc"
  end if
  write (u_phs, "(A,A)")  "   md5sum_process    = ", '""'
  write (u_phs, "(A,A)")  "   md5sum_model_par  = ", '""'
  write (u_phs, "(A,A)")  "   md5sum_phs_config = ", '""'
  write (u_phs, "(A)")  "   sqrts         = 1000"
  write (u_phs, "(A)")  "   m_threshold_s =   50"
  write (u_phs, "(A)")  "   m_threshold_t =  100"
```

```
    write (u_phs, "(A)")  "   off_shell = 2"
    write (u_phs, "(A)")  "   t_channel = 6"
    write (u_phs, "(A)")  "   keep_nonresonant = T"
    write (u_phs, "(A)")  "  grove #1"
    write (u_phs, "(A)")  "    tree 3"
  end subroutine write_test_phs_file
```

⟨*PHS wood: execute tests*⟩≡

```
  call test (phs_wood_1, "phs_wood_1", &
      "phase-space configuration", &
      u, results)
```

⟨*PHS wood: test declarations*⟩≡

```
  public :: phs_wood_1
```

⟨*PHS wood: tests*⟩≡

```
  subroutine phs_wood_1 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(process_constants_t) :: process_data
    class(phs_config_t), allocatable :: phs_data
    type(mapping_defaults_t) :: mapping_defaults
    real(default) :: sqrts
    integer :: u_phs, iostat
    character(32) :: buffer

    write (u, "(A)")  "* Test output: phs_wood_1"
    write (u, "(A)")  "*   Purpose: initialize and display &
        &phase-space configuration data"
    write (u, "(A)")

    call model%init_test ()

    call syntax_phs_forest_init ()

    write (u, "(A)")  "* Initialize a process"
    write (u, "(A)")

    call init_test_process_data (var_str ("phs_wood_1"), process_data)

    write (u, "(A)")  "* Create a scratch phase-space file"
    write (u, "(A)")

    u_phs = free_unit ()
    open (u_phs, status = "scratch", action = "readwrite")
    call write_test_phs_file (u_phs, var_str ("phs_wood_1"))
    rewind (u_phs)
    do
       read (u_phs, "(A)", iostat = iostat)  buffer
       if (iostat /= 0)  exit
       write (u, "(A)") trim (buffer)
    end do

    write (u, "(A)")
    write (u, "(A)")  "* Setup phase-space configuration object"
```

```
    write (u, "(A)")

    mapping_defaults%step_mapping = .false.

    allocate (phs_wood_config_t :: phs_data)
    call phs_data%init (process_data, model)
    select type (phs_data)
    type is (phs_wood_config_t)
       call phs_data%set_input (u_phs)
       call phs_data%set_mapping_defaults (mapping_defaults)
    end select

    sqrts = 1000._default
    call phs_data%configure (sqrts)

    call phs_data%write (u)
    write (u, "(A)")

    select type (phs_data)
    type is (phs_wood_config_t)
       call phs_data%write_forest (u)
    end select

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    close (u_phs)
    call phs_data%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_wood_1"

  end subroutine phs_wood_1
```

Phase space evaluation

Compute kinematics for given parameters, also invert the calculation.

⟨*PHS wood: execute tests*⟩+≡

```
  call test (phs_wood_2, "phs_wood_2", &
       "phase-space evaluation", &
       u, results)
```

⟨*PHS wood: test declarations*⟩+≡

```
  public :: phs_wood_2
```

⟨*PHS wood: tests*⟩+≡

```
  subroutine phs_wood_2 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(flavor_t) :: flv
    type(process_constants_t) :: process_data
    real(default) :: sqrts, E
    class(phs_config_t), allocatable, target :: phs_data
```

```
class(phs_t), pointer :: phs => null ()
type(vector4_t), dimension(2) :: p, q
integer :: u_phs

write (u, "(A)")  "* Test output: phs_wood_2"
write (u, "(A)")  "*   Purpose: test simple single-channel phase space"
write (u, "(A)")

call model%init_test ()
call flv%init (25, model)

write (u, "(A)")  "* Initialize a process and a matching &
     &phase-space configuration"
write (u, "(A)")

call init_test_process_data (var_str ("phs_wood_2"), process_data)
u_phs = free_unit ()
open (u_phs, status = "scratch", action = "readwrite")
call write_test_phs_file (u_phs, var_str ("phs_wood_2"))
rewind (u_phs)

allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data, model)
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_input (u_phs)
end select

sqrts = 1000._default
call phs_data%configure (sqrts)

call phs_data%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize the phase-space instance"
write (u, "(A)")

call phs_data%allocate_instance (phs)
call phs%init (phs_data)

call phs%write (u, verbose=.true.)

write (u, "(A)")
write (u, "(A)")  "* Set incoming momenta"
write (u, "(A)")

E = sqrts / 2
p(1) = vector4_moving (E, sqrt (E**2 - flv%get_mass ()**2), 3)
p(2) = vector4_moving (E,-sqrt (E**2 - flv%get_mass ()**2), 3)

call phs%set_incoming_momenta (p)
call phs%compute_flux ()
call phs%write (u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Compute phase-space point &
         &for x = 0.125, 0.5"
    write (u, "(A)")

    call phs%evaluate_selected_channel (1, [0.125_default, 0.5_default])
    call phs%evaluate_other_channels (1)
    call phs%write (u)
    write (u, "(A)")
    select type (phs)
    type is (phs_wood_t)
       call phs%write_forest (u)
    end select

    write (u, "(A)")
    write (u, "(A)")  "* Inverse kinematics"
    write (u, "(A)")

    call phs%get_outgoing_momenta (q)
    call phs%final ()
    deallocate (phs)

    call phs_data%allocate_instance (phs)
    call phs%init (phs_data)

    call phs%set_incoming_momenta (p)
    call phs%compute_flux ()
    call phs%set_outgoing_momenta (q)

    call phs%inverse ()
    call phs%write (u)
    write (u, "(A)")
    select type (phs)
    type is (phs_wood_t)
       call phs%write_forest (u)
    end select

    call phs%final ()
    deallocate (phs)

    close (u_phs)
    call phs_data%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_wood_2"

  end subroutine phs_wood_2
```

Phase-space generation

Generate phase space for a simple process.
⟨*PHS wood: execute tests*⟩+≡

```
call test (phs_wood_3, "phs_wood_3", &
     "phase-space generation", &
     u, results)
```

⟨*PHS wood: test declarations*⟩+≡

```
public :: phs_wood_3
```

⟨*PHS wood: tests*⟩+≡

```
subroutine phs_wood_3 (u)
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(process_constants_t) :: process_data
  type(phs_parameters_t) :: phs_par
  class(phs_config_t), allocatable :: phs_data
  integer :: iostat
  character(80) :: buffer

  write (u, "(A)")  "* Test output: phs_wood_3"
  write (u, "(A)")  "*   Purpose: generate a phase-space configuration"
  write (u, "(A)")

  call model%init_test ()

  call syntax_phs_forest_init ()

  write (u, "(A)")  "* Initialize a process and phase-space parameters"
  write (u, "(A)")

  call init_test_process_data (var_str ("phs_wood_3"), process_data)
  allocate (phs_wood_config_t :: phs_data)
  call phs_data%init (process_data, model)

  phs_par%sqrts = 1000
  select type (phs_data)
  type is (phs_wood_config_t)
     call phs_data%set_parameters (phs_par)
     phs_data%io_unit_keep_open = .true.
  end select

  write (u, "(A)")
  write (u, "(A)")  "* Generate a scratch phase-space file"
  write (u, "(A)")

  call phs_data%configure (phs_par%sqrts)

  select type (phs_data)
  type is (phs_wood_config_t)
     rewind (phs_data%io_unit)
     do
        read (phs_data%io_unit, "(A)", iostat = iostat)  buffer
        if (iostat /= 0)  exit
        write (u, "(A)") trim (buffer)
     end do
  end select
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call phs_data%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_wood_3"

  end subroutine phs_wood_3
```

Nontrivial process

Generate phase space for a $2 \to 3$ process.

⟨*PHS wood: execute tests*⟩+≡

```
  call test (phs_wood_4, "phs_wood_4", &
       "nontrivial process", &
       u, results)
```

⟨*PHS wood: test declarations*⟩+≡

```
  public :: phs_wood_4
```

⟨*PHS wood: tests*⟩+≡

```
  subroutine phs_wood_4 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(process_constants_t) :: process_data
    type(phs_parameters_t) :: phs_par
    class(phs_config_t), allocatable, target :: phs_data
    integer :: iostat
    character(80) :: buffer
    class(phs_t), pointer :: phs => null ()
    real(default) :: E, pL
    type(vector4_t), dimension(2) :: p
    type(vector4_t), dimension(3) :: q

    write (u, "(A)")  "* Test output: phs_wood_4"
    write (u, "(A)")  "*   Purpose: generate a phase-space configuration"
    write (u, "(A)")

    call model%init_test ()

    call syntax_phs_forest_init ()

    write (u, "(A)")  "* Initialize a process and phase-space parameters"
    write (u, "(A)")

    process_data%id = "phs_wood_4"
    process_data%model_name = "Test"
    process_data%n_in = 2
    process_data%n_out = 3
    process_data%n_flv = 1
    allocate (process_data%flv_state (process_data%n_in + process_data%n_out, &
         process_data%n_flv))
```

```
process_data%flv_state(:,1) = [25, 25, 25, 6, -6]

allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data, model)

phs_par%sqrts = 1000
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_parameters (phs_par)
   phs_data%io_unit_keep_open = .true.
end select

write (u, "(A)")
write (u, "(A)")  "* Generate a scratch phase-space file"
write (u, "(A)")

call phs_data%configure (phs_par%sqrts)

select type (phs_data)
type is (phs_wood_config_t)
   rewind (phs_data%io_unit)
   do
      read (phs_data%io_unit, "(A)", iostat = iostat)  buffer
      if (iostat /= 0)  exit
      write (u, "(A)") trim (buffer)
   end do
end select

write (u, "(A)")
write (u, "(A)")  "* Initialize the phase-space instance"
write (u, "(A)")

call phs_data%allocate_instance (phs)
call phs%init (phs_data)

write (u, "(A)")  "* Set incoming momenta"
write (u, "(A)")

select type (phs_data)
type is (phs_wood_config_t)
   E = phs_data%sqrts / 2
   pL = sqrt (E**2 - phs_data%flv(1,1)%get_mass ()**2)
end select
p(1) = vector4_moving (E, pL, 3)
p(2) = vector4_moving (E, -pL, 3)

call phs%set_incoming_momenta (p)
call phs%compute_flux ()

write (u, "(A)")  "* Compute phase-space point &
     &for x = 0.1, 0.2, 0.3, 0.4, 0.5"
write (u, "(A)")

call phs%evaluate_selected_channel (1, &
```

```
        [0.1_default, 0.2_default, 0.3_default, 0.4_default, 0.5_default])
    call phs%evaluate_other_channels (1)
    call phs%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Inverse kinematics"
    write (u, "(A)")

    call phs%get_outgoing_momenta (q)
    call phs%final ()
    deallocate (phs)

    call phs_data%allocate_instance (phs)
    call phs%init (phs_data)

    call phs%set_incoming_momenta (p)
    call phs%compute_flux ()
    call phs%set_outgoing_momenta (q)

    call phs%inverse ()
    call phs%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call phs%final ()
    deallocate (phs)

    call phs_data%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_wood_4"

  end subroutine phs_wood_4
```

Equivalences

Generate phase space for a simple process, including channel equivalences.

⟨*PHS wood: execute tests*⟩+≡

```
  call test (phs_wood_5, "phs_wood_5", &
      "equivalences", &
      u, results)
```

⟨*PHS wood: test declarations*⟩+≡

```
  public :: phs_wood_5
```

⟨*PHS wood: tests*⟩+≡

```
  subroutine phs_wood_5 (u)
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(process_constants_t) :: process_data
    type(phs_parameters_t) :: phs_par
    class(phs_config_t), allocatable :: phs_data
```

```
    write (u, "(A)")  "* Test output: phs_wood_5"
    write (u, "(A)")  "*   Purpose: generate a phase-space configuration"
    write (u, "(A)")

    call model%init_test ()

    call syntax_phs_forest_init ()

    write (u, "(A)")  "* Initialize a process and phase-space parameters"
    write (u, "(A)")

    call init_test_process_data (var_str ("phs_wood_5"), process_data)
    allocate (phs_wood_config_t :: phs_data)
    call phs_data%init (process_data, model)

    phs_par%sqrts = 1000
    select type (phs_data)
    type is (phs_wood_config_t)
       call phs_data%set_parameters (phs_par)
       call phs_data%enable_equivalences ()
    end select

    write (u, "(A)")
    write (u, "(A)")  "* Generate a scratch phase-space file"
    write (u, "(A)")

    call phs_data%configure (phs_par%sqrts)
    call phs_data%write (u)
    write (u, "(A)")

    select type (phs_data)
    type is (phs_wood_config_t)
       call phs_data%write_forest (u)
    end select

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call phs_data%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_wood_5"

  end subroutine phs_wood_5
```

MD5 sum checks

Generate phase space for a simple process. Repeat this with and without parameter change.

⟨*PHS wood: execute tests*⟩+≡

```
    call test (phs_wood_6, "phs_wood_6", &
         "phase-space generation", &
         u, results)
```

⟨*PHS wood: test declarations*⟩+≡

```
    public :: phs_wood_6
```

⟨*PHS wood: tests*⟩+≡

```
    subroutine phs_wood_6 (u)
      integer, intent(in) :: u
      type(model_data_t), target :: model
      type(process_constants_t) :: process_data
      type(phs_parameters_t) :: phs_par
      class(phs_config_t), allocatable :: phs_data
      logical :: exist, found, match
      integer :: u_phs
      character(*), parameter :: filename = "phs_wood_6_p.phs"

      write (u, "(A)")  "* Test output: phs_wood_6"
      write (u, "(A)")  "*   Purpose: generate and check  phase-space file"
      write (u, "(A)")

      call model%init_test ()

      call syntax_phs_forest_init ()

      write (u, "(A)")  "* Initialize a process and phase-space parameters"
      write (u, "(A)")

      call init_test_process_data (var_str ("phs_wood_6"), process_data)
      process_data%id = "phs_wood_6_p"
      process_data%md5sum = "1234567890abcdef1234567890abcdef"
      allocate (phs_wood_config_t :: phs_data)
      call phs_data%init (process_data, model)

      phs_par%sqrts = 1000
      select type (phs_data)
      type is (phs_wood_config_t)
         call phs_data%set_parameters (phs_par)
      end select

      write (u, "(A)")  "* Remove previous phs file, if any"
      write (u, "(A)")

      inquire (file = filename, exist = exist)
      if (exist) then
         u_phs = free_unit ()
         open (u_phs, file = filename, action = "write")
         close (u_phs, status = "delete")
      end if

      write (u, "(A)")  "* Check phase-space file (should fail)"
      write (u, "(A)")

      select type (phs_data)
```

```
type is (phs_wood_config_t)
   call phs_data%read_phs_file (exist, found, match)
   write (u, "(1x,A,L1)")  "exist = ", exist
   write (u, "(1x,A,L1)")  "found = ", found
   write (u, "(1x,A,L1)")  "match = ", match
end select

write (u, "(A)")
write (u, "(A)")  "* Generate a phase-space file"
write (u, "(A)")

call phs_data%configure (phs_par%sqrts)

write (u, "(1x,A,A,A)")  "MD5 sum (process)    = '", &
     phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (model par)  = '", &
     phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (phs config) = '", &
     phs_data%md5sum_phs_config, "'"

write (u, "(A)")
write (u, "(A)")  "* Check MD5 sum"
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data, model)
phs_par%sqrts = 1000
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_parameters (phs_par)
   phs_data%sqrts = phs_par%sqrts
   phs_data%par%sqrts = phs_par%sqrts
end select
call phs_data%compute_md5sum ()

write (u, "(1x,A,A,A)")  "MD5 sum (process)    = '", &
     phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (model par)  = '", &
     phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (phs config) = '", &
     phs_data%md5sum_phs_config, "'"

select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%read_phs_file (exist, found, match)
   write (u, "(1x,A,L1)")  "exist = ", exist
   write (u, "(1x,A,L1)")  "found = ", found
   write (u, "(1x,A,L1)")  "match = ", match
end select

write (u, "(A)")
write (u, "(A)")  "* Modify sqrts and check MD5 sum"
```

```
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data, model)
phs_par%sqrts = 500
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_parameters (phs_par)
   phs_data%sqrts = phs_par%sqrts
   phs_data%par%sqrts = phs_par%sqrts
end select
call phs_data%compute_md5sum ()

write (u, "(1x,A,A,A)")  "MD5 sum (process)    = '", &
     phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (model par)  = '", &
     phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (phs config) = '", &
     phs_data%md5sum_phs_config, "'"

select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%read_phs_file (exist, found, match)
   write (u, "(1x,A,L1)")  "exist = ", exist
   write (u, "(1x,A,L1)")  "found = ", found
   write (u, "(1x,A,L1)")  "match = ", match
end select

write (u, "(A)")
write (u, "(A)")  "* Modify process and check MD5 sum"
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
process_data%md5sum = "77777777777777777777777777777777"
allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data, model)
phs_par%sqrts = 1000
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_parameters (phs_par)
   phs_data%sqrts = phs_par%sqrts
   phs_data%par%sqrts = phs_par%sqrts
end select
call phs_data%compute_md5sum ()

write (u, "(1x,A,A,A)")  "MD5 sum (process)    = '", &
     phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (model par)  = '", &
     phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (phs config) = '", &
     phs_data%md5sum_phs_config, "'"
```

```
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%read_phs_file (exist, found, match)
   write (u, "(1x,A,L1)")  "exist = ", exist
   write (u, "(1x,A,L1)")  "found = ", found
   write (u, "(1x,A,L1)")  "match = ", match
end select

write (u, "(A)")
write (u, "(A)")  "* Modify phs parameter and check MD5 sum"
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
allocate (phs_wood_config_t :: phs_data)
process_data%md5sum = "1234567890abcdef1234567890abcdef"
call phs_data%init (process_data, model)
phs_par%sqrts = 1000
phs_par%off_shell = 17
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_parameters (phs_par)
   phs_data%sqrts = phs_par%sqrts
   phs_data%par%sqrts = phs_par%sqrts
end select
call phs_data%compute_md5sum ()

write (u, "(1x,A,A,A)")  "MD5 sum (process)    = '", &
     phs_data%md5sum_process, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (model par)  = '", &
     phs_data%md5sum_model_par, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (phs config) = '", &
     phs_data%md5sum_phs_config, "'"

select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%read_phs_file (exist, found, match)
   write (u, "(1x,A,L1)")  "exist = ", exist
   write (u, "(1x,A,L1)")  "found = ", found
   write (u, "(1x,A,L1)")  "match = ", match
end select

write (u, "(A)")
write (u, "(A)")  "* Modify model parameter and check MD5 sum"
write (u, "(A)")

call phs_data%final ()
deallocate (phs_data)
allocate (phs_wood_config_t :: phs_data)
call model%set_par (var_str ("ms"), 100._default)
call phs_data%init (process_data, model)
phs_par%sqrts = 1000
phs_par%off_shell = 1
```

```
    select type (phs_data)
    type is (phs_wood_config_t)
       call phs_data%set_parameters (phs_par)
       phs_data%sqrts = phs_par%sqrts
       phs_data%par%sqrts = phs_par%sqrts
    end select
    call phs_data%compute_md5sum ()

    write (u, "(1x,A,A,A)")  "MD5 sum (process)    = '", &
         phs_data%md5sum_process, "'"
    write (u, "(1x,A,A,A)")  "MD5 sum (model par)  = '", &
         phs_data%md5sum_model_par, "'"
    write (u, "(1x,A,A,A)")  "MD5 sum (phs config) = '", &
         phs_data%md5sum_phs_config, "'"

    select type (phs_data)
    type is (phs_wood_config_t)
       call phs_data%read_phs_file (exist, found, match)
       write (u, "(1x,A,L1)")  "exist = ", exist
       write (u, "(1x,A,L1)")  "found = ", found
       write (u, "(1x,A,L1)")  "match = ", match
    end select

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call phs_data%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: phs_wood_6"

  end subroutine phs_wood_6
```

⟨*PHS wood: execute vis tests*⟩≡

```
  call test (phs_wood_vis_1, "phs_wood_vis_1", &
       "visualizing phase space channels", &
       u, results)
```

⟨*PHS wood: test declarations*⟩+≡

```
  public :: phs_wood_vis_1
```

⟨*PHS wood: tests*⟩+≡

```
  subroutine phs_wood_vis_1 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_data_t), target :: model
    type(process_constants_t) :: process_data
    class(phs_config_t), allocatable :: phs_data
    type(mapping_defaults_t) :: mapping_defaults
    type(string_t) :: vis_file, pdf_file, ps_file
    real(default) :: sqrts
    logical :: exist, exist_pdf, exist_ps
    integer :: u_phs, iostat, u_vis
    character(95) :: buffer
```

```
write (u, "(A)")  "* Test output: phs_wood_vis_1"
write (u, "(A)")  "*   Purpose: visualizing the &
     &phase-space configuration"
write (u, "(A)")

call os_data_init (os_data)
call model%init_test ()

call syntax_phs_forest_init ()

write (u, "(A)")  "* Initialize a process"
write (u, "(A)")

call init_test_process_data (var_str ("phs_wood_vis_1"), process_data)

write (u, "(A)")  "* Create a scratch phase-space file"
write (u, "(A)")

u_phs = free_unit ()
open (u_phs, status = "scratch", action = "readwrite")
call write_test_phs_file (u_phs, var_str ("phs_wood_vis_1"))
rewind (u_phs)
do
   read (u_phs, "(A)", iostat = iostat)  buffer
   if (iostat /= 0)  exit
   write (u, "(A)") trim (buffer)
end do

write (u, "(A)")
write (u, "(A)")  "* Setup phase-space configuration object"
write (u, "(A)")

mapping_defaults%step_mapping = .false.

allocate (phs_wood_config_t :: phs_data)
call phs_data%init (process_data, model)
select type (phs_data)
type is (phs_wood_config_t)
   call phs_data%set_input (u_phs)
   call phs_data%set_mapping_defaults (mapping_defaults)
   phs_data%os_data = os_data
   phs_data%io_unit = 0
   phs_data%io_unit_keep_open = .true.
   phs_data%vis_channels = .true.
end select

sqrts = 1000._default
call phs_data%configure (sqrts)

call phs_data%write (u)
write (u, "(A)")

select type (phs_data)
```

```
      type is (phs_wood_config_t)
         call phs_data%write_forest (u)
      end select

      vis_file = "phs_wood_vis_1_phs.tex"
      ps_file  = "phs_wood_vis_1_phs.ps"
      pdf_file = "phs_wood_vis_1_phs.pdf"
      inquire (file = char (vis_file), exist = exist)
      if (exist) then
         u_vis = free_unit ()
         open (u_vis, file = char (vis_file), action = "read", status = "old")
         iostat = 0
         do while (iostat == 0)
            read (u_vis, "(A)", iostat = iostat)  buffer
            if (iostat == 0)  write (u, "(A)")  trim (buffer)
         end do
         close (u_vis)
      else
         write (u, "(A)")  "[Visualize LaTeX file is missing]"
      end if
      inquire (file = char (ps_file), exist = exist_ps)
      if (exist_ps) then
         write (u, "(A)")  "[Visualize Postscript file exists and is nonempty]"
      else
         write (u, "(A)")  "[Visualize Postscript file is missing/non-regular]"
      end if
      inquire (file = char (pdf_file), exist = exist_pdf)
      if (exist_pdf) then
         write (u, "(A)")  "[Visualize PDF file exists and is nonempty]"
      else
         write (u, "(A)")  "[Visualize PDF file is missing/non-regular]"
      end if

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      close (u_phs)
      call phs_data%final ()
      call model%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: phs_wood_vis_1"

   end subroutine phs_wood_vis_1
```

18.9 The FKS phase space

⟨phs_fks.f90⟩≡
⟨*File header*⟩

```
module phs_fks
```

```
⟨Use kinds⟩
⟨Use strings⟩
  use constants
  use diagnostics
  use io_units, only: given_output_unit
  use format_utils, only: write_separator
  use lorentz
  use physics_defs
  use flavors
  use pdg_arrays, only: is_colored
  use models, only: model_t
  use sf_mappings
  use sf_base
  use phs_base
  use resonances, only: resonance_contributors_t, resonance_history_t
  use phs_forests, only: phs_forest_final
  use phs_wood
  use cascades
  use process_constants
  use process_libraries
  use ttv_formfactors, only: generate_on_shell_decay_threshold, m1s_to_mpole

⟨Standard module head⟩

⟨phs fks: public⟩

⟨phs fks: parameters⟩

⟨phs fks: types⟩

⟨phs fks: interfaces⟩

contains

⟨phs fks: procedures⟩

end module phs_fks
```

Another container for the $x_{\oplus}$- and $x_{\ominus}$-values for initial-state phase spaces.

```
⟨phs fks: public⟩≡
    public :: isr_kinematics_t
```

```
⟨phs fks: types⟩≡
    type :: isr_kinematics_t
      integer :: n_in
      real(default), dimension(2) :: x = one
      real(default), dimension(2) :: z = zero
      real(default), dimension(2) :: z_coll = zero
      real(default) :: sqrts_born = zero
      real(default) :: beam_energy = zero
      real(default) :: fac_scale = zero
      real(default), dimension(2) :: jacobian = one
    end type isr_kinematics_t
```

⟨*phs fks: public*⟩+≡
```
public :: phs_point_set_t
```
⟨*phs fks: types*⟩+≡
```
type :: phs_point_set_t
   type(phs_point_t), dimension(:), allocatable :: phs_point
   logical :: initialized = .false.
contains
⟨phs fks: phs point set: TBP⟩
end type phs_point_set_t
```

⟨*phs fks: phs point set: TBP*⟩≡
```
procedure :: init => phs_point_set_init
```
⟨*phs fks: procedures*⟩≡
```
subroutine phs_point_set_init (phs_point_set, n_particles, n_phs)
  class(phs_point_set_t), intent(out) :: phs_point_set
  integer, intent(in) :: n_particles, n_phs
  integer :: i_phs
  allocate (phs_point_set%phs_point (n_phs))
  do i_phs = 1, n_phs
     phs_point_set%phs_point(i_phs) = n_particles
  end do
  phs_point_set%initialized = .true.
end subroutine phs_point_set_init
```

⟨*phs fks: phs point set: TBP*⟩+≡
```
procedure :: write => phs_point_set_write
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_point_set_write (phs_point_set, i_phs, contributors, unit, show_mass, &
       testflag, check_conservation, ultra, n_in)
  class(phs_point_set_t), intent(in) :: phs_point_set
  integer, intent(in), optional :: i_phs
  integer, intent(in), dimension(:), optional :: contributors
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: show_mass
  logical, intent(in), optional :: testflag, ultra
  logical, intent(in), optional :: check_conservation
  integer, intent(in), optional :: n_in
  integer :: i, u
  type(vector4_t) :: p_sum
  u = given_output_unit (unit); if (u < 0) return
  if (present (i_phs)) then
     call phs_point_set%phs_point(i_phs)%write &
          (unit = u, show_mass = show_mass, testflag = testflag, &
          check_conservation = check_conservation, ultra = ultra, n_in = n_in)
  else
     do i = 1, size(phs_point_set%phs_point)
        call phs_point_set%phs_point(i)%write &
             (unit = u, show_mass = show_mass, testflag = testflag, &
             check_conservation = check_conservation, ultra = ultra, n_in = n_in)
     end do
  end if
  if (present (contributors)) then
```

```
      p_sum = vector4_null
      call msg_debug (D_SUBTRACTION, "Invariant masses for real emission: ")
      associate (p => phs_point_set%phs_point(i_phs)%p)
         do i = 1, size (contributors)
            p_sum = p_sum + p(contributors(i))
         end do
         p_sum = p_sum + p(size(p))
      end associate
      if (debug_active (D_SUBTRACTION)) &
           call vector4_write (p_sum, unit = unit, show_mass = show_mass, &
                testflag = testflag, ultra = ultra)
  end if
end subroutine phs_point_set_write
```

⟨*phs fks: phs point set: TBP*⟩+≡

```
procedure :: get_n_momenta => phs_point_set_get_n_momenta
```

⟨*phs fks: procedures*⟩+≡

```
elemental function phs_point_set_get_n_momenta (phs_point_set, i_res) result (n)
  integer :: n
  class(phs_point_set_t), intent(in) :: phs_point_set
  integer, intent(in) :: i_res
  n = phs_point_set%phs_point(i_res)%n_momenta
end function phs_point_set_get_n_momenta
```

⟨*phs fks: phs point set: TBP*⟩+≡

```
procedure :: get_momenta => phs_point_set_get_momenta
```

⟨*phs fks: procedures*⟩+≡

```
pure function phs_point_set_get_momenta (phs_point_set, i_phs) result (p)
  type(vector4_t), dimension(:), allocatable :: p
  class(phs_point_set_t), intent(in) :: phs_point_set
  integer, intent(in) :: i_phs
  allocate (p (phs_point_set%phs_point(i_phs)%n_momenta), &
       source = phs_point_set%phs_point(i_phs)%p)
end function phs_point_set_get_momenta
```

⟨*phs fks: phs point set: TBP*⟩+≡

```
procedure :: get_momentum => phs_point_set_get_momentum
```

⟨*phs fks: procedures*⟩+≡

```
pure function phs_point_set_get_momentum (phs_point_set, i_phs, i_mom) result (p)
  type(vector4_t) :: p
  class(phs_point_set_t), intent(in) :: phs_point_set
  integer, intent(in) :: i_phs, i_mom
  p = phs_point_set%phs_point(i_phs)%p(i_mom)
end function phs_point_set_get_momentum
```

⟨*phs fks: phs point set: TBP*⟩+≡

```
procedure :: get_energy => phs_point_set_get_energy
```

⟨*phs fks: procedures*⟩+≡
```
pure function phs_point_set_get_energy (phs_point_set, i_phs, i_mom) result (E)
  real(default) :: E
  class(phs_point_set_t), intent(in) :: phs_point_set
  integer, intent(in) :: i_phs, i_mom
  E = phs_point_set%phs_point(i_phs)%p(i_mom)%p(0)
end function phs_point_set_get_energy
```

⟨*phs fks: phs point set: TBP*⟩+≡
```
procedure :: get_sqrts => phs_point_set_get_sqrts
```

⟨*phs fks: procedures*⟩+≡
```
function phs_point_set_get_sqrts (phs_point_set, i_phs) result (sqrts)
  real(default) :: sqrts
  class(phs_point_set_t), intent(in) :: phs_point_set
  integer, intent(in) :: i_phs
  associate (p => phs_point_set%phs_point(i_phs)%p)
     sqrts = (p(1) + p(2))**1
  end associate
end function phs_point_set_get_sqrts
```

⟨*phs fks: phs point set: TBP*⟩+≡
```
generic :: set_momenta => set_momenta_p, set_momenta_phs_point
procedure :: set_momenta_p => phs_point_set_set_momenta_p
```

⟨*phs fks: procedures*⟩+≡
```
subroutine phs_point_set_set_momenta_p (phs_point_set, i_phs, p)
  class(phs_point_set_t), intent(inout) :: phs_point_set
  integer, intent(in) :: i_phs
  type(vector4_t), intent(in), dimension(:) :: p
  phs_point_set%phs_point(i_phs)%p = p
end subroutine phs_point_set_set_momenta_p
```

⟨*phs fks: phs point set: TBP*⟩+≡
```
procedure :: set_momenta_phs_point => phs_point_set_set_momenta_phs_point
```

⟨*phs fks: procedures*⟩+≡
```
subroutine phs_point_set_set_momenta_phs_point (phs_point_set, i_phs, p)
  class(phs_point_set_t), intent(inout) :: phs_point_set
  integer, intent(in) :: i_phs
  type(phs_point_t), intent(in) :: p
  phs_point_set%phs_point(i_phs) = p
end subroutine phs_point_set_set_momenta_phs_point
```

⟨*phs fks: phs point set: TBP*⟩+≡
```
procedure :: get_n_particles => phs_point_set_get_n_particles
```

⟨*phs fks: procedures*⟩+≡
```
function phs_point_set_get_n_particles (phs_point_set, i) result (n_particles)
  integer :: n_particles
  class(phs_point_set_t), intent(in) :: phs_point_set
  integer, intent(in), optional :: i
  integer :: j
  j = 1; if (present (i)) j = i
```

```
    n_particles = size (phs_point_set%phs_point(j)%p)
  end function phs_point_set_get_n_particles
```

⟨*phs fks: phs point set: TBP*⟩+≡

```
  procedure :: get_n_phs => phs_point_set_get_n_phs
```

⟨*phs fks: procedures*⟩+≡

```
  function phs_point_set_get_n_phs (phs_point_set) result (n_phs)
    integer :: n_phs
    class(phs_point_set_t), intent(in) :: phs_point_set
    n_phs = size (phs_point_set%phs_point)
  end function phs_point_set_get_n_phs
```

⟨*phs fks: phs point set: TBP*⟩+≡

```
  procedure :: get_invariant_mass => phs_point_set_get_invariant_mass
```

⟨*phs fks: procedures*⟩+≡

```
  function phs_point_set_get_invariant_mass (phs_point_set, i_phs, i_part) result (m2)
    real(default) :: m2
    class(phs_point_set_t), intent(in) :: phs_point_set
    integer, intent(in) :: i_phs
    integer, intent(in), dimension(:) :: i_part
    type(vector4_t) :: p
    integer :: i
    p = vector4_null
    do i = 1, size (i_part)
       p = p + phs_point_set%phs_point(i_phs)%p(i_part(i))
    end do
    m2 = p**2
  end function phs_point_set_get_invariant_mass
```

⟨*phs fks: phs point set: TBP*⟩+≡

```
  procedure :: write_phs_point => phs_point_set_write_phs_point
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_point_set_write_phs_point (phs_point_set, i_phs, unit, show_mass, &
     testflag, check_conservation, ultra, n_in)
    class(phs_point_set_t), intent(in) :: phs_point_set
    integer, intent(in) :: i_phs
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: show_mass
    logical, intent(in), optional :: testflag, ultra
    logical, intent(in), optional :: check_conservation
    integer, intent(in), optional :: n_in
    call phs_point_set%phs_point(i_phs)%write (unit, show_mass, testflag, &
       check_conservation, ultra, n_in)
  end subroutine phs_point_set_write_phs_point
```

⟨*phs fks: phs point set: TBP*⟩+≡

```
  procedure :: final => phs_point_set_final
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_point_set_final (phs_point_set)
  class(phs_point_set_t), intent(inout) :: phs_point_set
  integer :: i
  do i = 1, size (phs_point_set%phs_point)
     call phs_point_set%phs_point(i)%final ()
  end do
  deallocate (phs_point_set%phs_point)
  phs_point_set%initialized = .false.
end subroutine phs_point_set_final
```

⟨*phs fks: public*⟩+≡

```
public :: real_jacobian_t
```

⟨*phs fks: types*⟩+≡

```
type :: real_jacobian_t
  real(default), dimension(4) :: jac = 1._default
end type real_jacobian_t
```

⟨*phs fks: public*⟩+≡

```
public :: real_kinematics_t
```

⟨*phs fks: types*⟩+≡

```
type :: real_kinematics_t
  logical :: supply_xi_max = .true.
  real(default) :: xi_tilde
  real(default) :: phi
  real(default), dimension(:), allocatable :: xi_max, y
  real(default) :: xi_mismatch, y_mismatch
  type(real_jacobian_t), dimension(:), allocatable :: jac
  real(default) :: jac_mismatch
  type(phs_point_set_t) :: p_born_cms
  type(phs_point_set_t) :: p_born_lab
  type(phs_point_set_t) :: p_real_cms
  type(phs_point_set_t) :: p_real_lab
  type(phs_point_set_t) :: p_born_onshell
  type(phs_point_set_t), dimension(2) :: p_real_onshell
  integer, dimension(:), allocatable :: alr_to_i_phs
  real(default), dimension(3) :: x_rad
  real(default), dimension(:), allocatable :: jac_rand
  real(default), dimension(:), allocatable :: y_soft
  real(default) :: cms_energy2
  type(vector4_t), dimension(:), allocatable :: xi_ref_momenta
contains
⟨phs fks: real kinematics: TBP⟩
end type real_kinematics_t
```

⟨*phs fks: real kinematics: TBP*⟩≡

```
procedure :: init => real_kinematics_init
```

⟨*phs fks: procedures*⟩+≡

```
subroutine real_kinematics_init (r, n_tot, n_phs, n_alr, n_contr)
  class(real_kinematics_t), intent(inout) :: r
  integer, intent(in) :: n_tot, n_phs, n_alr, n_contr
```

```
    allocate (r%xi_max (n_phs))
    allocate (r%y (n_phs))
    allocate (r%y_soft (n_phs))
    call r%p_born_cms%init (n_tot - 1, 1)
    call r%p_born_lab%init (n_tot - 1, 1)
    call r%p_real_cms%init (n_tot, n_phs)
    call r%p_real_lab%init (n_tot, n_phs)
    allocate (r%jac (n_phs), r%jac_rand (n_phs))
    allocate (r%alr_to_i_phs (n_alr))
    allocate (r%xi_ref_momenta (n_contr))
    r%alr_to_i_phs = 0
    r%xi_tilde = zero; r%xi_mismatch = zero
    r%xi_max = zero
    r%y = zero; r%y_mismatch = zero
    r%y_soft = zero
    r%phi = zero
    r%cms_energy2 = zero
    r%xi_ref_momenta = vector4_null
    r%jac_mismatch = one
    r%jac_rand = one
  end subroutine real_kinematics_init
```

⟨*phs fks: real kinematics: TBP*⟩+≡

```
  procedure :: init_onshell => real_kinematics_init_onshell
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine real_kinematics_init_onshell (r, n_tot, n_phs)
    class(real_kinematics_t), intent(inout) :: r
    integer, intent(in) :: n_tot, n_phs
    call r%p_born_onshell%init (n_tot - 1, 1)
    call r%p_real_onshell(1)%init (n_tot, n_phs)
    call r%p_real_onshell(2)%init (n_tot, n_phs)
  end subroutine real_kinematics_init_onshell
```

⟨*phs fks: real kinematics: TBP*⟩+≡

```
  procedure :: write => real_kinematics_write
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine real_kinematics_write (r, unit)
    class(real_kinematics_t), intent(in) :: r
    integer, intent(in), optional :: unit
    integer :: u, i
    u = given_output_unit (unit); if (u < 0) return
    write (u,"(A)") "Real kinematics: "
    write (u,"(A,F5.3)") "xi_tilde: ", r%xi_tilde
    write (u,"(A,F5.3)") "phi: ", r%phi
    do i = 1, size (r%xi_max)
       write (u,"(A,I1,1X)") "i_phs: ", i
       write (u,"(A,100F5.3,1X)") "xi_max: ", r%xi_max(i)
       write (u,"(A,100F5.3,1X)") "y: ", r%y(i)
       write (u,"(A,100F5.3,1X)") "jac_rand: ", r%jac_rand(i)
       write (u,"(A,100F5.3,1X)") "y_soft: ", r%y_soft(i)
    end do
  end subroutine real_kinematics_write
```

The boost to the center-of-mass system only has a reasonable meaning above the threshold. Below the threshold, we do not apply boost at all, so that the top quarks stay in the rest frame. However, with top quarks exactly at rest, problems arise in the matrix elements (e.g. in the computation of angles). Therefore, we apply a boost which is not exactly 1, but has a tiny value differing from that.

⟨*phs fks: public*⟩+≡
```
public :: get_boost_for_threshold_projection
```

⟨*phs fks: procedures*⟩+≡
```
function get_boost_for_threshold_projection (p, sqrts, mtop) result (L)
  type(lorentz_transformation_t) :: L
  type(vector4_t), intent(in), dimension(:) :: p
  real(default), intent(in) :: sqrts, mtop
  type(vector4_t) :: p_tmp
  type(vector3_t) :: dir
  real(default) :: scale_factor, arg
  p_tmp = p(THR_POS_WP) + p(THR_POS_B)
  arg = sqrts**2 - four * mtop**2
  if (arg > zero) then
     scale_factor = sqrt (arg) / two
  else
     scale_factor = tiny_07*1000
  end if
  dir = scale_factor * create_unit_vector (p_tmp)
  p_tmp = [sqrts / two, dir%p]
  L = boost (p_tmp, mtop)
end function get_boost_for_threshold_projection
```

This routine recomputes the value of ϕ used to generate the real phase space.

⟨*phs fks: procedures*⟩+≡
```
function get_generation_phi (p_born, p_real, emitter, i_gluon) result (phi)
  real(default) :: phi
  type(vector4_t), intent(in), dimension(:) :: p_born, p_real
  integer, intent(in) :: emitter, i_gluon
  type(vector4_t) :: p1, p2, pp
  type(lorentz_transformation_t) :: rot_to_gluon, rot_to_z
  type(vector3_t) :: dir, z
  real(default) :: cpsi
  pp = p_real(emitter) + p_real(i_gluon)
  cpsi = (space_part_norm (pp)**2 - space_part_norm (p_real(emitter))**2 &
         + space_part_norm (p_real(i_gluon))**2) / &
         (two * space_part_norm (pp) * space_part_norm (p_real(i_gluon)))
  dir = create_orthogonal (space_part (p_born(emitter)))
  rot_to_gluon = rotation (cpsi, sqrt (one - cpsi**2), dir)
  pp = rot_to_gluon * p_born(emitter)
  z%p = [0, 0, 1]
  rot_to_z = rotation_to_2nd &
       (space_part (p_born(emitter)) / space_part_norm (p_born(emitter)), z)
  p1 = rot_to_z * pp / space_part_norm (pp)
  p2 = rot_to_z * p_real(i_gluon)
  phi = azimuthal_distance (p1, p2)
  if (phi < zero) phi = twopi - abs(phi)
```

```
end function get_generation_phi
```

⟨*phs fks: real kinematics: TBP*⟩+≡

```
procedure :: apply_threshold_projection_real => real_kinematics_apply_threshold_projection_real
```

⟨*phs fks: procedures*⟩+≡

```
subroutine real_kinematics_apply_threshold_projection_real (r, i_phs, mtop, L_to_cms, invert)
  class(real_kinematics_t), intent(inout) :: r
  integer, intent(in) :: i_phs
  real(default), intent(in) :: mtop
  type(lorentz_transformation_t), intent(in), dimension(:) :: L_to_cms
  logical, intent(in) :: invert
  integer :: leg, other_leg
  type(vector4_t), dimension(4) :: k_tmp
  type(vector4_t), dimension(4) :: k_decay_onshell_real
  type(vector4_t), dimension(3) :: k_decay_onshell_born
  do leg = 1, 2
     other_leg = 3 - leg
     associate (p_real => r%p_real_cms%phs_point(i_phs)%p, &
          p_real_onshell => r%p_real_onshell(leg)%phs_point(i_phs)%p)
        p_real_onshell(1:2) = p_real(1:2)
        k_tmp(1) = p_real(7)
        k_tmp(2) = p_real(ass_quark(leg))
        k_tmp(3) = p_real(ass_boson(leg))
        k_tmp(4) = [mtop, zero, zero, zero]
        call generate_on_shell_decay_threshold (k_tmp(1:3), &
             k_tmp(4), k_decay_onshell_real (2:4))
        k_decay_onshell_real (1) = k_tmp(4)
        k_tmp(1) = p_real(ass_quark(other_leg))
        k_tmp(2) = p_real(ass_boson(other_leg))
        k_decay_onshell_born = create_two_particle_decay (mtop**2, k_tmp(1), k_tmp(2))
        p_real_onshell(THR_POS_GLUON) = L_to_cms(leg) * k_decay_onshell_real (2)
        p_real_onshell(ass_quark(leg)) = L_to_cms(leg) * k_decay_onshell_real(3)
        p_real_onshell(ass_boson(leg)) = L_to_cms(leg) * k_decay_onshell_real(4)
        p_real_onshell(ass_quark(other_leg)) = L_to_cms(leg) * k_decay_onshell_born (2)
        p_real_onshell(ass_boson(other_leg)) = L_to_cms(leg) * k_decay_onshell_born (3)
        if (invert) then
           call vector4_invert_direction (p_real_onshell (ass_quark(other_leg)))
           call vector4_invert_direction (p_real_onshell (ass_boson(other_leg)))
        end if
     end associate
  end do
end subroutine real_kinematics_apply_threshold_projection_real
```

⟨*phs fks: real kinematics: TBP*⟩+≡

```
procedure :: apply_threshold_projection_born => real_kinematics_apply_threshold_projection_born
```

⟨*phs fks: procedures*⟩+≡

```
subroutine real_kinematics_apply_threshold_projection_born (r, mtop, L_to_cms)
  class(real_kinematics_t), intent(inout) :: r
  real(default), intent(in) :: mtop
  type(lorentz_transformation_t), intent(in) :: L_to_cms
  associate (p_born => r%p_born_cms%phs_point(1)%p, &
       p_born_onshell => r%p_born_onshell%phs_point(1)%p)
```

```
      call threshold_projection_born (mtop, L_to_cms, p_born, p_born_onshell)
    end associate
  end subroutine real_kinematics_apply_threshold_projection_born
```

⟨*phs fks: public*⟩+≡

```
  public :: threshold_projection_born
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine threshold_projection_born (mtop, L_to_cms, p_in, p_onshell)
    real(default), intent(in) :: mtop
    type(lorentz_transformation_t), intent(in) :: L_to_cms
    type(vector4_t), intent(in), dimension(:) :: p_in
    type(vector4_t), intent(out), dimension(:) :: p_onshell
    type(vector4_t), dimension(3) :: k_decay_onshell
    type(vector4_t) :: p_tmp_1, p_tmp_2
    type(lorentz_transformation_t) :: L_to_cms_inv
    p_onshell(1:2) = p_in(1:2)
    L_to_cms_inv = inverse (L_to_cms)
    p_tmp_1 = L_to_cms_inv * p_in(THR_POS_B)
    p_tmp_2 = L_to_cms_inv * p_in(THR_POS_WP)
    k_decay_onshell = create_two_particle_decay (mtop**2, &
         p_tmp_1, p_tmp_2)
    p_onshell([THR_POS_B, THR_POS_WP]) = k_decay_onshell([2, 3])
    p_tmp_1 = L_to_cms * p_in(THR_POS_BBAR)
    p_tmp_2 = L_to_cms * p_in(THR_POS_WM)
    k_decay_onshell = create_two_particle_decay (mtop**2, &
         p_tmp_1, p_tmp_2)
    p_onshell([THR_POS_BBAR, THR_POS_WM]) = k_decay_onshell([2, 3])
    p_onshell([THR_POS_WP, THR_POS_B]) = L_to_cms * p_onshell([THR_POS_WP, THR_POS_B])
    p_onshell([THR_POS_WM, THR_POS_BBAR]) = L_to_cms_inv * p_onshell([THR_POS_WM, THR_POS_BBAR])
  end subroutine threshold_projection_born
```

This routine computes the bounds of the Dalitz region for massive emitters, see below. It is also used by `Powheg`, so the routine is public. The input parameter `m2` corresponds to the squared mass of the emitter and `p` is the four-momentum of the emitter.

⟨*phs fks: public*⟩+≡

```
  public :: compute_dalitz_bounds
```

⟨*phs fks: procedures*⟩+≡

```
  pure subroutine compute_dalitz_bounds (q0, m2, mrec2, z1, z2, k0_rec_max)
    real(default), intent(in) :: q0, m2, mrec2
    real(default), intent(out) :: z1, z2, k0_rec_max
    k0_rec_max = (q0**2 - m2 + mrec2) / (two * q0)
    z1 = (k0_rec_max + sqrt(k0_rec_max**2 - mrec2)) / q0
    z2 = (k0_rec_max - sqrt(k0_rec_max**2 - mrec2)) / q0
  end subroutine compute_dalitz_bounds
```

Compute the `kt2` of a given emitter

⟨*phs fks: real kinematics: TBP*⟩+≡

```
  procedure :: kt2 => real_kinematics_kt2
```

⟨*phs fks: procedures*⟩+≡

```
function real_kinematics_kt2 &
   (real_kinematics, i_phs, emitter, kt2_type, xi, y) result (kt2)
  real(default) :: kt2
  class(real_kinematics_t), intent(in) :: real_kinematics
  integer, intent(in) :: emitter, i_phs, kt2_type
  real(default), intent(in), optional :: xi, y
  real(default) :: xii, yy
  real(default) :: q, E_em, z, z1, z2, m2, mrec2, k0_rec_max
  type(vector4_t) :: p_emitter
  if (present (y)) then
     yy = y
  else
     yy = real_kinematics%y (i_phs)
  end if
  if (present (xi)) then
     xii = xi
  else
     xii = real_kinematics%xi_tilde * real_kinematics%xi_max (i_phs)
  end if
  select case (kt2_type)
  case (FSR_SIMPLE)
     kt2 = real_kinematics%cms_energy2 / two * xii**2 * (1 - yy)
  case (FSR_MASSIVE)
     q = sqrt (real_kinematics%cms_energy2)
     p_emitter = real_kinematics%p_born_cms%phs_point(1)%p(emitter)
     mrec2 = (q - p_emitter%p(0))**2 - sum (p_emitter%p(1:3)**2)
     m2 = p_emitter**2
     E_em = energy (p_emitter)
     call compute_dalitz_bounds (q, m2, mrec2, z1, z2, k0_rec_max)
     z = z2 - (z2 - z1) * (one + yy) / two
     kt2 = xii**2 * q**3 * (one - z) / &
        (two * E_em - z * xii * q)
  case (FSR_MASSLESS_RECOILER)
     kt2 = real_kinematics%cms_energy2 / two * xii**2 * (1 - yy**2) / two
  case default
     kt2 = zero
     call msg_bug ("kt2_type must be set to a known value")
  end select
end function real_kinematics_kt2
```

⟨*phs fks: parameters*⟩≡

```
integer, parameter, public :: FSR_SIMPLE = 1
integer, parameter, public :: FSR_MASSIVE = 2
integer, parameter, public :: FSR_MASSLESS_RECOILER = 3
```

⟨*phs fks: real kinematics: TBP*⟩+≡

```
procedure :: final => real_kinematics_final
```

⟨*phs fks: procedures*⟩+≡

```
subroutine real_kinematics_final (real_kin)
  class(real_kinematics_t), intent(inout) :: real_kin
  if (allocated (real_kin%xi_max)) deallocate (real_kin%xi_max)
  if (allocated (real_kin%y)) deallocate (real_kin%y)
  if (allocated (real_kin%alr_to_i_phs)) deallocate (real_kin%alr_to_i_phs)
```

```
    if (allocated (real_kin%jac_rand)) deallocate (real_kin%jac_rand)
    if (allocated (real_kin%y_soft)) deallocate (real_kin%y_soft)
    if (allocated (real_kin%xi_ref_momenta)) deallocate (real_kin%xi_ref_momenta)
    call real_kin%p_born_cms%final (); call real_kin%p_born_lab%final ()
    call real_kin%p_real_cms%final (); call real_kin%p_real_lab%final ()
  end subroutine real_kinematics_final
```

⟨*phs fks: parameters*⟩+≡

```
  integer, parameter, public :: I_XI = 1
  integer, parameter, public :: I_Y = 2
  integer, parameter, public :: I_PHI = 3

  integer, parameter, public :: PHS_MODE_UNDEFINED = 0
  integer, parameter, public :: PHS_MODE_ADDITIONAL_PARTICLE = 1
  integer, parameter, public :: PHS_MODE_COLLINEAR_REMNANT = 2
```

⟨*phs fks: public*⟩+≡

```
  public :: phs_fks_config_t
```

⟨*phs fks: types*⟩+≡

```
  type, extends (phs_wood_config_t) :: phs_fks_config_t
    integer :: mode = PHS_MODE_UNDEFINED
    character(32) :: md5sum_born_config
    logical :: make_dalitz_plot = .false.
  contains
  ⟨phs fks: fks config: TBP⟩
  end type phs_fks_config_t
```

⟨*phs fks: fks config: TBP*⟩≡

```
  procedure :: clear_phase_space => fks_config_clear_phase_space
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine fks_config_clear_phase_space (phs_config)
    class(phs_fks_config_t), intent(inout) :: phs_config
  end subroutine fks_config_clear_phase_space
```

⟨*phs fks: fks config: TBP*⟩+≡

```
  procedure :: write => phs_fks_config_write
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_config_write (object, unit, include_id)
    class(phs_fks_config_t), intent(in) :: object
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: include_id
    integer :: u
    u = given_output_unit (unit)
    call object%phs_wood_config_t%write (u)
    write (u, "(A,A)") "Extra Born md5sum: ", object%md5sum_born_config
  end subroutine phs_fks_config_write
```

⟨*phs fks: fks config: TBP*⟩+≡

```
  procedure :: set_mode => phs_fks_config_set_mode
```

⟨*phs fks: procedures*⟩+≡

```
    subroutine phs_fks_config_set_mode (phs_config, mode)
      class(phs_fks_config_t), intent(inout) :: phs_config
      integer, intent(in) :: mode
      select case (mode)
      case (NLO_REAL, NLO_MISMATCH)
         phs_config%mode = PHS_MODE_ADDITIONAL_PARTICLE
      case (NLO_DGLAP)
         phs_config%mode = PHS_MODE_COLLINEAR_REMNANT
      end select
    end subroutine phs_fks_config_set_mode
```

⟨*phs fks: fks config: TBP*⟩+≡

```
    procedure :: configure => phs_fks_config_configure
```

⟨*phs fks: procedures*⟩+≡

```
    subroutine phs_fks_config_configure (phs_config, sqrts, &
          sqrts_fixed, cm_frame, azimuthal_dependence, rebuild, &
          ignore_mismatch, nlo_type)
      class(phs_fks_config_t), intent(inout) :: phs_config
      real(default), intent(in) :: sqrts
      logical, intent(in), optional :: sqrts_fixed
      logical, intent(in), optional :: cm_frame
      logical, intent(in), optional :: azimuthal_dependence
      logical, intent(in), optional :: rebuild
      logical, intent(in), optional :: ignore_mismatch
      integer, intent(in), optional :: nlo_type
      if (phs_config%extension_mode == EXTENSION_NONE) then
         select case (phs_config%mode)
         case (PHS_MODE_ADDITIONAL_PARTICLE)
            phs_config%n_par = phs_config%n_par + 3
         case (PHS_MODE_COLLINEAR_REMNANT)
            phs_config%n_par = phs_config%n_par + 1
         end select
      end if
  !!! Channel equivalences not accessible yet
      phs_config%provides_equivalences = .false.
      call phs_config%compute_md5sum ()
    end subroutine phs_fks_config_configure
```

⟨*phs fks: fks config: TBP*⟩+≡

```
    procedure :: startup_message => phs_fks_config_startup_message
```

⟨*phs fks: procedures*⟩+≡

```
    subroutine phs_fks_config_startup_message (phs_config, unit)
      class(phs_fks_config_t), intent(in) :: phs_config
      integer, intent(in), optional :: unit
      call phs_config%phs_wood_config_t%startup_message (unit)
    end subroutine phs_fks_config_startup_message
```

⟨*phs fks: fks config: TBP*⟩+≡

```
    procedure, nopass :: allocate_instance => phs_fks_config_allocate_instance
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_config_allocate_instance (phs)
  class(phs_t), intent(inout), pointer :: phs
  allocate (phs_fks_t :: phs)
end subroutine phs_fks_config_allocate_instance
```

If the phase space is generated from file, but we want to have resonance histories, we must force the cascade sets to be generated. However, it must be assured that Born flavors are used for this.

⟨*phs fks: fks config: TBP*⟩+≡

```
procedure :: generate_phase_space_extra => phs_fks_config_generate_phase_space_extra
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_config_generate_phase_space_extra (phs_config)
  class(phs_fks_config_t), intent(inout) :: phs_config
  integer :: off_shell, extra_off_shell
  type(flavor_t), dimension(:,:), allocatable :: flv_born
  integer :: i, j
  integer :: n_state, n_flv_born
  allocate (phs_config%cascade_set)
  n_flv_born = size (phs_config%flv, 1) - 1
  n_state = size (phs_config%flv, 2)
  allocate (flv_born (n_flv_born, n_state))
  do i = 1, n_flv_born
     do j = 1, n_state
        flv_born(i, j) = phs_config%flv(i, j)
     end do
  end do
  off_shell = phs_config%par%off_shell
  do extra_off_shell = 0, max (n_flv_born - 2, 0)
     phs_config%par%off_shell = off_shell + extra_off_shell
     call cascade_set_generate (phs_config%cascade_set, &
        phs_config%model, phs_config%n_in, phs_config%n_out - 1, &
        flv_born, phs_config%par, phs_config%fatal_beam_decay)
     if (cascade_set_is_valid (phs_config%cascade_set)) exit
  end do
  if (.not. cascade_set_is_valid (phs_config%cascade_set)) &
     call msg_fatal ("Resonance extraction: Phase space generation failed")
end subroutine phs_fks_config_generate_phase_space_extra
```

⟨*phs fks: fks config: TBP*⟩+≡

```
procedure :: set_born_config => phs_fks_config_set_born_config
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_config_set_born_config (phs_config, phs_cfg_born)
  class(phs_fks_config_t), intent(inout) :: phs_config
  type(phs_wood_config_t), intent(in), target :: phs_cfg_born
  call msg_debug (D_PHASESPACE, "phs_fks_config_set_born_config")
  phs_config%forest = phs_cfg_born%forest
  phs_config%n_channel = phs_cfg_born%n_channel
  allocate (phs_config%channel (phs_config%n_channel))
  phs_config%channel = phs_cfg_born%channel
  phs_config%n_par = phs_cfg_born%n_par
  phs_config%n_state = phs_cfg_born%n_state
```

```
    phs_config%sqrts = phs_cfg_born%sqrts
    phs_config%par = phs_cfg_born%par
    phs_config%sqrts_fixed = phs_cfg_born%sqrts_fixed
    phs_config%azimuthal_dependence = phs_cfg_born%azimuthal_dependence
    phs_config%provides_chains = phs_cfg_born%provides_chains
    phs_config%cm_frame = phs_cfg_born%cm_frame
    phs_config%vis_channels = phs_cfg_born%vis_channels
    allocate (phs_config%chain (size (phs_cfg_born%chain)))
    phs_config%chain = phs_cfg_born%chain
    phs_config%model => phs_cfg_born%model
    if (allocated (phs_cfg_born%cascade_set)) then
       allocate (phs_config%cascade_set)
       phs_config%cascade_set = phs_cfg_born%cascade_set
    end if
    phs_config%md5sum_born_config = phs_cfg_born%md5sum_phs_config
  end subroutine phs_fks_config_set_born_config
```

⟨*phs fks: fks config: TBP*⟩+≡

```
  procedure :: get_resonance_histories => phs_fks_config_get_resonance_histories
```

⟨*phs fks: procedures*⟩+≡

```
  function phs_fks_config_get_resonance_histories (phs_config) result (resonance_histories)
    type(resonance_history_t), dimension(:), allocatable :: resonance_histories
    class(phs_fks_config_t), intent(inout) :: phs_config
    if (allocated (phs_config%cascade_set)) then
       call cascade_set_get_resonance_histories &
          (phs_config%cascade_set, n_filter = 2, res_hists = resonance_histories)
    else
       call msg_debug (D_PHASESPACE, "Have to rebuild phase space for resonance histories")
       call phs_config%generate_phase_space_extra ()
       call cascade_set_get_resonance_histories &
          (phs_config%cascade_set, n_filter = 2, res_hists = resonance_histories)
    end if
  end function phs_fks_config_get_resonance_histories
```

⟨*phs fks: public*⟩+≡

```
  public :: dalitz_plot_t
```

⟨*phs fks: types*⟩+≡

```
  type :: dalitz_plot_t
     integer :: unit = -1
     type(string_t) :: filename
     logical :: active = .false.
     logical :: inverse = .false.
  contains
  ⟨phs fks: dalitz plot: TBP⟩
  end type dalitz_plot_t
```

⟨*phs fks: dalitz plot: TBP*⟩≡

```
  procedure :: init => dalitz_plot_init
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine dalitz_plot_init (plot, unit, filename, inverse)
    class(dalitz_plot_t), intent(inout) :: plot
```

```
    integer, intent(in) :: unit
    type(string_t), intent(in) :: filename
    logical, intent(in) :: inverse
    plot%active = .true.
    plot%unit = unit
    plot%inverse = inverse
    open (plot%unit, file = char (filename), action = "write")
  end subroutine dalitz_plot_init
```

⟨*phs fks: dalitz plot: TBP*⟩+≡

```
  procedure :: write_header => dalitz_plot_write_header
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine dalitz_plot_write_header (plot)
    class(dalitz_plot_t), intent(in) :: plot
    write (plot%unit, "(A36)") "### Dalitz plot generated by WHIZARD"
    if (plot%inverse) then
       write (plot%unit, "(A10,1x,A4)") "### k0_n+1", "k0_n"
    else
       write (plot%unit, "(A8,1x,A6)") "### k0_n", "k0_n+1"
    end if
  end subroutine dalitz_plot_write_header
```

⟨*phs fks: dalitz plot: TBP*⟩+≡

```
  procedure :: register => dalitz_plot_register
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine dalitz_plot_register (plot, k0_n, k0_np1)
    class(dalitz_plot_t), intent(in) :: plot
    real(default), intent(in) :: k0_n, k0_np1
    if (plot%inverse) then
       write (plot%unit, "(F8.4,1X,F8.4)") k0_np1, k0_n
    else
       write (plot%unit, "(F8.4,1X,F8.4)") k0_np1, k0_n
    end if
  end subroutine dalitz_plot_register
```

⟨*phs fks: dalitz plot: TBP*⟩+≡

```
  procedure :: final => dalitz_plot_final
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine dalitz_plot_final (plot)
    class(dalitz_plot_t), intent(inout) :: plot
    logical :: opened
    plot%active = .false.
    plot%inverse = .false.
    if (plot%unit >= 0) then
       inquire (unit = plot%unit, opened = opened)
       if (opened) close (plot%unit)
    end if
    plot%filename = var_str (’’)
    plot%unit = -1
  end subroutine dalitz_plot_final
```

⟨*phs fks: parameters*⟩+≡

```
integer, parameter, public :: GEN_REAL_PHASE_SPACE = 1
integer, parameter, public :: GEN_SOFT_MISMATCH = 2
integer, parameter, public :: GEN_SOFT_LIMIT_TEST = 3
integer, parameter, public :: GEN_COLL_LIMIT_TEST = 4
integer, parameter, public :: GEN_ANTI_COLL_LIMIT_TEST = 5
integer, parameter, public :: GEN_SOFT_COLL_LIMIT_TEST = 6
integer, parameter, public :: GEN_SOFT_ANTI_COLL_LIMIT_TEST = 7

real(default), parameter :: xi_tilde_test_soft = 0.0001_default
real(default), parameter :: xi_tilde_test_coll = 0.5_default
real(default), parameter :: y_test_soft = 0.5_default
real(default), parameter :: y_test_coll = 0.999_default
```

Very soft or collinear phase-space points can become a problem for matrix elements providers, as some scalar products cannot be evaluated properly. Here, a nonsensical result can spoil the whole integration. We therefore check the scalar products appearing to be below a certain tolerance.

⟨*phs fks: public*⟩+≡

```
public :: check_scalar_products
```

⟨*phs fks: procedures*⟩+≡

```
function check_scalar_products (p) result (valid)
  logical :: valid
  type(vector4_t), intent(in), dimension(:) :: p
  real(default), parameter :: tolerance = 1E-6_default
  integer :: i, j
  valid = .true.
  do i = 1, size (p)
     do j = i, size (p)
        if (i /= j) then
           if (abs(p(i) * p(j)) < tolerance) then
              valid = .false.
              exit
           end if
        end if
     end do
  end do
end function check_scalar_products
```

`xi_min` should be set to a non-zero value in order to avoid phase-space points with `p_real(emitter) = 0`.

⟨*phs fks: public*⟩+≡

```
public :: phs_fks_generator_t
```

⟨*phs fks: types*⟩+≡

```
type :: phs_fks_generator_t
  integer, dimension(:), allocatable :: emitters
  type(real_kinematics_t), pointer :: real_kinematics => null()
  type(isr_kinematics_t), pointer :: isr_kinematics => null()
  integer :: n_in
  real(default) :: xi_min = tiny_07
  real(default) :: y_max = one
  real(default) :: sqrts
```

```
    real(default) :: E_gluon
    real(default) :: mrec2
    real(default), dimension(:), allocatable :: m2
    logical :: massive_phsp = .false.
    logical, dimension(:), allocatable :: is_massive
    logical :: singular_jacobian = .false.
    integer :: i_fsr_first = -1
    type(resonance_contributors_t), dimension(:), allocatable :: resonance_contributors !!! Put so
    integer :: mode = GEN_REAL_PHASE_SPACE
  contains
  ⟨phs fks: phs fks generator: TBP⟩
  end type phs_fks_generator_t
```

⟨*phs fks: phs fks generator: TBP*⟩≡

```
  procedure :: connect_kinematics => phs_fks_generator_connect_kinematics
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_generator_connect_kinematics &
         (generator, isr_kinematics, real_kinematics, massive_phsp)
    class(phs_fks_generator_t), intent(inout) :: generator
    type(isr_kinematics_t), intent(in), pointer :: isr_kinematics
    type(real_kinematics_t), intent(in), pointer :: real_kinematics
    logical, intent(in) :: massive_phsp
    generator%real_kinematics => real_kinematics
    generator%isr_kinematics => isr_kinematics
    generator%massive_phsp = massive_phsp
  end subroutine phs_fks_generator_connect_kinematics
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
  procedure :: compute_isr_kinematics => phs_fks_generator_compute_isr_kinematics
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_generator_compute_isr_kinematics (generator, r, p_in)
    class(phs_fks_generator_t), intent(inout) :: generator
    real(default), intent(in) :: r
    type(vector4_t), dimension(2), intent(in), optional :: p_in
    integer :: em
    type(vector4_t), dimension(2) :: p

    if (present (p_in)) then
       p = p_in
    else
       p = generator%real_kinematics%p_born_lab%phs_point(1)%p(1:2)
    end if

    associate (isr => generator%isr_kinematics)
       do em = 1, 2
          isr%x(em) = p(em)%p(0) / isr%beam_energy
          isr%z(em) = one - (one - isr%x(em)) * r
          isr%jacobian(em) =  one - isr%x(em)
       end do
       isr%sqrts_born = (p(1) + p(2))**1
    end associate
  end subroutine phs_fks_generator_compute_isr_kinematics
```

⟨*phs fks: phs fks generator: TBP*⟩+≡
```
procedure :: final => phs_fks_generator_final
```

⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_generator_final (generator)
  class(phs_fks_generator_t), intent(inout) :: generator
  if (allocated (generator%emitters)) deallocate (generator%emitters)
  if (associated (generator%real_kinematics)) nullify (generator%real_kinematics)
  if (associated (generator%isr_kinematics)) nullify (generator%isr_kinematics)
  if (allocated (generator%m2)) deallocate (generator%m2)
  generator%massive_phsp = .false.
  if (allocated (generator%is_massive)) deallocate (generator%is_massive)
  generator%singular_jacobian = .false.
  generator%i_fsr_first = -1
  if (allocated (generator%resonance_contributors)) &
         deallocate (generator%resonance_contributors)
  generator%mode = GEN_REAL_PHASE_SPACE
end subroutine phs_fks_generator_final
```

A resonance phase space is uniquely specified via the resonance contributors and the corresponding emitters. The `phs_identifier` type also checks whether the given contributor-emitter configuration has already been evaluated to avoid duplicate computations.

⟨*phs fks: public*⟩+≡
```
public :: phs_identifier_t
```

⟨*phs fks: types*⟩+≡
```
type :: phs_identifier_t
   integer, dimension(:), allocatable :: contributors
   integer :: emitter = -1
   logical :: evaluated = .false.
contains
⟨phs fks: phs identifier: TBP⟩
end type phs_identifier_t
```

⟨*phs fks: phs identifier: TBP*⟩≡
```
generic :: init => init_from_emitter, init_from_emitter_and_contributors
procedure :: init_from_emitter => phs_identifier_init_from_emitter
procedure :: init_from_emitter_and_contributors &
   => phs_identifier_init_from_emitter_and_contributors
```

⟨*phs fks: procedures*⟩+≡
```
subroutine phs_identifier_init_from_emitter (phs_id, emitter)
  class(phs_identifier_t), intent(out) :: phs_id
  integer, intent(in) :: emitter
  phs_id%emitter = emitter
end subroutine phs_identifier_init_from_emitter

subroutine phs_identifier_init_from_emitter_and_contributors &
   (phs_id, emitter, contributors)
   class(phs_identifier_t), intent(out) :: phs_id
   integer, intent(in) :: emitter
   integer, intent(in), dimension(:) :: contributors
   allocate (phs_id%contributors (size (contributors)))
```

```
      phs_id%contributors = contributors
      phs_id%emitter = emitter
   end subroutine phs_identifier_init_from_emitter_and_contributors
```

⟨*phs fks: phs identifier: TBP*⟩+≡

```
   procedure :: check => phs_identifier_check
```

⟨*phs fks: procedures*⟩+≡

```
   function phs_identifier_check (phs_id, emitter, contributors) result (check)
     logical :: check
     class(phs_identifier_t), intent(in) :: phs_id
     integer, intent(in) :: emitter
     integer, intent(in), dimension(:), optional :: contributors
     check = phs_id%emitter == emitter
     if (present (contributors)) then
        if (.not. allocated (phs_id%contributors)) &
           call msg_fatal ("Phs identifier: contributors not allocated!")
        check = check .and. all (phs_id%contributors == contributors)
     end if
   end function phs_identifier_check
```

⟨*phs fks: phs identifier: TBP*⟩+≡

```
   procedure :: write => phs_identifier_write
```

⟨*phs fks: procedures*⟩+≡

```
   subroutine phs_identifier_write (phs_id, unit)
     class(phs_identifier_t), intent(in) :: phs_id
     integer, intent(in), optional :: unit
     integer :: u, i
     u = given_output_unit (unit); if (u < 0) return
     write (u, '(A)') 'phs_identifier: '
     write (u, '(A,1X,I1)') 'Emitter: ', phs_id%emitter
     if (allocated (phs_id%contributors)) then
        write (u, '(A)', advance = 'no') 'Resonance contributors: '
        do i = 1, size (phs_id%contributors)
           write (u, '(I1,1X)', advance = 'no') phs_id%contributors(i)
        end do
     else
        write (u, '(A)') 'No Contributors allocated'
     end if
   end subroutine phs_identifier_write
```

⟨*phs fks: public*⟩+≡

```
   public :: check_for_phs_identifier
```

⟨*phs fks: procedures*⟩+≡

```
   subroutine check_for_phs_identifier (phs_id, n_in, emitter, contributors, phs_exist, i_phs)
      type(phs_identifier_t), intent(in), dimension(:) :: phs_id
      integer, intent(in) :: n_in, emitter
      integer, intent(in), dimension(:), optional :: contributors
      logical, intent(out) :: phs_exist
      integer, intent(out) :: i_phs
      integer :: i
      phs_exist = .false.
      i_phs = -1
```

```
      do i = 1, size (phs_id)
         if (phs_id(i)%emitter < 0) then
            i_phs = i
            exit
         end if
         phs_exist = phs_id(i)%emitter == emitter
         if (present (contributors)) &
            phs_exist = phs_exist .and. all (phs_id(i)%contributors == contributors)
         if (phs_exist) then
            i_phs = i
            exit
         end if
      end do
   end subroutine check_for_phs_identifier
```

The fks phase space type contains the wood phase space and separately the in- and outcoming momenta for the real process and the corresponding Born momenta. Additionally, there are the variables ξ,ξ_{max}, y and ϕ which are used to create the real phase space, as well as the jacobian and its corresponding soft and collinear limit. Lastly, the array `ch_to_em` connects each channel with an emitter.

⟨*phs fks: public*⟩+≡

```
   public :: phs_fks_t
```

⟨*phs fks: types*⟩+≡

```
   type, extends (phs_wood_t) :: phs_fks_t
     integer :: mode = PHS_MODE_UNDEFINED
     type(vector4_t), dimension(:), allocatable :: p_born
     type(vector4_t), dimension(:), allocatable :: q_born
     type(vector4_t), dimension(:), allocatable :: p_real
     type(vector4_t), dimension(:), allocatable :: q_real
     type(vector4_t), dimension(:), allocatable :: p_born_tot
     type(phs_fks_generator_t) :: generator
     logical :: perform_generation = .true.
     real(default) :: r_isr
     type(phs_identifier_t), dimension(:), allocatable :: phs_identifiers
   contains
   ⟨phs fks: phs fks: TBP⟩
   end type phs_fks_t
```

⟨*phs fks: interfaces*⟩≡

```
   interface compute_beta
     module procedure compute_beta_massless
     module procedure compute_beta_massive
   end interface

   interface get_xi_max_fsr
     module procedure get_xi_max_fsr_massless
     module procedure get_xi_max_fsr_massive
   end interface
```

⟨*phs fks: phs fks: TBP*⟩≡
```
procedure :: write => phs_fks_write
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_write (object, unit, verbose)
  class(phs_fks_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u, i, n_id
  u = given_output_unit (unit)
  call object%base_write ()
  n_id = size (object%phs_identifiers)
  if (n_id == 0) then
     write (u, "(A)") "No phs identifiers allocated! "
  else
     do i = 1, n_id
        call object%phs_identifiers(i)%write (u)
     end do
  end if
end subroutine phs_fks_write
```

Initializer for the phase space. Calls the initialization of the corresponding Born phase space, sets up the channel-emitter-association and allocates space for the momenta.

⟨*phs fks: phs fks: TBP*⟩+≡
```
procedure :: init => phs_fks_init
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_init (phs, phs_config)
  class(phs_fks_t), intent(out) :: phs
  class(phs_config_t), intent(in), target :: phs_config

  call phs%base_init (phs_config)
  select type (phs_config)
  type is (phs_fks_config_t)
     phs%config => phs_config
     phs%forest = phs_config%forest
  end select

  select type(phs)
  type is (phs_fks_t)
    select type (phs_config)
    type is (phs_fks_config_t)
       phs%mode = phs_config%mode
    end select

    select case (phs%mode)
    case (PHS_MODE_ADDITIONAL_PARTICLE)
       phs%n_r_born = phs%config%n_par - 3
    case (PHS_MODE_COLLINEAR_REMNANT)
       phs%n_r_born = phs%config%n_par - 1
    end select
  end select
end subroutine phs_fks_init
```

⟨*phs fks: phs fks: TBP*⟩+≡
```
procedure :: allocate_momenta => phs_fks_allocate_momenta
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_allocate_momenta (phs, phs_config, data_is_born)
  class(phs_fks_t), intent(inout) :: phs
  class(phs_config_t), intent(in) :: phs_config
  logical, intent(in) :: data_is_born
  integer :: n_out_born
  allocate (phs%p_born (phs_config%n_in))
  allocate (phs%p_real (phs_config%n_in))
  select case (phs%mode)
  case (PHS_MODE_ADDITIONAL_PARTICLE)
     if (data_is_born) then
        n_out_born = phs_config%n_out
     else
        n_out_born = phs_config%n_out - 1
     end if
     allocate (phs%q_born (n_out_born))
     allocate (phs%q_real (n_out_born + 1))
     allocate (phs%p_born_tot (phs_config%n_in + n_out_born))
  end select
end subroutine phs_fks_allocate_momenta
```

⟨*phs fks: phs fks: TBP*⟩+≡
```
procedure :: set_incoming_momenta => phs_fks_set_incoming_momenta
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_set_incoming_momenta (phs, p)
  class(phs_fks_t), intent(inout) :: phs
  type(vector4_t), dimension(:), intent(in) :: p
  call phs%phs_wood_t%set_incoming_momenta (p)
end subroutine phs_fks_set_incoming_momenta
```

Evaluate selected channel. First, the subroutine calls the evaluation procedure of the underlying Born phase space, using $n_r - 3$ random numbers. Then, the remaining three random numbers are used to create ξ, y and ϕ, from which the real momenta are calculated from the Born momenta.

⟨*phs fks: phs fks: TBP*⟩+≡
```
procedure :: evaluate_selected_channel => phs_fks_evaluate_selected_channel
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_evaluate_selected_channel (phs, c_in, r_in)
  class(phs_fks_t), intent(inout) :: phs
  integer, intent(in) :: c_in
  real(default), intent(in), dimension(:) :: r_in
  integer :: n_in

  call phs%phs_wood_t%evaluate_selected_channel (c_in, r_in)
  phs%r(:,c_in) = r_in

  phs%q_defined = phs%phs_wood_t%q_defined
  if (.not. phs%q_defined) return
```

```
    if (phs%perform_generation) then
       select case (phs%mode)
       case (PHS_MODE_ADDITIONAL_PARTICLE)
          n_in = phs%config%n_in
          phs%p_born = phs%phs_wood_t%p
          phs%q_born = phs%phs_wood_t%q
          phs%p_born_tot (1: n_in) = phs%p_born
          phs%p_born_tot (n_in + 1 :) = phs%q_born
          call phs%set_reference_frames ()
          call phs%set_isr_kinematics ()
       case (PHS_MODE_COLLINEAR_REMNANT)
          call phs%compute_isr_kinematics (r_in(phs%n_r_born + 1))
          phs%r_isr = r_in(phs%n_r_born + 1)
       end select
    end if
  end subroutine phs_fks_evaluate_selected_channel
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
  procedure :: evaluate_other_channels => phs_fks_evaluate_other_channels
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_evaluate_other_channels (phs, c_in)
    class(phs_fks_t), intent(inout) :: phs
    integer, intent(in) :: c_in
    call phs%phs_wood_t%evaluate_other_channels (c_in)
    phs%r_defined = .true.
  end subroutine phs_fks_evaluate_other_channels
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
  procedure :: get_mcpar => phs_fks_get_mcpar
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_get_mcpar (phs, c, r)
    class(phs_fks_t), intent(in) :: phs
    integer, intent(in) :: c
    real(default), dimension(:), intent(out) :: r
    r(1 : phs%n_r_born) = phs%r(1 : phs%n_r_born,c)
    select case (phs%mode)
    case (PHS_MODE_ADDITIONAL_PARTICLE)
       r(phs%n_r_born + 1 :) = phs%r_real
    case (PHS_MODE_COLLINEAR_REMNANT)
       r(phs%n_r_born + 1 :) = phs%r_isr
    end select
  end subroutine phs_fks_get_mcpar
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
  procedure :: set_beam_energy => phs_fks_set_beam_energy
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_set_beam_energy (phs)
    class(phs_fks_t), intent(inout) :: phs
    call phs%generator%set_sqrts_hat (phs%config%sqrts)
  end subroutine phs_fks_set_beam_energy
```

⟨*phs fks: phs fks: TBP*⟩+≡
```
procedure :: set_emitters => phs_fks_set_emitters
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_set_emitters (phs, emitters)
  class(phs_fks_t), intent(inout) :: phs
  integer, intent(in), dimension(:), allocatable :: emitters
  call phs%generator%set_emitters (emitters)
end subroutine phs_fks_set_emitters
```

⟨*phs fks: phs fks: TBP*⟩+≡
```
procedure :: set_momenta => phs_fks_set_momenta
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_set_momenta (phs, p)
  class(phs_fks_t), intent(inout) :: phs
  type(vector4_t), intent(in), dimension(:) :: p
  integer :: n_in, n_tot_born
  select case (phs%mode)
  case (PHS_MODE_ADDITIONAL_PARTICLE)
     n_in = phs%config%n_in; n_tot_born = phs%config%n_tot - 1
     phs%p_born = p(1 : n_in)
     phs%q_born = p(n_in + 1 : n_tot_born)
     phs%p_born_tot = p
  end select
end subroutine phs_fks_set_momenta
```

⟨*phs fks: phs fks: TBP*⟩+≡
```
procedure :: setup_masses => phs_fks_setup_masses
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_setup_masses (phs, n_tot)
  class(phs_fks_t), intent(inout) :: phs
  integer, intent(in) :: n_tot
  call phs%generator%setup_masses (n_tot)
end subroutine phs_fks_setup_masses
```

⟨*phs fks: phs fks: TBP*⟩+≡
```
procedure :: get_born_momenta => phs_fks_get_born_momenta
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_get_born_momenta (phs, p)
  class(phs_fks_t), intent(inout) :: phs
  type(vector4_t), intent(out), dimension(:) :: p
  select case (phs%mode)
  case (PHS_MODE_ADDITIONAL_PARTICLE)
     p(1 : phs%config%n_in) = phs%p_born
     p(phs%config%n_in + 1 :) = phs%q_born
  case (PHS_MODE_COLLINEAR_REMNANT)
     p(1:phs%config%n_in) = phs%phs_wood_t%p
     p(phs%config%n_in + 1 : ) = phs%phs_wood_t%q
  end select
  if (.not. phs%config%cm_frame) p = phs%lt_cm_to_lab * p
end subroutine phs_fks_get_born_momenta
```

⟨*phs fks: phs fks: TBP*⟩+≡
```
procedure :: get_outgoing_momenta => phs_fks_get_outgoing_momenta
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_get_outgoing_momenta (phs, q)
  class(phs_fks_t), intent(in) :: phs
  type(vector4_t), intent(out), dimension(:) :: q
  select case (phs%mode)
  case (PHS_MODE_ADDITIONAL_PARTICLE)
     q = phs%q_real
  case (PHS_MODE_COLLINEAR_REMNANT)
     q = phs%phs_wood_t%q
  end select
end subroutine phs_fks_get_outgoing_momenta
```

⟨*phs fks: phs fks: TBP*⟩+≡
```
procedure :: get_incoming_momenta => phs_fks_get_incoming_momenta
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_get_incoming_momenta (phs, p)
  class(phs_fks_t), intent(in) :: phs
  type(vector4_t), intent(inout), dimension(:), allocatable :: p
  p = phs%p_real
end subroutine phs_fks_get_incoming_momenta
```

⟨*phs fks: phs fks: TBP*⟩+≡
```
procedure :: set_isr_kinematics => phs_fks_set_isr_kinematics
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_set_isr_kinematics (phs, p_born)
  class(phs_fks_t), intent(inout) :: phs
  type(vector4_t), dimension(2), intent(in), optional :: p_born
  call phs%generator%set_isr_kinematics (p_born)
end subroutine phs_fks_set_isr_kinematics
```

⟨*phs fks: phs fks: TBP*⟩+≡
```
procedure :: generate_radiation_variables => &
   phs_fks_generate_radiation_variables
```
⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_generate_radiation_variables (phs, r_in, threshold)
  class(phs_fks_t), intent(inout) :: phs
  real(default), intent(in), dimension(:) :: r_in
  logical, intent(in) :: threshold
  type(vector4_t), dimension(:), allocatable :: p_born
  if (size (r_in) /= 3) call msg_fatal &
       ("Real kinematics need to be generated using three random numbers!")
  select case (phs%mode)
  case (PHS_MODE_ADDITIONAL_PARTICLE)
     allocate (p_born (size (phs%p_born_tot)))
     if (threshold) then
        p_born = phs%get_onshell_projected_momenta ()
     else
        p_born = phs%p_born_tot
```

```
      end if
      call phs%generator%generate_radiation_variables &
           (r_in, p_born, phs%phs_identifiers, threshold)
      phs%r_real = r_in
    end select
  end subroutine phs_fks_generate_radiation_variables
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
  procedure :: compute_xi_ref_momenta => phs_fks_compute_xi_ref_momenta
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_compute_xi_ref_momenta (phs, contributors)
    class(phs_fks_t), intent(inout) :: phs
    type(resonance_contributors_t), intent(in), dimension(:), optional :: contributors
    if (phs%mode == PHS_MODE_ADDITIONAL_PARTICLE) &
         call phs%generator%compute_xi_ref_momenta (phs%p_born_tot, contributors)
  end subroutine phs_fks_compute_xi_ref_momenta
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
  procedure :: compute_xi_ref_momenta_threshold => phs_fks_compute_xi_ref_momenta_threshold
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_compute_xi_ref_momenta_threshold (phs)
    class(phs_fks_t), intent(inout) :: phs
    select case (phs%mode)
    case (PHS_MODE_ADDITIONAL_PARTICLE)
       call phs%generator%compute_xi_ref_momenta_threshold &
            (phs%get_onshell_projected_momenta ())
    end select
  end subroutine phs_fks_compute_xi_ref_momenta_threshold
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
  procedure :: compute_cms_energy => phs_fks_compute_cms_energy
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_compute_cms_energy (phs)
    class(phs_fks_t), intent(inout) :: phs
    if (phs%mode == PHS_MODE_ADDITIONAL_PARTICLE) &
         call phs%generator%compute_cms_energy (phs%p_born_tot)
  end subroutine phs_fks_compute_cms_energy
```

When initial-state radiation is involved, either due to beamnstrahlung or QCD corrections, it is important to have access to both the phase space points in the center-of-mass and lab frame.

⟨*phs fks: phs fks: TBP*⟩+≡

```
  procedure :: set_reference_frames => phs_fks_set_reference_frames
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_set_reference_frames (phs)
    class(phs_fks_t), intent(inout) :: phs
    type(lorentz_transformation_t) :: lt_lab_to_cms
    associate (real_kinematics => phs%generator%real_kinematics)
       if (phs%config%cm_frame) then
```

```
            real_kinematics%p_born_cms%phs_point(1)%p = phs%p_born_tot
            real_kinematics%p_born_lab%phs_point(1)%p = phs%p_born_tot
         else
            real_kinematics%p_born_lab%phs_point(1)%p = phs%p_born_tot
            lt_lab_to_cms = inverse (phs%lt_cm_to_lab)
            real_kinematics%p_born_cms%phs_point(1)%p = &
               lt_lab_to_cms * phs%p_born_tot
         end if
      end associate
   end subroutine phs_fks_set_reference_frames
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
   procedure :: i_phs_is_isr => phs_fks_i_phs_is_isr
```

⟨*phs fks: procedures*⟩+≡

```
   function phs_fks_i_phs_is_isr (phs, i_phs) result (is_isr)
     logical :: is_isr
     class(phs_fks_t), intent(in) :: phs
     integer, intent(in) :: i_phs
     is_isr = phs%phs_identifiers(i_phs)%emitter <= phs%generator%n_in
   end function phs_fks_i_phs_is_isr
```

18.9.1 Creation of the real phase space - FSR

At this point, the Born phase space has been generated, as well as the three random variables ξ, y and ϕ. The question is how the real phase space is generated for a final-state emission configuration. We work with two different sets of momenta, the Born configuration $\left\{\bar{k}_\oplus, \bar{k}_\ominus, \bar{k}_1, ..., \bar{k}_n\right\}$ and the real configuration $\left\{k_\oplus, k_\ominus, k_1, ..., k_n, k_{n+1}\right\}$. We define the momentum of the emitter to be on the n-th position and the momentum of the radiated particle to be at position $n+1$. The magnitude of the spatial component of k is denoted by $\underline{k}$.

For final-state emissions, it is $\bar{k}_\oplus = k_\oplus$ and $\bar{k}_\ominus = k_\ominus$. Thus, the center-of-mass systems coincide and it is

$$q = \sum_{i=1}^{n} \bar{k}_i = \sum_{i=1}^{n+1} k_i, \tag{18.65}$$

with $\vec{q} = 0$ and $q^2 = \left(q^0\right)^2$.

We want to construct the real phase space from the Born phase space using three random numbers. They are defined as follows:

- $\xi = \frac{2k^0_{n+1}}{\sqrt{s}} \in [0, \xi_{max}]$, where k_{n+1} denotes the four-momentum of the radiated particle.
- $y = \cos\theta = \frac{\vec{k}_n \cdot \vec{k}_{n+1}}{\underline{k}_n \underline{k}_{n+1}}$ is the splitting angle.
- The angle between tho two splitting particles in the transversal plane, $phi \in [0, 2\pi]$.

Further, $k_{rec} = \sum_{i=1}^{n-1} k_i$ denotes the sum of all recoiling momenta.

⟨*phs fks: phs fks generator: TBP*⟩+≡
```
generic :: generate_fsr => generate_fsr_default, generate_fsr_resonances
```

⟨*phs fks: phs fks generator: TBP*⟩+≡
```
procedure :: generate_fsr_default => phs_fks_generator_generate_fsr_default
```

⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_generator_generate_fsr_default (generator, emitter, i_phs, &
   p_born, p_real, xi_y_phi, no_jacobians)
  class(phs_fks_generator_t), intent(inout) :: generator
  integer, intent(in) :: emitter, i_phs
  type(vector4_t), intent(in), dimension(:) :: p_born
  type(phs_point_t), intent(inout) :: p_real
  real(default), intent(in), dimension(3), optional :: xi_y_phi
  logical, intent(in), optional :: no_jacobians
  real(default) :: q0

  call generator%generate_fsr_in (p_born, p_real)
  if (generator%n_in == 1) then
     q0 = energy (p_born(1))
  else
     q0 = energy (p_born(1) + p_born(2))
  end if

  generator%i_fsr_first = generator%n_in + 1
  call generator%generate_fsr_out (emitter, i_phs, p_born, p_real, q0, &
       xi_y_phi = xi_y_phi, no_jacobians = no_jacobians)
  if (debug_active (D_PHASESPACE)) then
     call vector4_check_momentum_conservation (p_real%p, generator%n_in, &
         rel_smallness = 1000 * tiny_07, abs_smallness = tiny_07)
  end if
end subroutine phs_fks_generator_generate_fsr_default
```

⟨*phs fks: phs fks generator: TBP*⟩+≡
```
procedure :: generate_fsr_resonances => phs_fks_generator_generate_fsr_resonances
```

⟨*phs fks: procedures*⟩+≡
```
subroutine phs_fks_generator_generate_fsr_resonances (generator, &
     emitter, i_phs, i_con, p_born, p_real, xi_y_phi, no_jacobians)
  class(phs_fks_generator_t), intent(inout) :: generator
  integer, intent(in) :: emitter, i_phs
  integer, intent(in) :: i_con
  type(vector4_t), intent(in), dimension(:) :: p_born
  type(phs_point_t), intent(inout) :: p_real
  real(default), intent(in), dimension(3), optional :: xi_y_phi
  logical, intent(in), optional :: no_jacobians
  integer, dimension(:), allocatable :: resonance_list
  integer, dimension(size(p_born)) :: inv_resonance_list
  type(vector4_t), dimension(:), allocatable :: p_tmp_born
  type(phs_point_t) :: p_tmp_real
  type(vector4_t) :: p_resonance
  real(default) :: q0
  integer :: i, j, nlegborn, nlegreal
  integer :: i_emitter
```

```
type(lorentz_transformation_t) :: boost_to_resonance
integer :: n_resonant_particles
call msg_debug2 (D_PHASESPACE, "phs_fks_generator_generate_fsr_resonances")
nlegborn = size (p_born); nlegreal = nlegborn + 1
allocate (resonance_list (size (generator%resonance_contributors(i_con)%c)))
resonance_list = generator%resonance_contributors(i_con)%c
n_resonant_particles = size (resonance_list)

if (.not. any (resonance_list == emitter)) then
   call msg_fatal ("Emitter must be included in the resonance list!")
else
   do i = 1, n_resonant_particles
      if (resonance_list (i) == emitter) i_emitter = i
   end do
end if

inv_resonance_list = &
   create_inverse_resonance_list (nlegborn, resonance_list)

p_tmp_real = n_resonant_particles + 1
allocate (p_tmp_born (n_resonant_particles))
p_tmp_born = vector4_null
j = 1
do i = 1, n_resonant_particles
   p_tmp_born(j) = p_born (resonance_list(i))
   j = j + 1
end do

call generator%generate_fsr_in (p_born, p_real)

p_resonance = generator%real_kinematics%xi_ref_momenta(i_con)
q0 = p_resonance**1

boost_to_resonance = inverse (boost (p_resonance, q0))
p_tmp_born = boost_to_resonance * p_tmp_born

generator%i_fsr_first = 1
call generator%generate_fsr_out (emitter, i_phs, p_tmp_born, p_tmp_real, &
     q0, i_emitter, xi_y_phi)
p_tmp_real = inverse (boost_to_resonance) * p_tmp_real

do i = generator%n_in + 1, nlegborn
   if (any (resonance_list == i)) then
      p_real%p(i) = p_tmp_real%p(inv_resonance_list (i))
   else
      p_real%p(i) = p_born (i)
   end if
end do
p_real%p(nlegreal) = p_tmp_real%p (n_resonant_particles + 1)

if (debug_active (D_PHASESPACE)) then
   call vector4_check_momentum_conservation (p_real%p, generator%n_in, &
        rel_smallness = 1000 * tiny_07, abs_smallness = tiny_07)
end if
```

```
contains

  function create_inverse_resonance_list (nlegborn, resonance_list) &
     result (inv_resonance_list)
     integer, intent(in) :: nlegborn
     integer, intent(in), dimension(:) :: resonance_list
     integer, dimension(nlegborn) :: inv_resonance_list
     integer :: i, j
     inv_resonance_list = 0
     j = 1
     do i = 1, nlegborn
        if (any (i == resonance_list)) then
           inv_resonance_list (i) = j
           j = j + 1
        end if
     end do
  end function create_inverse_resonance_list

  function boosted_energy () result (E)
    real(default) :: E
    type(vector4_t) :: p_boost
    p_boost = boost_to_resonance * p_resonance
    E = p_boost%p(0)
  end function boosted_energy
end subroutine phs_fks_generator_generate_fsr_resonances
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
procedure :: generate_fsr_threshold => phs_fks_generator_generate_fsr_threshold
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_generator_generate_fsr_threshold (generator, &
     emitter, i_phs, p_born, p_real, xi_y_phi)
  class(phs_fks_generator_t), intent(inout) :: generator
  integer, intent(in) :: emitter, i_phs
  type(vector4_t), intent(in), dimension(:) :: p_born
  type(phs_point_t), intent(inout) :: p_real
  real(default), intent(in), dimension(3), optional :: xi_y_phi
  type(vector4_t), dimension(2) :: p_tmp_born
  type(phs_point_t) :: p_tmp_real
  integer :: nlegborn, nlegreal
  type(vector4_t) :: p_top
  real(default) :: q0
  type(lorentz_transformation_t) :: boost_to_top
  integer :: leg, other_leg
  real(default) :: sqrts, mtop
  call msg_debug2 (D_PHASESPACE, "phs_fks_generator_generate_fsr_resonances")
  nlegborn = size (p_born); nlegreal = nlegborn + 1

  leg = thr_leg(emitter); other_leg = 3 - leg
  p_tmp_real = 3
  p_tmp_born(1) = p_born (ass_boson(leg))
  p_tmp_born(2) = p_born (ass_quark(leg))
```

```
      call generator%generate_fsr_in (p_born, p_real)

      p_top = generator%real_kinematics%xi_ref_momenta(leg)

      q0 = p_top**1
      sqrts = two * p_born(1)%p(0)
      mtop = m1s_to_mpole (sqrts)
      if (sqrts**2 - four * mtop**2 > zero) then
         boost_to_top = inverse (boost (p_top, q0))
      else
         boost_to_top = identity
      end if
      p_tmp_born = boost_to_top * p_tmp_born

      generator%i_fsr_first = 1
      call generator%generate_fsr_out (emitter, i_phs, p_tmp_born, &
           p_tmp_real, q0, 2, xi_y_phi)
      p_tmp_real = inverse (boost_to_top) * p_tmp_real

      p_real%p(ass_boson(leg)) = p_tmp_real%p(1)
      p_real%p(ass_quark(leg)) = p_tmp_real%p(2)
      p_real%p(ass_boson(other_leg)) = p_born(ass_boson(other_leg))
      p_real%p(ass_quark(other_leg)) = p_born(ass_quark(other_leg))
      p_real%p(THR_POS_GLUON) = p_tmp_real%p(3)

    end subroutine phs_fks_generator_generate_fsr_threshold
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
    procedure :: generate_fsr_in => phs_fks_generator_generate_fsr_in
```

⟨*phs fks: procedures*⟩+≡

```
    subroutine phs_fks_generator_generate_fsr_in (generator, p_born, p_real)
      class(phs_fks_generator_t), intent(inout) :: generator
      type(vector4_t), intent(in), dimension(:) :: p_born
      type(phs_point_t), intent(inout) :: p_real
      integer :: i
      do i = 1, generator%n_in
         p_real%p(i) = p_born(i)
      end do
    end subroutine phs_fks_generator_generate_fsr_in
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
    procedure :: generate_fsr_out => phs_fks_generator_generate_fsr_out
```

⟨*phs fks: procedures*⟩+≡

```
    subroutine phs_fks_generator_generate_fsr_out (generator, &
        emitter, i_phs, p_born, p_real, q0, p_emitter_index, xi_y_phi, no_jacobians)
      class(phs_fks_generator_t), intent(inout) :: generator
      integer, intent(in) :: emitter, i_phs
      type(vector4_t), intent(in), dimension(:) :: p_born
      type(phs_point_t), intent(inout) :: p_real
      real(default), intent(in) :: q0
      integer, intent(in), optional :: p_emitter_index
      real(default), intent(in), dimension(3), optional :: xi_y_phi
```

```
logical, intent(in), optional :: no_jacobians
real(default) :: xi, y, phi
integer :: nlegborn, nlegreal
real(default) :: uk_np1, uk_n
real(default) :: uk_rec, k_rec0
type(vector3_t) :: k_n_born, k
real(default) :: uk_n_born, uk, k2, k0_n
real(default) :: cpsi, beta
type(vector3_t) :: vec, vec_orth
type(lorentz_transformation_t) :: rot
integer :: i, p_em
logical :: compute_jac
p_em = emitter; if (present (p_emitter_index)) p_em = p_emitter_index
compute_jac = .true.
if (present (no_jacobians)) compute_jac = .not. no_jacobians
if (generator%i_fsr_first < 0) &
   call msg_fatal ("FSR generator is called for outgoing particles but "&
      &"i_fsr_first is not set!")

if (present (xi_y_phi)) then
   xi = xi_y_phi(I_XI)
   y = xi_y_phi(I_Y)
   phi = xi_y_phi(I_PHI)
else
   associate (rad_var => generator%real_kinematics)
      xi = rad_var%xi_tilde
      if (rad_var%supply_xi_max) xi = xi * rad_var%xi_max(i_phs)
      y = rad_var%y(i_phs)
      phi = rad_var%phi
   end associate
end if

nlegborn = size (p_born)
nlegreal = nlegborn + 1
generator%E_gluon = q0 * xi / two
uk_np1 = generator%E_gluon
k_n_born = p_born(p_em)%p(1:3)
uk_n_born = k_n_born**1

generator%mrec2 = (q0 - p_born(p_em)%p(0))**2 &
     - space_part_norm(p_born(p_em))**2
if (generator%is_massive(emitter)) then
   call generator%compute_emitter_kinematics (y, emitter, &
        i_phs, q0, k0_n, uk_n, uk, compute_jac)
else
   call generator%compute_emitter_kinematics (y, q0, uk_n, uk)
   generator%real_kinematics%y_soft(i_phs) = y
   k0_n = uk_n
end if

call msg_debug2 (D_PHASESPACE, "phs_fks_generator_generate_fsr_out")
call debug_input_values ()

vec = uk_n / uk_n_born * k_n_born
```

```
      vec_orth = create_orthogonal (vec)
      p_real%p(p_em)%p(0) = k0_n
      p_real%p(p_em)%p(1:3) = vec%p(1:3)
      cpsi = (uk_n**2 + uk**2 - uk_np1**2) / (two * uk_n * uk)
      !!! This is to catch the case where cpsi = 1, but numerically
      !!! turns out to be slightly larger than 1.
      call check_cpsi_bound (cpsi)
      rot = rotation (cpsi, - sqrt (one - cpsi**2), vec_orth)
      p_real%p(p_em) = rot * p_real%p(p_em)
      vec = uk_np1 / uk_n_born * k_n_born
      vec_orth = create_orthogonal (vec)
      p_real%p(nlegreal)%p(0) = uk_np1
      p_real%p(nlegreal)%p(1:3) = vec%p(1:3)
      cpsi = (uk_np1**2 + uk**2 - uk_n**2) / (two * uk_np1 * uk)
      call check_cpsi_bound (cpsi)
      rot = rotation (cpsi, sqrt (one - cpsi**2), vec_orth)
      p_real%p(nlegreal) = rot * p_real%p(nlegreal)
      call construct_recoiling_momenta ()
      if (compute_jac) call compute_jacobians ()

    contains
```

⟨*phs fks: generator generate fsr out procedures*⟩

```
    end subroutine phs_fks_generator_generate_fsr_out
```

⟨*phs fks: generator generate fsr out procedures*⟩≡

```
    subroutine debug_input_values ()
      if (debug2_active (D_PHASESPACE)) then
         call generator%write ()
         print *, 'emitter =    ', emitter
         print *, 'p_born:'
         call vector4_write_set (p_born)
         print *, 'p_real:'
         call p_real%write()
         print *, 'q0 =    ', q0
         if (present(p_emitter_index)) then
            print *, 'p_emitter_index =    ', p_emitter_index
         else
            print *, 'p_emitter_index not given'
         end if
      end if
    end subroutine debug_input_values
```

⟨*phs fks: generator generate fsr out procedures*⟩+≡

```
    subroutine check_cpsi_bound (cpsi)
      real(default), intent(inout) :: cpsi
      if (cpsi > one) then
         cpsi = one
      else if (cpsi < -one) then
         cpsi = - one
      end if
    end subroutine check_cpsi_bound
```

Construction of the recoiling momenta. The reshuffling of momenta must not change the invariant mass of the recoiling system, which means $k_{\rm rec}^2 = \bar{k}_{\rm rec}^{\;2}$. Therefore, the momenta are related by a boost, $\bar{k}_i = \Lambda k_i$. The boost parameter is

$$\beta = \frac{q^2 - (k_{\rm rec}^0 + \underline{k}_{\rm rec})^2}{q^2 + (k_{\rm rec}^0 + \underline{k}_{\rm rec})^2}$$

⟨*phs fks: generator generate fsr out procedures*⟩+≡

```
subroutine construct_recoiling_momenta ()
  type(lorentz_transformation_t) :: lambda
  k_rec0 = q0 - p_real%p(p_em)%p(0) - p_real%p(nlegreal)%p(0)
  uk_rec = sqrt (k_rec0**2 - generator%mrec2)
  if (generator%is_massive(emitter)) then
     beta = compute_beta (q0**2, k_rec0, uk_rec, &
            p_born(p_em)%p(0), uk_n_born)
  else
     beta = compute_beta (q0**2, k_rec0, uk_rec)
  end if
  k = p_real%p(p_em)%p(1:3) + p_real%p(nlegreal)%p(1:3)
  vec%p(1:3) = one / uk * k%p(1:3)
  lambda = boost (beta / sqrt(one - beta**2), vec)
  do i = generator%i_fsr_first, nlegborn
    if (i /= p_em) then
       p_real%p(i) = lambda * p_born(i)
    end if
  end do
  vec%p(1:3) = p_born(p_em)%p(1:3) / uk_n_born
  rot = rotation (cos(phi), sin(phi), vec)
  p_real%p(nlegreal) = rot * p_real%p(nlegreal)
  p_real%p(p_em) = rot * p_real%p(p_em)
end subroutine construct_recoiling_momenta
```

The factor $\frac{q^2}{(4\pi)^3}$ is not included here since it is supplied during phase space generation. Also, we already divide by ξ.

⟨*phs fks: generator generate fsr out procedures*⟩+≡

```
subroutine compute_jacobians ()
  associate (jac => generator%real_kinematics%jac(i_phs))
     if (generator%is_massive(emitter)) then
        jac%jac(1) = jac%jac(1) * four / q0 / uk_n_born / xi
     else
        k2 = two * uk_n * uk_np1* (one - y)
        jac%jac(1) = uk_n**2 / uk_n_born / (uk_n - k2 / (two * q0))
     end if
     jac%jac(2) = one
     jac%jac(3) = one - xi / two * q0 / uk_n_born
  end associate
end subroutine compute_jacobians
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
procedure :: generate_fsr => phs_fks_generate_fsr
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_generate_fsr (phs, emitter, i_phs, p_real, i_con, &
       xi_y_phi, no_jacobians)
  class(phs_fks_t), intent(inout) :: phs
  integer, intent(in) :: emitter, i_phs
  type(phs_point_t), intent(inout) :: p_real
  integer, intent(in), optional :: i_con
  real(default), intent(in), dimension(3), optional :: xi_y_phi
  logical, intent(in), optional :: no_jacobians
  type(vector4_t), dimension(:), allocatable :: p
  associate (generator => phs%generator)
     allocate (p (1 : generator%real_kinematics%p_born_cms%get_n_particles()), &
        source = generator%real_kinematics%p_born_cms%phs_point(1)%p)
     generator%real_kinematics%supply_xi_max = .true.
     if (present (i_con)) then
        call generator%generate_fsr (emitter, i_phs, i_con, p, p_real, &
             xi_y_phi, no_jacobians)
     else
        call generator%generate_fsr (emitter, i_phs, p, p_real, &
             xi_y_phi, no_jacobians)
     end if
     generator%real_kinematics%p_real_cms%phs_point(i_phs)%p = p_real%p
     if (.not. phs%config%cm_frame) p_real = phs%lt_cm_to_lab * p_real
     generator%real_kinematics%p_real_lab%phs_point(i_phs)%p = p_real%p
  end associate
end subroutine phs_fks_generate_fsr
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
procedure :: threshold_projection => phs_fks_threshold_projection
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_threshold_projection (phs, mtop)
  class(phs_fks_t), intent(inout) :: phs
  real(default), intent(in) :: mtop
  type(lorentz_transformation_t) :: L
  associate (generator => phs%generator)
     L = get_boost_for_threshold_projection &
          (generator%real_kinematics%p_born_cms%get_momenta (1), phs%get_sqrts (), mtop)
     call generator%real_kinematics%apply_threshold_projection_born (mtop, L)
  end associate
end subroutine phs_fks_threshold_projection
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
procedure :: get_onshell_projected_momenta => phs_fks_get_onshell_projected_momenta
```

⟨*phs fks: procedures*⟩+≡

```
pure function phs_fks_get_onshell_projected_momenta (phs) result (p)
  type(vector4_t), dimension(:), allocatable :: p
  class(phs_fks_t), intent(in) :: phs
  !!! !!! !!! Workaround for standard-semantics ifort 16.0 bug
  allocate(p (size (phs%generator%real_kinematics%p_born_onshell%phs_point(1)%p)))
  p = phs%generator%real_kinematics%p_born_onshell%phs_point(1)%p
end function phs_fks_get_onshell_projected_momenta
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
procedure :: generate_fsr_threshold => phs_fks_generate_fsr_threshold
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_generate_fsr_threshold (phs, emitter, i_phs, p_real)
  class(phs_fks_t), intent(inout) :: phs
  integer, intent(in) :: emitter, i_phs
  type(phs_point_t), intent(inout), optional :: p_real
  type(vector4_t), dimension(:), allocatable :: p_born
  type(phs_point_t) :: pp
  integer :: leg
  associate (generator => phs%generator)
     generator%real_kinematics%supply_xi_max = .true.
     allocate (p_born (1 : generator%real_kinematics%p_born_cms%get_n_particles()))
     p_born = generator%real_kinematics%p_born_onshell%get_momenta (1)
     pp = size (p_born) + 1
     call generator%generate_fsr_threshold (emitter, i_phs, p_born, pp)
     leg = thr_leg(emitter)
     call generator%real_kinematics%p_real_onshell(leg)%set_momenta (i_phs, pp)
     if (present (p_real)) p_real = pp
  end associate
end subroutine phs_fks_generate_fsr_threshold
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
generic :: compute_xi_max => compute_xi_max_internal, compute_xi_max_with_output
procedure :: compute_xi_max_internal => phs_fks_compute_xi_max_internal
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_compute_xi_max_internal (phs, p, threshold)
  class(phs_fks_t), intent(inout) :: phs
  type(vector4_t), intent(in), dimension(:) :: p
  logical, intent(in) :: threshold
  integer :: i_phs, i_con, emitter
  do i_phs = 1, size (phs%phs_identifiers)
     associate (phs_id => phs%phs_identifiers(i_phs), generator => phs%generator)
        emitter = phs_id%emitter
        if (threshold) then
           call generator%compute_xi_max (emitter, i_phs, p, &
                generator%real_kinematics%xi_max(i_phs), i_con = thr_leg(emitter))
        else if (allocated (phs_id%contributors)) then
           do i_con = 1, size (phs_id%contributors)
              call generator%compute_xi_max (emitter, i_phs, p, &
                   generator%real_kinematics%xi_max(i_phs), i_con = 1)
           end do
        else
           call generator%compute_xi_max (emitter, i_phs, p, &
                generator%real_kinematics%xi_max(i_phs))
        end if
     end associate
  end do
end subroutine phs_fks_compute_xi_max_internal
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
procedure :: compute_xi_max_with_output => phs_fks_compute_xi_max_with_output
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_compute_xi_max_with_output (phs, emitter, i_phs, y, p, xi_max)
  class(phs_fks_t), intent(inout) :: phs
  integer, intent(in) :: i_phs, emitter
  real(default), intent(in) :: y
  type(vector4_t), intent(in), dimension(:) :: p
  real(default), intent(out) :: xi_max
  call phs%generator%compute_xi_max (emitter, i_phs, p, xi_max, y_in = y)
end subroutine phs_fks_compute_xi_max_with_output
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
generic :: compute_emitter_kinematics => &
   compute_emitter_kinematics_massless, &
   compute_emitter_kinematics_massive
procedure :: compute_emitter_kinematics_massless => &
   phs_fks_generator_compute_emitter_kinematics_massless
procedure :: compute_emitter_kinematics_massive => &
   phs_fks_generator_compute_emitter_kinematics_massive
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_generator_compute_emitter_kinematics_massless &
   (generator, y, q0, uk_em, uk)
  class(phs_fks_generator_t), intent(inout) :: generator
  real(default), intent(in) :: y, q0
  real(default), intent(out) :: uk_em, uk
  real(default) :: k0_np1, q2

  k0_np1 = generator%E_gluon
  q2 = q0**2

  uk_em = (q2 - generator%mrec2 - two * q0 * k0_np1) / (two * (q0 - k0_np1 * (one - y)))
  uk = sqrt (uk_em**2 + k0_np1**2 + two * uk_em * k0_np1 * y)
end subroutine phs_fks_generator_compute_emitter_kinematics_massless

subroutine phs_fks_generator_compute_emitter_kinematics_massive &
    (generator, y, em, i_phs, q0, k0_em, uk_em, uk, compute_jac)
  class(phs_fks_generator_t), intent(inout) :: generator
  real(default), intent(in) :: y
  integer, intent(in) :: em, i_phs
  real(default), intent(in) :: q0
  real(default), intent(inout) :: k0_em, uk_em, uk
  logical, intent(in) :: compute_jac
  real(default) :: k0_np1, q2, mrec2, m2
  real(default) :: k0_rec_max, k0_em_max, k0_rec, uk_rec
  real(default) :: z, z1, z2

  k0_np1 = generator%E_gluon
  q2 = q0**2
  mrec2 = generator%mrec2
  m2 = generator%m2(em)

  k0_rec_max = (q2 - m2 + mrec2) / (two * q0)
  k0_em_max = (q2 + m2 - mrec2)  /(two * q0)
  z1 = (k0_rec_max + sqrt (k0_rec_max**2 - mrec2)) / q0
```

```
    z2 = (k0_rec_max - sqrt (k0_rec_max**2 - mrec2)) / q0
    z = z2 - (z2 - z1) * (one + y) / two
    k0_em = k0_em_max - k0_np1 * z
    k0_rec = q0 - k0_np1 - k0_em
    uk_em = sqrt(k0_em**2 - m2)
    uk_rec = sqrt(k0_rec**2 - mrec2)
    uk = uk_rec
    if (compute_jac) &
         generator%real_kinematics%jac(i_phs)%jac = q0 * (z1 - z2) / four * k0_np1
    generator%real_kinematics%y_soft(i_phs) = &
       (two * q2 * z - q2 - mrec2 + m2) / (sqrt(k0_em_max**2 - m2) * q0) / two
  end subroutine phs_fks_generator_compute_emitter_kinematics_massive
```

⟨*phs fks: procedures*⟩+≡

```
  function recompute_xi_max (q0, mrec2, m2, y) result (xi_max)
    real(default) :: xi_max
    real(default), intent(in) :: q0, mrec2, m2, y
    real(default) :: q2, k0_np1_max, k0_rec_max
    real(default) :: z1, z2, z
    q2 = q0**2
    k0_rec_max = (q2 - m2 + mrec2) / (two * q0)
    z1 = (k0_rec_max + sqrt (k0_rec_max**2 - mrec2)) / q0
    z2 = (k0_rec_max - sqrt (k0_rec_max**2 - mrec2)) / q0
    z = z2 - (z2 - z1) * (one + y) / 2
    k0_np1_max = - (q2 * z**2 - two * q0 * k0_rec_max * z + mrec2) / (two * q0 * z * (one - z))
    xi_max = two * k0_np1_max / q0
  end function recompute_xi_max
```

⟨*phs fks: procedures*⟩+≡

```
  function compute_beta_massless (q2, k0_rec, uk_rec) result (beta)
    real(default), intent(in) :: q2, k0_rec, uk_rec
    real(default) :: beta
    beta = (q2 - (k0_rec + uk_rec)**2) / (q2 + (k0_rec + uk_rec)**2)
  end function compute_beta_massless

  function compute_beta_massive (q2, k0_rec, uk_rec, &
     k0_em_born, uk_em_born) result (beta)
    real(default), intent(in) :: q2, k0_rec, uk_rec
    real(default), intent(in) :: k0_em_born, uk_em_born
    real(default) :: beta
    real(default) :: k0_rec_born, uk_rec_born, alpha
    k0_rec_born = sqrt(q2) - k0_em_born
    uk_rec_born = uk_em_born
    alpha = (k0_rec + uk_rec) / (k0_rec_born + uk_rec_born)
    beta = (one - alpha**2) / (one + alpha**2)
  end function compute_beta_massive
```

The momentum of the radiated particle is computed according to

$$\underline{k}_n = \frac{q^2 - M_{\rm rec}^2 - 2q^0\underline{k}_{n+1}}{2(q^0 - \underline{k}_{n+1}(1-y))}, \tag{18.66}$$

with $k = k_n + k_{n+1}$ and $M_{\rm rec}^2 = k_{\rm rec}^2 = (q-k)^2$. Because of $\bar{\boldsymbol{k}}_n \parallel \boldsymbol{k}_n + \boldsymbol{k}_{n+1}$

we find $M_{\rm rec}^2 = \left(q - \bar{k}_n\right)^2$. Equation **??** follows from the fact that $\left(\boldsymbol{k} - \boldsymbol{k}_n\right)^2 = \boldsymbol{k}_{n+1}^2$, which is equivalent to $\boldsymbol{k}_n \cdot \boldsymbol{k} = \frac{1}{2}\left(\underline{k}_n^2 + \underline{k}^2 - \underline{k}_{n+1}^2\right)$.
$\boldsymbol{k}_n$ and $\boldsymbol{k}_{n+1}$ are obtained by first setting up vectors parallel to $\bar{\boldsymbol{k}}_n$,

$$\boldsymbol{k}_n' = \underline{k}_n \frac{\bar{\boldsymbol{k}}_n}{\underline{\bar{k}}_n}, \quad \boldsymbol{k}_{n+1}' = \underline{k}_{n+1} \frac{\bar{\boldsymbol{k}}_n}{\underline{\bar{k}}_n},$$

and then rotating these vectors by an amount of $\cos\psi_n = \frac{\boldsymbol{k}_n \cdot \boldsymbol{k}}{\underline{k}_n \underline{k}}$. The emitted particle cannot have more momentum than the emitter has in the Born phase space. Thus, there is an upper bound for ξ, determined by the condition $k_{n+1}^0 = \underline{\bar{k}}_n$, which is equal to

$$\xi_{\rm max} = \frac{2}{\underline{\bar{k}}_n} q^0.$$

⟨*phs fks: procedures*⟩+≡

```
pure function get_xi_max_fsr_massless (p_born, q0, emitter) result (xi_max)
  type(vector4_t), intent(in), dimension(:) :: p_born
  real(default), intent(in) :: q0
  integer, intent(in) :: emitter
  real(default) :: xi_max
  real(default) :: uk_n_born
  uk_n_born = space_part_norm (p_born(emitter))
  xi_max = two * uk_n_born / q0
end function get_xi_max_fsr_massless
```

The computation of $\xi_{\rm max}$ for massive emitters is described in arXiv:1202.0465. Let's recapitulate it here.

We consider the Dalitz-domain created by k_{n+1}^0, k_n^0 and $k_{\rm rec}^0$ and introduce the parameterization

$$k_n^0 = \bar{k}_n^0 - z k_{n+1}^0$$

Then, for each value of z, there exists a maximum value of $\underline{k}_{n+1}$ from which $\xi_{\rm max}$ can be extracted via $\xi_{\rm max} = 2k_{n+1}^0/q$. It is determined by the condition

$$\underline{k}_{n+1} \pm \underline{k}_n \pm \underline{k}_{\rm rec} = 0.$$

This can be manipulated to yield

$$\left(\underline{k}_{n+1}^2 + \underline{k}_n^2 - \underline{k}_{\rm rec}^2\right)^2 = 4\underline{k}_{n+1}^2 \underline{k}_n^2.$$

Here we can use $\underline{k}_n^2 = \left(k_n^0\right)^2 - m^2$ and $\underline{k}_{\rm rec}^2 = \left(q - k_n^0 - k_{n+1}^0\right)^2 - M_{\rm rec}^2$, as well as the above parameterization of k_n^0, to obtain

$$4\underline{k}_{n+1}^2 \left(2\underline{k}_{n+1} q z(1-z) + q^2 z^2 - 2q\bar{k}_{\rm rec}^0 z + M_{\rm rec}^2\right) = 0.$$

Solving for k_{n+1}^0 gives

$$k_{n+1}^0 = \frac{2q\bar{k}_{\rm rec}^0 z - q^2 z^2 - M_{\rm rec}^2}{2qz(1-z)}. \tag{18.67}$$

It is still open how to compute z. For this, consider that the right-hand-side of equation (18.67) vanishes for

$$z_{1,2} = \left(\bar{k}_{\rm rec}^0 \pm \sqrt{\left(\bar{k}_{\rm rec}^0\right)^2 - M_{\rm rec}^2}\right)/q,$$

which corresponds to the borders of the Dalitz-region where the gluon momentum vanishes. Thus we define

$$z = z_2 - \frac{1}{2}(z_2 - z_1)(1 + y).$$

⟨*phs fks: procedures*⟩+≡

```
pure function get_xi_max_fsr_massive (p_born, q0, emitter, m2, y) result (xi_max)
  real(default) :: xi_max
  type(vector4_t), intent(in), dimension(:) :: p_born
  real(default), intent(in) :: q0
  integer, intent(in) :: emitter
  real(default), intent(in) :: m2, y
  real(default) :: mrec2
  real(default) :: k0_rec_max
  real(default) :: z, z1, z2
  real(default) :: k0_np1_max
  associate (p => p_born(emitter)%p)
     mrec2 = (q0 - p(0))**2 - p(1)**2 - p(2)**2 - p(3)**2
  end associate
  call compute_dalitz_bounds (q0, m2, mrec2, z1, z2, k0_rec_max)
  z = z2 - (z2 - z1) * (one + y) / two
  k0_np1_max = - (q0**2 * z**2 - two * q0 * k0_rec_max * z + mrec2) &
     / (two * q0 * z * (one - z))
  xi_max = two * k0_np1_max / q0
end function get_xi_max_fsr_massive
```

⟨*phs fks: parameters*⟩+≡

```
integer, parameter, public :: I_PLUS = 1
integer, parameter, public :: I_MINUS = 2
```

⟨*phs fks: procedures*⟩+≡

```
function get_xi_max_isr (xb, y) result (xi_max)
  real(default), dimension(2), intent(in) :: xb
  real(default), intent(in) :: y
  real(default) :: xb_plus, xb_minus
  real(default) :: xi_max
  real(default) :: plus_val, minus_val
  real(default) :: onepy, onemy

  xb_plus = xb(I_PLUS); xb_minus = xb(I_MINUS)
  onepy = one + y; onemy = one - y

  plus_val = two * onepy * xb_plus**2 / &
             (sqrt ((one + xb_plus**2)**2 * onemy**2 + 16 * y * xb_plus**2) &
             + onemy * (one - xb_plus**2))
  minus_val = two * onemy * xb_minus**2 / &
              (sqrt ((one + xb_minus**2)**2 * onepy**2 - 16 * y * xb_minus**2) &
             + onepy * (one - xb_minus**2))
  xi_max = one - max (plus_val, minus_val)
end function get_xi_max_isr
```

⟨*phs fks: procedures*⟩+≡

```
recursive function get_xi_max_isr_decay (p) result (xi_max)
   real(default) :: xi_max
   type(vector4_t), dimension(:), intent(in) :: p
   integer :: n_tot
   type(vector4_t), dimension(:), allocatable :: p_dec_new
   n_tot = size (p)
   if (n_tot == 3) then
      xi_max = xi_max_one_to_two (p(1), p(2), p(3))
   else
      allocate (p_dec_new (n_tot - 1))
      p_dec_new(1) = sum (p (3 : ))
      p_dec_new(2 : n_tot - 1) = p (3 : n_tot)
      xi_max = min (xi_max_one_to_two (p(1), p(2), sum(p(3 : ))), &
         get_xi_max_isr_decay (p_dec_new))
   end if
contains
  function xi_max_one_to_two (p_in, p_out1, p_out2) result (xi_max)
    real(default) :: xi_max
    type(vector4_t), intent(in) :: p_in, p_out1, p_out2
    real(default) :: m_in, m_out1, m_out2
    m_in = p_in**1
    m_out1 = p_out1**1; m_out2 = p_out2**1
    xi_max = one - (m_out1 + m_out2)**2 / m_in**2
  end function xi_max_one_to_two
end function get_xi_max_isr_decay
```

18.9.2 Creation of the real phase space - ISR

⟨*phs fks: phs fks: TBP*⟩+≡

```
procedure :: generate_isr => phs_fks_generate_isr
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_generate_isr (phs, i_phs, p_born, p_real)
  class(phs_fks_t), intent(inout) :: phs
  integer, intent(in) :: i_phs
  type(vector4_t), intent(in), dimension(:) :: p_born
  type(phs_point_t), intent(inout) :: p_real
  type(vector4_t) :: p0, p1
  type(lorentz_transformation_t) :: lt
  real(default) :: sqrts_hat

  associate (generator => phs%generator)
     select case (generator%n_in)
     case (1)
        call generator%compute_xi_max (1, i_phs, p_born, &
             generator%real_kinematics%xi_max(i_phs))
        call generator%generate_isr_decay (i_phs, p_born, p_real)
        phs%config%cm_frame = .true.
     case (2)
        !!! TODO (cw-2016-11-03): Check this - isn't xi_max overwritten each time?
        call generator%compute_xi_max (1, i_phs, p_born, &
             generator%real_kinematics%xi_max(i_phs))
        call generator%compute_xi_max (2, i_phs, p_born, &
```

```
            generator%real_kinematics%xi_max(i_phs))
         call generator%generate_isr (i_phs, p_born, p_real)
      end select
      phs%generator%real_kinematics%p_real_lab%phs_point(i_phs)%p = p_real%p
      if (.not. phs%config%cm_frame) then
         sqrts_hat = (p_real%p(1) + p_real%p(2))**1
         p0 = p_real%p(1) + p_real%p(2)
         lt = boost (p0, sqrts_hat)
         p1 = inverse(lt) * p_real%p(1)
         lt = lt * rotation_to_2nd (3, space_part (p1))
         phs%generator%real_kinematics%p_real_cms%phs_point(i_phs)%p = &
            inverse (lt) * p_real%p
      else
         phs%generator%real_kinematics%p_real_cms%phs_point(i_phs)%p = p_real%p
      end if
   end associate
end subroutine phs_fks_generate_isr
```

The real phase space for an inital-state emission involved in a decay process is generated by first setting the gluon momentum like in the scattering case by using its angular coordinates y and ϕ and then adjusting the gluon energy with ξ. The emitter momentum is kept identical to the Born case, i.e. $p_{\rm in} = \bar{p}_{\rm in}$, so that after the emission it has momentum $p_{\rm virt} = p_{\rm in} - p_{\rm g}$ and invariant mass $m^2 = p_{\rm virt}^2$. Note that the final state momenta have to remain on-shell, so that $p_1^2 = \bar{p}_1^2 = m_1^2$ and $p_2^2 = \bar{p}_2^2 = m_2^2$. Let Λ be the boost from into the rest frame of the emitter after emission, i.e. $\Lambda p_{\rm virt} = (m, 0, 0, 0)$. In this reference frame, the spatial components of the final-state momenta sum up to zero, and their magnitude is

$$p = \frac{\sqrt{\lambda(m^2, m_1^2, m_2^2)}}{2m},$$

a fact already used in the evaluation of the phase space trees of `phs_forest`. Obviously, from this, the final-state energies can be deferred via $E_i^2 = m_i^2 - p^2$. In the next step, the $p_{1,2}$ are set up as vectors $(E, 0, 0, \pm p)$ along the z-axis and then rotated about the same azimuthal and polar angles as in the Born system. Finally, the momenta are boosted out of the rest frame by multiplying with Λ.

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
procedure :: generate_isr_decay => phs_fks_generator_generate_isr_decay
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_generator_generate_isr_decay (generator, i_phs, p_born, p_real)
   class(phs_fks_generator_t), intent(inout) :: generator
   integer, intent(in) :: i_phs
   type(vector4_t), intent(in), dimension(:) :: p_born
   type(phs_point_t), intent(inout) :: p_real
   real(default) :: xi_max, xi, y, phi
   integer :: nlegborn, nlegreal
   real(default) :: k0_np1
   real(default) :: msq_in
   type(vector4_t) :: p_virt
   real(default) :: jac_real

  associate (rad_var => generator%real_kinematics)
```

```
      xi_max = rad_var%xi_max(i_phs)
      xi = rad_var%xi_tilde * xi_max
      y = rad_var%y(i_phs)
      phi = rad_var%phi
      rad_var%y_soft(i_phs) = y
    end associate

    nlegborn = size (p_born)
    nlegreal = nlegborn + 1

    msq_in = p_born(1)**2
    generator%real_kinematics%jac(i_phs)%jac = one

    p_real%p(1) = p_born(1)
    k0_np1 = p_real%p(1)%p(0) * xi / two
    p_real%p(nlegreal)%p(0) = k0_np1
    p_real%p(nlegreal)%p(1) = k0_np1 * sqrt(one - y**2) * sin(phi)
    p_real%p(nlegreal)%p(2) = k0_np1 * sqrt(one - y**2) * cos(phi)
    p_real%p(nlegreal)%p(3) = k0_np1 * y

    p_virt = p_real%p(1) - p_real%p(nlegreal)
    jac_real = one
    call generate_on_shell_decay (p_virt, &
         p_born(2 : nlegborn), p_real%p(2 : nlegreal - 1), 1, &
         msq_in, jac_real)

    associate (jac => generator%real_kinematics%jac(i_phs))
      jac%jac(1) = jac_real
      jac%jac(2) = one
    end associate

  end subroutine phs_fks_generator_generate_isr_decay
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
  procedure :: generate_isr_factorized => phs_fks_generator_generate_isr_factorized
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_generator_generate_isr_factorized (generator, i_phs, emitter, p_born, p_real)
    class(phs_fks_generator_t), intent(inout) :: generator
    integer, intent(in) :: i_phs, emitter
    type(vector4_t), intent(in), dimension(:) :: p_born
    type(phs_point_t), intent(inout) :: p_real
    type(vector4_t), dimension(:), allocatable :: p_tmp_born
    type(phs_point_t) :: p_tmp_real
    type(vector4_t) :: p_top
    type(lorentz_transformation_t) :: boost_to_rest_frame
    integer, parameter :: nlegreal = 7 !!! Factorized phase space so far only required for ee -> b

    allocate (p_tmp_born (3)); p_tmp_born = vector4_null
    p_tmp_real = 4
    p_real%p(1:2) = p_born(1:2)
    if (emitter == THR_POS_B) then
      p_top = p_born (THR_POS_WP) + p_born (THR_POS_B)
      p_tmp_born(2) = p_born (THR_POS_WP)
```

```
      p_tmp_born(3) = p_born (THR_POS_B)
    else if (emitter == THR_POS_BBAR) then
      p_top = p_born (THR_POS_WM) + p_born (THR_POS_BBAR)
      p_tmp_born(2) = p_born (THR_POS_WM)
      p_tmp_born(3) = p_born (THR_POS_BBAR)
    else
      call msg_fatal ("Threshold computation requires emitters to be at position 5 and 6 " // &
        "Please check if your process specification fulfills this requirement.")
    end if
    p_tmp_born (1) = p_top
    boost_to_rest_frame = inverse (boost (p_top, p_top**1))
    p_tmp_born = boost_to_rest_frame * p_tmp_born
    call generator%compute_xi_max_isr_factorized (i_phs, p_tmp_born)
    call generator%generate_isr_decay (i_phs, p_tmp_born, p_tmp_real)
    p_tmp_real = inverse (boost_to_rest_frame) * p_tmp_real
    if (emitter == THR_POS_B) then
      p_real%p(THR_POS_WP) = p_tmp_real%p (2)
      p_real%p(THR_POS_B) = p_tmp_real%p (3)
      p_real%p(THR_POS_WM) = p_born (THR_POS_WM)
      p_real%p(THR_POS_BBAR) = p_born (THR_POS_BBAR)
    !!! Exception has been handled above
    else
      p_real%p(THR_POS_WM) = p_tmp_real%p (2)
      p_real%p(THR_POS_BBAR) = p_tmp_real%p (3)
      p_real%p(THR_POS_WP) = p_born (THR_POS_WP)
      p_real%p(THR_POS_B) = p_born (THR_POS_B)
    end if
    p_real%p(nlegreal) = p_tmp_real%p (4)
  end subroutine phs_fks_generator_generate_isr_factorized
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
  procedure :: generate_isr => phs_fks_generator_generate_isr
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_generator_generate_isr (generator, i_phs, p_born, p_real)
    !!! Important: Import momenta in the lab frame
    class(phs_fks_generator_t), intent(inout) :: generator
    integer, intent(in) :: i_phs
    type(vector4_t), intent(in) , dimension(:) :: p_born
    type(phs_point_t), intent(inout) :: p_real
    real(default) :: xi_max, xi_tilde, xi, y, phi
    integer :: nlegborn, nlegreal
    real(default) :: sqrts_real
    real(default) :: k0_np1
    type(lorentz_transformation_t) :: lambda_transv, lambda_longit, lambda_longit_inv
    real(default) :: x_plus, x_minus, xb_plus, xb_minus
    real(default) :: onemy, onepy
    integer :: i
    real(default) :: xi_plus, xi_minus
    real(default) :: beta_gamma
    type(vector3_t) :: beta_vec

    associate (rad_var => generator%real_kinematics)
      xi_max = rad_var%xi_max(i_phs)
```

```
  xi_tilde = rad_var%xi_tilde
  xi = xi_tilde * xi_max
  y = rad_var%y(i_phs)
  onemy = one - y; onepy = one + y
  phi = rad_var%phi
  rad_var%y_soft(i_phs) = y
end associate

nlegborn = size (p_born)
nlegreal = nlegborn + 1
generator%isr_kinematics%sqrts_born = (p_born(1) + p_born(2))**1

!!! Initial state real momenta
xb_plus = generator%isr_kinematics%x(I_PLUS)
xb_minus = generator%isr_kinematics%x(I_MINUS)
x_plus = xb_plus / sqrt(one - xi) * sqrt ((two - xi * onemy) / (two - xi * onepy))
x_minus = xb_minus / sqrt(one - xi) * sqrt ((two - xi * onepy) / (two - xi * onemy))
xi_plus = xi_tilde * (one - xb_plus)
xi_minus = xi_tilde * (one - xb_minus)
p_real%p(I_PLUS) = x_plus / xb_plus * p_born(I_PLUS)
p_real%p(I_MINUS) = x_minus / xb_minus * p_born(I_MINUS)
generator%isr_kinematics%z(I_PLUS) = x_plus / xb_plus
generator%isr_kinematics%z(I_MINUS) = x_minus / xb_minus
generator%isr_kinematics%z_coll(I_PLUS) = one / (one - xi_plus)
generator%isr_kinematics%z_coll(I_MINUS) = one / (one - xi_minus)

!!! Create radiation momentum
sqrts_real = generator%isr_kinematics%sqrts_born / sqrt (one - xi)
k0_np1 = sqrts_real * xi / two
p_real%p(nlegreal)%p(0) = k0_np1
p_real%p(nlegreal)%p(1) = k0_np1 * sqrt (one - y**2) * sin(phi)
p_real%p(nlegreal)%p(2) = k0_np1 * sqrt (one - y**2) * cos(phi)
p_real%p(nlegreal)%p(3) = k0_np1 * y

call get_boost_parameters (p_real%p, beta_gamma, beta_vec)
lambda_longit = create_longitudinal_boost (beta_gamma, beta_vec, inverse = .true.)
p_real%p(nlegreal) = lambda_longit * p_real%p(nlegreal)

call get_boost_parameters (p_born, beta_gamma, beta_vec)
lambda_longit = create_longitudinal_boost (beta_gamma, beta_vec, inverse = .false.)
forall (i = 3 : nlegborn) &
    p_real%p(i) = lambda_longit * p_born(i)

lambda_transv = create_transversal_boost (p_real%p(nlegreal), xi, sqrts_real)
forall (i = 3 : nlegborn) &
     p_real%p(i) = lambda_transv * p_real%p(i)

lambda_longit_inv = create_longitudinal_boost (beta_gamma, beta_vec, inverse = .true.)
forall (i = 3 : nlegborn) &
     p_real%p(i) = lambda_longit_inv * p_real%p(i)

!!! Compute jacobians
associate (jac => generator%real_kinematics%jac(i_phs))
   !!! Additional 1 / (1 -xi) factor because in the real jacobian,
```

```
      !!! there is s_real in the numerator
      !!! We also have to adapt the flux factor, which is 1/2s_real for the real component
      !!! The reweighting factor is s_born / s_real, cancelling the (1-x) factor from above
      jac%jac(1) = one / (one - xi)
      jac%jac(2) = one
      jac%jac(3) = one / (one - xi_plus)**2
      jac%jac(4) = one / (one - xi_minus)**2
    end associate
  contains
    subroutine get_boost_parameters (p, beta_gamma, beta_vec)
      type(vector4_t), intent(in), dimension(:) :: p
      real(default), intent(out) :: beta_gamma
      type(vector3_t), intent(out) :: beta_vec
      beta_vec = (p(1)%p(1:3) + p(2)%p(1:3)) / (p(1)%p(0) + p(2)%p(0))
      beta_gamma = beta_vec**1 / sqrt (one - beta_vec**2)
      beta_vec = beta_vec / beta_vec**1
    end subroutine get_boost_parameters

    function create_longitudinal_boost (beta_gamma, beta_vec, inverse) result (lambda)
      real(default), intent(in) :: beta_gamma
      type(vector3_t), intent(in) :: beta_vec
      logical, intent(in) :: inverse
      type(lorentz_transformation_t) :: lambda
      if (inverse) then
        lambda = boost (beta_gamma, beta_vec)
      else
        lambda = boost (-beta_gamma, beta_vec)
      end if
    end function create_longitudinal_boost

    function create_transversal_boost (p_rad, xi, sqrts_real) result (lambda)
      type(vector4_t), intent(in) :: p_rad
      real(default), intent(in) :: xi, sqrts_real
      type(lorentz_transformation_t) :: lambda
      type(vector3_t) :: vec_transverse
      real(default) :: pt2, beta, beta_gamma
      pt2 = transverse_part (p_rad)**2
      beta = one / sqrt (one + sqrts_real**2 * (one - xi) / pt2)
      beta_gamma = beta / sqrt (one - beta**2)
      vec_transverse%p(1:2) = p_rad%p(1:2)
      vec_transverse%p(3) = zero
      vec_transverse = normalize (vec_transverse)
      lambda = boost (-beta_gamma, vec_transverse)
    end function create_transversal_boost
  end subroutine phs_fks_generator_generate_isr
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
  procedure :: set_sqrts_hat => phs_fks_generator_set_sqrts_hat
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_generator_set_sqrts_hat (generator, sqrts)
    class(phs_fks_generator_t), intent(inout) :: generator
    real(default), intent(in) :: sqrts
    generator%sqrts = sqrts
```

```
    end subroutine phs_fks_generator_set_sqrts_hat
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
    procedure :: set_emitters => phs_fks_generator_set_emitters
```

⟨*phs fks: procedures*⟩+≡

```
    subroutine phs_fks_generator_set_emitters (generator, emitters)
      class(phs_fks_generator_t), intent(inout) :: generator
      integer, intent(in), dimension(:), allocatable ::  emitters
      allocate (generator%emitters (size (emitters)))
      generator%emitters = emitters
    end subroutine phs_fks_generator_set_emitters
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
    procedure :: setup_masses => phs_fks_generator_setup_masses
```

⟨*phs fks: procedures*⟩+≡

```
    subroutine phs_fks_generator_setup_masses (generator, n_tot)
      class (phs_fks_generator_t), intent(inout) :: generator
      integer, intent(in) :: n_tot
      if (.not. allocated (generator%m2)) then
         allocate (generator%is_massive (n_tot))
         allocate (generator%m2 (n_tot))
         generator%is_massive = .false.
         generator%m2 = zero
      end if
    end subroutine phs_fks_generator_setup_masses
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
    procedure :: set_xi_and_y_bounds => phs_fks_generator_set_xi_and_y_bounds
```

⟨*phs fks: procedures*⟩+≡

```
    subroutine phs_fks_generator_set_xi_and_y_bounds (generator, xi_min, y_max)
      class(phs_fks_generator_t), intent(inout) :: generator
      real(default), intent(in) :: xi_min, y_max
      generator%xi_min = xi_min
      generator%y_max = y_max
    end subroutine phs_fks_generator_set_xi_and_y_bounds
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
    procedure :: set_isr_kinematics => phs_fks_generator_set_isr_kinematics
```

⟨*phs fks: procedures*⟩+≡

```
    subroutine phs_fks_generator_set_isr_kinematics (generator, p_born)
      class(phs_fks_generator_t), intent(inout) :: generator
      type(vector4_t), dimension(2), intent(in), optional :: p_born
      type(vector4_t), dimension(2) :: p

      if (present (p_born)) then
         p = p_born
      else
         p = generator%real_kinematics%p_born_lab%phs_point(1)%p(1:2)
      end if
```

```
    generator%isr_kinematics%x = p%p(0) / generator%isr_kinematics%beam_energy
  end subroutine phs_fks_generator_set_isr_kinematics
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
  procedure :: generate_radiation_variables => &
     phs_fks_generator_generate_radiation_variables
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_generator_generate_radiation_variables &
     (generator, r_in, p_born, phs_identifiers, threshold)
    class(phs_fks_generator_t), intent(inout) :: generator
    real(default), intent(in), dimension(:) :: r_in
    type(vector4_t), intent(in), dimension(:) :: p_born
    type(phs_identifier_t), intent(in), dimension(:) :: phs_identifiers
    logical, intent(in), optional :: threshold

    if (any (generator%emitters <= 2) .and. generator%n_in > 1) &
        call generator%set_isr_kinematics &
           (generator%real_kinematics%p_born_lab%phs_point(1)%p(1:2))

    associate (rad_var => generator%real_kinematics)
       rad_var%phi = r_in (I_PHI) * twopi
       select case (generator%mode)
       case (GEN_REAL_PHASE_SPACE)
          rad_var%jac_rand = twopi
          call generator%compute_y_real_phs (r_in(I_Y), p_born, phs_identifiers, &
               rad_var%jac_rand, rad_var%y, threshold)
       case (GEN_SOFT_MISMATCH)
          rad_var%jac_mismatch = twopi
          call generator%compute_y_mismatch (r_in(I_Y), rad_var%jac_mismatch, &
               rad_var%y_mismatch, rad_var%y_soft)
       case default
          call generator%compute_y_test (rad_var%y)
       end select
       call generator%compute_xi_tilde (r_in(I_XI))
       call generator%set_masses (p_born, phs_identifiers)
    end associate
  end subroutine phs_fks_generator_generate_radiation_variables
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
  procedure :: compute_xi_ref_momenta => phs_fks_generator_compute_xi_ref_momenta
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_generator_compute_xi_ref_momenta &
         (generator, p_born, resonance_contributors)
    class(phs_fks_generator_t), intent(inout) :: generator
    type(vector4_t), intent(in), dimension(:) :: p_born
    type(resonance_contributors_t), intent(in), dimension(:), optional &
       :: resonance_contributors
    integer :: i_con, n_contributors
    if (present (resonance_contributors)) then
       n_contributors = size (resonance_contributors)
       if (.not. allocated (generator%resonance_contributors)) &
          allocate (generator%resonance_contributors (n_contributors))
```

```
      do i_con = 1, n_contributors
         generator%real_kinematics%xi_ref_momenta(i_con) = &
              get_resonance_momentum (p_born, resonance_contributors(i_con)%c)
         generator%resonance_contributors(i_con) = resonance_contributors(i_con)
      end do
   else
      generator%real_kinematics%xi_ref_momenta(1) = sum (p_born(1:generator%n_in))
   end if
end subroutine phs_fks_generator_compute_xi_ref_momenta
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
procedure :: compute_xi_ref_momenta_threshold &
     => phs_fks_generator_compute_xi_ref_momenta_threshold
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_generator_compute_xi_ref_momenta_threshold (generator, p_born)
  class(phs_fks_generator_t), intent(inout) :: generator
  type(vector4_t), intent(in), dimension(:) :: p_born
  generator%real_kinematics%xi_ref_momenta(1) = p_born(THR_POS_WP) + p_born(THR_POS_B)
  generator%real_kinematics%xi_ref_momenta(2) = p_born(THR_POS_WM) + p_born(THR_POS_BBAR)
end subroutine phs_fks_generator_compute_xi_ref_momenta_threshold
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
procedure :: compute_cms_energy => phs_fks_generator_compute_cms_energy
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_generator_compute_cms_energy (generator, p_born)
  class(phs_fks_generator_t), intent(inout) :: generator
  type(vector4_t), intent(in), dimension(:) :: p_born
  type(vector4_t) :: p_sum
  p_sum = sum (p_born (1 : generator%n_in))
  generator%real_kinematics%cms_energy2 = p_sum**2
end subroutine phs_fks_generator_compute_cms_energy
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
procedure :: compute_xi_max => phs_fks_generator_compute_xi_max
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_generator_compute_xi_max (generator, emitter, &
     i_phs, p, xi_max, i_con, y_in)
  class(phs_fks_generator_t), intent(inout) :: generator
  integer, intent(in) :: i_phs, emitter
  type(vector4_t), intent(in), dimension(:) :: p
  real(default), intent(out) :: xi_max
  integer, intent(in), optional :: i_con
  real(default), intent(in), optional :: y_in
  real(default) :: q0
  type(vector4_t), dimension(:), allocatable :: pp
  type(vector4_t) :: p_res
  type(lorentz_transformation_t) :: L_to_resonance
  real(default) :: y
  if (.not. any (generator%emitters == emitter)) return
  allocate (pp (size (p)))
  associate (rad_var => generator%real_kinematics)
```

```
      if (present (i_con)) then
         q0 = rad_var%xi_ref_momenta(i_con)**1
      else
         q0 = energy (sum (p(1:generator%n_in)))
      end if
      if (present (y_in)) then
         y = y_in
      else
         y = rad_var%y(i_phs)
      end if
      select case (generator%n_in)
      case (1)
         if (emitter > 1) then
            if (generator%is_massive(emitter)) then
               xi_max = get_xi_max_fsr (p, q0, emitter, &
                    generator%m2(emitter), y)
            else
               xi_max = get_xi_max_fsr (p, q0, emitter)
            end if
         else
            xi_max = get_xi_max_isr_decay (p)
         end if
      case (2)
         if (present (i_con)) then
            p_res = rad_var%xi_ref_momenta(i_con)
            L_to_resonance = inverse (boost (p_res, q0))
            pp = L_to_resonance * p
         else
            pp = p
         end if
         if (emitter <= 2) then
            xi_max = get_xi_max_isr (generator%isr_kinematics%x, y)
         else
            if (generator%is_massive(emitter)) then
               xi_max = get_xi_max_fsr (pp, q0, emitter, generator%m2(emitter), y)
            else
               xi_max = get_xi_max_fsr (pp, q0, emitter)
            end if
         end if
      case default
         call msg_fatal ("Real phase space: " // &
            "Only one or two initial particles supported")
      end select
   end associate
end subroutine phs_fks_generator_compute_xi_max
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
procedure :: compute_xi_max_isr_factorized &
   => phs_fks_generator_compute_xi_max_isr_factorized
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_generator_compute_xi_max_isr_factorized &
   (generator, i_phs, p)
  class(phs_fks_generator_t), intent(inout) :: generator
```

```
    integer, intent(in) :: i_phs
    type(vector4_t), intent(in), dimension(:) :: p
    generator%real_kinematics%xi_max(i_phs) = get_xi_max_isr_decay (p)
  end subroutine phs_fks_generator_compute_xi_max_isr_factorized
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
  procedure :: set_masses => phs_fks_generator_set_masses
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_generator_set_masses (generator, p, phs_identifiers)
    class(phs_fks_generator_t), intent(inout) :: generator
    type(phs_identifier_t), intent(in), dimension(:) :: phs_identifiers
    type(vector4_t), intent(in), dimension(:) :: p
    integer :: emitter, i_phs
    do i_phs = 1, size (phs_identifiers)
       emitter = phs_identifiers(i_phs)%emitter
       if (any (generator%emitters == emitter) .and. emitter > 0) then
          if (generator%is_massive (emitter) .and. emitter > generator%n_in) &
             generator%m2(emitter) = p(emitter)**2
       end if
    end do
  end subroutine phs_fks_generator_set_masses
```

⟨*phs fks: public*⟩+≡

```
  public :: compute_y_from_emitter
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine compute_y_from_emitter (r_y, p, emitter, massive, &
         y_max, jac_rand, y, contributors, threshold)
    real(default), intent(in) :: r_y
    type(vector4_t), intent(in), dimension(:) :: p
    integer, intent(in) :: emitter
    logical, intent(in) :: massive
    real(default), intent(in) :: y_max
    real(default), intent(inout) :: jac_rand
    real(default), intent(out) :: y
    integer, intent(in), dimension(:), allocatable, optional :: contributors
    logical, intent(in), optional :: threshold
    logical :: thr, resonance
    type(vector4_t) :: p_res, p_em
    real(default) :: q0
    type(lorentz_transformation_t) :: boost_to_resonance
    integer :: i
    real(default) :: beta, one_m_beta, one_p_beta
    thr = .false.; if (present (threshold)) thr = threshold
    p_res = vector4_null
    if (present (contributors)) then
       resonance = allocated (contributors)
    else
       resonance = .false.
    end if
    if (massive) then
       if (resonance) then
          do i = 1, size (contributors)
```

```
            p_res = p_res + p(contributors(i))
         end do
         q0 = p_res**1
         boost_to_resonance = inverse (boost (p_res, q0))
         p_em = boost_to_resonance * p(emitter)
      else if (thr) then
         !p_res = generator%real_kinematics%xi_ref_momenta (thr_leg (emitter))
         p_res = p(ass_boson(thr_leg(emitter))) + p(ass_quark(thr_leg(emitter)))
         q0 = p_res**1
         boost_to_resonance = inverse (boost (p_res, q0))
         p_em = boost_to_resonance * p(emitter)
      else
         q0 = energy (p(1) + p(2))
         p_em = p(emitter)
      end if
      beta = beta_emitter (q0, p_em)
      one_m_beta = one - beta
      one_p_beta = one + beta
      y = one / beta * (one - one_p_beta * &
             exp ( - r_y * log(one_p_beta / one_m_beta)))
      jac_rand = jac_rand * &
             (one - beta * y) * log(one_p_beta / one_m_beta) / beta
   else
      y = (one - two * r_y) * y_max
      jac_rand = jac_rand * 3 * (one - y**2)
      y = 1.5_default * (y - y**3 / 3)
   end if
end subroutine compute_y_from_emitter
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
procedure :: compute_y_real_phs => phs_fks_generator_compute_y_real_phs
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_generator_compute_y_real_phs (generator, r_y, p, phs_identifiers, &
     jac_rand, y, threshold)
  class(phs_fks_generator_t), intent(inout) :: generator
  real(default), intent(in) :: r_y
  type(vector4_t), intent(in), dimension(:) :: p
  type(phs_identifier_t), intent(in), dimension(:) :: phs_identifiers
  real(default), intent(inout), dimension(:) :: jac_rand
  real(default), intent(out), dimension(:) :: y
  logical, intent(in), optional :: threshold
  real(default) :: beta, one_p_beta, one_m_beta
  type(lorentz_transformation_t) :: boost_to_resonance
  real(default) :: q0
  type(vector4_t) :: p_res, p_em
  integer :: i, i_phs, emitter
  logical :: thr
  logical :: construct_massive_fsr = .false.
  thr = .false.; if (present (threshold)) thr = threshold
  do i_phs = 1, size (phs_identifiers)
     emitter = phs_identifiers(i_phs)%emitter
     !!! We need this additional check because of decay phase spaces
     !!! t -> bW has a massive emitter at position 1, which should
```

```
        !!! not be treated here.
        construct_massive_fsr = emitter > generator%n_in
        if (construct_massive_fsr) construct_massive_fsr = &
           construct_massive_fsr .and. generator%is_massive (emitter)
        call compute_y_from_emitter (r_y, p, emitter, construct_massive_fsr, &
             generator%y_max, jac_rand(i_phs), y(i_phs), &
             phs_identifiers(i_phs)%contributors, threshold)
      end do
    end subroutine phs_fks_generator_compute_y_real_phs
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
    procedure :: compute_y_mismatch => phs_fks_generator_compute_y_mismatch
```

⟨*phs fks: procedures*⟩+≡

```
    subroutine phs_fks_generator_compute_y_mismatch (generator, r_y, jac_rand, y, y_soft)
      class(phs_fks_generator_t), intent(inout) :: generator
      real(default), intent(in) :: r_y
      real(default), intent(inout) :: jac_rand
      real(default), intent(out) :: y
      real(default), intent(out), dimension(:) :: y_soft
      y = (one - two * r_y) * generator%y_max
      jac_rand = jac_rand * 3 * (one - y**2)
      y = 1.5_default * (y - y**3 / 3)
      y_soft = y
    end subroutine phs_fks_generator_compute_y_mismatch
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
    procedure :: compute_y_test => phs_fks_generator_compute_y_test
```

⟨*phs fks: procedures*⟩+≡

```
    subroutine phs_fks_generator_compute_y_test (generator, y)
      class(phs_fks_generator_t), intent(inout) :: generator
      real(default), intent(out), dimension(:):: y
      select case (generator%mode)
      case (GEN_SOFT_LIMIT_TEST)
         y = y_test_soft
      case (GEN_COLL_LIMIT_TEST)
         y = y_test_coll
      case (GEN_ANTI_COLL_LIMIT_TEST)
         y = - y_test_coll
      case (GEN_SOFT_COLL_LIMIT_TEST)
         y = y_test_coll
      case (GEN_SOFT_ANTI_COLL_LIMIT_TEST)
         y = - y_test_coll
      end select
    end subroutine phs_fks_generator_compute_y_test
```

⟨*phs fks: public*⟩+≡

```
    public :: beta_emitter
```

⟨*phs fks: procedures*⟩+≡

```
    pure function beta_emitter (q0, p) result (beta)
      real(default), intent(in) :: q0
      type(vector4_t), intent(in) :: p
```

```
    real(default) :: beta
    real(default) :: m2, mrec2, k0_max
    m2 = p**2
    mrec2 = (q0 - p%p(0))**2 - p%p(1)**2 - p%p(2)**2 - p%p(3)**2
    k0_max = (q0**2 - mrec2 + m2) / (two * q0)
    beta = sqrt(one - m2 / k0_max**2)
  end function beta_emitter
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
  procedure :: compute_xi_tilde => phs_fks_generator_compute_xi_tilde
```

⟨*phs fks: procedures*⟩+≡

```
  pure subroutine phs_fks_generator_compute_xi_tilde (generator, r)
    class(phs_fks_generator_t), intent(inout) :: generator
    real(default), intent(in) :: r
    real(default) :: deno
    associate (rad_var => generator%real_kinematics)
      select case (generator%mode)
      case (GEN_REAL_PHASE_SPACE)
        if (generator%singular_jacobian) then
          rad_var%xi_tilde = (one - generator%xi_min) - (one - r)**2 * &
            (one - two * generator%xi_min)
          rad_var%jac_rand = rad_var%jac_rand * two * (one - r) * &
            (one - two * generator%xi_min)
        else
          rad_var%xi_tilde = generator%xi_min + r * (one - generator%xi_min)
          rad_var%jac_rand = rad_var%jac_rand * (one - generator%xi_min)
        end if
      case (GEN_SOFT_MISMATCH)
        deno = one - r
        if (deno < tiny_13) deno = tiny_13
        rad_var%xi_mismatch = generator%xi_min + r / deno
        rad_var%jac_mismatch = rad_var%jac_mismatch / deno**2
      case (GEN_SOFT_LIMIT_TEST)
        rad_var%xi_tilde = r * two * xi_tilde_test_soft
        rad_var%jac_rand = two * xi_tilde_test_soft
      case (GEN_COLL_LIMIT_TEST)
        rad_var%xi_tilde = xi_tilde_test_coll
        rad_var%jac_rand = xi_tilde_test_coll
      case (GEN_ANTI_COLL_LIMIT_TEST)
        rad_var%xi_tilde = xi_tilde_test_coll
        rad_var%jac_rand = xi_tilde_test_coll
      case (GEN_SOFT_COLL_LIMIT_TEST)
        rad_var%xi_tilde = r * two * xi_tilde_test_soft
        rad_var%jac_rand = two * xi_tilde_test_soft
      case (GEN_SOFT_ANTI_COLL_LIMIT_TEST)
        rad_var%xi_tilde = r * two * xi_tilde_test_soft
        rad_var%jac_rand = two * xi_tilde_test_soft
      end select
    end associate
  end subroutine phs_fks_generator_compute_xi_tilde
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
  procedure :: prepare_generation => phs_fks_generator_prepare_generation
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_generator_prepare_generation (generator, r_in, i_phs, &
   emitter, p_born, phs_identifiers, contributors, i_con)
  class(phs_fks_generator_t), intent(inout) :: generator
  real(default), dimension(3), intent(in) :: r_in
  integer, intent(in) :: i_phs, emitter
  type(vector4_t), intent(in), dimension(:) :: p_born
  type(phs_identifier_t), intent(in), dimension(:) :: phs_identifiers
  type(resonance_contributors_t), intent(in), dimension(:), optional :: contributors
  integer, intent(in), optional :: i_con
  call generator%generate_radiation_variables (r_in, p_born, phs_identifiers)
  call generator%compute_xi_ref_momenta (p_born, contributors)
  call generator%compute_xi_max (emitter, i_phs, p_born, &
       generator%real_kinematics%xi_max(i_phs), i_con = i_con)
end subroutine phs_fks_generator_prepare_generation
```

Get `xi` and `y` from an external routine (e.g. `powheg`) and generate an FSR phase space. Note that the flag `supply\_xi\_max` is set to `.false.` because it is assumed that the upper bound on `xi` has already been taken into account during its generation.

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
procedure :: generate_fsr_from_xi_and_y => &
   phs_fks_generator_generate_fsr_from_xi_and_y
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_generator_generate_fsr_from_xi_and_y (generator, xi, y, &
   phi, emitter, i_phs, p_born, p_real)
  class(phs_fks_generator_t), intent(inout) :: generator
  real(default), intent(in) :: xi, y, phi
  integer, intent(in) :: emitter, i_phs
  type(vector4_t), intent(in), dimension(:) :: p_born
  type(phs_point_t), intent(inout) :: p_real
  associate (rad_var => generator%real_kinematics)
     rad_var%supply_xi_max = .false.
     rad_var%xi_tilde = xi
     rad_var%y(i_phs) = y
     rad_var%phi = phi
  end associate
  call generator%set_sqrts_hat (p_born(1)%p(0) + p_born(2)%p(0))
  call generator%generate_fsr (emitter, i_phs, p_born, p_real)
end subroutine phs_fks_generator_generate_fsr_from_xi_and_y
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
procedure :: get_radiation_variables => &
   phs_fks_generator_get_radiation_variables
```

⟨*phs fks: procedures*⟩+≡

```
pure subroutine phs_fks_generator_get_radiation_variables (generator, &
   i_phs, xi, y, phi)
  class(phs_fks_generator_t), intent(in) :: generator
  integer, intent(in) :: i_phs
  real(default), intent(out) :: xi, y
  real(default), intent(out), optional :: phi
  associate (rad_var => generator%real_kinematics)
```

```
      xi = rad_var%xi_max(i_phs) * rad_var%xi_tilde
      y = rad_var%y(i_phs)
      if (present (phi)) phi = rad_var%phi
    end associate
  end subroutine phs_fks_generator_get_radiation_variables
```

⟨*phs fks: phs fks generator: TBP*⟩+≡

```
  procedure :: write => phs_fks_generator_write
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_generator_write (generator, unit)
    class(phs_fks_generator_t), intent(in) :: generator
    integer, intent(in), optional :: unit
    integer :: u
    type(string_t) :: massive_phsp
    u = given_output_unit (unit); if (u < 0) return
    if (generator%massive_phsp) then
       massive_phsp = " massive "
    else
       massive_phsp = " massless "
    end if
    write (u, "(A)") char ("This is a generator for a" &
         // massive_phsp // "phase space")
    if (associated (generator%real_kinematics)) then
       call generator%real_kinematics%write ()
    else
       write (u, "(A)") "Warning: There are no real " // &
            "kinematics associated with this generator"
    end if
    call write_separator (u)
    write (u, "(A,F5.3)") "sqrts: ", generator%sqrts
    write (u, "(A,F5.3)") "E_gluon: ", generator%E_gluon
    write (u, "(A,F5.3)") "mrec2: ", generator%mrec2
  end subroutine phs_fks_generator_write
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
  procedure :: compute_isr_kinematics => phs_fks_compute_isr_kinematics
```

⟨*phs fks: procedures*⟩+≡

```
  subroutine phs_fks_compute_isr_kinematics (phs, r)
    class(phs_fks_t), intent(inout) :: phs
    real(default), intent(in) :: r
    if (.not. phs%config%cm_frame) then
      call phs%generator%compute_isr_kinematics (r, phs%lt_cm_to_lab * phs%phs_wood_t%p)
    else
       call phs%generator%compute_isr_kinematics (r, phs%phs_wood_t%p)
    end if
  end subroutine phs_fks_compute_isr_kinematics
```

⟨*phs fks: phs fks: TBP*⟩+≡

```
  procedure :: final => phs_fks_final
```

⟨*phs fks: procedures*⟩+≡

```
subroutine phs_fks_final (object)
  class(phs_fks_t), intent(inout) :: object
  call phs_forest_final (object%forest)
  call object%generator%final ()
end subroutine phs_fks_final
```

⟨*phs fks: public*⟩+≡

```
public :: get_filtered_resonance_histories
```

⟨*phs fks: procedures*⟩+≡

```
subroutine filter_particles_from_resonances (res_hist, exclusion_list, &
  model, res_hist_filtered)
  type(resonance_history_t), intent(in), dimension(:) :: res_hist
  type(string_t), intent(in), dimension(:) :: exclusion_list
  type(model_t), intent(in) :: model
  type(resonance_history_t), intent(out), dimension(:), allocatable :: res_hist_filtered
  integer :: i_hist, i_flv, i_new, n_orig
  logical, dimension(size (res_hist)) :: to_filter
  type(flavor_t) :: flv
  to_filter = .false.
  n_orig = size (res_hist)
  do i_flv = 1, size (exclusion_list)
     call flv%init (exclusion_list (i_flv), model)
     do i_hist = 1, size (res_hist)
        if (res_hist(i_hist)%has_flavor (flv)) to_filter (i_hist) = .true.
     end do
  end do
  allocate (res_hist_filtered (n_orig - count (to_filter)))
  i_new = 1
  do i_hist = 1, size (res_hist)
     if (.not. to_filter (i_hist)) then
        res_hist_filtered (i_new) = res_hist (i_hist)
        i_new = i_new + 1
     end if
  end do
end subroutine filter_particles_from_resonances
```

⟨*phs fks: procedures*⟩+≡

```
subroutine clean_resonance_histories (res_hist, n_in, flv, res_hist_clean, success)
  type(resonance_history_t), intent(in), dimension(:) :: res_hist
  integer, intent(in) :: n_in
  integer, intent(in), dimension(:) :: flv
  type(resonance_history_t), intent(out), dimension(:), allocatable :: res_hist_clean
  logical, intent(out) :: success
  integer :: i_hist
  type(resonance_history_t), dimension(:), allocatable :: res_hist_colored, res_hist_contracted

  call msg_debug (D_SUBTRACTION, "resonance_mapping_init")
  if (debug_active (D_SUBTRACTION)) then
     call msg_debug (D_SUBTRACTION, "Original resonances:")
     do i_hist = 1, size(res_hist)
        call res_hist(i_hist)%write ()
```

```
      end do
   end if

   call remove_uncolored_resonances ()
   call contract_resonances (res_hist_colored, res_hist_contracted)
   call remove_subresonances (res_hist_contracted, res_hist_clean)
   !!! Here, we are still not sure whether we actually would rather use
   !!! call remove_multiple_resonances (res_hist_contracted, res_hist_clean)
   if (debug_active (D_SUBTRACTION)) then
      call msg_debug (D_SUBTRACTION, "Resonances after removing uncolored and duplicates: ")
      do i_hist = 1, size (res_hist_clean)
         call res_hist_clean(i_hist)%write ()
      end do
   end if
   if (size (res_hist_clean) == 0) then
      call msg_warning ("No resonances found. Proceed in usual FKS mode.")
      success = .false.
   else
      success = .true.
   end if

contains
   subroutine remove_uncolored_resonances ()
     type(resonance_history_t), dimension(:), allocatable :: res_hist_tmp
     integer :: n_hist, nleg_out, n_removed
     integer :: i_res, i_hist
     n_hist = size (res_hist)
     nleg_out = size (flv) - n_in
     allocate (res_hist_tmp (n_hist))
     allocate (res_hist_colored (n_hist))
     do i_hist = 1, n_hist
        res_hist_tmp(i_hist) = res_hist(i_hist)
        call res_hist_tmp(i_hist)%add_offset (n_in)
        n_removed = 0
        do i_res = 1, res_hist_tmp(i_hist)%n_resonances
           associate (resonance => res_hist_tmp(i_hist)%resonances(i_res - n_removed))
              if (.not. any (is_colored (flv (resonance%contributors%c))) &
                 .or. size (resonance%contributors%c) == nleg_out) then
                    call res_hist_tmp(i_hist)%remove_resonance (i_res - n_removed)
                    n_removed = n_removed + 1
              end if
           end associate
        end do
        if (allocated (res_hist_tmp(i_hist)%resonances)) then
           if (any (res_hist_colored == res_hist_tmp(i_hist))) then
              cycle
           else
              do i_res = 1, res_hist_tmp(i_hist)%n_resonances
                 associate (resonance => res_hist_tmp(i_hist)%resonances(i_res))
                    call res_hist_colored(i_hist)%add_resonance (resonance)
                 end associate
              end do
           end if
        end if
```

```
  end do
end subroutine remove_uncolored_resonances

subroutine contract_resonances (res_history_in, res_history_out)
  type(resonance_history_t), intent(in), dimension(:) :: res_history_in
  type(resonance_history_t), intent(out), dimension(:), allocatable :: res_history_out
  logical, dimension(:), allocatable :: i_non_zero
  integer :: n_hist_non_zero, n_hist
  integer :: i_hist_new
  n_hist = size (res_history_in); n_hist_non_zero = 0
  allocate (i_non_zero (n_hist))
  i_non_zero = .false.
  do i_hist = 1, n_hist
     if (res_history_in(i_hist)%n_resonances /= 0) then
        n_hist_non_zero = n_hist_non_zero + 1
        i_non_zero(i_hist) = .true.
     end if
  end do
  allocate (res_history_out (n_hist_non_zero))
  i_hist_new = 1
  do i_hist = 1, n_hist
     if (i_non_zero (i_hist)) then
        res_history_out (i_hist_new) = res_history_in (i_hist)
        i_hist_new = i_hist_new + 1
     end if
  end do
end subroutine contract_resonances

subroutine remove_subresonances (res_history_in, res_history_out)
  type(resonance_history_t), intent(in), dimension(:) :: res_history_in
  type(resonance_history_t), intent(out), dimension(:), allocatable :: res_history_out
  logical, dimension(:), allocatable :: i_non_sub_res
  integer :: n_hist, n_hist_non_sub_res
  integer :: i_hist1, i_hist2
  logical :: is_not_subres
  n_hist = size (res_history_in); n_hist_non_sub_res = 0
  allocate (i_non_sub_res (n_hist)); i_non_sub_res = .false.
  do i_hist1 = 1, n_hist
     is_not_subres = .true.
     do i_hist2 = 1, n_hist
        if (i_hist1 == i_hist2) cycle
        is_not_subres = is_not_subres .and. &
           .not.(res_history_in(i_hist2) .contains. res_history_in(i_hist1))
     end do
     if (is_not_subres) then
        n_hist_non_sub_res = n_hist_non_sub_res + 1
        i_non_sub_res (i_hist1) = .true.
     end if
  end do

  allocate (res_history_out (n_hist_non_sub_res))
  i_hist2 = 1
  do i_hist1 = 1, n_hist
     if (i_non_sub_res (i_hist1)) then
```

```
            res_history_out (i_hist2) = res_history_in (i_hist1)
            i_hist2 = i_hist2 + 1
         end if
      end do
   end subroutine remove_subresonances

   subroutine remove_multiple_resonances (res_history_in, res_history_out)
      type(resonance_history_t), intent(in), dimension(:) :: res_history_in
      type(resonance_history_t), intent(out), dimension(:), allocatable :: res_history_out
      integer :: n_hist, n_hist_single
      logical, dimension(:), allocatable :: i_hist_single
      integer :: i_hist, j
      n_hist = size (res_history_in)
      n_hist_single = 0
      allocate (i_hist_single (n_hist)); i_hist_single = .false.
      do i_hist = 1, n_hist
         if (res_history_in(i_hist)%n_resonances == 1) then
            n_hist_single = n_hist_single + 1
            i_hist_single(i_hist) = .true.
         end if
      end do

      allocate (res_history_out (n_hist_single))
      j = 1
      do i_hist = 1, n_hist
         if (i_hist_single(i_hist)) then
            res_history_out(j) = res_history_in(i_hist)
            j = j + 1
         end if
      end do
   end subroutine remove_multiple_resonances
end subroutine clean_resonance_histories
```

⟨*phs fks: procedures*⟩+≡

```
subroutine get_filtered_resonance_histories (phs_config, n_in, flv_state, model, &
   excluded_resonances, resonance_histories_filtered, success)
   type(phs_fks_config_t), intent(inout) :: phs_config
   integer, intent(in) :: n_in
   integer, intent(in), dimension(:,:), allocatable :: flv_state
   type(model_t), intent(in) :: model
   type(string_t), intent(in), dimension(:), allocatable :: excluded_resonances
   type(resonance_history_t), intent(out), dimension(:), &
      allocatable :: resonance_histories_filtered
   logical, intent(out) :: success
   type(resonance_history_t), dimension(:), allocatable :: resonance_histories
   type(resonance_history_t), dimension(:), allocatable :: &
      resonance_histories_clean!, resonance_histories_filtered
   allocate (resonance_histories (size (phs_config%get_resonance_histories ())))
   resonance_histories = phs_config%get_resonance_histories ()
   call clean_resonance_histories (resonance_histories, &
      n_in, flv_state (:,1), resonance_histories_clean, success)
   if (success .and. allocated (excluded_resonances)) then
      call filter_particles_from_resonances (resonance_histories_clean, &
         excluded_resonances, model, resonance_histories_filtered)
```

```
  else
     allocate (resonance_histories_filtered (size (resonance_histories_clean)))
     resonance_histories_filtered = resonance_histories_clean
  end if
end subroutine get_filtered_resonance_histories
```

18.9.3 Unit tests

Test module for FKS phase space, followed by the corresponding implementation module.

⟨phs_fks_ut.f90⟩≡

```
⟨File header⟩

module phs_fks_ut
  use unit_tests
  use phs_fks_uti

⟨Standard module head⟩

⟨phs fks: public test⟩

contains

⟨phs fks: test driver⟩

end module phs_fks_ut
```

⟨phs_fks_uti.f90⟩≡

```
⟨File header⟩

module phs_fks_uti

⟨Use kinds⟩
  use format_utils, only: write_separator, pac_fmt
  use format_defs, only: FMT_15, FMT_19
  use numeric_utils, only: nearly_equal
  use constants, only: tiny_07, zero, one, two
  use lorentz

  use physics_defs, only: THR_POS_B, THR_POS_BBAR, THR_POS_WP, THR_POS_WM, THR_POS_GLUON
  use physics_defs, only: thr_leg

  use resonances, only: resonance_contributors_t
  use phs_fks

⟨Standard module head⟩

⟨phs fks: test declarations⟩

contains

⟨phs fks: tests⟩

end module phs_fks_uti
```

API: driver for the unit tests below.

⟨*phs fks: public test*⟩≡

```
  public :: phs_fks_generator_test
```

⟨*phs fks: test driver*⟩≡

```
  subroutine phs_fks_generator_test (u, results)
    integer, intent(in) :: u
```

```
    type(test_results_t), intent(inout) :: results
    call test(phs_fks_generator_1, "phs_fks_generator_1", &
         "Test the generation of FKS phase spaces", u, results)
    call test(phs_fks_generator_2, "phs_fks_generator_2", &
         "Test the generation of an ISR FKS phase space", u, results)
    call test(phs_fks_generator_3, "phs_fks_generator_3", &
         "Test the generation of a real phase space for decays", &
         u, results)
    call test(phs_fks_generator_4, "phs_fks_generator_4", &
         "Test the generation of an FSR phase space with "&
         &"conserved invariant resonance masses", u, results)
    call test(phs_fks_generator_5, "phs_fks_generator_5", &
         "Test on-shell projection of a Born phase space and the generation"&
         &" of a real phase-space from that", u, results)
    call test(phs_fks_generator_6, "phs_fks_generator_6", &
         "Test the generation of a real phase space for 1 -> 3 decays", &
         u, results)
  end subroutine phs_fks_generator_test
```

⟨*phs fks: test declarations*⟩≡

```
  public :: phs_fks_generator_1
```

⟨*phs fks: tests*⟩≡

```
  subroutine phs_fks_generator_1 (u)
    integer, intent(in) :: u
    type(phs_fks_generator_t) :: generator
    type(vector4_t), dimension(:), allocatable :: p_born
    type(phs_point_t) :: p_real
    integer :: emitter, i_phs
    real(default) :: x1, x2, x3
    real(default), parameter :: sqrts = 250.0_default
    type(phs_identifier_t), dimension(2) :: phs_identifiers
    write (u, "(A)") "* Test output: phs_fks_generator_1"
    write (u, "(A)") "* Purpose: Create massless fsr phase space"
    write (u, "(A)")

    allocate (p_born (4))
    p_born(1)%p(0) = 125.0_default
    p_born(1)%p(1:2) = 0.0_default
    p_born(1)%p(3) = 125.0_default
    p_born(2)%p(0) = 125.0_default
    p_born(2)%p(1:2) = 0.0_default
    p_born(2)%p(3) = -125.0_default
    p_born(3)%p(0) = 125.0_default
    p_born(3)%p(1) = -39.5618_default
    p_born(3)%p(2) = -20.0791_default
    p_born(3)%p(3) = -114.6957_default
    p_born(4)%p(0) = 125.0_default
    p_born(4)%p(1:3) = -p_born(3)%p(1:3)

    allocate (generator%isr_kinematics)
    generator%n_in = 2

    call generator%set_sqrts_hat (sqrts)
```

```
    write (u, "(A)") "* Use four-particle phase space containing: "
    call vector4_write_set (p_born, u, testflag = .true., ultra = .true.)
    write (u, "(A)") "************************"
    write (u, "(A)")

    x1 = 0.5_default; x2 = 0.25_default; x3 = 0.75_default
    write (u, "(A)" ) "* Use random numbers: "
    write (u, "(A,F3.2,1X,A,F3.2,1X,A,F3.2)") &
       "x1: ", x1, "x2: ", x2, "x3: ", x3

    allocate (generator%real_kinematics)
    call generator%real_kinematics%init (4, 2, 2, 1)

    allocate (generator%emitters (2))
    generator%emitters(1) = 3; generator%emitters(2) = 4
    allocate (generator%m2 (4))
    generator%m2 = zero
    allocate (generator%is_massive (4))
    generator%is_massive(1:2) = .false.
    generator%is_massive(3:4) = .true.
    phs_identifiers(1)%emitter = 3
    phs_identifiers(2)%emitter = 4
    call generator%compute_xi_ref_momenta (p_born)
    call generator%generate_radiation_variables ([x1,x2,x3], p_born, phs_identifiers)
    do i_phs = 1, 2
       emitter = phs_identifiers(i_phs)%emitter
       call generator%compute_xi_max (emitter, i_phs, p_born, &
            generator%real_kinematics%xi_max(i_phs))
    end do
    write (u, "(A)")  &
         "* With these, the following radiation variables have been produced:"
    associate (rad_var => generator%real_kinematics)
      write (u, "(A,F3.2)") "xi_tilde: ", rad_var%xi_tilde
      write (u, "(A,F3.2)") "y: " , rad_var%y(1)
      write (u, "(A,F3.2)") "phi: ", rad_var%phi
    end associate
    call write_separator (u)
    write (u, "(A)") "Produce real momenta: "
    i_phs = 1; emitter = phs_identifiers(i_phs)%emitter
    write (u, "(A,I1)") "emitter: ", emitter
    p_real = 5
    call generator%generate_fsr (emitter, i_phs, p_born, p_real)
    call vector4_write_set (p_real%p, u, testflag = .true., ultra = .true.)
    call write_separator (u)
    write (u, "(A)")
    write (u, "(A)") "* Test output end: phs_fks_generator_1"

  end subroutine phs_fks_generator_1
```

⟨*phs fks: test declarations*⟩+≡

```
  public :: phs_fks_generator_2
```

⟨*phs fks: tests*⟩+≡

```
subroutine phs_fks_generator_2 (u)
  integer, intent(in) :: u
  type(phs_fks_generator_t) :: generator
  type(vector4_t), dimension(:), allocatable :: p_born
  type(phs_point_t) :: p_real
  integer :: emitter, i_phs
  real(default) :: x1, x2, x3
  real(default), parameter :: sqrts_hadronic = 250.0_default
  type(phs_identifier_t), dimension(2) :: phs_identifiers
  write (u, "(A)") "* Test output: phs_fks_generator_2"
  write (u, "(A)") "* Purpose: Create massless ISR phase space"
  write (u, "(A)")


  allocate (p_born (4))
  p_born(1)%p(0) = 114.661_default
  p_born(1)%p(1:2) = 0.0_default
  p_born(1)%p(3) = 114.661_default
  p_born(2)%p(0) = 121.784_default
  p_born(2)%p(1:2) = 0.0_default
  p_born(2)%p(3) = -121.784_default
  p_born(3)%p(0) = 115.148_default
  p_born(3)%p(1) = -46.250_default
  p_born(3)%p(2) = -37.711_default
  p_born(3)%p(3) = 98.478_default
  p_born(4)%p(0) = 121.296_default
  p_born(4)%p(1:2) = -p_born(3)%p(1:2)
  p_born(4)%p(3) = -105.601_default

  phs_identifiers(1)%emitter = 1
  phs_identifiers(2)%emitter = 2

  allocate (generator%emitters (2))
  allocate (generator%isr_kinematics)
  generator%n_in = 2
  generator%emitters(1) = 1; generator%emitters(2) = 2
  generator%sqrts = sqrts_hadronic
  generator%isr_kinematics%beam_energy = sqrts_hadronic / two
  call generator%set_sqrts_hat (sqrts_hadronic)
  call generator%set_isr_kinematics (p_born)

  write (u, "(A)") "* Use four-particle phase space containing: "
  call vector4_write_set (p_born, u, testflag = .true., ultra = .true.)
  write (u, "(A)") "************************"
  write (u, "(A)")

  x1=0.5_default; x2=0.25_default; x3=0.65_default
  write (u, "(A)" ) "* Use random numbers: "
  write (u, "(A,F3.2,1X,A,F3.2,1X,A,F3.2)") &
     "x1: ", x1, "x2: ", x2, "x3: ", x3

  allocate (generator%real_kinematics)
  call generator%real_kinematics%init (4, 2, 2, 1)
  call generator%real_kinematics%p_born_lab%set_momenta (1, p_born)
```

```
    allocate (generator%m2 (2))
    generator%m2(1) = 0._default; generator%m2(2) = 0._default
    allocate (generator%is_massive (4))
    generator%is_massive = .false.
    call generator%generate_radiation_variables ([x1,x2,x3], p_born, phs_identifiers)
    call generator%compute_xi_ref_momenta (p_born)
    do i_phs = 1, 2
       emitter = phs_identifiers(i_phs)%emitter
       call generator%compute_xi_max (emitter, i_phs, p_born, &
            generator%real_kinematics%xi_max(i_phs))
    end do
    write (u, "(A)")  &
         "* With these, the following radiation variables have been produced:"
    associate (rad_var => generator%real_kinematics)
      write (u, "(A,F3.2)") "xi_tilde: ", rad_var%xi_tilde
      write (u, "(A,F3.2)") "y: " , rad_var%y(1)
      write (u, "(A,F3.2)") "phi: ", rad_var%phi
    end associate
    write (u, "(A)") "Initial-state momentum fractions: "
    associate (xb => generator%isr_kinematics%x)
       write (u, "(A,F3.2)") "x_born_plus: ", xb(1)
       write (u, "(A,F3.2)") "x_born_minus: ", xb(2)
    end associate
    call write_separator (u)
    write (u, "(A)") "Produce real momenta: "
    i_phs = 1; emitter = phs_identifiers(i_phs)%emitter
    write (u, "(A,I1)") "emitter: ", emitter
    p_real = 5
    call generator%generate_isr (i_phs, p_born, p_real)
    call vector4_write_set (p_real%p, u, testflag = .true., ultra = .true.)
    call write_separator (u)
    write (u, "(A)")
    write (u, "(A)") "* Test output end: phs_fks_generator_2"

  end subroutine phs_fks_generator_2
```

⟨*phs fks: test declarations*⟩+≡

```
  public :: phs_fks_generator_3
```

⟨*phs fks: tests*⟩+≡

```
  subroutine phs_fks_generator_3 (u)
    integer, intent(in) :: u
    type(phs_fks_generator_t) :: generator
    type(vector4_t), dimension(:), allocatable :: p_born
    type(phs_point_t) :: p_real
    real(default) :: x1, x2, x3
    real(default) :: mB, mW, mT
    integer :: i, emitter, i_phs
    type(phs_identifier_t), dimension(2) :: phs_identifiers

    write (u, "(A)") "* Test output: phs_fks_generator_3"
    write (u, "(A)") "* Puropse: Create real phase space for particle decays"
    write (u, "(A)")
```

```
allocate (p_born(3))
p_born(1)%p(0) = 172._default
p_born(1)%p(1) = 0._default
p_born(1)%p(2) = 0._default
p_born(1)%p(3) = 0._default
p_born(2)%p(0) = 104.72866679_default
p_born(2)%p(1) = 45.028053213_default
p_born(2)%p(2) = 29.450337581_default
p_born(2)%p(3) = -5.910229156_default
p_born(3)%p(0) = 67.271333209_default
p_born(3)%p(1:3) = -p_born(2)%p(1:3)

generator%n_in = 1

mB = 4.2_default
mW = 80.376_default
mT = 172._default

generator%sqrts = mT

write (u, "(A)") "* Use three-particle phase space containing: "
call vector4_write_set (p_born, u, testflag = .true., ultra = .true.)
write (u, "(A)") "**********************"
write (u, "(A)")

x1 = 0.5_default; x2 = 0.25_default; x3 = 0.6_default
write (u, "(A)") "* Use random numbers: "
write (u, "(A,F3.2,1X,A,F3.2,A,1X,F3.2)") &
   "x1: ", x1, "x2: ", x2, "x3: ", x3

allocate (generator%real_kinematics)
call generator%real_kinematics%init (3, 2, 2, 1)
call generator%real_kinematics%p_born_lab%set_momenta (1, p_born)

allocate (generator%emitters(2))
generator%emitters(1) = 1
generator%emitters(2) = 3
allocate (generator%m2 (3), generator%is_massive(3))
generator%m2(1) = mT**2
generator%m2(2) = mW**2
generator%m2(3) = mB**2
generator%is_massive = .true.
phs_identifiers(1)%emitter = 1
phs_identifiers(2)%emitter = 3

call generator%generate_radiation_variables ([x1,x2,x3], p_born, phs_identifiers)
call generator%compute_xi_ref_momenta (p_born)
do i_phs = 1, 2
   emitter = phs_identifiers(i_phs)%emitter
   call generator%compute_xi_max (emitter, i_phs, p_born, &
        generator%real_kinematics%xi_max(i_phs))
end do
```

```
    write (u, "(A)") &
       "* With these, the following radiation variables have been produced: "
    associate (rad_var => generator%real_kinematics)
      write (u, "(A,F4.2)") "xi_tilde: ", rad_var%xi_tilde
      do i = 1, 2
         write (u, "(A,I1,A,F5.2)") "i: ", i, "y: " , rad_var%y(i)
      end do
      write (u, "(A,F4.2)") "phi: ", rad_var%phi
    end associate

    call write_separator (u)
    write (u, "(A)") "Produce real momenta via initial-state emission: "
    i_phs = 1; emitter = phs_identifiers(i_phs)%emitter
    write (u, "(A,I1)") "emitter: ", emitter
    p_real = 4
    call generator%generate_isr_decay (i_phs, p_born, p_real)
    call pacify (p_real%p, 1E-6_default)
    call vector4_write_set (p_real%p, u, testflag = .true., ultra = .true.)
    call write_separator(u)
    write (u, "(A)") "Produce real momenta via final-state emisson: "
    i_phs = 2; emitter = phs_identifiers(i_phs)%emitter
    write (u, "(A,I1)") "emitter: ", emitter
    call generator%generate_fsr (emitter, i_phs, p_born, p_real)
    call pacify (p_real%p, 1E-6_default)
    call vector4_write_set (p_real%p, u, testflag = .true., ultra = .true.)
    write (u, "(A)")
    write (u, "(A)") "* Test output end: phs_fks_generator_3"

  end subroutine phs_fks_generator_3
```

⟨*phs fks: test declarations*⟩+≡

```
  public :: phs_fks_generator_4
```

⟨*phs fks: tests*⟩+≡

```
  subroutine phs_fks_generator_4 (u)
    integer, intent(in) :: u
    type(phs_fks_generator_t) :: generator
    type(vector4_t), dimension(:), allocatable :: p_born
    type(phs_point_t) :: p_real
    integer, dimension(:), allocatable :: emitters
    integer, dimension(:,:), allocatable :: resonance_lists
    type(resonance_contributors_t), dimension(2) :: alr_contributors
    real(default) :: x1, x2, x3
    real(default), parameter :: sqrts = 250.0_default
    integer, parameter :: nlegborn = 6
    integer :: i_phs, i_con, emitter
    real(default) :: m_inv_born, m_inv_real
    character(len=7) :: fmt
    type(phs_identifier_t), dimension(2) :: phs_identifiers

    call pac_fmt (fmt, FMT_19, FMT_15, .true.)

    write (u, "(A)") "* Test output: phs_fks_generator_4"
    write (u, "(A)") "* Purpose: Create FSR phase space with fixed resonances"
```

```
write (u, "(A)")

allocate (p_born (nlegborn))
p_born(1)%p(0) = 250._default
p_born(1)%p(1) = 0._default
p_born(1)%p(2) = 0._default
p_born(1)%p(3) = 250._default
p_born(2)%p(0) = 250._default
p_born(2)%p(1) = 0._default
p_born(2)%p(2) = 0._default
p_born(2)%p(3) = -250._default
p_born(3)%p(0) = 145.91184486_default
p_born(3)%p(1) = 50.39727589_default
p_born(3)%p(2) = 86.74156041_default
p_born(3)%p(3) = -69.03608748_default
p_born(4)%p(0) = 208.1064784_default
p_born(4)%p(1) = -44.07610020_default
p_born(4)%p(2) = -186.34264578_default
p_born(4)%p(3) = 13.48038407_default
p_born(5)%p(0) = 26.25614471_default
p_born(5)%p(1) = -25.12258068_default
p_born(5)%p(2) = -1.09540228_default
p_born(5)%p(3) = -6.27703505_default
p_born(6)%p(0) = 119.72553196_default
p_born(6)%p(1) = 18.80140499_default
p_born(6)%p(2) = 100.69648766_default
p_born(6)%p(3) = 61.83273846_default

allocate (generator%isr_kinematics)
generator%n_in = 2

call generator%set_sqrts_hat (sqrts)

write (u, "(A)") "* Test process: e+ e- -> W+ W- b b~"
write (u, "(A)") "* Resonance pairs: (3,5) and (4,6)"
write (u, "(A)") "* Use four-particle phase space containing: "
call vector4_write_set (p_born, u, testflag = .true., ultra = .true.)
write (u, "(A)") "******************************"
write (u, "(A)")

x1 = 0.5_default; x2 = 0.25_default; x3 = 0.75_default
write (u, "(A)") "* Use random numbers: "
write (u, "(A,F3.2,1X,A,F3.2,1X,A,F3.2)") &
   "x1: ", x1, "x2: ", x2, "x3: ", x3

allocate (generator%real_kinematics)
call generator%real_kinematics%init (nlegborn, 2, 2, 2)

allocate (generator%emitters (2))
generator%emitters(1) = 5; generator%emitters(2) = 6
allocate (generator%m2 (nlegborn))
generator%m2 = p_born**2
allocate (generator%is_massive (nlegborn))
generator%is_massive (1:2) = .false.
```

```
  generator%is_massive (3:6) = .true.

  phs_identifiers(1)%emitter = 5
  phs_identifiers(2)%emitter = 6
  do i_phs = 1, 2
     allocate (phs_identifiers(i_phs)%contributors (2))
  end do
  allocate (resonance_lists (2, 2))
  resonance_lists (1,:) = [3,5]
  resonance_lists (2,:) = [4,6]
  !!! Here is obviously some redundance. Surely we can improve on this.
  do i_phs = 1, 2
     phs_identifiers(i_phs)%contributors = resonance_lists(i_phs,:)
  end do
  do i_con = 1, 2
     allocate (alr_contributors(i_con)%c (size (resonance_lists(i_con,:))))
     alr_contributors(i_con)%c = resonance_lists(i_con,:)
  end do
  call generator%generate_radiation_variables &
     ([x1, x2, x3], p_born, phs_identifiers)

  p_real = nlegborn + 1
  call generator%compute_xi_ref_momenta (p_born, alr_contributors)
  !!! Keep the distinction between i_phs and i_con because in general,
  !!! they are not the same.
  do i_phs = 1, 2
     i_con = i_phs
     emitter = phs_identifiers(i_phs)%emitter
     write (u, "(A,I1,1X,A,I1,A,I1,A)") &
        "* Generate FSR phase space for emitter ", emitter, &
        "and resonance pair (",  resonance_lists (i_con, 1), ",", &
        resonance_lists (i_con, 2), ")"
     call generator%compute_xi_max (emitter, i_phs, p_born, &
          generator%real_kinematics%xi_max(i_phs), i_con = i_con)
     call generator%generate_fsr (emitter, i_phs, i_con, p_born, p_real)
     call vector4_write_set (p_real%p, u, testflag = .true., ultra = .true.)
     call write_separator(u)
     write (u, "(A)") "* Check if resonance masses are conserved: "
     m_inv_born = compute_resonance_mass (p_born, resonance_lists (i_con,:))
     m_inv_real = compute_resonance_mass (p_real%p, resonance_lists (i_con,:), 7)
     write (u, "(A,1X, " // fmt // ")") "m_inv_born = ", m_inv_born
     write (u, "(A,1X, " // fmt // ")") "m_inv_real = ", m_inv_real
     if (abs (m_inv_born - m_inv_real) < tiny_07) then
        write (u, "(A)") " Success! "
     else
        write (u, "(A)") " Failure! "
     end if
     call write_separator(u)
     call write_separator(u)
  end do
  call p_real%final ()
  write (u, "(A)")
  write (u, "(A)") "* Test output end: phs_fks_generator_4"
end subroutine phs_fks_generator_4
```

⟨*phs fks: test declarations*⟩+≡

```
public :: phs_fks_generator_5
```

⟨*phs fks: tests*⟩+≡

```
subroutine phs_fks_generator_5 (u)
  use ttv_formfactors, only: init_parameters
  integer, intent(in) :: u
  type(phs_fks_generator_t) :: generator
  type(vector4_t), dimension(:), allocatable :: p_born
  type(vector4_t), dimension(:), allocatable :: p_born_onshell
  type(phs_point_t) :: p_real
  real(default) :: x1, x2, x3
  real(default) :: mB, mW, mtop, mcheck
  integer :: i, emitter, i_phs
  type(phs_identifier_t), dimension(2) :: phs_identifiers
  type(lorentz_transformation_t) :: L_to_cms
  real(default), parameter :: sqrts = 360._default
  real(default), parameter :: momentum_tolerance = 1E-10_default
  real(default) :: mpole, gam_out

  write (u, "(A)") "* Test output: phs_fks_generator_5"
  write (u, "(A)") "* Puropse: Perform threshold on-shell projection of "
  write (u, "(A)") "*          Born momenta and create a real phase-space "
  write (u, "(A)") "*          point from those. "
  write (u, "(A)")

  allocate (p_born(6), p_born_onshell(6))
  p_born(1)%p(0) = sqrts / two
  p_born(1)%p(1:2) = zero
  p_born(1)%p(3) = sqrts / two
  p_born(2)%p(0) = sqrts / two
  p_born(2)%p(1:2) = zero
  p_born(2)%p(3) = -sqrts / two
  p_born(3)%p(0) = 117.1179139230_default
  p_born(3)%p(1) = 56.91215483880_default
  p_born(3)%p(2) = -40.02386013017_default
  p_born(3)%p(3) = -49.07634310496_default
  p_born(4)%p(0) = 98.91904548743_default
  p_born(4)%p(1) = 56.02241403836_default
  p_born(4)%p(2) = -8.302977504723_default
  p_born(4)%p(3) = -10.50293716131_default
  p_born(5)%p(0) = 62.25884689208_default
  p_born(5)%p(1) = -60.00786540278_default
  p_born(5)%p(2) = 4.753602375910_default
  p_born(5)%p(3) = 15.32916731546_default
  p_born(6)%p(0) = 81.70419369751_default
  p_born(6)%p(1) = -52.92670347439_default
  p_born(6)%p(2) = 43.57323525898_default
  p_born(6)%p(3) = 44.25011295081_default

  generator%n_in = 2

  mB = 4.2_default
```

```
mW = 80.376_default
mtop = 172._default

generator%sqrts = sqrts

!!! Dummy-initialization of the threshold model because generate_fsr_threshold
!!! uses m1s_to_mpole to determine if it is above or below threshold.
call init_parameters (mpole, gam_out, mtop, one, one / 1.5_default, 125._default, &
     0.47_default, 0.118_default, 91._default, 80._default, 4.2_default, &
     one, one, one, one, zero, zero, zero, zero, zero, zero, .false., zero)

write (u, "(A)") "* Use four-particle phase space containing: "
call vector4_write_set (p_born, u, testflag = .true., ultra = .true.)
call vector4_check_momentum_conservation &
     (p_born, 2, unit = u, abs_smallness = momentum_tolerance, verbose = .true.)
write (u, "(A)") "**********************"
write (u, "(A)")

allocate (generator%real_kinematics)
call generator%real_kinematics%init (7, 2, 2, 2)
call generator%real_kinematics%init_onshell (7, 2)
generator%real_kinematics%p_born_cms%phs_point(1)%p = p_born

write (u, "(A)") "Get boost projection system -> CMS: "
L_to_cms = get_boost_for_threshold_projection (p_born, sqrts, mtop)
call L_to_cms%write (u, testflag = .true., ultra = .true.)
write (u, "(A)") "**********************"
write (u, "(A)")

write (u, "(A)") "* Perform onshell-projection:"
call generator%real_kinematics%apply_threshold_projection_born (mtop, L_to_cms)
call generator%real_kinematics%p_born_onshell%write (1, unit = u, testflag = .true., &
     ultra = .true.)
associate (p => generator%real_kinematics%p_born_onshell%phs_point(1)%p)
   p_born_onshell = p
   call check_phsp (p, 0)
end associate

allocate (generator%emitters (2))
generator%emitters(1) = THR_POS_B; generator%emitters(2) = THR_POS_BBAR

allocate (generator%m2 (6), generator%is_massive(6))
generator%m2 = p_born**2
generator%is_massive (1:2) = .false.
generator%is_massive (3:6) = .true.

phs_identifiers(1)%emitter = THR_POS_B
phs_identifiers(2)%emitter = THR_POS_BBAR

x1 = 0.5_default; x2 = 0.25_default; x3 = 0.6_default
write (u, "(A)") "* Use random numbers: "
write (u, "(A,F3.2,1X,A,F3.2,A,1X,F3.2)") &
   "x1: ", x1, "x2: ", x2, "x3: ", x3
```

```
  call generator%generate_radiation_variables ([x1,x2,x3], p_born_onshell, phs_identifiers)
  do i_phs = 1, 2
     emitter = phs_identifiers(i_phs)%emitter
     call generator%compute_xi_ref_momenta_threshold (p_born_onshell)
     call generator%compute_xi_max (emitter, i_phs, p_born_onshell, &
          generator%real_kinematics%xi_max(i_phs), i_con = thr_leg(emitter))
  end do
  write (u, "(A)") &
     "* With these, the following radiation variables have been produced: "
  associate (rad_var => generator%real_kinematics)
    write (u, "(A,F4.2)") "xi_tilde: ", rad_var%xi_tilde
    write (u, "(A)") "xi_max: "
    write (u, "(2F5.2)") rad_var%xi_max(1), rad_var%xi_max(2)
    write (u, "(A)") "y: "
    write (u, "(2F5.2)") rad_var%y(1), rad_var%y(2)
    write (u, "(A,F4.2)") "phi: ", rad_var%phi
  end associate

  call write_separator (u)
  write (u, "(A)") "* Produce real momenta from on-shell phase space: "
  p_real = 7
  do i_phs = 1, 2
     emitter = phs_identifiers(i_phs)%emitter
     write (u, "(A,I1)") "emitter: ", emitter
     call generator%generate_fsr_threshold (emitter, i_phs, p_born_onshell, p_real)
     call check_phsp (p_real%p, emitter)
  end do

  call write_separator(u)
  write (u, "(A)")
  write (u, "(A)") "* Test output end: phs_fks_generator_5"

contains
  subroutine check_phsp (p, emitter)
     type(vector4_t), intent(inout), dimension(:) :: p
     integer, intent(in) :: emitter
     type(vector4_t) :: pp
     real(default) :: E_tot
     logical :: check
     write (u, "(A)") "* Check momentum conservation: "
     call vector4_check_momentum_conservation &
          (p, 2, unit = u, abs_smallness = momentum_tolerance, verbose = .true.)
     write (u, "(A)") "* Check invariant masses: "
     write (u, "(A)", advance = "no") "inv(W+, b, gl): "
     pp = p(THR_POS_WP) + p(THR_POS_B)
     if (emitter == THR_POS_B) pp = pp + p(THR_POS_GLUON)
     if (nearly_equal (pp**1, mtop)) then
        write (u, "(A)") "CHECK"
     else
        write (u, "(A,F7.3)") "FAIL: ", pp**1
     end if
     write (u, "(A)", advance = "no") "inv(W-, bbar): "
     pp = p(THR_POS_WM) + p(THR_POS_BBAR)
```

```
      if (emitter == THR_POS_BBAR) pp = pp + p(THR_POS_GLUON)
      if (nearly_equal (pp**1, mtop)) then
         write (u, "(A)") "CHECK"
      else
         write (u, "(A,F7.3)") "FAIL: ", pp**1
      end if
      write (u, "(A)") "* Sum of energies equal to sqrts?"
      E_tot = sum(p(1:2)%p(0)); check = nearly_equal (E_tot, sqrts)
      write (u, "(A,L1)") "Initial state: ", check
      if (.not. check) write (u, "(A,F7.3)") "E_tot: ", E_tot
      if (emitter > 0) then
         E_tot = sum(p(3:7)%p(0))
      else
         E_tot = sum(p(3:6)%p(0))
      end if
      check = nearly_equal (E_tot, sqrts)
      write (u, "(A,L1)") "Final state  : ", check
      if (.not. check) write (u, "(A,F7.3)") "E_tot: ", E_tot
      call pacify (p, 1E-6_default)
      call vector4_write_set (p, u, testflag = .true., ultra = .true.)

    end subroutine check_phsp
  end subroutine phs_fks_generator_5
```

⟨*phs fks: test declarations*⟩+≡

```
  public :: phs_fks_generator_6
```

⟨*phs fks: tests*⟩+≡

```
  subroutine phs_fks_generator_6 (u)
    integer, intent(in) :: u
    type(phs_fks_generator_t) :: generator
    type(vector4_t), dimension(:), allocatable :: p_born
    type(phs_point_t) :: p_real
    real(default) :: x1, x2, x3
    real(default) :: mB, mW, mT
    integer :: i, emitter, i_phs
    type(phs_identifier_t), dimension(2) :: phs_identifiers

    write (u, "(A)") "* Test output: phs_fks_generator_6"
    write (u, "(A)") "* Puropse: Create real phase space for particle decays"
    write (u, "(A)")

    allocate (p_born(4))
    p_born(1)%p(0) = 173.1_default
    p_born(1)%p(1) = zero
    p_born(1)%p(2) = zero
    p_born(1)%p(3) = zero
    p_born(2)%p(0) = 68.17074462929_default
    p_born(2)%p(1) = -37.32578717617_default
    p_born(2)%p(2) = 30.99675959336_default
    p_born(2)%p(3) = -47.70321718398_default
    p_born(3)%p(0) = 65.26639312326_default
    p_born(3)%p(1) = -1.362927648502_default
    p_born(3)%p(2) = -33.25327150840_default
```

```
p_born(3)%p(3) = 56.14324922494_default
p_born(4)%p(0) = 39.66286224745_default
p_born(4)%p(1) = 38.68871482467_default
p_born(4)%p(2) = 2.256511915049_default
p_born(4)%p(3) = -8.440032040958_default

generator%n_in = 1

mB = 4.2_default
mW = 80.376_default
mT = 173.1_default

generator%sqrts = mT

write (u, "(A)") "* Use four-particle phase space containing: "
call vector4_write_set (p_born, u, testflag = .true., ultra = .true.)
write (u, "(A)") "**********************"
write (u, "(A)")

x1=0.5_default; x2=0.25_default; x3=0.6_default
write (u, "(A)") "* Use random numbers: "
write (u, "(A,F3.2,1X,A,F3.2,A,1X,F3.2)") &
   "x1: ", x1, "x2: ", x2, "x3: ", x3

allocate (generator%real_kinematics)
call generator%real_kinematics%init (3, 2, 2, 1)
call generator%real_kinematics%p_born_lab%set_momenta (1, p_born)

allocate (generator%emitters(2))
generator%emitters(1) = 1
generator%emitters(2) = 2
allocate (generator%m2 (4), generator%is_massive(4))
generator%m2(1) = mT**2
generator%m2(1) = mB**2
generator%m2(3) = zero
generator%m2(4) = zero
generator%is_massive(1:2) = .true.
generator%is_massive(3:4) = .false.
phs_identifiers(1)%emitter = 1
phs_identifiers(2)%emitter = 2

call generator%generate_radiation_variables ([x1,x2,x3], p_born, phs_identifiers)
call generator%compute_xi_ref_momenta (p_born)
do i_phs = 1, 2
   emitter = phs_identifiers(i_phs)%emitter
   call generator%compute_xi_max (emitter, i_phs, p_born, &
        generator%real_kinematics%xi_max(i_phs))
end do

write (u, "(A)") &
   "* With these, the following radiation variables have been produced: "
associate (rad_var => generator%real_kinematics)
  write (u, "(A,F4.2)") "xi_tilde: ", rad_var%xi_tilde
  do i = 1, 2
```

```
      write (u, "(A,I1,A,F5.2)") "i: ", i, "y: " , rad_var%y(i)
    end do
    write (u, "(A,F4.2)") "phi: ", rad_var%phi
  end associate

  call write_separator (u)
  write (u, "(A)") "Produce real momenta via initial-state emission: "
  i_phs = 1; emitter = phs_identifiers(i_phs)%emitter
  write (u, "(A,I1)") "emitter: ", emitter
  p_real = 5
  call generator%generate_isr_decay (i_phs, p_born, p_real)
  call pacify (p_real%p, 1E-6_default)
  call vector4_write_set (p_real%p, u, testflag = .true., ultra = .true.)
  call write_separator(u)
  write (u, "(A)") "Produce real momenta via final-state emisson: "
  i_phs = 2; emitter = phs_identifiers(i_phs)%emitter
  write (u, "(A,I1)") "emitter: ", emitter
  call generator%generate_fsr (emitter, i_phs, p_born, p_real)
  call pacify (p_real%p, 1E-6_default)
  call vector4_write_set (p_real%p, u, testflag = .true., ultra = .true.)
  write (u, "(A)")
  write (u, "(A)") "* Test output end: phs_fks_generator_6"

end subroutine phs_fks_generator_6
```

18.10 Dispatch

⟨dispatch_phase_space.f90⟩≡

```
⟨File header⟩

module dispatch_phase_space

⟨Use kinds⟩
⟨Use strings⟩
  use io_units, only: free_unit
  use variables, only: var_list_t
  use os_interface, only: os_data_t
  use diagnostics

  use sf_mappings, only: sf_channel_t
  use beam_structures, only: beam_structure_t
  use dispatch_beams, only: sf_prop_t, strfun_mode

  use mappings
  use phs_forests, only: phs_parameters_t
  use phs_base
  use phs_single
  use phs_wood
  use phs_fks

⟨Standard module head⟩
```

⟨*Dispatch phs: public*⟩

```
contains
```

⟨*Dispatch phs: procedures*⟩

```
end module dispatch_phase_space
```

Allocate a phase-space object according to the variable `$phs_method`.

⟨*Dispatch phs: public*⟩≡

```
public :: dispatch_phs
```

⟨*Dispatch phs: procedures*⟩≡

```
subroutine dispatch_phs (phs, var_list, os_data, process_id, &
      mapping_defaults, phs_par, phs_method_in)
  class(phs_config_t), allocatable, intent(inout) :: phs
  type(var_list_t), intent(in) :: var_list
  type(os_data_t), intent(in) :: os_data
  type(string_t), intent(in) :: process_id
  type(mapping_defaults_t), intent(in), optional :: mapping_defaults
  type(phs_parameters_t), intent(in), optional :: phs_par
  type(string_t), intent(in), optional :: phs_method_in
  type(string_t) :: phs_method, phs_file, run_id
  logical :: use_equivalences, vis_channels, fatal_beam_decay
  integer :: u_phs
  logical :: exist
  if (present (phs_method_in)) then
     phs_method = phs_method_in
  else
     phs_method = &
          var_list%get_sval (var_str ("$phs_method"))
  end if
  phs_file = &
       var_list%get_sval (var_str ("$phs_file"))
  use_equivalences = &
       var_list%get_lval (var_str ("?use_vamp_equivalences"))
  vis_channels = &
       var_list%get_lval (var_str ("?vis_channels"))
  fatal_beam_decay = &
       var_list%get_lval (var_str ("?fatal_beam_decay"))
  run_id = &
       var_list%get_sval (var_str ("$run_id"))
  select case (char (phs_method))
  case ("single")
     allocate (phs_single_config_t :: phs)
     if (vis_channels) then
        call msg_warning ("Visualizing phase space channels not " // &
             "available for method 'single'.")
     end if
  case ("fks")
    allocate (phs_fks_config_t :: phs)
  case ("wood", "default")
     call dispatch_wood ()
  case default
     call msg_fatal ("Phase space: parameterization method '" &
```

```
          // char (phs_method) // "' not implemented")
    end select
  contains
  ⟨Dispatch phs: dispatch phs: procedures⟩
  end subroutine dispatch_phs
```

⟨*Dispatch phs: dispatch phs: procedures*⟩≡

```
  subroutine dispatch_wood ()
    allocate (phs_wood_config_t :: phs)
    select type (phs)
    type is (phs_wood_config_t)
       if (phs_file /= "") then
          inquire (file = char (phs_file), exist = exist)
          if (exist) then
             call msg_message ("Phase space: reading configuration from '" &
                  // char (phs_file) // "'")
             u_phs = free_unit ()
             open (u_phs, file = char (phs_file), &
                  action = "read", status = "old")
             call phs%set_input (u_phs)
          else
             call msg_fatal ("Phase space: configuration file '" &
                  // char (phs_file) // "' not found")
          end if
       end if
       if (present (phs_par)) &
            call phs%set_parameters (phs_par)
       if (use_equivalences) &
            call phs%enable_equivalences ()
       if (present (mapping_defaults)) &
            call phs%set_mapping_defaults (mapping_defaults)
       phs%vis_channels = vis_channels
       phs%fatal_beam_decay = fatal_beam_decay
       phs%os_data = os_data
       phs%run_id = run_id
    end select
  end subroutine dispatch_wood
```

Configure channel mappings, using some conditions from the phase space configuration. If there are no structure functions, we enable a default setup with a single (dummy) structure-function channel. Otherwise, we look at the channel collection that we got from the phase-space configuration step. Each entry should be translated into an independent structure-function channel, where typically there is one default entry, which could be mapped using a standard s-channel mapping if the structure function setup recommends this, and other entries with s-channel resonances. The latter need to be translated into global mappings from the structure-function chain.

⟨*Dispatch phs: public*⟩+≡

```
  public :: dispatch_sf_channels
```

⟨*Dispatch phs: procedures*⟩+≡

```
  subroutine dispatch_sf_channels (sf_channel, sf_string, sf_prop, coll, &
         var_list, sqrts, beam_structure)
```

```
type(sf_channel_t), dimension(:), allocatable, intent(out) :: sf_channel
type(string_t), intent(out) :: sf_string
type(sf_prop_t), intent(in) :: sf_prop
type(phs_channel_collection_t), intent(in) :: coll
type(var_list_t), intent(in) :: var_list
real(default), intent(in) :: sqrts
type(beam_structure_t), intent(in) :: beam_structure
type(beam_structure_t) :: beam_structure_tmp
class(channel_prop_t), allocatable :: prop
integer :: n_strfun, n_sf_channel, i
logical :: sf_allow_s_mapping, circe1_map, circe1_generate
logical :: s_mapping_enable, endpoint_mapping, power_mapping
logical :: single_parameter
integer, dimension(:), allocatable :: s_mapping, single_mapping
real(default) :: s_mapping_power
real(default) :: circe1_mapping_slope, endpoint_mapping_slope
real(default) :: power_mapping_eps
beam_structure_tmp = beam_structure
call beam_structure_tmp%write ()
call beam_structure_tmp%expand (strfun_mode)
n_strfun = beam_structure_tmp%get_n_record ()
sf_string = beam_structure_tmp%to_string (sf_only = .true.)
sf_allow_s_mapping = &
     var_list%get_lval (var_str ("?sf_allow_s_mapping"))
circe1_generate = &
     var_list%get_lval (var_str ("?circe1_generate"))
circe1_map = &
     var_list%get_lval (var_str ("?circe1_map"))
circe1_mapping_slope = &
     var_list%get_rval (var_str ("circe1_mapping_slope"))
s_mapping_enable = .false.
s_mapping_power = 1
endpoint_mapping = .false.
endpoint_mapping_slope = 1
power_mapping = .false.
single_parameter = .false.
select case (char (sf_string))
case ("", "[any particles]")
case ("pdf_builtin, none", &
     "pdf_builtin_photon, none", &
     "none, pdf_builtin", &
     "none, pdf_builtin_photon", &
     "lhapdf, none", &
     "lhapdf_photon, none", &
     "none, lhapdf", &
     "none, lhapdf_photon")
     single_parameter = .true.
case ("pdf_builtin, none => none, pdf_builtin", &
      "pdf_builtin, none => none, pdf_builtin_photon", &
      "pdf_builtin_photon, none => none, pdf_builtin", &
      "pdf_builtin_photon, none => none, pdf_builtin_photon", &
      "lhapdf, none => none, lhapdf", &
      "lhapdf, none => none, lhapdf_photon", &
      "lhapdf_photon, none => none, lhapdf", &
```

```
      "lhapdf_photon, none => none, lhapdf_photon")
   allocate (s_mapping (2), source = [1, 2])
   s_mapping_enable = .true.
   s_mapping_power = 2
case ("pdf_builtin, none => none, pdf_builtin => epa, none => none, epa", &
      "pdf_builtin, none => none, pdf_builtin => ewa, none => none, ewa", &
      "pdf_builtin, none => none, pdf_builtin => ewa, none => none, epa", &
      "pdf_builtin, none => none, pdf_builtin => epa, none => none, ewa")
   allocate (s_mapping (2), source = [1, 2])
   s_mapping_enable = .true.
   s_mapping_power = 2
case ("isr, none", &
     "none, isr")
   allocate (single_mapping (1), source = [1])
   single_parameter = .true.
case ("isr, none => none, isr")
   allocate (s_mapping (2), source = [1, 2])
   power_mapping = .true.
   power_mapping_eps = minval (sf_prop%isr_eps)
case ("isr, none => none, isr => epa, none => none, epa", &
      "isr, none => none, isr => ewa, none => none, ewa", &
      "isr, none => none, isr => ewa, none => none, epa", &
      "isr, none => none, isr => epa, none => none, ewa")
   allocate (s_mapping (2), source = [1, 2])
   power_mapping = .true.
   power_mapping_eps = minval (sf_prop%isr_eps)
case ("circe1 => isr, none => none, isr => epa, none => none, epa", &
      "circe1 => isr, none => none, isr => ewa, none => none, ewa", &
      "circe1 => isr, none => none, isr => ewa, none => none, epa", &
      "circe1 => isr, none => none, isr => epa, none => none, ewa")
   if (circe1_generate) then
      allocate (s_mapping (2), source = [2, 3])
   else
      allocate (s_mapping (3), source = [1, 2, 3])
      endpoint_mapping = .true.
      endpoint_mapping_slope = circe1_mapping_slope
   end if
   power_mapping = .true.
   power_mapping_eps = minval (sf_prop%isr_eps)
case ("pdf_builtin, none => none, isr", &
     "pdf_builtin_photon, none => none, isr", &
     "lhapdf, none => none, isr", &
     "lhapdf_photon, none => none, isr")
   allocate (single_mapping (1), source = [2])
case ("isr, none => none, pdf_builtin", &
     "isr, none => none, pdf_builtin_photon", &
     "isr, none => none, lhapdf", &
     "isr, none => none, lhapdf_photon")
   allocate (single_mapping (1), source = [1])
case ("epa, none", &
      "none, epa")
   allocate (single_mapping (1), source = [1])
   single_parameter = .true.
case ("epa, none => none, epa")
```

```
   allocate (single_mapping (2), source = [1, 2])
case ("epa, none => none, isr", &
     "isr, none => none, epa", &
     "ewa, none => none, isr", &
     "isr, none => none, ewa")
   allocate (single_mapping (2), source = [1, 2])
case ("pdf_builtin, none => none, epa", &
     "pdf_builtin_photon, none => none, epa", &
     "lhapdf, none => none, epa", &
     "lhapdf_photon, none => none, epa")
   allocate (single_mapping (1), source = [2])
case ("pdf_builtin, none => none, ewa", &
     "pdf_builtin_photon, none => none, ewa", &
     "lhapdf, none => none, ewa", &
     "lhapdf_photon, none => none, ewa")
   allocate (single_mapping (1), source = [2])
case ("epa, none => none, pdf_builtin", &
     "epa, none => none, pdf_builtin_photon", &
     "epa, none => none, lhapdf", &
     "epa, none => none, lhapdf_photon")
   allocate (single_mapping (1), source = [1])
case ("ewa, none => none, pdf_builtin", &
     "ewa, none => none, pdf_builtin_photon", &
     "ewa, none => none, lhapdf", &
     "ewa, none => none, lhapdf_photon")
   allocate (single_mapping (1), source = [1])
case ("ewa, none", &
      "none, ewa")
   allocate (single_mapping (1), source = [1])
   single_parameter = .true.
case ("ewa, none => none, ewa")
   allocate (single_mapping (2), source = [1, 2])
case ("energy_scan, none => none, energy_scan")
   allocate (s_mapping (2), source = [1, 2])
case ("sf_test_1, none => none, sf_test_1")
   allocate (s_mapping (2), source = [1, 2])
case ("circe1")
   if (circe1_generate) then
      !!! no mapping
   else if (circe1_map) then
      allocate (s_mapping (1), source = [1])
      endpoint_mapping = .true.
      endpoint_mapping_slope = circe1_mapping_slope
   else
      allocate (s_mapping (1), source = [1])
      s_mapping_enable = .true.
   end if
case ("circe1 => isr, none => none, isr")
   if (circe1_generate) then
      allocate (s_mapping (2), source = [2, 3])
   else
      allocate (s_mapping (3), source = [1, 2, 3])
      endpoint_mapping = .true.
      endpoint_mapping_slope = circe1_mapping_slope
```

```
   end if
   power_mapping = .true.
   power_mapping_eps = minval (sf_prop%isr_eps)
case ("circe1 => isr, none", &
     "circe1 => none, isr")
   allocate (single_mapping (1), source = [2])
case ("circe1 => epa, none => none, epa")
   if (circe1_generate) then
      allocate (single_mapping (2), source = [2, 3])
   else
      call msg_fatal ("CIRCE/EPA: supported with ?circe1_generate=true &
           &only")
   end if
case ("circe1 => ewa, none => none, ewa")
   if (circe1_generate) then
      allocate (single_mapping (2), source = [2, 3])
   else
      call msg_fatal ("CIRCE/EWA: supported with ?circe1_generate=true &
           &only")
   end if
case ("circe1 => epa, none", &
     "circe1 => none, epa")
   if (circe1_generate) then
      allocate (single_mapping (1), source = [2])
   else
      call msg_fatal ("CIRCE/EPA: supported with ?circe1_generate=true &
           &only")
   end if
case ("circe1 => epa, none => none, isr", &
     "circe1 => isr, none => none, epa", &
     "circe1 => ewa, none => none, isr", &
     "circe1 => isr, none => none, ewa")
   if (circe1_generate) then
      allocate (single_mapping (2), source = [2, 3])
   else
      call msg_fatal ("CIRCE/EPA: supported with ?circe1_generate=true &
           &only")
   end if
case ("circe2", &
     "gaussian", &
     "beam_events")
   !!! no mapping
case ("circe2 => isr, none => none, isr", &
   "gaussian => isr, none => none, isr", &
   "beam_events => isr, none => none, isr")
   allocate (s_mapping (2), source = [2, 3])
   power_mapping = .true.
   power_mapping_eps = minval (sf_prop%isr_eps)
case ("circe2 => isr, none", &
     "circe2 => none, isr", &
     "gaussian => isr, none", &
     "gaussian => none, isr", &
     "beam_events => isr, none", &
     "beam_events => none, isr")
```

```
   allocate (single_mapping (1), source = [2])
case ("circe2 => epa, none => none, epa", &
     "gaussian => epa, none => none, epa", &
     "beam_events => epa, none => none, epa")
   allocate (single_mapping (2), source = [2, 3])
case ("circe2 => epa, none", &
     "circe2 => none, epa", &
     "circe2 => ewa, none", &
     "circe2 => none, ewa", &
     "gaussian => epa, none", &
     "gaussian => none, epa", &
     "gaussian => ewa, none", &
     "gaussian => none, ewa", &
     "beam_events => epa, none", &
     "beam_events => none, epa", &
     "beam_events => ewa, none", &
     "beam_events => none, ewa")
   allocate (single_mapping (1), source = [2])
case ("circe2 => epa, none => none, isr", &
     "circe2 => isr, none => none, epa", &
     "circe2 => ewa, none => none, isr", &
     "circe2 => isr, none => none, ewa", &
     "gaussian => epa, none => none, isr", &
     "gaussian => isr, none => none, epa", &
     "gaussian => ewa, none => none, isr", &
     "gaussian => isr, none => none, ewa", &
     "beam_events => epa, none => none, isr", &
     "beam_events => isr, none => none, epa", &
     "beam_events => ewa, none => none, isr", &
     "beam_events => isr, none => none, ewa")
   allocate (single_mapping (2), source = [2, 3])
case ("energy_scan")
case default
   call msg_fatal ("Beam structure: " &
        // char (sf_string) // " not supported")
end select
if (sf_allow_s_mapping .and. coll%n > 0) then
   n_sf_channel = coll%n
   allocate (sf_channel (n_sf_channel))
   do i = 1, n_sf_channel
      call sf_channel(i)%init (n_strfun)
      if (allocated (single_mapping)) then
         call sf_channel(i)%activate_mapping (single_mapping)
      end if
      if (allocated (prop))  deallocate (prop)
      call coll%get_entry (i, prop)
      if (allocated (prop)) then
         if (endpoint_mapping .and. power_mapping) then
            select type (prop)
            type is (resonance_t)
               call sf_channel(i)%set_eir_mapping (s_mapping, &
                    a = endpoint_mapping_slope, eps = power_mapping_eps, &
                    m = prop%mass / sqrts, w = prop%width / sqrts)
            type is (on_shell_t)
```

```
         call sf_channel(i)%set_eio_mapping (s_mapping, &
              a = endpoint_mapping_slope, eps = power_mapping_eps, &
              m = prop%mass / sqrts)
      end select
   else if (endpoint_mapping) then
      select type (prop)
      type is (resonance_t)
         call sf_channel(i)%set_epr_mapping (s_mapping, &
              a = endpoint_mapping_slope, &
              m = prop%mass / sqrts, w = prop%width / sqrts)
      type is (on_shell_t)
         call sf_channel(i)%set_epo_mapping (s_mapping, &
              a = endpoint_mapping_slope, &
              m = prop%mass / sqrts)
      end select
   else if (power_mapping) then
      select type (prop)
      type is (resonance_t)
         call sf_channel(i)%set_ipr_mapping (s_mapping, &
              eps = power_mapping_eps, &
              m = prop%mass / sqrts, w = prop%width / sqrts)
      type is (on_shell_t)
         call sf_channel(i)%set_ipo_mapping (s_mapping, &
              eps = power_mapping_eps, &
              m = prop%mass / sqrts)
      end select
   else if (allocated (s_mapping)) then
      select type (prop)
      type is (resonance_t)
         call sf_channel(i)%set_res_mapping (s_mapping, &
              m = prop%mass / sqrts, w = prop%width / sqrts, &
              single = single_parameter)
      type is (on_shell_t)
         call sf_channel(i)%set_os_mapping (s_mapping, &
              m = prop%mass / sqrts, &
              single = single_parameter)
      end select
   else if (allocated (single_mapping)) then
      select type (prop)
      type is (resonance_t)
         call sf_channel(i)%set_res_mapping (single_mapping, &
                m = prop%mass / sqrts, w = prop%width / sqrts, &
                single = single_parameter)
      type is (on_shell_t)
         call sf_channel(i)%set_os_mapping (single_mapping, &
              m = prop%mass / sqrts, &
              single = single_parameter)
      end select
   end if
else if (endpoint_mapping .and. power_mapping) then
   call sf_channel(i)%set_ei_mapping (s_mapping, &
        a = endpoint_mapping_slope, eps = power_mapping_eps)
else if (endpoint_mapping .and. .not. allocated (single_mapping)) then
   call sf_channel(i)%set_ep_mapping (s_mapping, &
```

```
                a = endpoint_mapping_slope)
         else if (power_mapping .and. .not. allocated (single_mapping)) then
            call sf_channel(i)%set_ip_mapping (s_mapping, &
                 eps = power_mapping_eps)
         else if (s_mapping_enable .and. .not. allocated (single_mapping)) then
            call sf_channel(i)%set_s_mapping (s_mapping, &
                 power = s_mapping_power)
         end if
      end do
   else if (sf_allow_s_mapping) then
      allocate (sf_channel (1))
      call sf_channel(1)%init (n_strfun)
      if (allocated (single_mapping)) then
         call sf_channel(1)%activate_mapping (single_mapping)
      else if (endpoint_mapping .and. power_mapping) then
         call sf_channel(i)%set_ei_mapping (s_mapping, &
              a = endpoint_mapping_slope, eps = power_mapping_eps)
      else if (endpoint_mapping) then
         call sf_channel(1)%set_ep_mapping (s_mapping, &
                a = endpoint_mapping_slope)
      else if (power_mapping) then
         call sf_channel(1)%set_ip_mapping (s_mapping, &
                eps = power_mapping_eps)
      else if (s_mapping_enable) then
         call sf_channel(1)%set_s_mapping (s_mapping, &
              power = s_mapping_power)
      end if
   else
      allocate (sf_channel (1))
      call sf_channel(1)%init (n_strfun)
      if (allocated (single_mapping)) then
         call sf_channel(1)%activate_mapping (single_mapping)
      end if
   end if
end subroutine dispatch_sf_channels
```

18.10.1 Unit tests

Test module, followed by the corresponding implementation module.

⟨dispatch_phs_ut.f90⟩≡

```
⟨File header⟩

module dispatch_phs_ut
  use unit_tests
  use dispatch_phs_uti

⟨Standard module head⟩

⟨Dispatch phs: public test⟩

contains
```

```
  ⟨Dispatch phs: test driver⟩

  end module dispatch_phs_ut
```

⟨dispatch_phs_uti.f90⟩≡

```
  ⟨File header⟩

  module dispatch_phs_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use variables
    use io_units, only: free_unit
    use os_interface, only: os_data_t, os_data_init
    use process_constants
    use model_data
    use models
    use phs_base
    use phs_forests
    use phs_wood
    use mappings
    use dispatch_phase_space

  ⟨Standard module head⟩

  ⟨Dispatch phs: test declarations⟩

  contains

  ⟨Dispatch phs: tests⟩

  end module dispatch_phs_uti
```

API: driver for the unit tests below.

⟨*Dispatch phs: public test*⟩≡

```
    public ::dispatch_phs_test
```

⟨*Dispatch phs: test driver*⟩≡

```
    subroutine dispatch_phs_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Dispatch phs: execute tests⟩
    end subroutine dispatch_phs_test
```

Select type: phase-space configuration object

⟨*Dispatch phs: execute tests*⟩≡

```
    call test (dispatch_phs_1, "dispatch_phs_1", &
         "phase-space configuration", &
         u, results)
```

⟨*Dispatch phs: test declarations*⟩≡

```
    public :: dispatch_phs_1
```

⟨*Dispatch phs: tests*⟩≡

```
subroutine dispatch_phs_1 (u)
  integer, intent(in) :: u
  type(var_list_t) :: var_list
  class(phs_config_t), allocatable :: phs
  type(phs_parameters_t) :: phs_par
  type(os_data_t) :: os_data
  type(mapping_defaults_t) :: mapping_defs

  write (u, "(A)")  "* Test output: dispatch_phs_1"
  write (u, "(A)")  "*   Purpose: select phase-space configuration method"
  write (u, "(A)")

  call var_list%init_defaults (0)

  write (u, "(A)")  "* Allocate PHS as phs_single_t"
  write (u, "(A)")

  call var_list%set_string (&
       var_str ("$phs_method"), &
       var_str ("single"), is_known = .true.)
  call dispatch_phs (phs, var_list, os_data, var_str ("dispatch_phs_1"))
  call phs%write (u)

  call phs%final ()
  deallocate (phs)

  write (u, "(A)")
  write (u, "(A)")  "* Allocate PHS as phs_wood_t"
  write (u, "(A)")

  call var_list%set_string (&
       var_str ("$phs_method"), &
       var_str ("wood"), is_known = .true.)
  call dispatch_phs (phs, var_list, os_data, var_str ("dispatch_phs_1"))
  call phs%write (u)

  call phs%final ()
  deallocate (phs)

  write (u, "(A)")
  write (u, "(A)")  "* Setting parameters for phs_wood_t"
  write (u, "(A)")

  phs_par%m_threshold_s = 123
  phs_par%m_threshold_t = 456
  phs_par%t_channel = 42
  phs_par%off_shell = 17
  phs_par%keep_nonresonant = .false.
  mapping_defs%energy_scale = 987
  mapping_defs%invariant_mass_scale = 654
  mapping_defs%momentum_transfer_scale = 321
  mapping_defs%step_mapping = .false.
  mapping_defs%step_mapping_exp = .false.
```

```
    mapping_defs%enable_s_mapping = .true.
    call dispatch_phs (phs, var_list, os_data, var_str ("dispatch_phs_1"), &
         mapping_defs, phs_par)
    call phs%write (u)

    call phs%final ()

    call var_list%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: dispatch_phs_1"

  end subroutine dispatch_phs_1
```

Phase-space configuration with file

⟨*Dispatch phs: execute tests*⟩+≡

```
  call test (dispatch_phs_2, "dispatch_phs_2", &
       "configure phase space using file", &
       u, results)
```

⟨*Dispatch phs: test declarations*⟩+≡

```
  public :: dispatch_phs_2
```

⟨*Dispatch phs: tests*⟩+≡

```
  subroutine dispatch_phs_2 (u)
    use phs_base_ut, only: init_test_process_data
    use phs_wood_ut, only: write_test_phs_file
    use phs_forests
    integer, intent(in) :: u
    type(var_list_t) :: var_list
    type(os_data_t) :: os_data
    type(process_constants_t) :: process_data
    type(model_list_t) :: model_list
    type(model_t), pointer :: model
    class(phs_config_t), allocatable :: phs
    integer :: u_phs

    write (u, "(A)")  "* Test output: dispatch_phs_2"
    write (u, "(A)")  "*   Purpose: select 'wood' phase-space &
         &for a test process"
    write (u, "(A)")  "*            and read phs configuration from file"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize a process"
    write (u, "(A)")

    call var_list%init_defaults (0)
    call os_data_init (os_data)
    call syntax_model_file_init ()
    call model_list%read_model &
         (var_str ("Test"), var_str ("Test.mdl"), os_data, model)

    call syntax_phs_forest_init ()
```

```
  call init_test_process_data (var_str ("dispatch_phs_2"), process_data)

  write (u, "(A)")  "* Write phase-space file"

  u_phs = free_unit ()
  open (u_phs, file = "dispatch_phs_2.phs", action = "write", status = "replace")
  call write_test_phs_file (u_phs, var_str ("dispatch_phs_2"))
  close (u_phs)

  write (u, "(A)")
  write (u, "(A)")  "* Allocate PHS as phs_wood_t"
  write (u, "(A)")

  call var_list%set_string (&
       var_str ("$phs_method"), &
       var_str ("wood"), is_known = .true.)
  call var_list%set_string (&
       var_str ("$phs_file"), &
       var_str ("dispatch_phs_2.phs"), is_known = .true.)
  call dispatch_phs (phs, var_list, os_data, var_str ("dispatch_phs_2"))

  call phs%init (process_data, model)
  call phs%configure (sqrts = 1000._default)

  call phs%write (u)
  write (u, "(A)")
  select type (phs)
  type is (phs_wood_config_t)
     call phs%write_forest (u)
  end select

  call phs%final ()

  call var_list%final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: dispatch_phs_2"

end subroutine dispatch_phs_2
```

Chapter 19

VEGAS Integration

The backbone integrator of WHIZARD is a object-oriented implemetation of the VEGAS algorithm.

⟨vegas.f90⟩≡

```
⟨File header⟩

module vegas
⟨Use kinds⟩
  use diagnostics
  use io_units
  use format_utils, only: write_indent
  use format_defs, only: FMT_14
  use rng_base, only: rng_t

⟨Standard module head⟩

⟨vegas: public⟩

⟨vegas: parameters⟩

⟨vegas: types⟩

⟨vegas: interfaces⟩

contains

⟨vegas: procedures⟩

end module vegas
```

19.1 Integration modes

VEGAS operates in three different modes: `vegas_mode_importance_only`, `vegas_mode_importance` or `vegas_mode_stratified`. The default mode is `vegas_mode_importance`, where the algorithm decides whether if it is possible to use importance sampling or stratified sampling. In low dimensions VEGAS uses strict stratified sampling.

⟨*vegas: parameters*⟩≡

```
integer, parameter, public :: VEGAS_MODE_IMPORTANCE = 0, &
     & VEGAS_MODE_STRATIFIED = 1, VEGAS_MODE_IMPORTANCE_ONLY = 2
```

19.2 Type: vegas_func_t

We define a abstract `func` type which only gives an interface to an `evaluate` procedure. The inside of implementation and also the optimization of are not a concern of the `vegas` implementation.

⟨*vegas: public*⟩≡
```
public :: vegas_func_t
```

⟨*vegas: types*⟩≡
```
type, abstract :: vegas_func_t
   !
 contains
   procedure(vegas_func_evaluate), deferred, pass, public :: evaluate
end type vegas_func_t
```

The only procedure called in `vegas` is `vegas_func_evaluate`. It takes a real value `x` and returns a real value `f`.

⟨*vegas: interfaces*⟩≡
```
abstract interface
   real(default) function vegas_func_evaluate (self, x) result (f)
     import :: default, vegas_func_t
     class(vegas_func_t), intent(inout) :: self
     real(default), dimension(:), intent(in) :: x
   end function vegas_func_evaluate
end interface
```

19.3 Type: vegas_config_t

We store the complete configuration in a transparent container. The `vegas_config_t` object inside VEGAS must not be directly accesible. We provide a get method which returns a copy of the `vegas_config_t` object.

Apart from the options which can be set by the constructor of `vegas_t` object, we store the run-time configuration `n_calls`, `calls_per_box`, `n_bins` and `n_boxes`. Those are calculated and set accordingly by VEGAS.

⟨*vegas: public*⟩+≡
```
public :: vegas_config_t
```

⟨*vegas: types*⟩+≡
```
type :: vegas_config_t
   integer :: n_dim = 0
   real(default) :: alpha = 1.5
   integer :: n_bins_max = 50
   integer :: iterations = 5
   integer :: mode = VEGAS_MODE_STRATIFIED
   integer :: calls_per_box = 0
   integer :: n_calls = 0
   integer :: n_calls_min = 20
```

```
    integer :: n_boxes = 1
    integer :: n_bins = 1
  contains
  ⟨vegas: vegas config: TBP⟩
end type vegas_config_t
```

Write out the configuration of the grid.

⟨*vegas: vegas config: TBP*⟩≡

```
procedure, public :: write => vegas_config_write
```

⟨*vegas: procedures*⟩≡

```
subroutine vegas_config_write (self, unit, indent)
  class(vegas_config_t), intent(in) :: self
  integer, intent(in), optional :: unit
  integer, intent(in), optional :: indent
  integer :: u, ind
  u = given_output_unit (unit)
  ind = 0; if (present (indent)) ind = indent
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Number of dimensions = ", self%n_dim
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 // ")") "Adaption power (alpha) = ", self%alpha
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Max. number of bins (per dim.) = ", self%n_bins_max
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Number of iterations = ", self%iterations
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Mode (stratified or importance) = ", self%mode
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Calls per box = ", self%calls_per_box
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Number of calls = ", self%n_calls
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Min. number of calls = ", self%n_calls_min
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Number of bins = ", self%n_bins
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Number of boxes = ", self%n_boxes
end subroutine vegas_config_write
```

19.4 Type: vegas_grid_t

We provide a simple and transparent grid container. The container can then later be used, to export the actual grid.

⟨*vegas: public*⟩+≡

```
public :: vegas_grid_t
```

⟨*vegas: types*⟩+≡

```
type :: vegas_grid_t
   integer :: n_dim = 0
   integer :: n_bins = 0
   real(default), dimension(:), allocatable :: x_lower
   real(default), dimension(:), allocatable :: x_upper
```

```
    real(default), dimension(:), allocatable :: delta_x
    real(default), dimension(:,:), allocatable :: xi
  contains
  ⟨vegas: vegas grid: TBP⟩
end type vegas_grid_t
```

Output.

⟨*vegas: vegas grid: TBP*⟩≡

```
procedure, public :: write => vegas_grid_write
```

⟨*vegas: procedures*⟩+≡

```
subroutine vegas_grid_write (self, unit, indent)
  class(vegas_grid_t), intent(in) :: self
  integer, intent(in), optional :: unit
  integer, intent(in), optional :: indent
  integer :: u, ind, i, j
  u = given_output_unit (unit)
  ind = 0; if (present (indent)) ind = indent
  write (u, descr_fmt) "begin x_lower"
  do j = 1, self%n_dim
     call write_indent (u, ind)
     write (u, double_array_fmt)  j, self%x_lower(j)
  end do
  write (u, descr_fmt) "end x_lower"
  write (u, descr_fmt) "begin x_upper"
  do j = 1, self%n_dim
     call write_indent (u, ind)
     write (u, double_array_fmt) j, self%x_upper(j)
  end do
  write (u, descr_fmt) "end x_upper"
  write (u, descr_fmt) "begin delta_x"
  do j = 1, self%n_dim
     call write_indent (u, ind)
     write (u, double_array_fmt)  j, self%delta_x(j)
  end do
  write (u, descr_fmt) "end delta_x"
  write (u, descr_fmt) "begin xi"
  do j = 1, self%n_dim
     do i = 1, self%n_bins + 1
        call write_indent (u, ind)
        write (u, double_array2_fmt) i, j, self%xi(i, j)
     end do
  end do
  write (u, descr_fmt) "end xi"
end subroutine vegas_grid_write
```

Find the probability for a given `x` in the unit hypercube. We use a binary search algorithm to find the correct bin along each axis.

⟨*vegas: vegas grid: TBP*⟩+≡

```
procedure, public :: get_probability => vegas_grid_get_probability
```

⟨*vegas: procedures*⟩+≡

```
function vegas_grid_get_probability (self, x) result (g)
  class(vegas_grid_t), intent(in) :: self
```

```
    real(default), dimension(:), intent(in) :: x
    real(default) :: g, y
    integer :: j, i_lower, i_higher, i_mid
    g = 1.
    ndim: do j = 1, self%n_dim
       y = (x(j) - self%x_lower(j)) / self%delta_x(j)
       if (y >= 0. .or. y <= 1.) then
          i_lower = 1
          i_higher = self%n_bins + 1
          search: do
             if (i_lower >= (i_higher - 1)) then
                g = g / (self%delta_x(j) * &
                     & self%n_bins * (self%xi(i_higher, j) - self%xi(i_higher - 1, j)))
                cycle ndim
             end if
             i_mid = (i_higher + i_lower) / 2
             if (y > self%xi(i_mid, j)) then
                i_lower = i_mid
             else
                i_higher = i_mid
             end if
          end do search
       else
          g = 0.
          exit ndim
       end if
    end do ndim
  end function vegas_grid_get_probability
```

19.5 Type: vegas_result_t

We store the result of the latest iteration(s) in a transparent container. The `vegas_result_t` object inside VEGAS must not be directly accessible. We export the a copy of the result via a get-method of the `vegas_t` object.

⟨*vegas: public*⟩+≡

```
  public :: vegas_result_t
```

⟨*vegas: types*⟩+≡

```
  type :: vegas_result_t
     integer :: it_start = 0
     integer :: it_num = 0
     integer :: samples = 0
     real(default) :: sum_int_wgtd = 0.
     real(default) :: sum_wgts
     real(default) :: sum_chi = 0.
     real(default) :: chi2 = 0.
     real(default) :: efficiency = 0.
     real(default) :: efficiency_pos = 0.
     real(default) :: efficiency_neg = 0.
     real(default) :: max_abs_f = 0.
     real(default) :: max_abs_f_pos = 0.
     real(default) :: max_abs_f_neg = 0.
```

```
    real(default) :: result = 0.
    real(default) :: std = 0.
  contains
  ⟨vegas: vegas result: TBP⟩
end type vegas_result_t
```

Write out the current status of the integration result.

⟨*vegas: vegas result: TBP*⟩≡

```
procedure, public :: write => vegas_result_write
```

⟨*vegas: procedures*⟩+≡

```
subroutine vegas_result_write (self, unit, indent)
  class(vegas_result_t), intent(in) :: self
  integer, intent(in), optional :: unit
  integer, intent(in), optional :: indent
  integer :: u, ind
  u = given_output_unit (unit)
  ind = 0; if (present (indent)) ind = indent
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Start iteration = ", self%it_start
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Iteration number = ", self%it_num
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Sample number = ", self%samples
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 //")") "Sum of weighted integrals = ", self%sum_int_wgtd
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 // ")") "Sum of weights = ", self%sum_wgts
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 // ")") "Sum of chi = ", self%sum_chi
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 // ")") "chi2 = ", self%chi2
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 // ")") "Overall efficiency = ", self%efficiency
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 // ")") "f-positive efficiency = ", self%efficiency_pos
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 // ")") "f-negative efficiency = ", self%efficiency_neg
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 // ")") "Maximum absolute overall value = ", self%max_abs_f
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 // ")") "Maximum absolute positive value = ", self%max_abs_f_pos
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 // ")") "Maximum absolute negative value = ", self%max_abs_f_neg
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 // ")") "Integral (of latest iteration) = ", self%result
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 // ")") "Standard deviation = ", self%std
end subroutine vegas_result_write
```

19.6 Type: vegas_t

The VEGAS object contains the methods for integration and grid resize- and refinement. We store the grid configuration and the (current) result in transparent containers alongside with the actual grid and the distribution.

The values of the distribution depend on the chosen mode whether the function value or the variance is stored. The distribution is used after each iteration to refine the grid.

⟨*vegas: public*⟩+≡
```
public :: vegas_t
```

⟨*vegas: types*⟩+≡
```
type :: vegas_t
   private
   type(vegas_config_t) :: config
   real(default) :: hypercube_volume = 0.
   real(default) :: jacobian = 0.
   real(default), dimension(:, :), allocatable :: d
   real(default), dimension(:), allocatable :: weight
   type(vegas_grid_t) :: grid
   integer, dimension(:), allocatable :: bin
   integer, dimension(:), allocatable :: box
   type(vegas_result_t) :: result
 contains
 ⟨vegas: vegas: TBP⟩
end type vegas_t
```

We overload the type constructor of `vegas_t` which initialises the mandatory argument `n_dim` and allocate memory for the grid.

⟨*vegas: interfaces*⟩+≡
```
interface vegas_t
   module procedure vegas_init
end interface vegas_t
```

⟨*vegas: procedures*⟩+≡
```
type(vegas_t) function vegas_init (n_dim, alpha, n_bins_max, iterations, mode) result (self)
  integer, intent(in) :: n_dim
  integer, intent(in), optional :: n_bins_max
  real(default), intent(in), optional :: alpha
  integer, intent(in), optional :: iterations
  integer, intent(in), optional :: mode
  self%config%n_dim = n_dim
  if (present (alpha)) self%config%alpha = alpha
  if (present (n_bins_max)) self%config%n_bins_max = n_bins_max
  if (present (iterations)) self%config%iterations = iterations
  if (present (mode)) self%config%mode = mode
  allocate (self%d(self%config%n_bins_max, n_dim), source=0.0_default)
  allocate (self%grid%x_upper(n_dim), source=1.0_default)
  allocate (self%grid%x_lower(n_dim), source=0.0_default)
  allocate (self%grid%delta_x(n_dim), source=1.0_default)
  allocate (self%grid%xi((self%config%n_bins_max + 1), n_dim), source=0.0_default)
  allocate (self%weight(self%config%n_bins_max), source=0.0_default)
  allocate (self%box(n_dim), source=1)
  allocate (self%bin(n_dim), source=1)
```

```
    self%config%n_bins = 1
    self%config%n_boxes = 1
    call self%set_limits (self%grid%x_lower, self%grid%x_upper)
    call self%reset_grid ()
    call self%reset_result ()
  end function vegas_init
```

Finalize the grid. Deallocate grid memory.

⟨*vegas: vegas: TBP*⟩≡

```
  procedure, public :: final => vegas_final
```

⟨*vegas: procedures*⟩+≡

```
  subroutine vegas_final (self)
    class(vegas_t), intent(inout) :: self
    deallocate (self%grid%x_upper)
    deallocate (self%grid%x_lower)
    deallocate (self%grid%delta_x)
    deallocate (self%d)
    deallocate (self%grid%xi)
    deallocate (self%weight)
    deallocate (self%box)
    deallocate (self%bin)
  end subroutine vegas_final
```

19.7 Get-/Set-methods

The VEGAS object prohibits direct access from outside. Communication is handle via get- or set-methods. Set the limits of integration. The defaults limits correspong the n-dimensionl unit hypercube.

Remark: After setting the limits, the grid is initialised, again. Previous results are lost due to recalculation of the overall jacobian.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: set_limits => vegas_set_limits
```

⟨*vegas: procedures*⟩+≡

```
  subroutine vegas_set_limits (self, x_lower, x_upper)
    class(vegas_t), intent(inout) :: self
    real(default), dimension(:), intent(in) :: x_lower
    real(default), dimension(:), intent(in) :: x_upper
    if (size (x_lower) /= self%config%n_dim &
         & .or. size (x_upper) /= self%config%n_dim) then
       write (msg_buffer, "(A, I5, A, I5, A, I5)") &
            "VEGAS: [set_limits] n_dim of new lower/upper integration limit&
            & does not match previously set n_dim. ", self%config%n_dim, " =/=&
            & ", size (x_lower), " =/= ", size (x_upper)
       call msg_fatal ()
    end if
    if (any(x_upper < x_lower)) then
       call msg_fatal ("VEGAS: [set_limits] upper limits are smaller than lower limits.")
    end if

    if (any((x_upper - x_lower) > huge(0._default))) then
```

```
      call msg_fatal ("VEGAS: [set_limits] upper and lower limits exceed rendering.")
    end if
    self%grid%x_upper = x_upper
    self%grid%x_lower = x_lower
    self%grid%delta_x = self%grid%x_upper - self%grid%x_lower
    self%hypercube_volume = product (self%grid%delta_x)
    call self%reset_result ()
  end subroutine vegas_set_limits
```

Set the number of calls. If the number of calls changed during different passes, we resize the grid preserving the probability density. We should reset the results after changing the number of calls which change the size of the grid and the running mode of VEGAS. But, this is a set method only for the number of calls.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: set_calls => vegas_set_n_calls
```

⟨*vegas: procedures*⟩+≡

```
  subroutine vegas_set_n_calls (self, n_calls)
    class(vegas_t), intent(inout) :: self
    integer, intent(in) :: n_calls
    if (.not. (n_calls > 0)) then
      write (msg_buffer, "(A, I5)") &
           "VEGAS: [set_calls] number of calls must be a positive number. Keep&
           & number of calls = ", self%config%n_calls
      call msg_warning ()
    else
      self%config%n_calls = max (n_calls, self%config%n_calls_min)
      if (self%config%n_calls /= n_calls) then
        write (msg_buffer, "(A,I5)") &
             "VEGAS: [set calls] number of calls is too few, reset to ", self%config%n_calls
        call msg_warning ()
      end if
      call self%init_grid ()
    end if
  end subroutine vegas_set_n_calls
```

Get the config object.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: get_config => vegas_get_config
```

⟨*vegas: procedures*⟩+≡

```
  subroutine vegas_get_config (self, config)
    class(vegas_t), intent(in) :: self
    type(vegas_config_t), intent(out) :: config
    config = self%config
  end subroutine vegas_get_config
```

Set non-runtime dependent configuration. It will no be possible to change `n_bins_max`.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: set_config => vegas_set_config
```

⟨*vegas: procedures*⟩+≡

```
subroutine vegas_set_config (self, config)
  class(vegas_t), intent(inout) :: self
  class(vegas_config_t), intent(in) :: config
  self%config%alpha = config%alpha
  self%config%iterations = config%iterations
  self%config%mode = config%mode
  self%config%n_calls_min = config%n_calls_min
 end subroutine vegas_set_config
```

Get the grid object and set `n_bins`, `n_dim` inside grid container.

⟨*vegas: vegas: TBP*⟩+≡

```
procedure, public :: get_grid => vegas_get_grid
```

⟨*vegas: procedures*⟩+≡

```
type(vegas_grid_t) function vegas_get_grid (self) result (grid)
  class(vegas_t), intent(in) :: self
  grid = self%grid
  grid%n_dim = self%config%n_dim
  grid%n_bins = self%config%n_bins
end function vegas_get_grid
```

Get the result object.

⟨*vegas: vegas: TBP*⟩+≡

```
procedure, public :: get_result => vegas_get_result
```

⟨*vegas: procedures*⟩+≡

```
subroutine vegas_get_result (self, result)
  class(vegas_t), intent(in) :: self
  type(vegas_result_t), intent(out) :: result
  result = self%result
end subroutine vegas_get_result
```

Get (actual) number of calls.

⟨*vegas: vegas: TBP*⟩+≡

```
procedure, public :: get_calls => vegas_get_n_calls
```

⟨*vegas: procedures*⟩+≡

```
elemental real(default) function vegas_get_n_calls (self) result (n_calls)
  class(vegas_t), intent(in) :: self
  n_calls = self%config%n_calls
end function vegas_get_n_calls
```

Get the cumulative result of the integration. Recalculate weighted average of the integration.

⟨*vegas: vegas: TBP*⟩+≡

```
procedure, public :: get_integral => vegas_get_integral
```

⟨*vegas: procedures*⟩+≡

```
elemental real(default) function vegas_get_integral (self) result (integral)
  class(vegas_t), intent(in) :: self
  integral = 0.
  if (self%result%sum_wgts > 0.) then
```

```
      integral = self%result%sum_int_wgtd / self%result%sum_wgts
    end if
  end function vegas_get_integral
```

Get the cumulative variance of the integration. Recalculate the variance.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: get_variance => vegas_get_variance
```

⟨*vegas: procedures*⟩+≡

```
  elemental real(default) function vegas_get_variance (self) result (variance)
    class(vegas_t), intent(in) :: self
    variance = 0.
    if (self%result%sum_wgts > 0.) then
       variance = 1.0 / self%result%sum_wgts
    end if
  end function vegas_get_variance
```

Get efficiency.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: get_efficiency => vegas_get_efficiency
```

⟨*vegas: procedures*⟩+≡

```
  elemental real(default) function vegas_get_efficiency (self) result (efficiency)
    class(vegas_t), intent(in) :: self
    efficiency = 0.
    if (self%result%efficiency > 0. ) then
       efficiency = self%result%efficiency
    end if
  end function vegas_get_efficiency
```

Get `f_max`.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: get_max_abs_f => vegas_get_max_abs_f
```

⟨*vegas: procedures*⟩+≡

```
  elemental real(default) function vegas_get_max_abs_f (self) result (max_abs_f)
    class(vegas_t), intent(in) :: self
    max_abs_f = 0.
    if (self%result%max_abs_f > 0.) then
       max_abs_f = self%result%max_abs_f
    end if
  end function vegas_get_max_abs_f
```

Get `f_max_pos`.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: get_max_abs_f_pos => vegas_get_max_abs_f_pos
```

⟨*vegas: procedures*⟩+≡

```
  elemental real(default) function vegas_get_max_abs_f_pos (self) result (max_abs_f)
    class(vegas_t), intent(in) :: self
    max_abs_f = 0.
    if (self%result%max_abs_f_pos > 0.) then
       max_abs_f = self%result%max_abs_f_pos
```

```
    end if
  end function vegas_get_max_abs_f_pos
```

Get f_max_neg.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: get_max_abs_f_neg => vegas_get_max_abs_f_neg
```

⟨*vegas: procedures*⟩+≡

```
  elemental real(default) function vegas_get_max_abs_f_neg (self) result (max_abs_f)
    class(vegas_t), intent(in) :: self
    max_abs_f = 0.
    if (self%result%max_abs_f_neg > 0.) then
       max_abs_f = self%result%max_abs_f_neg
    end if
  end function vegas_get_max_abs_f_neg
```

Get the current distribution.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: get_distribution => vegas_get_distribution
```

⟨*vegas: procedures*⟩+≡

```
  subroutine vegas_get_distribution (self, distribution)
    class(vegas_t), intent(in) :: self
    real(default), dimension(self%config%n_bins_max, self%config%n_dim), intent(out) :: distributi
    distribution = self%d
  end subroutine vegas_get_distribution
```

Set the distribution. Use with care! After setting the distribution, we refine the grid.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: set_distribution => vegas_set_distribution
```

⟨*vegas: procedures*⟩+≡

```
  subroutine vegas_set_distribution (self, distribution)
    class(vegas_t), intent(inout) :: self
    real(default), dimension(:, :), intent(in) :: distribution
    if (size (distribution, 1) > self%config%n_bins_max) then
       write (msg_buffer, "(A, I5)") &
            "[set_distribution] new distribution size exceeds", self%config%n_bins_max
       call msg_bug ()
    end if
    self%d = distribution(1:self%config%n_bins_max, 1:self%config%n_dim)
    call self%refine ()
  end subroutine vegas_set_distribution
```

19.8 Grid resize- and refinement

Before integration the grid itself must be initialised. Given the number of n_calls and n_dim we prepare the grid for the integration.

The grid is binned according to the VEGAS mode and `n_calls`. If the mode is not set to `vegas_importance_only`, the grid is divided in to equal boxes. We try for 2 calls per box

$$boxes = \sqrt[n_{dim}]{\frac{calls}{2}}. \tag{19.1}$$

If the numbers of boxes exceeds the number of bins, which is the case for low dimensions, the algorithm switches to stratified sampling. Otherwise, we are still using importance sampling, but keep the boxes for book keeping. If the number of bins changes from the previous invocation, bins are expanded or contracted accordingly, while preserving bin density.

⟨*vegas: vegas: TBP*⟩+≡

```
procedure, private :: init_grid => vegas_init_grid
```

⟨*vegas: procedures*⟩+≡

```
subroutine vegas_init_grid (self)
  class(vegas_t), intent(inout) :: self
  integer :: n_bins, n_boxes, box_per_bin, n_total_boxes
  n_bins = self%config%n_bins_max
  n_boxes = 1
  if (self%config%mode /= VEGAS_MODE_IMPORTANCE_ONLY) then
     ! We try for 2 calls per box
     n_boxes = max (floor ((self%config%n_calls / 2.)**(1. / self%config%n_dim)), 1)
     self%config%mode = VEGAS_MODE_IMPORTANCE
     if (2 * n_boxes >= self%config%n_bins_max) then
        ! if n_bins/box < 2
        box_per_bin = max (n_boxes / self%config%n_bins_max, 1)
        n_bins =  min (n_boxes / box_per_bin, self%config%n_bins_max)
        n_boxes = box_per_bin * n_bins
        self%config%mode = VEGAS_MODE_STRATIFIED
     end if
  end if
  n_total_boxes = n_boxes**self%config%n_dim
  self%config%calls_per_box = max (floor (real (self%config%n_calls) / n_total_boxes), 2)
  self%config%n_calls = self%config%calls_per_box * n_total_boxes
  !  Total volume of x-space/(average n_calls per bin)
  self%jacobian = self%hypercube_volume * real(n_bins, default)&
       &**self%config%n_dim / real(self%config%n_calls, default)
  self%config%n_boxes = n_boxes
  if (n_bins /= self%config%n_bins) then
     call self%resize (n_bins)
  end if
end subroutine vegas_init_grid
```

Reset the cumulative result, and efficiency and max. grid values.

⟨*vegas: vegas: TBP*⟩+≡

```
procedure, public :: reset_result => vegas_reset_result
```

⟨*vegas: procedures*⟩+≡

```
subroutine vegas_reset_result (self)
  class(vegas_t), intent(inout) :: self
  self%result%sum_int_wgtd = 0.
  self%result%sum_wgts = 0.
  self%result%sum_chi = 0.
```

```
    self%result%it_num = 0
    self%result%samples = 0
    self%result%chi2 = 0
    self%result%efficiency = 0.
    self%result%efficiency_pos = 0.
    self%result%efficiency_neg = 0.
    self%result%max_abs_f = 0.
    self%result%max_abs_f_pos = 0.
    self%result%max_abs_f_neg = 0.
  end subroutine vegas_reset_result
```

Reset the grid. Purge the adapted grid and the distribution. Furthermore, reset the results. The maximal size of the grid remains.

Note: Handle `vegas_reset_grid` with great care! Instead of reusing an old object, create a new one.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: reset_grid => vegas_reset_grid
```

⟨*vegas: procedures*⟩+≡

```
  subroutine vegas_reset_grid (self)
    class(vegas_t), intent(inout) :: self
    self%config%n_bins = 1
    self%d = 0._default
    self%grid%xi = 0._default
    self%grid%xi(1, :) = 0._default
    self%grid%xi(2, :) = 1._default
    call self%reset_result ()
  end subroutine vegas_reset_grid
```

Resize the grid to new number of `bins`. Bins are expanded or contracted accordingly, while preserving bin density. The resizing weight is the ratio of old `n_bins` over new `n_bins`.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, private :: resize => vegas_resize_grid
```

⟨*vegas: procedures*⟩+≡

```
  subroutine vegas_resize_grid (self, n_bins)
    class(vegas_t), intent(inout) :: self
    integer, intent(in) :: n_bins
    integer :: i, j, k
    real(default) :: pts_per_bin, d_width
    real(default), dimension(n_bins + 1) :: xi_new
    pts_per_bin = real(self%config%n_bins, default) / real(n_bins, default)
    do j = 1, self%config%n_dim
       d_width = 0.
       i = 2
       do k = 1, self%config%n_bins
          d_width = d_width + 1.
          associate (x_upper => self%grid%xi(k + 1, j), x_lower => self%grid%xi(k, j))
            do while (d_width > pts_per_bin .and. i <= n_bins)
               d_width = d_width - pts_per_bin
               xi_new(i) = x_upper - (x_upper - x_lower) * d_width
               i = i + 1
```

```
            end do
          end associate
       end do
       self%grid%xi(:, j) = 0. ! Reset grid explicitly
       self%grid%xi(2:n_bins, j) = xi_new(2:n_bins)
       self%grid%xi(n_bins + 1, j) = 1.
    end do
    self%config%n_bins = n_bins
  end subroutine vegas_resize_grid
```

Refine the grid to match the distribution `d`.

Average the distribution over neighbouring bins, then contract or expand the bins. The averaging dampens high fluctuations amog the integrand or the variance.

We make the type-bound procedure public accessible because the multi-channel integration refines each grid after integration over all grids.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: refine => vegas_refine_grid
```

⟨*vegas: procedures*⟩+≡

```
  subroutine vegas_refine_grid (self)
    class(vegas_t), intent(inout) :: self
    integer :: i, j, k
    real(default) :: oldg, newg, rc
    real(default) :: grid_total_jacobian, total_weight
    real(default) :: pts_per_bin, x_old, x_new, d_width
    real(default), dimension(self%config%n_bins_max + 1) :: xi_new
    ndim: do j = 1, self%config%n_dim
       oldg = self%d(1, j)
       newg = self%d(2, j)
       self%d(1, j) = (oldg + newg) / 2.
       ! d[i][j] = (d[i-1][j]+d[i][j]+d[i+1][j])/3
       do i = 2, self%config%n_bins - 1
          rc = oldg + newg
          oldg = newg
          newg = self%d(i + 1, j)
          self%d(i, j) = (rc + newg) / 3
       end do
       self%d(self%config%n_bins, j) = (newg + oldg) / 2.
       grid_total_jacobian = sum(self%d(:, j))
       self%weight = 0.
       do i = 1, self%config%n_bins
          if (self%d(i, j) > 0.) then
             oldg = grid_total_jacobian / self%d(i, j)
             self%weight(i) = ((oldg - 1.) / oldg / log(oldg))**self%config%alpha
          end if
       end do
       total_weight = sum(self%weight)
       pts_per_bin = total_weight / self%config%n_bins
       xi_new = self%grid%xi(:, j)
       x_new = 0.
       d_width = 0.
       i = 2
       do k = 1, self%config%n_bins
```

```
        d_width = d_width + self%weight(k)
        associate (x_upper => self%grid%xi(k + 1, j), x_lower => self%grid%xi(k, j))
          do while (d_width > pts_per_bin .and. i <= self%config%n_bins)
             d_width = d_width - pts_per_bin
             xi_new(i) = x_upper - (x_upper - x_lower) * d_width / self%weight(k)
             i = i + 1
          end do
        end associate
      end do
      self%grid%xi(:, j) = 0. ! Reset explicitly the grid
      self%grid%xi(2:self%config%n_bins, j) = xi_new(2:self%config%n_bins)
      self%grid%xi(self%config%n_bins + 1, j) = 1.
    end do ndim
  end subroutine vegas_refine_grid
```

19.9 Integration

Integrate `func`, in the previous set bounds `x_lower` to `x_upper`, with `n_calls`. Use results from previous invocations of `integrate` with `opt_reset_result = .false.` and better with subsequent calls.

Before we walk through the hybercube, we initialise the grid (at a central position).

We step through the (equidistant) boxes which ensure we do not miss any place in the n-dim. hypercube. In each box we sample `calls_per_box` random points and transform them to bin coordinates. The total integral and the total (sample) variance over each box i is then calculated by

$$E(I)_i = \sum_j^{\text{calls per box}} I_{i,j},$$

$$V(I)_i = \text{calls per box} \frac{\sum_j^{\text{calls per box}} {}^2}{I}_{i,j} - (\sum_j^{\text{calls per box}} I_{i,j}) ** 2 \frac{\text{calls per box}}{\text{calls per box} - 1}.$$

The stratification of the n-dimensional hybercube allows a simple parallelisation of the algorithm (R. Kreckel, "Parallelization of adaptive MC integrators", Computer Physics Communications, vol. 106, no. 3, pp. 258266, Nov. 1997.). We have to ensure that all boxes are sampled, but the number of boxes to distribute is too large. We allow each thread to sample a fraction r of all boxes k such that $r \ll k$. Furthermore, we constrain that the number of process p is much smaller than r.

The overall constraint is

$$p \ll r \ll k. \tag{19.2}$$

We divide the intgeration into a parallel and a perpendicular subspace. The number of parallel dimensions is $D_\parallel = \lfloor \frac{D}{2} \rfloor$.

⟨*vegas: vegas: TBP*⟩+≡

```
procedure, public :: integrate => vegas_integrate
```

⟨*vegas: procedures*⟩+≡

```
subroutine vegas_integrate (self, func, rng, iterations, opt_reset_result,&
    & opt_refine_grid, opt_verbose, result, abserr)
  class(vegas_t), intent(inout) :: self
  class(vegas_func_t), intent(inout) :: func
  class(rng_t), intent(inout) :: rng
  integer, intent(in), optional :: iterations
  logical, intent(in), optional :: opt_reset_result
  logical, intent(in), optional :: opt_refine_grid
  logical, intent(in), optional :: opt_verbose
  real(default), optional, intent(out) :: result, abserr
  integer :: it, j, k
  real(default), dimension(self%config%n_dim) :: x
  real(default) :: fval, fval_sq, bin_volume
  real(default) :: fval_box, fval_sq_box
  real(default) :: total_integral, total_sq_integral, total_variance, chi, wgt
  real(default) :: cumulative_int, cumulative_std
  real(default) :: sum_abs_f_pos, max_abs_f_pos
  real(default) :: sum_abs_f_neg, max_abs_f_neg
  logical :: reset_result = .true.
  logical :: refine_grid = .true.
  logical :: verbose = .false.
  integer :: n_dim_par
  logical :: box_success
  if (present (iterations)) self%config%iterations = iterations
  if (present (opt_reset_result)) reset_result = opt_reset_result
  if (present (opt_refine_grid)) refine_grid = opt_refine_grid
  if (present (opt_verbose)) verbose = opt_verbose
  call self%init_grid ()
  if (reset_result) call self%reset_result
  self%result%it_start = self%result%it_num
  cumulative_int = 0.
  cumulative_std = 0.
  if (verbose) then
     call msg_message ("Results: [it, calls, integral, error, chi^2, eff.]")
  end if
  n_dim_par = floor (self%config%n_dim / 2.)
  iteration: do it = 1, self%config%iterations
     self%result%it_num = self%result%it_start + it
     self%d = 0.
     self%box = 1
     self%bin = 1
     total_integral = 0.
     total_sq_integral = 0.
     total_variance = 0.
     sum_abs_f_pos = 0.
     max_abs_f_pos = 0.
     sum_abs_f_neg = 0.
     max_abs_f_neg = 0.
     box_success = .true.
     loop_over_par_boxes: do while (box_success)
        loop_over_perp_boxes: do while (box_success)
           fval_box = 0._default
           fval_sq_box = 0._default
```

```
      do k = 1, self%config%calls_per_box
         call self%random_point (rng, x, bin_volume)
         ! Call the function, yeah, call it...
         fval = self%jacobian * bin_volume * func%evaluate (x)
         fval_sq = fval**2
         fval_box = fval_box + fval
         fval_sq_box = fval_sq_box + fval_sq
         if (fval > 0.) then
            max_abs_f_pos = max(abs (fval), max_abs_f_pos)
            sum_abs_f_pos = sum_abs_f_pos + abs(fval)
         else
            max_abs_f_neg = max(abs (fval), max_abs_f_neg)
            sum_abs_f_neg = sum_abs_f_neg + abs(fval)
         end if
         if (self%config%mode /= VEGAS_MODE_STRATIFIED) then
            call self%accumulate_distribution (fval_sq)
         end if
      end do
      fval_sq_box = sqrt (fval_sq_box * self%config%calls_per_box)
      ! (a - b) * (a + b) = a**2 - b**2
      fval_sq_box = (fval_sq_box - fval_box) * (fval_sq_box + fval_box)
      if (fval_sq_box <= 0.0) fval_sq_box = tiny (1.0_default)
      total_integral = total_integral + fval_box
      total_sq_integral = total_sq_integral + fval_sq_box
      if (self%config%mode == VEGAS_MODE_STRATIFIED) then
         call self%accumulate_distribution (fval_sq_box)
      end if
      call increment_box_coord (self%box(n_dim_par + 1:self%config&
           &%n_dim), box_success)
   end do loop_over_perp_boxes
   call increment_box_coord (self%box(1:n_dim_par), box_success)
end do loop_over_par_boxes
! Compute final results for this iterations
total_variance = total_sq_integral / (self%config%calls_per_box - 1.)
! Ensure variance is always positive and larger than zero.
if (total_variance < tiny (1._default) / epsilon (1._default) &
     & * max (total_integral**2, 1._default)) then
   total_variance = tiny (1._default) / epsilon (1._default) &
        & * max (total_integral**2, 1._default)
end if
wgt = 1. / total_variance
! Reuse of variable!
total_sq_integral = total_integral**2
self%result%result = total_integral
self%result%std = sqrt (total_variance)
self%result%samples = self%result%samples + 1
if (self%result%samples == 1) then
   self%result%chi2 = 0._default
else
   chi = total_integral
   if (self%result%sum_wgts > 0) chi = chi - self%result%sum_int_wgtd / self%result%sum_wgt
   self%result%chi2 = self%result%chi2 * (self%result%samples - 2.0_default)
   self%result%chi2 = (wgt / (1._default + (wgt / self%result%sum_wgts))) &
        & * chi**2
```

```
      self%result%chi2 = self%result%chi2 / (self%result%samples - 1._default)
   end if
   self%result%sum_wgts = self%result%sum_wgts + wgt
   self%result%sum_int_wgtd = self%result%sum_int_wgtd + (total_integral * wgt)
   self%result%sum_chi = self%result%sum_chi + (total_sq_integral * wgt)
   cumulative_int = self%result%sum_int_wgtd / self%result%sum_wgts
   cumulative_std = sqrt (1. / self%result%sum_wgts)
   self%result%max_abs_f_pos = max_abs_f_pos
   self%result%max_abs_f_neg = max_abs_f_neg
   self%result%efficiency_pos = 0.
   if (max_abs_f_pos > 0.) then
      self%result%efficiency_pos = sum_abs_f_pos / (self%config%n_calls * max_abs_f_pos)
   end if
   self%result%efficiency_neg = 0.
   if (max_abs_f_neg > 0.) then
      self%result%efficiency_neg = sum_abs_f_neg / (self%config%n_calls * max_abs_f_neg)
   end if
   self%result%max_abs_f = max (self%result%max_abs_f_pos, self%result%max_abs_f_neg)
   self%result%efficiency = self%result%efficiency_pos + self%result%efficiency_neg
   if (verbose) then
      write (msg_buffer, "(I0,1x,I0,1x, 4(" // FMT_14 // ",1x))") &
           & it, self%config%n_calls, cumulative_int, cumulative_std, &
           & self%result%chi2, self%result%efficiency
      call msg_message ()
   end if
   if (refine_grid) call self%refine ()
 end do iteration
 if (present(result)) result = cumulative_int
 if (present(abserr)) abserr = abs(cumulative_std)
contains
  ⟨vegas: vegas integrate: procedures⟩
end subroutine vegas_integrate
```

Increment the box coordinates by 1. If we reach the largest value for the current axis (starting with the largest dimension number), we reset the counter to 1 and increment the next axis counter by 1. And so on, until we reach the maximum counter value of the axis with the lowest dimension, then we set `success` to false and the box coord is set to 1.

⟨*vegas: vegas integrate: procedures*⟩≡

```
subroutine increment_box_coord (box, success)
  integer, dimension(:), intent(inout) :: box
  logical, intent(out) :: success
  integer :: j
  success = .true.
  do j = size(box), 1, -1
     box(j) = box(j) + 1
     if (box(j) <= self%config%n_boxes) return
     box(j) = 1
  end do
  success = .false.
end subroutine increment_box_coord
```

Obtain a random point inside the n-dimensional hypercube, transform onto the

correct interval and calculate the bin volume.

⟨*vegas: vegas: TBP*⟩+≡

```
procedure, private :: random_point => vegas_random_point
```

⟨*vegas: procedures*⟩+≡

```
subroutine vegas_random_point (self, rng, x, bin_volume)
  class(vegas_t), intent(inout) :: self
  class(rng_t), intent(inout) :: rng
  real(default), dimension(self%config%n_dim), intent(out) :: x
  real(default), intent(out) :: bin_volume
  integer :: j
  real(default) :: r, y, z, bin_width
  bin_volume = 1.
  ndim: do j = 1, self%config%n_dim
     call rng%generate (r)
     z = ((self%box(j) - 1 + r) / self%config%n_boxes) * self%config%n_bins + 1
     self%bin(j) = max (min (int (z), self%config%n_bins), 1)
     if (self%bin(j) == 1) then
        bin_width = self%grid%xi(2, j)
        y = (z - self%bin(j)) * bin_width
     else
        bin_width = self%grid%xi(self%bin(j) + 1, j) - self%grid%xi(self%bin(j), j)
        y = self%grid%xi(self%bin(j), j) + (z - self%bin(j)) * bin_width
     end if
     x(j) = self%grid%x_lower(j) + y * self%grid%delta_x(j)
     bin_volume = bin_volume * bin_width
  end do ndim
end subroutine vegas_random_point
```

Obtain a random point inside the n-dimensional hyper-cube. We neglect stratification and generate the random point in the most simple way. Hence, we do not need to know in which box we are actually sampling. This is useful for only for event generation.

⟨*vegas: vegas: TBP*⟩+≡

```
procedure, private :: simple_random_point => vegas_simple_random_point
```

⟨*vegas: procedures*⟩+≡

```
subroutine vegas_simple_random_point (self, rng, x, bin_volume)
  class(vegas_t), intent(inout) :: self
  class(rng_t), intent(inout) :: rng
  real(default), dimension(self%config%n_dim), intent(out) :: x
  real(default), intent(out) :: bin_volume
  integer :: j, k
  real(default) :: r, y, z, bin_width
  bin_volume = 1.
  ndim: do j = 1, self%config%n_dim
     call rng%generate (r)
     z = r * self%config%n_bins + 1
     k = max (min (int (z), self%config%n_bins), 1)
     if (k == 1) then
        bin_width = self%grid%xi(2, j)
        y = (z - 1) * bin_width
     else
        bin_width = self%grid%xi(k + 1, j) - self%grid%xi(k, j)
```

```
         y = self%grid%xi(k, j) + (z - k) * bin_width
      end if
      x(j) = self%grid%x_lower(j) + y * self%grid%delta_x(j)
      bin_volume = bin_volume * bin_width
   end do ndim
end subroutine vegas_simple_random_point
```

⟨*vegas: vegas: TBP*⟩+≡

```
procedure, private :: accumulate_distribution => vegas_accumulate_distribution
```

⟨*vegas: procedures*⟩+≡

```
subroutine vegas_accumulate_distribution (self, y)
   class(vegas_t), intent(inout) :: self
   real(default), intent(in) :: y
   integer :: j
   do j = 1, self%config%n_dim
      self%d(self%bin(j), j) = self%d(self%bin(j), j) + y
   end do
end subroutine vegas_accumulate_distribution
```

Generate random event. We accept on the rate

$$\frac{|f(x)|}{\max_x |f(x)|}. \tag{19.3}$$

We keep separate maximum weights for positive and negative weights, and use them, accordingly. In the case of unweighted event generation, if the current weight exceeds the the maximum weight, we update the maximum, accordingly.

⟨*vegas: vegas: TBP*⟩+≡

```
procedure, public :: generate_event => vegas_generate_event
```

⟨*vegas: procedures*⟩+≡

```
subroutine vegas_generate_event (self, func, rng, x, weight)
   class(vegas_t), intent(inout) :: self
   class(vegas_func_t), intent(inout) :: func
   class(rng_t), intent(inout) :: rng
   real(default), dimension(self%config%n_dim), intent(out) :: x
   real(default), intent(out), optional :: weight
   real(default) :: bin_volume
   real(default) :: fval, max_abs_f
   real(default) :: r
   generate: do
      call self%simple_random_point (rng, x, bin_volume)
      fval = self%jacobian * bin_volume * func%evaluate (x)
      if (present(weight)) then
         weight = fval
         exit generate
      end if
      if (fval > 0.) then
         if (abs (fval) > self%result%max_abs_f_pos) then
            self%result%max_abs_f_pos = abs (fval)
            write (msg_buffer, "(A,1X," // FMT_14 // ")") &
                 & "[VEGAS] Adjust maximal absolute value for event&
```

```
                & generation.", self%result%max_abs_f_pos
            call msg_warning ()
            exit generate
         end if
         max_abs_f = self%result%max_abs_f_pos
      else
         if (abs (fval) > self%result%max_abs_f_neg) then
            self%result%max_abs_f_neg = abs (fval)
            write (msg_buffer, "(A,1X," // FMT_14 // ")") "[VEGAS] Adjust&
                & maximal absolute value for event generation.", self%result&
                &%max_abs_f_neg
            call msg_warning ()
            exit generate
         end if
         max_abs_f = self%result%max_abs_f_neg
      end if
      call rng%generate (r)
      ! Do not use division, because max_abs_f could be zero.
      if (max_abs_f * r <= abs(fval)) then
         exit generate
      end if
   end do generate
end subroutine vegas_generate_event
```

19.10 I/0 operation

Write grid to file. We use the original VAMP formater.

⟨*vegas: parameters*⟩+≡

```
character(len=*), parameter, private :: &
     descr_fmt =         "(1X,A)", &
     integer_fmt =       "(1X,A18,1X,I15)", &
     integer_array_fmt = "(1X,I18,1X,I15)", &
     logical_fmt =       "(1X,A18,1X,L1)", &
     double_fmt =        "(1X,A18,1X," // FMT_14 // ")", &
     double_array_fmt =  "(1X,I18,1X," // FMT_14 // ")", &
     double_array2_fmt =  "(1X,2(1X,I8),1X," // FMT_14 // ")"
```

⟨*vegas: vegas: TBP*⟩+≡

```
procedure, public :: write_grid => vegas_write_grid
```

⟨*vegas: procedures*⟩+≡

```
subroutine vegas_write_grid (self, unit)
  class(vegas_t), intent(in) :: self
  integer, intent(in), optional :: unit
  integer :: u
  integer :: i, j
  u = given_output_unit (unit)
  write (u, descr_fmt) "begin type(vegas_t)"
  write (u, integer_fmt) "n_dim =", self%config%n_dim
  write (u, integer_fmt) "n_bins_max =", self%config%n_bins_max
  write (u, double_fmt) "alpha =", self%config%alpha
  write (u, integer_fmt) "iterations =", self%config%iterations
```

```
write (u, integer_fmt) "mode =", self%config%mode
write (u, integer_fmt) "calls_per_box =", self%config%calls_per_box
write (u, integer_fmt) "n_calls =", self%config%n_calls
write (u, integer_fmt) "n_calls_min =", self%config%n_calls_min
write (u, integer_fmt) "n_boxes =", self%config%n_boxes
write (u, integer_fmt) "n_bins =", self%config%n_bins
write (u, integer_fmt) "it_start =", self%result%it_start
write (u, integer_fmt) "it_num =", self%result%it_num
write (u, integer_fmt) "samples =", self%result%samples
write (u, double_fmt) "sum_int_wgtd =", self%result%sum_int_wgtd
write (u, double_fmt) "sum_wgts =", self%result%sum_wgts
write (u, double_fmt) "sum_chi =", self%result%sum_chi
write (u, double_fmt) "chi2 =", self%result%chi2
write (u, double_fmt) "efficiency =", self%result%efficiency
write (u, double_fmt) "efficiency =", self%result%efficiency_pos
write (u, double_fmt) "efficiency =", self%result%efficiency_neg
write (u, double_fmt) "max_abs_f =", self%result%max_abs_f
write (u, double_fmt) "max_abs_f_pos =", self%result%max_abs_f_pos
write (u, double_fmt) "max_abs_f_neg =", self%result%max_abs_f_neg
write (u, double_fmt) "result =", self%result%result
write (u, double_fmt) "std =", self%result%std
write (u, double_fmt) "hypercube_volume =", self%hypercube_volume
write (u, double_fmt) "jacobian =", self%jacobian
write (u, descr_fmt) "begin x_lower"
do j = 1, self%config%n_dim
   write (u, double_array_fmt) j, self%grid%x_lower(j)
end do
write (u, descr_fmt) "end x_lower"
write (u, descr_fmt) "begin x_upper"
do j = 1, self%config%n_dim
   write (u, double_array_fmt) j, self%grid%x_upper(j)
end do
write (u, descr_fmt) "end x_upper"
write (u, descr_fmt) "begin delta_x"
do j = 1, self%config%n_dim
   write (u, double_array_fmt) j, self%grid%delta_x(j)
end do
write (u, descr_fmt) "end delta_x"
write (u, integer_fmt) "n_bins =", self%config%n_bins
write (u, descr_fmt) "begin bin"
do j = 1, self%config%n_dim
   write (u, integer_array_fmt) j, self%bin(j)
end do
write (u, descr_fmt) "end n_bin"
write (u, integer_fmt) "n_boxes =", self%config%n_boxes
write (u, descr_fmt) "begin box"
do j = 1, self%config%n_dim
   write (u, integer_array_fmt) j, self%box(j)
end do
write (u, descr_fmt) "end box"
write (u, descr_fmt) "begin d"
do j = 1, self%config%n_dim
   do i = 1, self%config%n_bins_max
      write (u, double_array2_fmt) i, j, self%d(i, j)
```

```
      end do
    end do
    write (u, descr_fmt) "end d"
    write (u, descr_fmt) "begin xi"
    do j = 1, self%config%n_dim
      do i = 1, self%config%n_bins_max + 1
        write (u,  double_array2_fmt) i, j, self%grid%xi(i, j)
      end do
    end do
    write (u, descr_fmt) "end xi"
    write (u, descr_fmt) "end type(vegas_t)"
  end subroutine vegas_write_grid
```

Read grid configuration from file.

⟨*vegas: vegas: TBP*⟩+≡

```
  procedure, public :: read_grid => vegas_read_grid
```

⟨*vegas: procedures*⟩+≡

```
  subroutine vegas_read_grid (self, unit)
    class(vegas_t), intent(out) :: self
    integer, intent(in) :: unit
    integer :: i, j
    character(len=80) :: buffer
    integer :: ibuffer, jbuffer
    read (unit, descr_fmt) buffer
    read (unit, integer_fmt) buffer, ibuffer
    read (unit, integer_fmt) buffer, jbuffer
    select type(self)
    type is (vegas_t)
      self = vegas_t (n_dim = ibuffer, n_bins_max = jbuffer)
    end select
    read (unit, double_fmt) buffer, self%config%alpha
    read (unit, integer_fmt) buffer, self%config%iterations
    read (unit, integer_fmt) buffer, self%config%mode
    read (unit, integer_fmt) buffer, self%config%calls_per_box
    read (unit, integer_fmt) buffer, self%config%n_calls
    read (unit, integer_fmt) buffer, self%config%n_calls_min
    read (unit, integer_fmt) buffer, self%config%n_boxes
    read (unit, integer_fmt) buffer, self%config%n_bins
    read (unit, integer_fmt) buffer, self%result%it_start
    read (unit, integer_fmt) buffer, self%result%it_num
    read (unit, integer_fmt) buffer, self%result%samples
    read (unit, double_fmt) buffer, self%result%sum_int_wgtd
    read (unit, double_fmt) buffer, self%result%sum_wgts
    read (unit, double_fmt) buffer, self%result%sum_chi
    read (unit, double_fmt) buffer, self%result%chi2
    read (unit, double_fmt) buffer, self%result%efficiency
    read (unit, double_fmt) buffer, self%result%efficiency_pos
    read (unit, double_fmt) buffer, self%result%efficiency_neg
    read (unit, double_fmt) buffer, self%result%max_abs_f
    read (unit, double_fmt) buffer, self%result%max_abs_f_pos
    read (unit, double_fmt) buffer, self%result%max_abs_f_neg
    read (unit, double_fmt) buffer, self%result%result
    read (unit, double_fmt) buffer, self%result%std
```

```
    read (unit, double_fmt) buffer, self%hypercube_volume
    read (unit, double_fmt) buffer, self%jacobian
    read (unit, descr_fmt) buffer
    do j = 1, self%config%n_dim
       read (unit, double_array_fmt) jbuffer, self%grid%x_lower(j)
    end do
    read (unit, descr_fmt) buffer
    read (unit, descr_fmt) buffer
    do j = 1, self%config%n_dim
       read (unit, double_array_fmt) jbuffer, self%grid%x_upper(j)
    end do
    read (unit, descr_fmt) buffer
    read (unit, descr_fmt) buffer
    do j = 1, self%config%n_dim
       read (unit, double_array_fmt) jbuffer, self%grid%delta_x(j)
    end do
    read (unit, descr_fmt) buffer
    read (unit, integer_fmt) buffer, self%config%n_bins
    read (unit, descr_fmt) buffer
    do j = 1, self%config%n_dim
       read (unit, integer_array_fmt) jbuffer, self%bin(j)
    end do
    read (unit, descr_fmt) buffer
    read (unit, integer_fmt) buffer, self%config%n_boxes
    read (unit, descr_fmt) buffer
    do j = 1, self%config%n_dim
       read (unit, integer_array_fmt) jbuffer, self%box(j)
    end do
    read (unit, descr_fmt) buffer
    read (unit, descr_fmt) buffer
    do j = 1, self%config%n_dim
       do i = 1, self%config%n_bins_max
          read (unit, double_array2_fmt) ibuffer, jbuffer, self%d(i, j)
       end do
    end do
    read (unit, descr_fmt) buffer
    read (unit, descr_fmt) buffer
    do j = 1, self%config%n_dim
       do i = 1, self%config%n_bins_max + 1
          read (unit,  double_array2_fmt) ibuffer, jbuffer, self%grid%xi(i, j)
       end do
    end do
    read (unit, descr_fmt) buffer
    read (unit, descr_fmt) buffer
  end subroutine vegas_read_grid
```

19.11 Unit tests

Test module, followed by the corresponding implementation module.

⟨vegas_ut.f90⟩≡
  ⟨*File header*⟩

```
module vegas_ut
  use unit_tests
  use vegas_uti

⟨Standard module head⟩

⟨vegas: public test⟩

contains
⟨vegas: test driver⟩
end module vegas_ut
```

⟨vegas_uti.f90⟩≡

```
⟨File header⟩

module vegas_uti
⟨Use kinds⟩
  use io_units
  use constants, only: pi
  use format_defs, only: FMT_10, FMT_12
  use rng_base
  use rng_tao
  use vegas

⟨Standard module head⟩

⟨vegas: test declaration⟩

⟨vegas: test types⟩

contains
⟨vegas: tests⟩
end module vegas_uti
```

API: driver for the unit tests below.

⟨vegas: public test⟩≡

```
  public :: vegas_test
```

⟨vegas: test driver⟩≡

```
  subroutine vegas_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
    ⟨vegas: execute tests⟩
  end subroutine vegas_test
```

Test function

We use the example from the Monte Carlo Examples of the GSL library

$$I = \int_{-pi}^{+pi} dk_x/(2pi) \int_{-pi}^{+pi} dk_y/(2pi) \int_{-pi}^{+pi} dk_z/(2pi) 1/(1-cos(k_x)cos(k_y)cos(k_z)). \tag{19.4}$$

The integral is reduced to region (0,0,0) → (π, π, π) and multiplied by 8.

⟨*vegas: test types*⟩≡
```
type, extends (vegas_func_t) :: vegas_test_func_t
   !
 contains
 ⟨vegas: vegas test func: TBP⟩
end type vegas_test_func_t
```

Evaluate the integrand.

⟨*vegas: vegas test func: TBP*⟩≡
```
procedure, public :: evaluate => vegas_test_func_evaluate
```

⟨*vegas: tests*⟩≡
```
real(default) function vegas_test_func_evaluate (self, x) result (f)
  class(vegas_test_func_t), intent(inout) :: self
  real(default), dimension(:), intent(in) :: x
  f = 1.0 / (pi**3)
  f = f / ( 1.0 - cos (x(1)) * cos (x(2)) * cos (x(3)))
end function vegas_test_func_evaluate
```

The second test function is the normalised n-dim. gaussian distribution.

⟨*vegas: test types*⟩+≡
```
type, extends (vegas_func_t) :: vegas_gaussian_test_func_t
   !
 contains
 ⟨vegas: vegas gaussian test func: TBP⟩
end type vegas_gaussian_test_func_t
```

Evaluate the integrand.

⟨*vegas: vegas gaussian test func: TBP*⟩≡
```
procedure, public :: evaluate => vegas_gaussian_evaluate
```

⟨*vegas: tests*⟩+≡
```
real(default) function vegas_gaussian_evaluate (self, x) result (f)
  class(vegas_gaussian_test_func_t), intent(inout) :: self
  real(default), dimension(:), intent(in) :: x
  real(default), parameter :: inv_sqrt_pi = 1._default / sqrt(pi)
  f = inv_sqrt_pi**size (x)
  f = f * exp (- dot_product(x, x))
end function vegas_gaussian_evaluate
```

The third test function is a three-dimensional polynomial function which factories. The function is defined in such a way that the integral in the unit range is normalised to zero.

$$f(x) = -\frac{8}{3}(x+1)*(y-1)*z \tag{19.5}$$

⟨*vegas: test types*⟩+≡
```
type, extends (vegas_func_t) :: vegas_polynomial_func_t
   !
 contains
 ⟨vegas: vegas polynomial func: TBP⟩
 end type vegas_polynomial_func_t
```

⟨*vegas: vegas polynomial func: TBP*⟩≡

```
procedure, public :: evaluate => vegas_polynomial_evaluate
```

⟨*vegas: tests*⟩+≡

```
real(default) function vegas_polynomial_evaluate (self, x) result (f)
  class(vegas_polynomial_func_t), intent(inout) :: self
  real(default), dimension(:), intent(in) :: x
  f = - 8. / 3. * (x(1) + 1.) * (x(2) - 1.) * x(3)
end function vegas_polynomial_evaluate
```

MC Integrator check

Initialise the VEGAS MC integrator and call to `vegas_init_grid` for the initialisation of the grid.

⟨*vegas: execute tests*⟩≡

```
call test (vegas_1, "vegas_1", "VEGAS initialisation and&
     & grid preparation", u, results)
```

⟨*vegas: test declaration*⟩≡

```
public :: vegas_1
```

⟨*vegas: tests*⟩+≡

```
subroutine vegas_1 (u)
  integer, intent(in) :: u
  type(vegas_t) :: mc_integrator
  class(rng_t), allocatable :: rng
  class(vegas_func_t), allocatable :: func
  real(default), dimension(3), parameter :: x_lower = 0., &
       x_upper = pi
  real(default) :: result, abserr

  write (u, "(A)") "* Test output: vegas_1"
  write (u, "(A)") "*   Purpose: initialise the VEGAS MC integrator and the grid"
  write (u, "(A)")

  write (u, "(A)") "* Initialise random number generator (default seed)"
  write (u, "(A)")

  allocate (rng_tao_t :: rng)
  call rng%init ()

  call rng%write (u)

  write (u, "(A)")
  write (u, "(A)") "* Initialise MC integrator with n_dim = 3"
  write (u, "(A)")

  allocate (vegas_test_func_t :: func)
  mc_integrator = vegas_t (3)

  write (u, "(A)")
  write (u, "(A)") "* Initialise grid with n_calls = 10000"
  write (u, "(A)")
```

```
    call mc_integrator%set_limits (x_lower, x_upper)
    call mc_integrator%set_calls (10000)

    write (u, "(A)")
    write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 10000 (Adaptation)"
    write (u, "(A)")

    call mc_integrator%integrate (func, rng, 3, result=result, abserr=abserr)
    write (u, "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr

    write (u, "(A)")
    write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 2000 (Precision)"
    write (u, "(A)")

    call mc_integrator%set_calls (2000)
    call mc_integrator%integrate (func, rng, 3, result=result, abserr=abserr)
    write (u, "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr


    write (u, "(A)")
    write (u, "(A)") "* Write grid"
    write (u, "(A)")

    call mc_integrator%write_grid (u)

    write (u, "(A)")
    write (u, "(A)") "* Cleanup"

    call mc_integrator%final ()
    call rng%final ()
    deallocate (rng)
  end subroutine vegas_1
```

Configuration and result check

Initialise the MC integrator. Get and write the config object, also the (empty) result object.

⟨*vegas: execute tests*⟩+≡

```
  call test (vegas_2, "vegas_2", "VEGAS configuration and result object", u, results)
```

⟨*vegas: test declaration*⟩+≡

```
  public :: vegas_2
```

⟨*vegas: tests*⟩+≡

```
  subroutine vegas_2 (u)
    integer, intent(in) :: u
    type(vegas_t) :: mc_integrator
    type(vegas_config_t) :: mc_integrator_config
    type(vegas_result_t) :: mc_integrator_result

    write (u, "(A)") "* Test output: vegas_2"
    write (u, "(A)") "*   Purpose: use transparent containers for&
         & configuration and result."
    write (u, "(A)")
```

```
  write (u, "(A)")
  write (u, "(A)") "* Initialise MC integrator with n_dim = 10"
  write (u, "(A)")

  mc_integrator = vegas_t (10)

  write (u, "(A)")
  write (u, "(A)") "* Initialise grid with n_calls = 10000 (Importance Sampling)"
  write (u, "(A)")

  call mc_integrator%set_calls (10000)

  write (u, "(A)")
  write (u, "(A)") "* Get VEGAS config object and write out"
  write (u, "(A)")

  call mc_integrator%get_config (mc_integrator_config)
  call mc_integrator_config%write (u)

  write (u, "(A)")
  write (u, "(A)") "* Get VEGAS empty result object and write out"
  write (u, "(A)")

  call mc_integrator%get_result (mc_integrator_result)
  call mc_integrator_result%write (u)

  write (u, "(A)")
  write (u, "(A)") "* Write grid"
  write (u, "(A)")

  call mc_integrator%write_grid (u)

  write (u, "(A)")
  write (u, "(A)") "* Cleanup"

  call mc_integrator%final ()
end subroutine vegas_2
```

Grid check

Initialise the MC integrator. Get and write the config object. Integrate the gaussian distribution. Get and write the result object. Before and after integration get the grid object and output both. Repeat with different number of dimensions.

⟨*vegas: execute tests*⟩+≡

```
call test (vegas_3, "vegas_3", "VEGAS integration of multi-dimensional gaussian", u, results)
```

⟨*vegas: test declaration*⟩+≡

```
public :: vegas_3
```

⟨*vegas: tests*⟩+≡

```
subroutine vegas_3 (u)
  integer, intent(in) :: u
```

```
type(vegas_t) :: mc_integrator
class(rng_t), allocatable :: rng
class(vegas_func_t), allocatable :: func
real(default), dimension(3), parameter :: x_lower_3 = -10._default, &
     x_upper_3 = 10._default
type(vegas_config_t) :: mc_integrator_config
type(vegas_grid_t) :: mc_integrator_grid
type(vegas_result_t) :: mc_integrator_result

real(default) :: result, abserr

write (u, "(A)") "* Test output: vegas_3"
write (u, "(A)") "*   Purpose: Integrate gaussian distribution."
write (u, "(A)")

allocate (rng_tao_t :: rng)
call rng%init ()

call rng%write (u)

write (u, "(A)")
write (u, "(A)") "* Initialise MC integrator with n_dim = 3"
write (u, "(A)")

allocate (vegas_gaussian_test_func_t :: func)
mc_integrator = vegas_t (3)

write (u, "(A)")
write (u, "(A)") "* Initialise grid with n_calls = 10000"
write (u, "(A)")

call mc_integrator%set_limits (x_lower_3, x_upper_3)
call mc_integrator%set_calls (10000)

write (u, "(A)")
write (u, "(A)") "* Get VEGAS config object and write out"
write (u, "(A)")

call mc_integrator%get_config (mc_integrator_config)
call mc_integrator_config%write (u)

write (u, "(A)")
write (u, "(A)") "* Get VEGAS grid object and write out"
write (u, "(A)")

mc_integrator_grid = mc_integrator%get_grid ()
call mc_integrator_grid%write (u)

write (u, "(A)")
write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 20000 (Adaptation)"
write (u, "(A)")

call mc_integrator%integrate (func, rng, 3, result=result, abserr=abserr)
write (u, "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr
```

```
  write (u, "(A)")
  write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 2000 (Precision)"
  write (u, "(A)")

  call mc_integrator%set_calls (2000)
  call mc_integrator%get_config (mc_integrator_config)
  call mc_integrator_config%write (u)

  write (u, "(A)")

  call mc_integrator%integrate (func, rng, 3, result=result, abserr=abserr)
  write (u, "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr


  write (u, "(A)")
  write (u, "(A)") "* Get VEGAS result object and write out"
  write (u, "(A)")

  call mc_integrator%get_result (mc_integrator_result)
  call mc_integrator_result%write (u)

  write (u, "(A)")
  write (u, "(A)") "* Get VEGAS grid object and write out"
  write (u, "(A)")

  mc_integrator_grid = mc_integrator%get_grid ()
  call mc_integrator_grid%write (u)

  write (u, "(A)")
  write (u, "(A)") "* Write grid"
  write (u, "(A)")

  call mc_integrator%write_grid (u)

  write (u, "(A)")
  write (u, "(A)") "* Cleanup"

  call mc_integrator%final ()
end subroutine vegas_3
```

Three-dimensional integration with polynomial function

Initialise the MC integrator. Get and write the config object. Integrate the factorisable polynomial function. Get and write the result object. Repeat with different number of dimensions.

⟨*vegas: execute tests*⟩+≡

```
call test (vegas_4, "vegas_4", "VEGAS integration of three&
     &-dimensional factorisable polynomial function", u, results)
```

⟨*vegas: test declaration*⟩+≡

```
public :: vegas_4
```

⟨*vegas: tests*⟩+≡

```
subroutine vegas_4 (u)
  integer, intent(in) :: u
  type(vegas_t) :: mc_integrator
  class(rng_t), allocatable :: rng
  class(vegas_func_t), allocatable :: func
  real(default), dimension(3), parameter :: x_lower_3 = 0._default, &
       x_upper_3 = 1._default
  type(vegas_config_t) :: mc_integrator_config
  type(vegas_result_t) :: mc_integrator_result

  real(default) :: result, abserr

  write (u, "(A)") "* Test output: vegas_4"
  write (u, "(A)") "*   Purpose: Integrate gaussian distribution."
  write (u, "(A)")

  allocate (rng_tao_t :: rng)
  call rng%init ()

  call rng%write (u)

  write (u, "(A)")
  write (u, "(A)") "* Initialise MC integrator with n_dim = 3"
  write (u, "(A)")

  allocate (vegas_polynomial_func_t :: func)
  mc_integrator = vegas_t (3)

  write (u, "(A)")
  write (u, "(A)") "* Initialise grid with n_calls = 2000"
  write (u, "(A)")

  call mc_integrator%set_limits (x_lower_3, x_upper_3)
  call mc_integrator%set_calls (2000)

  write (u, "(A)")
  write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 2000 (Adaptation)"
  write (u, "(A)")

  call mc_integrator%integrate (func, rng, 3, result=result, abserr=abserr)

  call mc_integrator%get_config (mc_integrator_config)
  call mc_integrator_config%write (u)

  write (u, "(A)")

  write (u, "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr

  write (u, "(A)")
  write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 20000 (Precision)"
  write (u, "(A)")

  call mc_integrator%set_calls (20000)
```

```
    call mc_integrator%integrate (func, rng, 3, result=result, abserr=abserr)

    call mc_integrator%get_config (mc_integrator_config)
    call mc_integrator_config%write (u)

    write (u, "(A)")
    write (u, "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr

    write (u, "(A)")
    write (u, "(A)") "* Get VEGAS result object and write out"
    write (u, "(A)")

    call mc_integrator%get_result (mc_integrator_result)
    call mc_integrator_result%write (u)

    write (u, "(A)")
    write (u, "(A)") "* Write grid"
    write (u, "(A)")

    call mc_integrator%write_grid (u)

    write (u, "(A)")
    write (u, "(A)") "* Cleanup"

    call mc_integrator%final ()
  end subroutine vegas_4
```

Event generation

Initialise the MC integrator. Integrate the gaussian distribution. Get and write the result object. Finally, generate events in accordance to the adapted grid and print them out.

⟨*vegas: execute tests*⟩+≡

```
  call test (vegas_5, "vegas_5", "VEGAS integration and event&
      & generation of multi-dimensional gaussian", u, results)
```

⟨*vegas: test declaration*⟩+≡

```
  public :: vegas_5
```

⟨*vegas: tests*⟩+≡

```
  subroutine vegas_5 (u)
    integer, intent(in) :: u
    type(vegas_t) :: mc_integrator
    class(rng_t), allocatable :: rng
    class(vegas_func_t), allocatable :: func
    real(default), dimension(1), parameter :: x_lower_1 = -10._default, &
        x_upper_1 = 10._default
    type(vegas_config_t) :: mc_integrator_config
    type(vegas_result_t) :: mc_integrator_result

    integer :: i, u_event
    real(default), dimension(1) :: event
```

```
real(default) :: result, abserr

write (u, "(A)") "* Test output: vegas_5"
write (u, "(A)") "*   Purpose: Integrate gaussian distribution."
write (u, "(A)")

allocate (rng_tao_t :: rng)
call rng%init ()

call rng%write (u)

write (u, "(A)")
write (u, "(A)") "* Initialise MC integrator with n_dim = 1"
write (u, "(A)")

allocate (vegas_gaussian_test_func_t :: func)
mc_integrator = vegas_t (1)

write (u, "(A)")
write (u, "(A)") "* Initialise grid with n_calls = 10000"
write (u, "(A)")

call mc_integrator%set_limits (x_lower_1, x_upper_1)
call mc_integrator%set_calls (1000)

write (u, "(A)")
write (u, "(A)") "* Integrate with n_it = 3 (Adaptation)"
write (u, "(A)")

call mc_integrator%integrate (func, rng, 3, opt_verbose=.true., result=result, abserr=abserr)
call mc_integrator%get_config (mc_integrator_config)
call mc_integrator_config%write (u)
write (u, "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr

write (u, "(A)")
write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 100000 (Precision)"
write (u, "(A)")

call mc_integrator%set_calls (10000)
call mc_integrator%integrate (func, rng, 3, opt_verbose=.true., result=result, abserr=abserr)
call mc_integrator%get_config (mc_integrator_config)
call mc_integrator_config%write (u)
write (u, "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr

write (u, "(A)")
write (u, "(A)") "* Get VEGAS result object and write out"
write (u, "(A)")

call mc_integrator%get_result (mc_integrator_result)
call mc_integrator_result%write (u)

write (u, "(A)")
write (u, "(A)") "* Generate 1000 events based on the adaptation."
write (u, "(A)")
```

```
    do i = 1, 1000
       call mc_integrator%generate_event (func, rng, event)
       write (u, "(2x,I4,1x," // FMT_10 // ")") i, event
    end do

    write (u, "(A)")
    write (u, "(A)") "* Cleanup"

    call mc_integrator%final ()
  end subroutine vegas_5
```

Grid I/O

Initialise the MC integrator. Get and write the config object. Integrate the factorisable polynomial function. Get and write the result object. Write grid to file and start with fresh grid.

⟨*vegas: execute tests*⟩+≡

```
  call test (vegas_6, "vegas_6", "VEGAS integrate and write grid, &
      & read grid and continue", u, results)
```

⟨*vegas: test declaration*⟩+≡

```
  public :: vegas_6
```

⟨*vegas: tests*⟩+≡

```
  subroutine vegas_6 (u)
    integer, intent(in) :: u
    type(vegas_t) :: mc_integrator
    class(rng_t), allocatable :: rng
    class(vegas_func_t), allocatable :: func
    real(default), dimension(3), parameter :: x_lower_3 = 0._default, &
        x_upper_3 = 1._default
    type(vegas_config_t) :: mc_integrator_config
    type(vegas_result_t) :: mc_integrator_result

    real(default) :: result, abserr
    integer :: unit

    write (u, "(A)") "* Test output: vegas_6"
    write (u, "(A)") "*   Purpose: Write and read grid, and continue."
    write (u, "(A)")

    allocate (rng_tao_t :: rng)
    call rng%init ()

    call rng%write (u)

    write (u, "(A)")
    write (u, "(A)") "* Initialise MC integrator with n_dim = 3"
    write (u, "(A)")

    allocate (vegas_polynomial_func_t :: func)
    mc_integrator = vegas_t (3)
```

```
write (u, "(A)")
write (u, "(A)") "* Initialise grid with n_calls = 2000"
write (u, "(A)")

call mc_integrator%set_limits (x_lower_3, x_upper_3)
call mc_integrator%set_calls (2000)

write (u, "(A)")
write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 2000 (Adaptation)"
write (u, "(A)")

call mc_integrator%integrate (func, rng, 3, result=result, abserr=abserr)

call mc_integrator%get_config (mc_integrator_config)
call mc_integrator_config%write (u)

write (u, "(A)")

write (u,  "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr

write (u, "(A)")
write (u, "(A)") "* Write grid to file vegas_io.grid"
write (u, "(A)")

unit = free_unit ()
open (unit, file = "vegas_io.grid", &
     action = "write", status = "replace")
call mc_integrator%write_grid (unit)
close (unit)

write (u, "(A)")
write (u, "(A)") "* Read grid from file vegas_io.grid"
write (u, "(A)")

call mc_integrator%final ()
open (unit, file = "vegas_io.grid", &
     action = "read", status = "old")
call mc_integrator%read_grid (unit)
close (unit)

write (u, "(A)")
write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 20000 (Precision)"
write (u, "(A)")

call mc_integrator%set_calls (20000)

call mc_integrator%integrate (func, rng, 3, result=result, abserr=abserr)

call mc_integrator%get_config (mc_integrator_config)
call mc_integrator_config%write (u)

write (u, "(A)")
write (u,  "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr
```

```
  write (u, "(A)")
  write (u, "(A)") "* Get VEGAS result object and write out"
  write (u, "(A)")

  call mc_integrator%get_result (mc_integrator_result)
  call mc_integrator_result%write (u)

  write (u, "(A)")
  write (u, "(A)") "* Cleanup"

  call mc_integrator%final ()
end subroutine vegas_6
```

19.12 VAMP2

We concentrate all configuration and run-time data in a derived-type, such that, `mci_t` can spwan each time a distinctive MCI VEGAS integrator object.

⟨`vamp2.f90`⟩≡
```
⟨File header⟩

module vamp2

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_utils, only: pac_fmt
  use format_utils, only: write_separator, write_indent
  use format_defs, only: FMT_12, FMT_14, FMT_17, FMT_19
  use diagnostics
  use rng_base

  use vegas

⟨Standard module head⟩

⟨vamp2: public⟩

⟨vamp2: parameters⟩

⟨vamp2: types⟩

⟨vamp2: interfaces⟩

contains

⟨vamp2: procedures⟩

end module vamp2
```

19.12.1 Type: vamp2_func_t

We extend `vegas_func_t` with the multi-channel weights and the `vegas_grid_t`, such that, the overall multi-channel weight can be calculated by the function itself.

We add an additional logicial `valid_x`, if it is set to `.false.`, we do not compute weighted function and just set the weighted integrand to zero. This behavior is in particular very useful, if a mapping is prohibited or fails. Or in the case of WHIZARD, a phase cut is applied.

⟨*vamp2: public*⟩≡
```
public :: vamp2_func_t
```

⟨*vamp2: types*⟩≡
```
type, abstract, extends(vegas_func_t) :: vamp2_func_t
   integer :: current_channel = 0
   integer :: n_dim = 0
   integer :: n_channel = 0
```

```
    integer :: n_calls = 0
    logical :: valid_x = .false.
    real(default), dimension(:, :), allocatable :: xi
    real(default), dimension(:), allocatable :: det
    real(default), dimension(:), allocatable :: wi
    real(default), dimension(:), allocatable :: gi
    type(vegas_grid_t), dimension(:), allocatable :: grids
    real(default) :: g = 0.
  contains
  ⟨vamp2: vamp2 func: TBP⟩
end type vamp2_func_t
```

Init.

⟨*vamp2: vamp2 func: TBP*⟩≡

```
procedure, public :: init => vamp2_func_init
```

⟨*vamp2: procedures*⟩≡

```
subroutine vamp2_func_init (self, n_dim, n_channel)
  class(vamp2_func_t), intent(out) :: self
  integer, intent(in) :: n_dim
  integer, intent(in) :: n_channel
  self%n_dim = n_dim
  self%n_channel = n_channel
  allocate (self%xi(n_dim, n_channel), source=0._default)
  allocate (self%det(n_channel), source=1._default)
  allocate (self%wi(n_channel), source=0._default)
  allocate (self%gi(n_channel), source=0._default)
  allocate (self%grids(n_channel))
end subroutine vamp2_func_init
```

Set current channel.

⟨*vamp2: vamp2 func: TBP*⟩+≡

```
procedure, public :: set_channel => vamp2_func_set_channel
```

⟨*vamp2: procedures*⟩+≡

```
subroutine vamp2_func_set_channel (self, channel)
  class(vamp2_func_t), intent(inout) :: self
  integer, intent(in) :: channel
  self%current_channel = channel
end subroutine vamp2_func_set_channel
```

Get number of function calls.

⟨*vamp2: vamp2 func: TBP*⟩+≡

```
procedure, public :: get_n_calls => vamp2_func_get_n_calls
```

⟨*vamp2: procedures*⟩+≡

```
integer function vamp2_func_get_n_calls (self) result (n_calls)
  class(vamp2_func_t), intent(in) :: self
  n_calls = self%n_calls
end function vamp2_func_get_n_calls
```

Reset number of calls.

⟨*vamp2: vamp2 func: TBP*⟩+≡

```
procedure, public :: reset_n_calls => vamp2_func_reset_n_calls
```

⟨*vamp2: procedures*⟩+≡

```
subroutine vamp2_func_reset_n_calls (self)
  class(vamp2_func_t), intent(inout) :: self
  self%n_calls = 0
end subroutine vamp2_func_reset_n_calls
```

Evaluate mappings. We defer this method to be implemented by the user. The result must be written to `xi` and `det`.

The mapping is defined by $\phi : U \to M$. We map $x \in M$ to the different mappings of the hypercube U_i, such that $x_i \in U_i$.

The mapping should determine, whether `x` is a valid point, e.g. can be mapped, or is restricted otherwise.

⟨*vamp2: vamp2 func: TBP*⟩+≡

```
procedure(vamp2_func_evaluate_maps), deferred :: evaluate_maps
```

⟨*vamp2: interfaces*⟩≡

```
abstract interface
  subroutine vamp2_func_evaluate_maps (self, x)
    import :: vamp2_func_t, default
    class(vamp2_func_t), intent(inout) :: self
    real(default), dimension(:), intent(in) :: x
  end subroutine vamp2_func_evaluate_maps
end interface
```

Evaluate channel weights.

⟨*vamp2: vamp2 func: TBP*⟩+≡

```
procedure, private :: evaluate_weight => vamp2_func_evaluate_weight
```

⟨*vamp2: procedures*⟩+≡

```
subroutine vamp2_func_evaluate_weight (self)
  class(vamp2_func_t), intent(inout) :: self
  integer :: ch
  self%g = 0.
  self%gi = 0.
  do ch = 1, self%n_channel
     if (self%wi(ch) /= 0) then
        self%gi(ch) = self%grids(ch)%get_probability (self%xi(:, ch))
     end if
  end do
  if (self%gi(self%current_channel) /= 0) then
     do ch = 1, self%n_channel
        if (self%wi(ch) /= 0 .and. self%det(ch) /= 0) then
           self%g = self%g + self%wi(ch) * self%gi(ch) / self%det(ch)
        end if
     end do
     self%g = self%g / self%gi(self%current_channel)
  end if
end subroutine vamp2_func_evaluate_weight
```

Evaluate function at `x`. We call this procedure in `vamp2_func_evaluate`.

⟨*vamp2: vamp2 func: TBP*⟩+≡

```
procedure(vamp2_func_evaluate_func), deferred :: evaluate_func
```

⟨*vamp2: interfaces*⟩+≡

```
abstract interface
   real(default) function vamp2_func_evaluate_func (self, x) result (f)
     import :: vamp2_func_t, default
     class(vamp2_func_t), intent(in) :: self
     real(default), dimension(:), intent(in) :: x
   end function vamp2_func_evaluate_func
end interface
```

⟨*vamp2: vamp2 func: TBP*⟩+≡

```
procedure, public :: evaluate => vamp2_func_evaluate
```

⟨*vamp2: procedures*⟩+≡

```
real(default) function vamp2_func_evaluate (self, x) result (f)
  class(vamp2_func_t), intent(inout) :: self
  real(default), dimension(:), intent(in) :: x
  call self%evaluate_maps (x)
  f = 0.
  self%gi = 0.
  self%g = 1
  if (self%valid_x) then
     call self%evaluate_weight ()
     f = self%evaluate_func (x) / self%g
  end if
  self%n_calls = self%n_calls + 1
end function vamp2_func_evaluate
```

19.12.2 Type: vamp2_config_t

This is a transparent container which incorporates and extends the definitions in `vegas_config`. The parent object can then be used to parametrise the VEGAS grids directly, where the new parameters are exclusively used in the multi-channel implementation of VAMP2. `n_calls_min` is calculated by `n_calls_min_per_channel` and `n_channel`. The channels weights (and the result `n_calls` for each channel) are calculated regarding `n_calls_threshold`.

⟨*vamp2: public*⟩+≡

```
public :: vamp2_config_t
```

⟨*vamp2: types*⟩+≡

```
type, extends(vegas_config_t) :: vamp2_config_t
   integer :: n_channel = 0
   integer :: n_calls_min_per_channel = 20
   integer :: n_calls_threshold = 10
   logical :: stratified = .true.
   real(default) :: beta = 0.5_default
   real(default) :: accuracy_goal = 0._default
   real(default) :: error_goal = 0._default
   real(default) :: rel_error_goal = 0._default
 contains
 ⟨vamp2: vamp2 config: TBP⟩
end type vamp2_config_t
```

Write.

⟨*vamp2: vamp2 config: TBP*⟩≡

```
procedure, public :: write => vamp2_config_write
```

⟨*vamp2: procedures*⟩+≡

```
subroutine vamp2_config_write (self, unit, indent)
  class(vamp2_config_t), intent(in) :: self
  integer, intent(in), optional :: unit
  integer, intent(in), optional :: indent
  integer :: u, ind
  u = given_output_unit (unit)
  ind = 0; if (present (indent)) ind = indent
  call self%vegas_config_t%write (unit, indent)
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Number of channels = ", self%n_channel
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Min. number of calls per channel (setting calls) = ", &
       self%n_calls_min_per_channel
  call write_indent (u, ind)
  write (u, "(2x,A,I0)") "Threshold number of calls (adapting weights) = ", &
       self%n_calls_threshold
  call write_indent (u, ind)
  write (u, "(2x,A,L1)") "Stratified = ", self%stratified
  call write_indent (u, ind)
  write (u, "(2x,A," // FMT_14 // ")") "Adaption power (beta) = ", &
       self%beta
  if (self%accuracy_goal > 0) then
     call write_indent (u, ind)
     write (u, "(2x,A," // FMT_14 // ")") "accuracy_goal         = ", &
          self%accuracy_goal
  end if
  if (self%error_goal > 0) then
     call write_indent (u, ind)
     write (u, "(2x,A," // FMT_14 // ")") "error_goal            = ", &
          self%error_goal
  end if
  if (self%rel_error_goal > 0) then
     call write_indent (u, ind)
     write (u, "(2x,A," // FMT_14 // ")") "rel_error_goal        = ", &
          self%rel_error_goal
  end if
end subroutine vamp2_config_write
```

19.12.3 Type: vamp2_result_t

This is a transparent container which incorporates and extends the definitions of `vegas_result_t`.

⟨*vamp2: public*⟩+≡

```
public :: vamp2_result_t
```

⟨*vamp2: types*⟩+≡

```
type, extends(vegas_result_t) :: vamp2_result_t
 contains
```

```
  ⟨vamp2: vamp2 result: TBP⟩
end type vamp2_result_t
```

Output.

⟨*vamp2: vamp2 result: TBP*⟩≡

```
procedure, public :: write => vamp2_result_write
```

⟨*vamp2: procedures*⟩+≡

```
subroutine vamp2_result_write (self, unit, indent)
  class(vamp2_result_t), intent(in) :: self
  integer, intent(in), optional :: unit
  integer, intent(in), optional :: indent
  integer :: u, ind
  u = given_output_unit (unit)
  ind = 0; if (present (indent)) ind = indent
  call self%vegas_result_t%write (unit, indent)
end subroutine vamp2_result_write
```

19.12.4 Type: vamp2_t

⟨*vamp2: public*⟩+≡

```
public :: vamp2_t
```

⟨*vamp2: types*⟩+≡

```
type :: vamp2_t
   private
   type(vamp2_config_t) :: config
   type(vegas_t), dimension(:), allocatable :: integrator
   real(default), dimension(:), allocatable :: weight
   real(default), dimension(:), allocatable :: integral
   real(default), dimension(:), allocatable :: variance
   real(default), dimension(:), allocatable :: efficiency
   type(vamp2_result_t) :: result
 contains
 ⟨vamp2: vamp2: TBP⟩
end type vamp2_t
```

⟨*vamp2: interfaces*⟩+≡

```
interface vamp2_t
   module procedure vamp2_init
end interface vamp2_t
```

Constructor.

⟨*vamp2: procedures*⟩+≡

```
type(vamp2_t) function vamp2_init (n_channel, n_dim, alpha, beta, n_bins_max,&
     & n_calls_min_per_channel, iterations, mode) result (self)
  integer, intent(in) :: n_channel
  integer, intent(in) :: n_dim
  integer, intent(in), optional :: n_bins_max
  integer, intent(in), optional :: n_calls_min_per_channel
  real(default), intent(in), optional :: alpha
  real(default), intent(in), optional :: beta
```

```
    integer, intent(in), optional :: iterations
    integer, intent(in), optional :: mode
    integer :: ch
    self%config%n_dim = n_dim
    self%config%n_channel = n_channel
    if (present (n_bins_max)) self%config%n_bins_max = n_bins_max
    if (present (n_calls_min_per_channel)) self%config%n_calls_min_per_channel = n_calls_min_per_c
    if (present (alpha)) self%config%alpha = alpha
    if (present (beta)) self%config%beta = beta
    if (present (iterations)) self%config%iterations = iterations
    if (present (mode)) self%config%mode = mode
    allocate (self%integrator(n_channel))
    allocate (self%weight(n_channel), source=0._default)
    allocate (self%integral(n_channel), source=0._default)
    allocate (self%variance(n_channel), source=0._default)
    allocate (self%efficiency(n_channel), source=0._default)
    do ch = 1, n_channel
       self%integrator(ch) = vegas_t (n_dim, alpha, n_bins_max, 1, mode)
    end do
    self%weight = 1. / self%config%n_channel
    call self%reset_result ()
  end function vamp2_init
```

⟨*vamp2: vamp2: TBP*⟩≡

```
  procedure, public :: final => vamp2_final
```

⟨*vamp2: procedures*⟩+≡

```
  subroutine vamp2_final (self)
    class(vamp2_t), intent(inout) :: self
    integer :: ch
    do ch = 1, self%config%n_channel
       call self%integrator(ch)%final ()
    end do
  end subroutine vamp2_final
```

Output.

⟨*vamp2: vamp2: TBP*⟩+≡

```
  procedure, public :: write => vamp2_write
```

⟨*vamp2: procedures*⟩+≡

```
  subroutine vamp2_write (self, unit, indent)
    class(vamp2_t), intent(in) :: self
    integer, intent(in), optional :: unit
    integer, intent(in), optional :: indent
    integer :: u, ind, ch
    u = given_output_unit (unit)
    ind = 0; if (present (indent)) ind = indent
    call write_indent (u, ind)
    write (u, "(A)") "VAMP2: VEGAS AMPlified 2"
    call write_indent (u, ind)
    call self%config%write (unit, indent)
    call self%result%write (unit, indent)
  end subroutine vamp2_write
```

Get the config object.

⟨*vamp2: vamp2: TBP*⟩+≡
```
procedure, public :: get_config => vamp2_get_config
```

⟨*vamp2: procedures*⟩+≡
```
subroutine vamp2_get_config (self, config)
  class(vamp2_t), intent(in) :: self
  type(vamp2_config_t), intent(out) :: config
  config = self%config
end subroutine vamp2_get_config
```

Set non-runtime dependent configuration. It will no be possible to change `n_bins_max`.

⟨*vamp2: vamp2: TBP*⟩+≡
```
procedure, public :: set_config => vamp2_set_config
```

⟨*vamp2: procedures*⟩+≡
```
subroutine vamp2_set_config (self, config)
  class(vamp2_t), intent(inout) :: self
  class(vamp2_config_t), intent(in) :: config
  integer :: ch
  self%config%n_calls_min_per_channel = config%n_calls_min_per_channel
  self%config%n_calls_threshold = config%n_calls_threshold
  self%config%n_calls_min = config%n_calls_min
  self%config%beta = config%beta
  self%config%accuracy_goal = config%accuracy_goal
  self%config%error_goal = config%error_goal
  self%config%rel_error_goal = config%rel_error_goal
  do ch = 1, self%config%n_channel
     call self%integrator(ch)%set_config (config)
  end do
end subroutine vamp2_set_config
```

Set the overall number of calls. The number of calls each channel is scaled by the channel weights

$$N_i = \alpha_i N. \tag{19.6}$$

⟨*vamp2: vamp2: TBP*⟩+≡
```
procedure, public :: set_calls => vamp2_set_n_calls
```

⟨*vamp2: procedures*⟩+≡
```
subroutine vamp2_set_n_calls (self, n_calls)
  class(vamp2_t), intent(inout) :: self
  integer, intent(in) :: n_calls
  integer :: ch
  self%config%n_calls_min = self%config%n_calls_min_per_channel &
       & * self%config%n_channel
  self%config%n_calls = max(n_calls, self%config%n_calls_min)
  if (self%config%n_calls > n_calls) then
     write (msg_buffer, "(A,I0)") "VAMP2: [set_calls] number of calls too few,&
          & reset to = ", self%config%n_calls
     call msg_message ()
  end if
  do ch = 1, self%config%n_channel
     call self%integrator(ch)%set_calls (max (nint (self%config%n_calls *&
```

```
            & self%weight(ch)), self%config%n_calls_min_per_channel))
      end do
    end subroutine vamp2_set_n_calls
```

Set limits. We only support same limits for all channels.

⟨*vamp2: vamp2: TBP*⟩+≡

```
    procedure, public :: set_limits => vamp2_set_limits
```

⟨*vamp2: procedures*⟩+≡

```
    subroutine vamp2_set_limits (self, x_upper, x_lower)
      class(vamp2_t), intent(inout) :: self
      real(default), dimension(:), intent(in) :: x_upper
      real(default), dimension(:), intent(in) :: x_lower
      integer :: ch
      do ch = 1, self%config%n_channel
         call self%integrator(ch)%set_limits (x_upper, x_lower)
      end do
    end subroutine vamp2_set_limits
```

Get the cumulative result of the integration. Recalculate weighted average of the integration.

⟨*vamp2: vamp2: TBP*⟩+≡

```
    procedure, public :: get_integral => vamp2_get_integral
```

⟨*vamp2: procedures*⟩+≡

```
    elemental real(default) function vamp2_get_integral (self) result (integral)
      class(vamp2_t), intent(in) :: self
      integral = 0.
      if (self%result%sum_wgts > 0.) then
         integral = self%result%sum_int_wgtd / self%result%sum_wgts
      end if
    end function vamp2_get_integral
```

Get the cumulative variance of the integration. Recalculate the variance.

⟨*vamp2: vamp2: TBP*⟩+≡

```
    procedure, public :: get_variance => vamp2_get_variance
```

⟨*vamp2: procedures*⟩+≡

```
    elemental real(default) function vamp2_get_variance (self) result (variance)
      class(vamp2_t), intent(in) :: self
      variance = 0.
      if (self%result%sum_wgts > 0.) then
         variance = 1.0 / self%result%sum_wgts
      end if
    end function vamp2_get_variance
```

Get efficiency.

⟨*vamp2: vamp2: TBP*⟩+≡

```
    procedure, public :: get_efficiency => vamp2_get_efficiency
```

⟨*vamp2: procedures*⟩+≡

```
elemental real(default) function vamp2_get_efficiency (self) result (efficiency)
  class(vamp2_t), intent(in) :: self
  efficiency = 0.
  if (self%result%efficiency > 0.) then
     efficiency = self%result%efficiency
  end if
end function vamp2_get_efficiency
```

Adapt. We adapt the weights due the contribution of variances with $\beta > 0$.

$$\alpha_i = \frac{\alpha_i V_i^\beta}{\sum_i \alpha_i V_i^\beta} \tag{19.7}$$

If `n_calls_threshold` is set, we rescale the weights in such a way, that the `n_calls` for each channel are greater than `n_calls_threshold`. We calculate the distance of the weights to the `weight_min` and reset those weights which are less than `weight_mins` to this value. The other values are accordingly resized to fit the boundary condition of the partition of unity.

⟨*vamp2: vamp2: TBP*⟩+≡

```
procedure, private :: adapt_weights => vamp2_adapt_weights
```

⟨*vamp2: procedures*⟩+≡

```
subroutine vamp2_adapt_weights (self)
  class(vamp2_t), intent(inout) :: self
  integer :: ch
  integer :: n_weights_underflow
  real(default) :: weight_min, sum_weights_underflow
  self%weight = self%weight * self%integrator%get_variance ()**self%config%beta
  if (sum (self%weight) == 0) self%weight = real(self%config%n_calls, default)
  self%weight = self%weight / sum(self%weight)
  if (self%config%n_calls_threshold /= 0) then
     weight_min = real(self%config%n_calls_threshold, default) &
          & / self%config%n_calls
     sum_weights_underflow = sum (self%weight, self%weight < weight_min)
     n_weights_underflow = count (self%weight < weight_min)
     where (self%weight < weight_min)
        self%weight = weight_min
     elsewhere
        self%weight = self%weight * (1. - n_weights_underflow * weight_min) &
             & / (1. - sum_weights_underflow)
     end where
  end if
  call self%set_calls (self%config%n_calls)
end subroutine vamp2_adapt_weights
```

Reset the cumulative result.

⟨*vamp2: vamp2: TBP*⟩+≡

```
procedure, public :: reset_result => vamp2_reset_result
```

⟨*vamp2: procedures*⟩+≡

```
subroutine vamp2_reset_result (self)
  class(vamp2_t), intent(inout) :: self
  self%result%sum_int_wgtd = 0.
```

```
    self%result%sum_wgts = 0.
    self%result%sum_chi = 0.
    self%result%it_num = 0
    self%result%samples = 0
    self%result%chi2 = 0
    self%result%efficiency = 0.
  end subroutine vamp2_reset_result
```

Integrate. We integrate each channel separately and combine the results

$$I = \sum_i \alpha_i I_i, \tag{19.8}$$

$$\sigma^2 = \sum_i \alpha_i^2 \sigma_i^2. \tag{19.9}$$

Although, the (population) variance is given by

$$\begin{aligned} \sigma^2 &= \frac{1}{N}\left(\sum_i \alpha_i I_i^2 - I^2\right) \\ &= \frac{1}{N-1}\left(\sum_i \left(N_i\sigma_i^2 + I_i^2\right) - I^2\right) \\ &= \frac{1}{N-1}\left(\sum_i \alpha_i\sigma_i^2 + \alpha_i I_i^2 - I^2\right), \end{aligned} \tag{19.10}$$

where we used $\sigma_i^2 = \frac{1}{N}\left(\langle I_i^2\rangle - \langle I_i\rangle^2\right)$, we use the approximation for numeric stability. The population variance relates to sample variance

$$s^2 = \frac{n}{n-1}\sigma^2, \tag{19.11}$$

which gives an unbiased error estimate.

Beside those adaption to multichannel, the overall processing of `total_integral`, `total_sq_integral` and `total_variance` is the same as in `vegas_integrate`.

⟨*vamp2: vamp2: TBP*⟩+≡

```
  procedure, public :: integrate => vamp2_integrate
```

⟨*vamp2: procedures*⟩+≡

```
  subroutine vamp2_integrate (self, func, rng, iterations, opt_reset_result,&
      & opt_refine_grid, opt_adapt_weight, opt_verbose, result, abserr)
    class(vamp2_t), intent(inout) :: self
    class(vamp2_func_t), intent(inout) :: func
    class(rng_t), intent(inout) :: rng
    integer, intent(in), optional :: iterations
    logical, intent(in), optional :: opt_reset_result
    logical, intent(in), optional :: opt_refine_grid
    logical, intent(in), optional :: opt_adapt_weight
    logical, intent(in), optional :: opt_verbose
    real(default), optional, intent(out) :: result, abserr
    integer :: it, ch
    real(default) :: total_integral, total_sq_integral, total_variance, chi, wgt
    real(default) :: cumulative_int, cumulative_std
```

```
logical :: reset_result = .true.
logical :: adapt_weight = .true.
logical :: refine_grid = .true.
logical :: verbose = .false.
if (present (iterations)) self%config%iterations = iterations
if (present (opt_reset_result)) reset_result = opt_reset_result
if (present (opt_adapt_weight)) adapt_weight = opt_adapt_weight
if (present (opt_refine_grid)) refine_grid = opt_refine_grid
if (present (opt_verbose)) verbose = opt_verbose
cumulative_int = 0.
cumulative_std = 0.
if (reset_result) call self%reset_result
if (verbose) then
   call msg_message ("Results: [it, calls, integral, error, chi^2, eff.]")
end if
iteration: do it = 1, self%config%iterations
   total_integral = 0._default
   total_sq_integral = 0._default
   total_variance = 0._default
   self%integral = 0._default
   self%variance = 0._default
   do ch = 1, self%config%n_channel
      func%wi(ch) = self%weight(ch)
      func%grids(ch) = self%integrator(ch)%get_grid ()
   end do
   channel: do ch = 1, self%config%n_channel
      ! Oh, do me! Integrate me! Integrate me, soooo hard!
      call func%set_channel (ch)
      call self%integrator(ch)%integrate ( &
           & func, rng, iterations, opt_refine_grid = .false., opt_verbose = verbose)
      total_integral = total_integral &
           & + self%weight(ch) * self%integrator(ch)%get_integral ()
      total_sq_integral = total_sq_integral &
           & + self%weight(ch) * self%integrator(ch)%get_integral ()**2
      total_variance = total_variance &
           & + self%weight(ch)**2 * self%config%n_calls * self%integrator(ch)%get_variance ()
   end do channel
   associate (result => self%result)
     ! a**2 - b**2 = (a - b) * (a + b)
     total_variance = sqrt (total_variance + total_sq_integral)
     total_variance = 1. / self%config%n_calls * &
          & (total_variance + total_integral) * (total_variance - total_integral)
     ! Ensure variance is always positive and larger than zero
     if (total_variance < tiny (1._default) / epsilon (1._default) * max (total_integral**2, 1
        total_variance = tiny (1._default) / epsilon (1._default) * max (total_integral**2, 1.
     end if
     wgt = 1. / total_variance
     result%result = total_integral
     result%std = sqrt (total_variance)
     result%samples = result%samples + 1
     if (result%samples == 1) then
        result%chi2 = 0._default
     else
        chi = total_integral
```

```
            if (result%sum_wgts > 0) chi = chi - result%sum_int_wgtd / result%sum_wgts
            result%chi2 = result%chi2 * (result%samples - 2.0_default)
            result%chi2 = (wgt / (1._default + (wgt / result%sum_wgts))) &
                 & * chi**2
            result%chi2 = result%chi2 / (result%samples - 1._default)
         end if
         result%sum_wgts = result%sum_wgts + wgt
         result%sum_int_wgtd = result%sum_int_wgtd + (total_integral * wgt)
         result%sum_chi = result%sum_chi + (total_sq_integral * wgt)
         result%max_abs_f = dot_product (self%weight * self%config%n_calls, self%integrator%get_ma
         result%max_abs_f_pos = dot_product (self%weight * self%config%n_calls, self%integrator%ge
         result%max_abs_f_neg = dot_product (self%weight * self%config%n_calls, self%integrator%ge
         result%efficiency = 0.
         if (result%max_abs_f > 0.) then
            result%efficiency = dot_product (self%weight, &
                 & (self%integrator%get_efficiency () * self%weight * self%config%n_calls * self%i
                 & / result%max_abs_f
            ! TODO pos. or. negative efficiency would be very nice.
         end if
         cumulative_int = result%sum_int_wgtd / result%sum_wgts
         cumulative_std = sqrt (1. / result%sum_wgts)
         if (verbose) then
            write (msg_buffer, "(I0,1x,I0,1x, 4(" // FMT_14 // ",1x))") &
                 & it, self%config%n_calls, cumulative_int, cumulative_std, &
                 & self%result%chi2, self%result%efficiency
            call msg_message ()
         end if
      end associate
      if (adapt_weight) then
         call self%adapt_weights ()
      end if
      if (refine_grid) then
         do ch = 1, self%config%n_channel
            call self%integrator(ch)%refine ()
         end do
      end if
   end do iteration
   if (present (result)) result = cumulative_int
   if (present (abserr)) abserr = abs (cumulative_std)
end subroutine vamp2_integrate
```

Generate event from multi-channel weight $w(x) = f(x)/g(x)$.

We select a channel using the a-priori weights and $f_i^{\max}$, to flatten possible unbalanced channel weight(s). The reade may have mercy for the completely misuse of weight.

⟨*vamp2: vamp2: TBP*⟩+≡

```
procedure, public :: generate_event => vamp2_generate_event
```

⟨*vamp2: procedures*⟩+≡

```
subroutine vamp2_generate_event (self, func, rng, x, opt_event_weight, opt_event_excess)
  class(vamp2_t), intent(inout) :: self
  class(vamp2_func_t), intent(inout) :: func
  class(rng_t), intent(inout) :: rng
  real(default), dimension(self%config%n_dim), intent(out)  :: x
```

```
real(default), intent(out), optional :: opt_event_weight
real(default), intent(out), optional :: opt_event_excess
integer :: ch, i
real(default) :: r, event_weight, max_abs_f
real(default), dimension(self%config%n_channel) :: weight
if (any (self%integrator%get_max_abs_f () > 0)) then
   weight = self%weight * self%integrator%get_max_abs_f ()
else
   weight = self%weight
end if
weight = weight / sum (weight)
call rng%generate (r)
nchannel: do ch = 1, self%config%n_channel
   r = r - weight(ch)
   if (r <= 0._default) exit nchannel
end do nchannel
ch = min (ch, self%config%n_channel)
call func%set_channel (ch)
! TODO move to a separat procedure and let this be done at initialisation
do i = 1, self%config%n_channel
   func%wi(i) = self%weight(i)
   func%grids(i) = self%integrator(i)%get_grid ()
end do
if (present (opt_event_excess)) opt_event_excess = 0
generate: do
   call self%integrator(ch)%generate_event (func, rng, x, event_weight)
   if (present (opt_event_weight)) then
      opt_event_weight = event_weight * self%weight(ch) / weight(ch)
      exit generate
   end if
   if (event_weight > 0.) then
      if (abs (event_weight) > self%integrator(ch)%get_max_abs_f_pos ()) then
         if (present (opt_event_excess)) then
            opt_event_excess = event_weight / self%integrator(ch)%get_max_abs_f_pos () - 1._de
         else
            write (msg_buffer, "(A,1X," // FMT_14 // ",A)") "[VAMP2] Event&
                 & generation: weight > 1 (", self%result%max_abs_f_pos, ")"
            call msg_warning ()
         end if
      end if
      max_abs_f = self%integrator(ch)%get_max_abs_f_pos ()
   else
      if (abs (event_weight) > self%integrator(ch)%get_max_abs_f_neg ()) then
         if (present (opt_event_excess)) then
            opt_event_excess = event_weight / self%integrator(ch)%get_max_abs_f_neg () - 1._de
         else
            write (msg_buffer, "(A,1X," // FMT_14 // ",A)") "[VAMP2] Event&
                 & generation: weight > 1 (", self%result%max_abs_f_neg, ")"
            call msg_warning ()
         end if
      end if
      max_abs_f = self%integrator(ch)%get_max_abs_f_neg ()
   end if
   call rng%generate (r)
```

```
      ! Do not use division, because max_abs_f could be zero.
      if (max_abs_f * r <= abs(event_weight)) then
         exit generate
      end if
   end do generate
end subroutine vamp2_generate_event
```

Write grids to unit.

⟨*vamp2: parameters*⟩≡

```
character(len=*), parameter, private :: &
   descr_fmt =         "(1X,A)", &
   integer_fmt =       "(1X,A18,1X,I15)", &
   integer_array_fmt = "(1X,I18,1X,I15)", &
   logical_fmt =       "(1X,A18,1X,L1)", &
   double_fmt =        "(1X,A18,1X," // FMT_14 // ")", &
   double_array_fmt =  "(1X,I18,1X," // FMT_14 // ")", &
   double_array2_fmt =  "(1X,2(1X,I8),1X," // FMT_14 // ")"
```

⟨*vamp2: vamp2: TBP*⟩+≡

```
procedure, public :: write_grids => vamp2_write_grids
```

⟨*vamp2: procedures*⟩+≡

```
subroutine vamp2_write_grids (self, unit)
  class(vamp2_t), intent(in) :: self
  integer, intent(in), optional :: unit
  integer :: u
  integer :: ch
  u = given_output_unit (unit)
  write (u, descr_fmt) "begin type(vamp2_t)"
  write (u, integer_fmt) "n_channel =", self%config%n_channel
  write (u, integer_fmt) "n_dim =", self%config%n_dim
  write (u, integer_fmt) "n_calls_min_ch =", self%config%n_calls_min_per_channel
  write (u, integer_fmt) "n_calls_thres =", self%config%n_calls_threshold
  write (u, logical_fmt) "stratified =", self%config%stratified
  write (u, double_fmt) "alpha =", self%config%alpha
  write (u, double_fmt) "beta =", self%config%beta
  write (u, integer_fmt) "n_bins_max =", self%config%n_bins_max
  write (u, integer_fmt) "iterations =", self%config%iterations
  write (u, integer_fmt) "n_calls =", self%config%n_calls
  write (u, integer_fmt) "it_start =", self%result%it_start
  write (u, integer_fmt) "it_num =", self%result%it_num
  write (u, integer_fmt) "samples =", self%result%samples
  write (u, double_fmt) "sum_int_wgtd =", self%result%sum_int_wgtd
  write (u, double_fmt) "sum_wgts =", self%result%sum_wgts
  write (u, double_fmt) "sum_chi =", self%result%sum_chi
  write (u, double_fmt) "chi2 =", self%result%chi2
  write (u, double_fmt) "efficiency =", self%result%efficiency
  write (u, double_fmt) "efficiency =", self%result%efficiency_pos
  write (u, double_fmt) "efficiency =", self%result%efficiency_neg
  write (u, double_fmt) "max_abs_f =", self%result%max_abs_f
  write (u, double_fmt) "max_abs_f_pos =", self%result%max_abs_f_pos
  write (u, double_fmt) "max_abs_f_neg =", self%result%max_abs_f_neg
  write (u, double_fmt) "result =", self%result%result
  write (u, double_fmt) "std =", self%result%std
  write (u, descr_fmt) "begin weight"
```

```
    do ch = 1, self%config%n_channel
       write (u, double_array_fmt) ch, self%weight(ch)
    end do
    write (u, descr_fmt) "end weight"
    write (u, descr_fmt) "begin integrator"
    do ch = 1, self%config%n_channel
       call self%integrator(ch)%write_grid (unit)
    end do
    write (u, descr_fmt) "end integrator"
    write (u, descr_fmt) "end type(vamp2_t)"
  end subroutine vamp2_write_grids
```

Read grids from unit.

⟨*vamp2: vamp2: TBP*⟩+≡

```
  procedure, public :: read_grids => vamp2_read_grids
```

⟨*vamp2: procedures*⟩+≡

```
  subroutine vamp2_read_grids (self, unit)
    class(vamp2_t), intent(out) :: self
    integer, intent(in), optional :: unit
    integer :: u
    integer :: ibuffer, jbuffer, ch
    character(len=80) :: buffer
    read (unit, descr_fmt) buffer
    read (unit, integer_fmt) buffer, ibuffer
    read (unit, integer_fmt) buffer, jbuffer
    select type (self)
    type is (vamp2_t)
       self = vamp2_t (n_channel = ibuffer, n_dim = jbuffer)
    end select
    read (unit, integer_fmt) buffer, self%config%n_calls_min_per_channel
    read (unit, integer_fmt) buffer, self%config%n_calls_threshold
    read (unit, logical_fmt) buffer, self%config%stratified
    read (unit, double_fmt) buffer, self%config%alpha
    read (unit, double_fmt) buffer, self%config%beta
    read (unit, integer_fmt) buffer, self%config%n_bins_max
    read (unit, integer_fmt) buffer, self%config%iterations
    read (unit, integer_fmt) buffer, self%config%n_calls
    read (unit, integer_fmt) buffer, self%result%it_start
    read (unit, integer_fmt) buffer, self%result%it_num
    read (unit, integer_fmt) buffer, self%result%samples
    read (unit, double_fmt) buffer, self%result%sum_int_wgtd
    read (unit, double_fmt) buffer, self%result%sum_wgts
    read (unit, double_fmt) buffer, self%result%sum_chi
    read (unit, double_fmt) buffer, self%result%chi2
    read (unit, double_fmt) buffer, self%result%efficiency
    read (unit, double_fmt) buffer, self%result%efficiency_pos
    read (unit, double_fmt) buffer, self%result%efficiency_neg
    read (unit, double_fmt) buffer, self%result%max_abs_f
    read (unit, double_fmt) buffer, self%result%max_abs_f_pos
    read (unit, double_fmt) buffer, self%result%max_abs_f_neg
    read (unit, double_fmt) buffer, self%result%result
    read (unit, double_fmt) buffer, self%result%std
    read (unit, descr_fmt) buffer
```

```
    do ch = 1, self%config%n_channel
       read (unit, double_array_fmt) ibuffer, self%weight(ch)
    end do
    read (unit, descr_fmt) buffer
    read (unit, descr_fmt) buffer
    do ch = 1, self%config%n_channel
       call self%integrator(ch)%read_grid (unit)
    end do
    read (unit, descr_fmt) buffer
    read (unit, descr_fmt) buffer
  end subroutine vamp2_read_grids
```

19.13 Unit tests

Test module, followed by the corresponding implementation module.

⟨vamp2_ut.f90⟩≡

```
⟨File header⟩

module vamp2_ut
  use unit_tests
  use vamp2_uti

⟨Standard module head⟩

⟨vamp2: public test⟩

contains
⟨vamp2: test driver⟩
end module vamp2_ut
```

⟨vamp2_uti.f90⟩≡

```
⟨File header⟩

module vamp2_uti
⟨Use kinds⟩
  use io_units
  use constants, only: pi
  use format_defs, only: FMT_12
  use rng_base
  use rng_tao
  use rng_stream
  use vegas, only: vegas_func_t
  use vamp2

⟨Standard module head⟩

⟨vamp2: test declaration⟩

⟨vamp2: test types⟩

contains
⟨vamp2: tests⟩
```

```
  end module vamp2_uti
```

API: driver for the unit tests below.

⟨*vamp2: public test*⟩≡

```
    public :: vamp2_test
```

⟨*vamp2: test driver*⟩≡

```
    subroutine vamp2_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
      ⟨vamp2: execute tests⟩
    end subroutine vamp2_test
```

Test function

We use the example from the Monte Carlo Examples of the GSL library

$$I = \int_{-pi}^{+pi} dk_x/(2pi) \int_{-pi}^{+pi} dk_y/(2pi) \int_{-pi}^{+pi} dk_z/(2pi) 1/(1-cos(k_x)cos(k_y)cos(k_z)). \tag{19.12}$$

The integral is reduced to region (0,0,0) → (π, π, π) and multiplied by 8.

⟨*vamp2: test types*⟩≡

```
     type, extends (vamp2_func_t) :: vamp2_test_func_t
       !
     contains
     ⟨vamp2: vamp2 test func: TBP⟩
  end type vamp2_test_func_t
```

⟨*vamp2: vamp2 test func: TBP*⟩≡

```
    procedure, public :: evaluate_maps => vamp2_test_func_evaluate_maps
```

⟨*vamp2: tests*⟩≡

```
    subroutine vamp2_test_func_evaluate_maps (self, x)
      class(vamp2_test_func_t), intent(inout) :: self
      real(default), dimension(:), intent(in) :: x
      self%xi(:, 1) = x
      self%det(1) = 1
      self%valid_x = .true.
    end subroutine vamp2_test_func_evaluate_maps
```

Evaluate the integrand.

⟨*vamp2: vamp2 test func: TBP*⟩+≡

```
    procedure, public :: evaluate_func => vamp2_test_func_evaluate
```

⟨*vamp2: tests*⟩+≡

```
    real(default) function vamp2_test_func_evaluate (self, x) result (f)
      class(vamp2_test_func_t), intent(in) :: self
      real(default), dimension(:), intent(in) :: x
      f = 1.0 / (pi**3)
      f = f / ( 1.0 - cos (x(1)) * cos (x(2)) * cos (x(3)))
    end function vamp2_test_func_evaluate
```

The second test function implements

$$f(\vec{x}) = 4\sin^2(\pi x_1)\sin^2(\pi x_2) + 2\sin^2(\pi v), \tag{19.13}$$

where

$$x = u^v y = u^{1-v} \tag{19.14}$$

$$u = xyv = \frac{1}{2}\left(1 + \frac{\log(x/y}{\log(xy)}\right). \tag{19.15}$$

The jacobian is $\frac{\partial(x,y)}{\partial(u,v)}$.

⟨*vamp2: test types*⟩+≡

```
type, extends(vamp2_func_t) :: vamp2_test_func_2_t
   !
 contains
 ⟨vamp2: vamp2 test func 2: TBP⟩
end type vamp2_test_func_2_t
```

Evaluate maps.

⟨*vamp2: vamp2 test func 2: TBP*⟩≡

```
procedure :: evaluate_maps => vamp2_test_func_2_evaluate_maps
```

⟨*vamp2: tests*⟩+≡

```
subroutine vamp2_test_func_2_evaluate_maps (self, x)
  class(vamp2_test_func_2_t), intent(inout) :: self
  real(default), dimension(:), intent(in) :: x
  select case (self%current_channel)
  case (1)
     self%xi(:, 1) = x
     self%xi(1, 2) = x(1) * x(2)
     self%xi(2, 2) = 0.5 * ( 1. + log(x(1) / x(2)) / log(x(1) * x(2)))
  case (2)
     self%xi(1, 1) = x(1)**x(2)
     self%xi(2, 1) = x(1)**(1. - x(2))
     self%xi(:, 2) = x
  end select
  self%det(1) = 1.
  self%det(2) = abs (log(self%xi(1, 2)))
  self%valid_x = .true.
end subroutine vamp2_test_func_2_evaluate_maps
```

Evaluate func.

⟨*vamp2: vamp2 test func 2: TBP*⟩+≡

```
procedure :: evaluate_func => vamp2_test_func_2_evaluate_func
```

⟨*vamp2: tests*⟩+≡

```
real(default) function vamp2_test_func_2_evaluate_func (self, x) result (f)
  class(vamp2_test_func_2_t), intent(in) :: self
  real(default), dimension(:), intent(in) :: x
  f = 4. * sin(pi * self%xi(1, 1))**2 * sin(pi * self%xi(2, 1))**2 + 2. * sin(pi * self%xi(2, 2)
end function vamp2_test_func_2_evaluate_func
```

MC Integrator check

We reproduce the first test case of VEGAS. Initialise the VAMP2 MC integrator and call to `vamp2_init_grid` for the initialisation of the grid.

⟨*vamp2: execute tests*⟩≡
```
call test (vamp2_1, "vamp2_1", "VAMP2 initialisation and&
     & grid preparation", u, results)
```

⟨*vamp2: test declaration*⟩≡
```
public :: vamp2_1
```

⟨*vamp2: tests*⟩+≡
```
subroutine vamp2_1 (u)
  integer, intent(in) :: u
  type(vamp2_t) :: mc_integrator
  class(rng_t), allocatable :: rng
  class(vamp2_func_t), allocatable :: func
  real(default), dimension(3), parameter :: x_lower = 0., &
       x_upper = pi
  real(default) :: result, abserr

  write (u, "(A)") "* Test output: vamp2_1"
  write (u, "(A)") "*   Purpose: initialise the VAMP2 MC integrator and the grid"
  write (u, "(A)")

  write (u, "(A)") "* Initialise random number generator (default seed)"
  write (u, "(A)")

  allocate (rng_stream_t :: rng)
  call rng%init ()

  call rng%write (u)

  write (u, "(A)")
  write (u, "(A)") "* Initialise MC integrator with n_channel = 1 and n_dim = 3"
  write (u, "(A)")

  allocate (vamp2_test_func_t :: func)
  call func%init (n_dim = 3, n_channel = 1)
  mc_integrator = vamp2_t (1, 3)
  call mc_integrator%write (u)

  write (u, "(A)")
  write (u, "(A)") "* Initialise grid with n_calls = 10000"
  write (u, "(A)")

  call mc_integrator%set_limits (x_lower, x_upper)
  call mc_integrator%set_calls (10000)

  write (u, "(A)")
  write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 10000 (Adaptation)"
  write (u, "(A)")

  call mc_integrator%integrate (func, rng, 3, result=result, abserr=abserr)
  write (u, "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr
```

```
    write (u, "(A)")
    write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 2000 (Precision)"
    write (u, "(A)")

    call mc_integrator%set_calls (2000)
    call mc_integrator%integrate (func, rng, 3, result=result, abserr=abserr)
    write (u,  "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr


    write (u, "(A)")
    write (u, "(A)") "* Write grids"
    write (u, "(A)")

    call mc_integrator%write_grids (u)

    write (u, "(A)")
    write (u, "(A)") "* Cleanup"

    call mc_integrator%final ()
    call rng%final ()
    deallocate (rng)
  end subroutine vamp2_1
```

Integrate a function with two-dimensional argument and two channels.

⟨*vamp2: execute tests*⟩+≡

```
  call test (vamp2_2, "vamp2_2", "VAMP2 intgeration of two-dimensional &
      & function with two channels", u, results)
```

⟨*vamp2: test declaration*⟩+≡

```
  public :: vamp2_2
```

⟨*vamp2: tests*⟩+≡

```
  subroutine vamp2_2 (u)
    integer, intent(in) :: u
    type(vamp2_t) :: mc_integrator
    class(rng_t), allocatable :: rng
    class(vamp2_func_t), allocatable :: func
    real(default), dimension(2), parameter :: x_lower = 0., &
        x_upper = 1.
    real(default) :: result, abserr

    write (u, "(A)") "* Test output: vamp2_2"
    write (u, "(A)") "*   Purpose:  intgeration of two-dimensional &
       & function with two channels"
    write (u, "(A)")

    write (u, "(A)") "* Initialise random number generator (default seed)"
    write (u, "(A)")

    allocate (rng_stream_t :: rng)
    call rng%init ()

    call rng%write (u)
```

```
    write (u, "(A)")
    write (u, "(A)") "* Initialise MC integrator with n_channel = 1 and n_dim = 3"
    write (u, "(A)")

    allocate (vamp2_test_func_2_t :: func)
    call func%init (n_dim = 2, n_channel = 2)
    mc_integrator = vamp2_t (2, 2)
    call mc_integrator%write (u)

    write (u, "(A)")
    write (u, "(A)") "* Initialise grid with n_calls = 10000"
    write (u, "(A)")

    call mc_integrator%set_limits (x_lower, x_upper)
    call mc_integrator%set_calls (1000)

    write (u, "(A)")
    write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 10000 (Adaptation)"
    write (u, "(A)")

    call mc_integrator%integrate (func, rng, 3, opt_verbose = .true., result=result, abserr=abserr
    write (u,  "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr

    write (u, "(A)")
    write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 2000 (Precision)"
    write (u, "(A)")

    call mc_integrator%set_calls (200)
    call mc_integrator%integrate (func, rng, 3, opt_verbose = .true., result=result, abserr=abserr
    write (u,  "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr


    write (u, "(A)")
    write (u, "(A)") "* Write grids"
    write (u, "(A)")

    call mc_integrator%write_grids (u)

    write (u, "(A)")
    write (u, "(A)") "* Cleanup"

    call mc_integrator%final ()
    call rng%final ()
    deallocate (rng)
  end subroutine vamp2_2
```

Integrate a function with two-dimensional argument and two channels.

⟨*vamp2: execute tests*⟩+≡

```
  call test (vamp2_3, "vamp2_3", "VAMP2 intgeration of two-dimensional &
      & function with two channels", u, results)
```

⟨*vamp2: test declaration*⟩+≡

```
  public :: vamp2_3
```

⟨*vamp2: tests*⟩+≡

```
subroutine vamp2_3 (u)
  integer, intent(in) :: u
  type(vamp2_t) :: mc_integrator
  class(rng_t), allocatable :: rng
  class(vamp2_func_t), allocatable :: func
  real(default), dimension(2), parameter :: x_lower = 0., &
       x_upper = 1.
  real(default) :: result, abserr
  integer :: unit

  write (u, "(A)") "* Test output: vamp2_3"
  write (u, "(A)") "*   Purpose:  intgeration of two-dimensional &
     & function with two channels"
  write (u, "(A)")

  write (u, "(A)") "* Initialise random number generator (default seed)"
  write (u, "(A)")

  allocate (rng_stream_t :: rng)
  call rng%init ()

  call rng%write (u)

  write (u, "(A)")
  write (u, "(A)") "* Initialise MC integrator with n_channel = 1 and n_dim = 3"
  write (u, "(A)")

  allocate (vamp2_test_func_2_t :: func)
  call func%init (n_dim = 2, n_channel = 2)
  mc_integrator = vamp2_t (2, 2)
  call mc_integrator%write (u)

  write (u, "(A)")
  write (u, "(A)") "* Initialise grid with n_calls = 10000"
  write (u, "(A)")

  call mc_integrator%set_limits (x_lower, x_upper)
  call mc_integrator%set_calls (1000)

  write (u, "(A)")
  write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 10000 (Adaptation)"
  write (u, "(A)")

  call mc_integrator%integrate (func, rng, 3, result=result, abserr=abserr)
  write (u,  "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr

  write (u, "(A)")
  write (u, "(A)") "* Write grid to file vegas_io.grid"
  write (u, "(A)")

  unit = free_unit ()
  open (unit, file = "vamp2_io.grids", &
       action = "write", status = "replace")
  call mc_integrator%write_grids (unit)
```

```
  close (unit)

  write (u, "(A)")
  write (u, "(A)") "* Read grid from file vegas_io.grid"
  write (u, "(A)")

  call mc_integrator%final ()
  open (unit, file = "vamp2_io.grids", &
       action = "read", status = "old")
  call mc_integrator%read_grids (unit)
  close (unit)

  write (u, "(A)")
  write (u, "(A)") "* Integrate with n_it = 3 and n_calls = 2000 (Precision)"
  write (u, "(A)")

  call mc_integrator%set_calls (200)
  call mc_integrator%integrate (func, rng, 3, result=result, abserr=abserr)
  write (u,  "(2x,A," // FMT_12 // ",A," // FMT_12 // ")") "Result: ", result, " +/- ", abserr


  write (u, "(A)")
  write (u, "(A)") "* Write grids"
  write (u, "(A)")

  call mc_integrator%write_grids (u)

  write (u, "(A)")
  write (u, "(A)") "* Cleanup"

  call mc_integrator%final ()
  call rng%final ()
  deallocate (rng)
end subroutine vamp2_3
```

Chapter 20

Multi-Channel Integration

The abstract representation of multi-channel Monte Carlo algorithms for integration and event generation.

Module `mci_base`: The abstract types and their methods. It provides a test integrator that is referenced in later unit tests.

iterations Container for defining integration call and pass settings.

integration_results This module handles results from integrating processes. It records passes and iterations, calculates statistical averages, and provides the user output of integration results.

These are the implementations:

Module `mci_midpoint`: A simple integrator that uses the midpoint rule to sample the integrand uniformly over the unit hypercube. There is only one integration channel, so this can be matched only to single-channel phase space.

Module `mci_vamp`: Interface for the VAMP package.

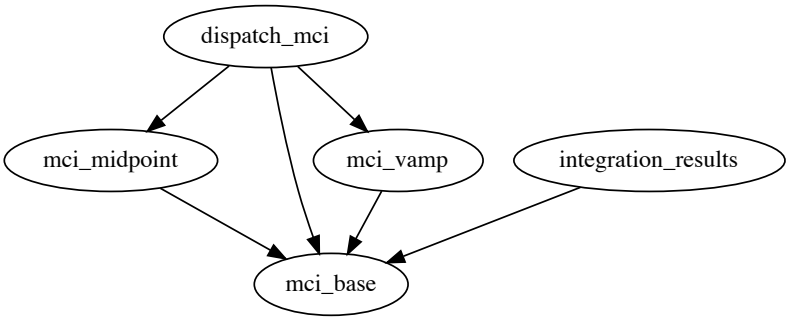

Figure 20.1: Module dependencies in `src/mci`.

20.1 Generic Integrator

This module provides a multi-channel integrator (MCI) base type, a corresponding configuration type, and methods for integration and event generation.

⟨mci_base.f90⟩≡

```
⟨File header⟩

module mci_base

  use kinds
  use io_units
  use format_utils, only: pac_fmt
  use format_defs, only: FMT_14, FMT_17
  use diagnostics
  use cputime
  use phs_base
  use rng_base

⟨Standard module head⟩

⟨MCI base: public⟩

⟨MCI base: types⟩

⟨MCI base: interfaces⟩

contains

⟨MCI base: procedures⟩

end module mci_base
```

20.1.1 MCI: integrator

The MCI object contains the methods for integration and event generation. For the actual work and data storage, it spawns an MCI instance object.

The base object contains the number of integration dimensions and the number of channels as configuration data. Further configuration data are stored in the concrete extensions.

The MCI sum contains all relevant information about the integrand. It can be used for comparing the current configuration against a previous one. If they match, we can skip an actual integration. (Implemented only for the VAMP version.)

There is a random-number generator (its state with associated methods) available as `rng`. It may or may not be used for integration. It will be used for event generation.

⟨*MCI base: public*⟩≡

```
public :: mci_t
```

⟨*MCI base: types*⟩≡

```
type, abstract :: mci_t
   integer :: n_dim = 0
```

```
    integer :: n_channel = 0
    integer :: n_chain = 0
    integer, dimension(:), allocatable :: chain
    real(default), dimension(:), allocatable :: chain_weights
    character(32) :: md5sum = ""
    logical :: integral_known = .false.
    logical :: error_known = .false.
    logical :: efficiency_known = .false.
    real(default) :: integral = 0
    real(default) :: error = 0
    real(default) :: efficiency = 0
    logical :: use_timer = .false.
    type(timer_t) :: timer
    class(rng_t), allocatable :: rng
  contains
  ⟨MCI base: mci: TBP⟩
end type mci_t
```

Finalizer: the random-number generator may need one.

⟨*MCI base: mci: TBP*⟩≡

```
procedure :: base_final => mci_final
procedure (mci_final), deferred :: final
```

⟨*MCI base: procedures*⟩≡

```
subroutine mci_final (object)
  class(mci_t), intent(inout) :: object
  if (allocated (object%rng))  call object%rng%final ()
end subroutine mci_final
```

Output: basic and extended output.

⟨*MCI base: mci: TBP*⟩+≡

```
procedure :: base_write => mci_write
procedure (mci_write), deferred :: write
```

⟨*MCI base: procedures*⟩+≡

```
subroutine mci_write (object, unit, pacify, md5sum_version)
  class(mci_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacify
  logical, intent(in), optional :: md5sum_version
  logical :: md5sum_ver
  integer :: u, i, j
  character(len=7) :: fmt
  call pac_fmt (fmt, FMT_17, FMT_14, pacify)
  u = given_output_unit (unit)
  md5sum_ver = .false.
  if (present (md5sum_version))  md5sum_ver = md5sum_version
  if (object%use_timer .and. .not. md5sum_ver) then
     write (u, "(2x)", advance="no")
     call object%timer%write (u)
  end if
  if (object%integral_known) then
     write (u, "(3x,A," // fmt // ")") &
          "Integral             = ", object%integral
```

```
    end if
    if (object%error_known) then
       write (u, "(3x,A," // fmt // ")") &
            "Error                = ", object%error
    end if
    if (object%efficiency_known) then
       write (u, "(3x,A," // fmt // ")")  &
            "Efficiency           = ", object%efficiency
    end if
    write (u, "(3x,A,I0)")  "Number of channels   = ", object%n_channel
    write (u, "(3x,A,I0)")  "Number of dimensions = ", object%n_dim
    if (object%n_chain > 0) then
       write (u, "(3x,A,I0)")  "Number of chains     = ", object%n_chain
       write (u, "(3x,A)")  "Chains:"
       do i = 1, object%n_chain
          write (u, "(5x,I0,':')", advance = "no")  i
          do j = 1, object%n_channel
             if (object%chain(j) == i) &
                  write (u, "(1x,I0)", advance = "no")  j
          end do
          write (u, "(A)")
       end do
    end if
  end subroutine mci_write
```

Print an informative message when starting integration.

⟨*MCI base: mci: TBP*⟩+≡

```
  procedure (mci_startup_message), deferred :: startup_message
  procedure :: base_startup_message => mci_startup_message
```

⟨*MCI base: procedures*⟩+≡

```
  subroutine mci_startup_message (mci, unit, n_calls)
    class(mci_t), intent(in) :: mci
    integer, intent(in), optional :: unit, n_calls
    if (mci%n_chain > 0) then
       write (msg_buffer, "(A,3(1x,I0,1x,A))") &
            "Integrator:", mci%n_chain, "chains,", &
            mci%n_channel, "channels,", &
            mci%n_dim, "dimensions"
    else
       write (msg_buffer, "(A,3(1x,I0,1x,A))") &
            "Integrator:", &
            mci%n_channel, "channels,", &
            mci%n_dim, "dimensions"
    end if
    call msg_message (unit = unit)
  end subroutine mci_startup_message
```

Dump type-specific info to a logfile.

⟨*MCI base: mci: TBP*⟩+≡

```
  procedure(mci_write_log_entry), deferred :: write_log_entry
```

⟨*MCI base: interfaces*⟩≡

```
  abstract interface
```

```
    subroutine mci_write_log_entry (mci, u)
      import
      class(mci_t), intent(in) :: mci
      integer, intent(in) :: u
    end subroutine mci_write_log_entry
end interface
```

In order to avoid dependencies on definite MCI implementations, we introduce a MD5 sum calculator.

⟨*MCI base: mci: TBP*⟩+≡

```
procedure(mci_compute_md5sum), deferred :: compute_md5sum
```

⟨*MCI base: interfaces*⟩+≡

```
abstract interface
    subroutine mci_compute_md5sum (mci, pacify)
      import
      class(mci_t), intent(inout) :: mci
      logical, intent(in), optional :: pacify
    end subroutine mci_compute_md5sum
end interface
```

Record the index of the MCI object within a process. For multi-component processes with more than one integrator, the integrator should know about its own index, so file names can be unique, etc. The default implementation does nothing, however.

⟨*MCI base: mci: TBP*⟩+≡

```
procedure :: record_index => mci_record_index
```

⟨*MCI base: procedures*⟩+≡

```
subroutine mci_record_index (mci, i_mci)
  class(mci_t), intent(inout) :: mci
  integer, intent(in) :: i_mci
end subroutine mci_record_index
```

There is no Initializer for the abstract type, but a generic setter for the number of channels and dimensions. We make two aliases available, to be able to override it.

⟨*MCI base: mci: TBP*⟩+≡

```
procedure :: set_dimensions => mci_set_dimensions
procedure :: base_set_dimensions => mci_set_dimensions
```

⟨*MCI base: procedures*⟩+≡

```
subroutine mci_set_dimensions (mci, n_dim, n_channel)
  class(mci_t), intent(inout) :: mci
  integer, intent(in) :: n_dim
  integer, intent(in) :: n_channel
  mci%n_dim = n_dim
  mci%n_channel = n_channel
end subroutine mci_set_dimensions
```

Declare particular dimensions as flat. This information can be used to simplify integration. When generating events, the flat dimensions should be sampled with uniform and uncorrelated distribution. It depends on the integrator what to do with that information.

⟨*MCI base: mci: TBP*⟩+≡
```
procedure (mci_declare_flat_dimensions), deferred :: declare_flat_dimensions
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
   subroutine mci_declare_flat_dimensions (mci, dim_flat)
     import
     class(mci_t), intent(inout) :: mci
     integer, dimension(:), intent(in) :: dim_flat
   end subroutine mci_declare_flat_dimensions
end interface
```

Declare particular channels as equivalent, possibly allowing for permutations or reflections of dimensions. We use the information stored in the `phs_channel_t` object array that the phase-space module provides.

(We do not test this here, deferring the unit test to the `mci_vamp` implementation where we actually use this feature.)

⟨*MCI base: mci: TBP*⟩+≡
```
procedure (mci_declare_equivalences), deferred :: declare_equivalences
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
   subroutine mci_declare_equivalences (mci, channel, dim_offset)
     import
     class(mci_t), intent(inout) :: mci
     type(phs_channel_t), dimension(:), intent(in) :: channel
     integer, intent(in) :: dim_offset
   end subroutine mci_declare_equivalences
end interface
```

Declare particular channels as chained together. The implementation may use this array for keeping their weights equal to each other, etc.

The chain array is an array sized by the number of channels. For each channel, there is an integer entry that indicates the correponding chains. The total number of chains is the maximum value of this entry.

⟨*MCI base: mci: TBP*⟩+≡
```
procedure :: declare_chains => mci_declare_chains
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_declare_chains (mci, chain)
  class(mci_t), intent(inout) :: mci
  integer, dimension(:), intent(in) :: chain
  allocate (mci%chain (size (chain)))
  mci%n_chain = maxval (chain)
  allocate (mci%chain_weights (mci%n_chain), source = 0._default)
  mci%chain = chain
end subroutine mci_declare_chains
```

Collect channel weights according to chains and store them in the `chain_weights` for output. We sum up the weights for all channels that share the same `chain` index and store the results in the `chain_weights` array.

⟨*MCI base: mci: TBP*⟩+≡
```
procedure :: collect_chain_weights => mci_collect_chain_weights
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_collect_chain_weights (mci, weight)
  class(mci_t), intent(inout) :: mci
  real(default), dimension(:), intent(in) :: weight
  integer :: i, c
  if (allocated (mci%chain)) then
     mci%chain_weights = 0
     do i = 1, size (mci%chain)
        c = mci%chain(i)
        mci%chain_weights(c) = mci%chain_weights(c) + weight(i)
     end do
  end if
end subroutine mci_collect_chain_weights
```

Check if there are chains.

⟨*MCI base: mci: TBP*⟩+≡
```
procedure :: has_chains => mci_has_chains
```

⟨*MCI base: procedures*⟩+≡
```
function mci_has_chains (mci) result (flag)
  class(mci_t), intent(in) :: mci
  logical :: flag
  flag = allocated (mci%chain)
end function mci_has_chains
```

Output of the chain weights, kept separate from the main `write` method.

[The formatting will work as long as the number of chains is less than 10^{10}...]

⟨*MCI base: mci: TBP*⟩+≡
```
procedure :: write_chain_weights => mci_write_chain_weights
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_write_chain_weights (mci, unit)
  class(mci_t), intent(in) :: mci
  integer, intent(in), optional :: unit
  integer :: u, i, n, n_digits
  character(4) :: ifmt
  u = given_output_unit (unit)
  if (allocated (mci%chain_weights)) then
     write (u, "(1x,A)")  "Weights of channel chains (groves):"
     n_digits = 0
     n = size (mci%chain_weights)
     do while (n > 0)
        n = n / 10
        n_digits = n_digits + 1
     end do
     write (ifmt, "(A1,I1)") "I", n_digits
     do i = 1, size (mci%chain_weights)
        write (u, "(3x," // ifmt // ",F13.10)")  i, mci%chain_weights(i)
```

```
      end do
    end if
  end subroutine mci_write_chain_weights
```

Set the MD5 sum, independent of initialization.

⟨*MCI base: mci: TBP*⟩+≡

```
  procedure :: set_md5sum => mci_set_md5sum
```

⟨*MCI base: procedures*⟩+≡

```
  subroutine mci_set_md5sum (mci, md5sum)
    class(mci_t), intent(inout) :: mci
    character(32), intent(in) :: md5sum
    mci%md5sum = md5sum
  end subroutine mci_set_md5sum
```

Initialize a new integration pass. This is not necessarily meaningful, so we provide an empty base method. The `mci_vamp` implementation overrides this.

⟨*MCI base: mci: TBP*⟩+≡

```
  procedure :: add_pass => mci_add_pass
```

⟨*MCI base: procedures*⟩+≡

```
  subroutine mci_add_pass (mci, adapt_grids, adapt_weights, final_pass)
    class(mci_t), intent(inout) :: mci
    logical, intent(in), optional :: adapt_grids
    logical, intent(in), optional :: adapt_weights
    logical, intent(in), optional :: final_pass
  end subroutine mci_add_pass
```

Allocate an instance with matching type. This must be deferred.

⟨*MCI base: mci: TBP*⟩+≡

```
  procedure (mci_allocate_instance), deferred :: allocate_instance
```

⟨*MCI base: interfaces*⟩+≡

```
  abstract interface
     subroutine mci_allocate_instance (mci, mci_instance)
       import
       class(mci_t), intent(in) :: mci
       class(mci_instance_t), intent(out), pointer :: mci_instance
     end subroutine mci_allocate_instance
  end interface
```

Import a random-number generator. We transfer the allocation of an existing generator state into the object. The generator state may already be initialized, or we can reset it by its `init` method.

⟨*MCI base: mci: TBP*⟩+≡

```
  procedure :: import_rng => mci_import_rng
```

⟨*MCI base: procedures*⟩+≡

```
  subroutine mci_import_rng (mci, rng)
    class(mci_t), intent(inout) :: mci
    class(rng_t), intent(inout), allocatable :: rng
    call move_alloc (rng, mci%rng)
  end subroutine mci_import_rng
```

Activate or deactivate the timer.

⟨*MCI base: mci: TBP*⟩+≡
```
procedure :: set_timer => mci_set_timer
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_set_timer (mci, active)
  class(mci_t), intent(inout) :: mci
  logical, intent(in) :: active
  mci%use_timer = active
end subroutine mci_set_timer
```

Start and stop signal for the timer, if active. The elapsed time can then be retrieved from the MCI record.

⟨*MCI base: mci: TBP*⟩+≡
```
procedure :: start_timer => mci_start_timer
procedure :: stop_timer => mci_stop_timer
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_start_timer (mci)
  class(mci_t), intent(inout) :: mci
  if (mci%use_timer)  call mci%timer%start ()
end subroutine mci_start_timer

subroutine mci_stop_timer (mci)
  class(mci_t), intent(inout) :: mci
  if (mci%use_timer)  call mci%timer%stop ()
end subroutine mci_stop_timer
```

Sampler test. Evaluate the sampler a given number of times. Results are discarded, so we don't need the MCI instance which would record them.

The evaluation channel is iterated, and the `x` parameters are randomly chosen.

⟨*MCI base: mci: TBP*⟩+≡
```
procedure :: sampler_test => mci_sampler_test
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_sampler_test (mci, sampler, n_calls)
  class(mci_t), intent(inout) :: mci
  class(mci_sampler_t), intent(inout), target :: sampler
  integer, intent(in) :: n_calls
  real(default), dimension(:), allocatable :: x_in, f
  real(default), dimension(:,:), allocatable :: x_out
  real(default) :: val
  integer :: i, c
  allocate (x_in (mci%n_dim))
  allocate (f (mci%n_channel))
  allocate (x_out (mci%n_dim, mci%n_channel))
  do i = 1, n_calls
     c = mod (i, mci%n_channel) + 1
     call mci%rng%generate_array (x_in)
     call sampler%evaluate (c, x_in, val, x_out, f)
  end do
end subroutine mci_sampler_test
```

Integrate: this depends on the implementation. We foresee a pacify flag to take care of small numerical noise on different platforms.

⟨*MCI base: mci: TBP*⟩+≡
```
procedure (mci_integrate), deferred :: integrate
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
   subroutine mci_integrate (mci, instance, sampler, &
        n_it, n_calls, results, pacify)
     import
     class(mci_t), intent(inout) :: mci
     class(mci_instance_t), intent(inout), target :: instance
     class(mci_sampler_t), intent(inout), target :: sampler
     integer, intent(in) :: n_it
     integer, intent(in) :: n_calls
     logical, intent(in), optional :: pacify
     class(mci_results_t), intent(inout), optional :: results
   end subroutine mci_integrate
end interface
```

Event generation. Depending on the implementation, event generation may or may not require a previous integration pass.

Instead of a black-box `simulate` method, we require an initializer, a finalizer, and procedures for generating a single event. This allows us to interface simulation event by event from the outside, and it facilitates the further processing of an event after successful generation. For integration, this is not necessary.

The initializer has `intent(inout)` for the `mci` passed object. The reason is that the initializer can read integration results and grids from file, where the results can modify the `mci` record.

⟨*MCI base: mci: TBP*⟩+≡
```
procedure (mci_prepare_simulation), deferred :: prepare_simulation
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
   subroutine mci_prepare_simulation (mci)
     import
     class(mci_t), intent(inout) :: mci
   end subroutine mci_prepare_simulation
end interface
```

The generated event will reside in in the `instance` object (overall results and weight) and in the `sampler` object (detailed data). In the real application, we can subsequently call methods of the `sampler` in order to further process the generated event.

The `target` attributes are required by the VAMP implementation, which uses pointers to refer to the instance and sampler objects from within the integration function.

⟨*MCI base: mci: TBP*⟩+≡
```
procedure (mci_generate), deferred :: generate_weighted_event
procedure (mci_generate), deferred :: generate_unweighted_event
```

⟨*MCI base: interfaces*⟩+≡

```
abstract interface
   subroutine mci_generate (mci, instance, sampler)
     import
     class(mci_t), intent(inout) :: mci
     class(mci_instance_t), intent(inout), target :: instance
     class(mci_sampler_t), intent(inout), target :: sampler
   end subroutine mci_generate
end interface
```

This is analogous, but we rebuild the event from the information stored in `state` instead of generating it.

Note: currently unused outside of tests, might be deleted later.

⟨*MCI base: mci: TBP*⟩+≡

```
procedure (mci_rebuild), deferred :: rebuild_event
```

⟨*MCI base: interfaces*⟩+≡

```
abstract interface
   subroutine mci_rebuild (mci, instance, sampler, state)
     import
     class(mci_t), intent(inout) :: mci
     class(mci_instance_t), intent(inout) :: instance
     class(mci_sampler_t), intent(inout) :: sampler
     class(mci_state_t), intent(in) :: state
   end subroutine mci_rebuild
end interface
```

Pacify: reduce numerical noise. The base implementation does nothing.

⟨*MCI base: mci: TBP*⟩+≡

```
procedure :: pacify => mci_pacify
```

⟨*MCI base: procedures*⟩+≡

```
subroutine mci_pacify (object, efficiency_reset, error_reset)
  class(mci_t), intent(inout) :: object
  logical, intent(in), optional :: efficiency_reset, error_reset
end subroutine mci_pacify
```

Return the value of the integral, error, efficiency, and time per call.

⟨*MCI base: mci: TBP*⟩+≡

```
procedure :: get_integral => mci_get_integral
procedure :: get_error => mci_get_error
procedure :: get_efficiency => mci_get_efficiency
procedure :: get_time => mci_get_time
```

⟨*MCI base: procedures*⟩+≡

```
function mci_get_integral (mci) result (integral)
  class(mci_t), intent(in) :: mci
  real(default) :: integral
  if (mci%integral_known) then
     integral = mci%integral
  else
     call msg_bug ("The integral is unknown. This is presumably a" // &
          "WHIZARD bug.")
```

```
  end if
end function mci_get_integral

function mci_get_error (mci) result (error)
  class(mci_t), intent(in) :: mci
  real(default) :: error
  if (mci%error_known) then
     error = mci%error
  else
     error = 0
  end if
end function mci_get_error

function mci_get_efficiency (mci) result (efficiency)
  class(mci_t), intent(in) :: mci
  real(default) :: efficiency
  if (mci%efficiency_known) then
     efficiency = mci%efficiency
  else
     efficiency = 0
  end if
end function mci_get_efficiency

function mci_get_time (mci) result (time)
  class(mci_t), intent(in) :: mci
  real(default) :: time
  if (mci%use_timer) then
     time = mci%timer
  else
     time = 0
  end if
end function mci_get_time
```

Return the MD5 sum of the configuration. This may be overridden in an extension, to return a different MD5 sum.

⟨*MCI base: mci: TBP*⟩+≡

```
procedure :: get_md5sum => mci_get_md5sum
```

⟨*MCI base: procedures*⟩+≡

```
pure function mci_get_md5sum (mci) result (md5sum)
  class(mci_t), intent(in) :: mci
  character(32) :: md5sum
  md5sum = mci%md5sum
end function mci_get_md5sum
```

20.1.2 MCI instance

The base type contains an array of channel weights. The value `mci_weight` is the combined MCI weight that corresponds to a particular sampling point.

For convenience, we also store the `x` and Jacobian values for this sampling point.

⟨*MCI base: public*⟩+≡

```
public :: mci_instance_t
```

⟨*MCI base: types*⟩+≡

```
type, abstract :: mci_instance_t
   logical :: valid = .false.
   real(default), dimension(:), allocatable :: w
   real(default), dimension(:), allocatable :: f
   real(default), dimension(:,:), allocatable :: x
   integer :: selected_channel = 0
   real(default) :: mci_weight = 0
   real(default) :: integrand  = 0
   logical :: negative_weights = .false.
 contains
 ⟨MCI base: mci instance: TBP⟩
end type mci_instance_t
```

Output: deferred

⟨*MCI base: mci instance: TBP*⟩≡

```
procedure (mci_instance_write), deferred :: write
```

⟨*MCI base: interfaces*⟩+≡

```
abstract interface
   subroutine mci_instance_write (object, unit, pacify)
     import
     class(mci_instance_t), intent(in) :: object
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: pacify
   end subroutine mci_instance_write
end interface
```

A finalizer, just in case.

⟨*MCI base: mci instance: TBP*⟩+≡

```
procedure (mci_instance_final), deferred :: final
```

⟨*MCI base: interfaces*⟩+≡

```
abstract interface
   subroutine mci_instance_final (object)
     import
     class(mci_instance_t), intent(inout) :: object
   end subroutine mci_instance_final
end interface
```

Init: basic initializer for the arrays, otherwise deferred. Assigning the `mci` object is also deferred, because it depends on the concrete type.

The weights are initialized with an uniform normalized value.

⟨*MCI base: mci instance: TBP*⟩+≡

```
procedure (mci_instance_base_init), deferred :: init
procedure :: base_init => mci_instance_base_init
```

⟨*MCI base: procedures*⟩+≡

```
subroutine mci_instance_base_init (mci_instance, mci)
  class(mci_instance_t), intent(out) :: mci_instance
  class(mci_t), intent(in), target :: mci
  allocate (mci_instance%w (mci%n_channel))
```

```
    allocate (mci_instance%f (mci%n_channel))
    allocate (mci_instance%x (mci%n_dim, mci%n_channel))
    if (mci%n_channel > 0) then
       call mci_instance%set_channel_weights &
            (spread (1._default, dim=1, ncopies=mci%n_channel))
    end if
    mci_instance%f = 0
    mci_instance%x = 0
  end subroutine mci_instance_base_init
```

Explicitly set the array of channel weights.

⟨*MCI base: mci instance: TBP*⟩+≡

```
  procedure :: set_channel_weights => mci_instance_set_channel_weights
```

⟨*MCI base: procedures*⟩+≡

```
  subroutine mci_instance_set_channel_weights (mci_instance, weights, sum_non_zero)
    class(mci_instance_t), intent(inout) :: mci_instance
    real(default), dimension(:), intent(in) :: weights
    logical, intent(out), optional :: sum_non_zero
    real(default) :: wsum
    wsum = sum (weights)
    if (wsum /= 0) then
       mci_instance%w = weights / wsum
       if (present (sum_non_zero)) sum_non_zero = .true.
    else
       if (present (sum_non_zero)) sum_non_zero = .false.
       call msg_warning ("MC sampler initialization:&
            & sum of channel weights is zero")
    end if
  end subroutine mci_instance_set_channel_weights
```

Compute the overall weight factor for a configuration of x values and Jacobians f. The x values come in `n_channel` rows with `n_dim` entries each. The f factors constitute an array with `n_channel` entries.

We assume that the x and f arrays are already stored inside the MC instance. The result is also stored there.

⟨*MCI base: mci instance: TBP*⟩+≡

```
  procedure (mci_instance_compute_weight), deferred :: compute_weight
```

⟨*MCI base: interfaces*⟩+≡

```
  abstract interface
     subroutine mci_instance_compute_weight (mci, c)
       import
       class(mci_instance_t), intent(inout) :: mci
       integer, intent(in) :: c
     end subroutine mci_instance_compute_weight
  end interface
```

Record the integrand as returned by the sampler. Depending on the implementation, this may merely copy the value, or do more complicated things.

We may need the MCI weight for the actual computations, so this should be called after the previous routine.

⟨*MCI base: mci instance: TBP*⟩+≡

```
procedure (mci_instance_record_integrand), deferred :: record_integrand
```

⟨*MCI base: interfaces*⟩+≡

```
abstract interface
   subroutine mci_instance_record_integrand (mci, integrand)
     import
     class(mci_instance_t), intent(inout) :: mci
     real(default), intent(in) :: integrand
   end subroutine mci_instance_record_integrand
end interface
```

Sample a point directly: evaluate the sampler, then compute the weight and the weighted integrand. Finally, record the integrand within the MCI instance.

If a signal (interrupt) was raised recently, we abort the calculation before entering the sampler. Thus, a previous calculation will have completed and any data are already recorded, but any new point can be discarded. If the `abort` flag is present, we may delay the interrupt, so we can do some cleanup.

⟨*MCI base: mci instance: TBP*⟩+≡

```
procedure :: evaluate => mci_instance_evaluate
```

⟨*MCI base: procedures*⟩+≡

```
subroutine mci_instance_evaluate (mci, sampler, c, x)
  class(mci_instance_t), intent(inout) :: mci
  class(mci_sampler_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x
  real(default) :: val
  call sampler%evaluate (c, x, val, mci%x, mci%f)
  mci%valid = sampler%is_valid ()
  if (mci%valid) then
     call mci%compute_weight (c)
     call mci%record_integrand (val)
  end if
end subroutine mci_instance_evaluate
```

Initiate and terminate simulation. In contrast to integration, we implement these as methods of the process instance, since the `mci` configuration object is unchanged.

The safety factor reduces the acceptance probability for unweighted events. The implementation of this feature depends on the concrete type.

⟨*MCI base: mci instance: TBP*⟩+≡

```
procedure (mci_instance_init_simulation), deferred :: init_simulation
procedure (mci_instance_final_simulation), deferred :: final_simulation
```

⟨*MCI base: interfaces*⟩+≡

```
abstract interface
   subroutine mci_instance_init_simulation (instance, safety_factor)
     import
     class(mci_instance_t), intent(inout) :: instance
     real(default), intent(in), optional :: safety_factor
   end subroutine mci_instance_init_simulation
end interface
```

```
abstract interface
   subroutine mci_instance_final_simulation (instance)
     import
     class(mci_instance_t), intent(inout) :: instance
   end subroutine mci_instance_final_simulation
end interface
```

Assuming that the sampler is in a completely defined state, just extract the data that `evaluate` would compute. Also record the integrand.

⟨*MCI base: mci instance: TBP*⟩+≡

```
procedure :: fetch => mci_instance_fetch
```

⟨*MCI base: procedures*⟩+≡

```
subroutine mci_instance_fetch (mci, sampler, c)
  class(mci_instance_t), intent(inout) :: mci
  class(mci_sampler_t), intent(in) :: sampler
  integer, intent(in) :: c
  real(default) :: val
  mci%valid = sampler%is_valid ()
  if (mci%valid) then
     call sampler%fetch (val, mci%x, mci%f)
     call mci%compute_weight (c)
     call mci%record_integrand (val)
  end if
end subroutine mci_instance_fetch
```

The value, i.e., the weighted integrand, is the integrand (which should be taken as-is from the sampler) multiplied by the MCI weight.

⟨*MCI base: mci instance: TBP*⟩+≡

```
procedure :: get_value => mci_instance_get_value
```

⟨*MCI base: procedures*⟩+≡

```
function mci_instance_get_value (mci) result (value)
  class(mci_instance_t), intent(in) :: mci
  real(default) :: value
  if (mci%valid) then
     value = mci%integrand * mci%mci_weight
  else
     value = 0
  end if
end function mci_instance_get_value
```

This is an extra routine. By default, the event weight is equal to the value returned by the previous routine. However, if we select a channel for event generation not just based on the channel weights, the event weight has to account for this bias, so the event weight that applies to event generation is different. In that case, we should override the default routine.

⟨*MCI base: mci instance: TBP*⟩+≡

```
procedure :: get_event_weight => mci_instance_get_value
```

Excess weight can occur during unweighted event generation, if the assumed maximum value of the integrand is too small. This excess should be normalized

in the same way as the event weight above (which for unweighted events becomes unity).

⟨*MCI base: mci instance: TBP*⟩+≡
```
procedure (mci_instance_get_event_excess), deferred :: get_event_excess
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
   function mci_instance_get_event_excess (mci) result (excess)
     import
     class(mci_instance_t), intent(in) :: mci
     real(default) :: excess
   end function mci_instance_get_event_excess
end interface
```

20.1.3 MCI state

This object can hold the relevant information that allows us to reconstruct the MCI instance without re-evaluating the sampler completely.

We store the `x_in` MC input parameter set, which coincides with the section of the complete `x` array that belongs to a particular channel. We also store the MC function value. When we want to reconstruct the state, we can use the input array to recover the complete `x` and `f` arrays (i.e., the kinematics), but do not need to recompute the MC function value (the dynamics).

The `mci_state_t` may be extended, to allow storing/recalling more information. In that case, we would override the type-bound procedures. However, the base type is also a concrete type and self-contained.

⟨*MCI base: public*⟩+≡
```
public :: mci_state_t
```

⟨*MCI base: types*⟩+≡
```
type :: mci_state_t
   integer :: selected_channel = 0
   real(default), dimension(:), allocatable :: x_in
   real(default) :: val
 contains
 ⟨MCI base: mci state: TBP⟩
end type mci_state_t
```

Output:

⟨*MCI base: mci state: TBP*⟩≡
```
procedure :: write => mci_state_write
```

⟨*MCI base: procedures*⟩+≡
```
subroutine mci_state_write (object, unit)
  class(mci_state_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "MCI state:"
  write (u, "(3x,A,I0)")  "Channel   = ", object%selected_channel
  write (u, "(3x,A,999(1x,F12.10))")  "x (in)    =", object%x_in
  write (u, "(3x,A,ES19.12)")  "Integrand = ", object%val
```

```
end subroutine mci_state_write
```

To store the object, we take the relevant section of the `x` array. The channel used for storing data is taken from the `instance` object, but it could be arbitrary in principle.

⟨*MCI base: mci instance: TBP*⟩+≡

```
procedure :: store => mci_instance_store
```

⟨*MCI base: procedures*⟩+≡

```
subroutine mci_instance_store (mci, state)
  class(mci_instance_t), intent(in) :: mci
  class(mci_state_t), intent(out) :: state
  state%selected_channel = mci%selected_channel
  allocate (state%x_in (size (mci%x, 1)))
  state%x_in = mci%x(:,mci%selected_channel)
  state%val = mci%integrand
end subroutine mci_instance_store
```

Recalling the state, we must consult the sampler in order to fully reconstruct the `x` and `f` arrays. The integrand value is known, and we also give it to the sampler, bypassing evaluation.

The final steps are equivalent to the `evaluate` method above.

⟨*MCI base: mci instance: TBP*⟩+≡

```
procedure :: recall => mci_instance_recall
```

⟨*MCI base: procedures*⟩+≡

```
subroutine mci_instance_recall (mci, sampler, state)
  class(mci_instance_t), intent(inout) :: mci
  class(mci_sampler_t), intent(inout) :: sampler
  class(mci_state_t), intent(in) :: state
  if (size (state%x_in) == size (mci%x, 1) &
       .and. state%selected_channel <= size (mci%x, 2)) then
     call sampler%rebuild (state%selected_channel, &
          state%x_in, state%val, mci%x, mci%f)
     call mci%compute_weight (state%selected_channel)
     call mci%record_integrand (state%val)
  else
     call msg_fatal ("Recalling event: mismatch in channel or dimension")
  end if
end subroutine mci_instance_recall
```

20.1.4 MCI sampler

A sampler is an object that implements a multi-channel parameterization of the unit hypercube. Specifically, it is able to compute, given a channel and a set of x MC parameter values, a the complete set of x values and associated Jacobian factors f for all channels.

Furthermore, the sampler should return a single real value, the integrand, for the given point in the hypercube.

It must implement a method `evaluate` for performing the above computations.

⟨*MCI base: public*⟩+≡
```
public :: mci_sampler_t
```

⟨*MCI base: types*⟩+≡
```
type, abstract :: mci_sampler_t
 contains
 ⟨MCI base: mci sampler: TBP⟩
end type mci_sampler_t
```

Output, deferred to the implementation.

⟨*MCI base: mci sampler: TBP*⟩≡
```
procedure (mci_sampler_write), deferred :: write
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
   subroutine mci_sampler_write (object, unit, testflag)
     import
     class(mci_sampler_t), intent(in) :: object
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: testflag
   end subroutine mci_sampler_write
end interface
```

The evaluation routine. Input is the channel index `c` and the one-dimensional parameter array `x_in`. Output are the integrand value `val`, the two-dimensional parameter array `x` and the Jacobian array `f`.

⟨*MCI base: mci sampler: TBP*⟩+≡
```
procedure (mci_sampler_evaluate), deferred :: evaluate
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
   subroutine mci_sampler_evaluate (sampler, c, x_in, val, x, f)
     import
     class(mci_sampler_t), intent(inout) :: sampler
     integer, intent(in) :: c
     real(default), dimension(:), intent(in) :: x_in
     real(default), intent(out) :: val
     real(default), dimension(:,:), intent(out) :: x
     real(default), dimension(:), intent(out) :: f
   end subroutine mci_sampler_evaluate
end interface
```

Query the validity of the sampling point. Can be called after `evaluate`.

⟨*MCI base: mci sampler: TBP*⟩+≡
```
procedure (mci_sampler_is_valid), deferred :: is_valid
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
   function mci_sampler_is_valid (sampler) result (valid)
     import
     class(mci_sampler_t), intent(in) :: sampler
     logical :: valid
```

```
      end function mci_sampler_is_valid
   end interface
```

The shortcut. Again, the channel index `c` and the parameter array `x_in` are input. However, we also provide the integrand value `val`, and we just require that the complete parameter array `x` and Jacobian array `f` are recovered.

⟨*MCI base: mci sampler: TBP*⟩+≡

```
   procedure (mci_sampler_rebuild), deferred :: rebuild
```

⟨*MCI base: interfaces*⟩+≡

```
   abstract interface
      subroutine mci_sampler_rebuild (sampler, c, x_in, val, x, f)
        import
        class(mci_sampler_t), intent(inout) :: sampler
        integer, intent(in) :: c
        real(default), dimension(:), intent(in) :: x_in
        real(default), intent(in) :: val
        real(default), dimension(:,:), intent(out) :: x
        real(default), dimension(:), intent(out) :: f
      end subroutine mci_sampler_rebuild
   end interface
```

This routine should extract the important data from a sampler that has been filled by other means. We fetch the integrand value `val`, the two-dimensional parameter array `x` and the Jacobian array `f`.

⟨*MCI base: mci sampler: TBP*⟩+≡

```
   procedure (mci_sampler_fetch), deferred :: fetch
```

⟨*MCI base: interfaces*⟩+≡

```
   abstract interface
      subroutine mci_sampler_fetch (sampler, val, x, f)
        import
        class(mci_sampler_t), intent(in) :: sampler
        real(default), intent(out) :: val
        real(default), dimension(:,:), intent(out) :: x
        real(default), dimension(:), intent(out) :: f
      end subroutine mci_sampler_fetch
   end interface
```

20.1.5 Results record

This is an abstract type which allows us to implement callback: each integration results can optionally be recorded to an instance of this object. The actual object may store a new result, average results, etc. It may also display a result on-line or otherwise, whenever the `record` method is called.

⟨*MCI base: public*⟩+≡

```
   public :: mci_results_t
```

⟨*MCI base: types*⟩+≡

```
   type, abstract :: mci_results_t
    contains
    ⟨MCI base: mci results: TBP⟩
```

```
end type mci_results_t
```

The output routine is deferred. We provide an extra `verbose` flag, which could serve any purpose.

⟨*MCI base: mci results: TBP*⟩≡
```
procedure (mci_results_write), deferred :: write
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
   subroutine mci_results_write (object, unit, verbose, suppress)
     import
     class(mci_results_t), intent(in) :: object
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: verbose, suppress
   end subroutine mci_results_write
end interface
```

This is the `record` method, which can be called directly from the integrator.

⟨*MCI base: mci results: TBP*⟩+≡
```
procedure (mci_results_record), deferred :: record
```

⟨*MCI base: interfaces*⟩+≡
```
abstract interface
   subroutine mci_results_record (object, n_it, &
        n_calls, integral, error, efficiency, chain_weights, suppress)
     import
     class(mci_results_t), intent(inout) :: object
     integer, intent(in) :: n_it
     integer, intent(in) :: n_calls
     real(default), intent(in) :: integral
     real(default), intent(in) :: error
     real(default), intent(in) :: efficiency
     real(default), dimension(:), intent(in), optional :: chain_weights
     logical, intent(in), optional :: suppress
   end subroutine mci_results_record
end interface
```

20.1.6 Unit tests

Test module, followed by the corresponding implementation module.

⟨`mci_base_ut.f90`⟩≡
```
⟨File header⟩

module mci_base_ut
  use unit_tests
  use mci_base_uti

⟨Standard module head⟩

⟨MCI base: public test⟩

⟨MCI base: public test auxiliary⟩
```

```
contains

⟨MCI base: test driver⟩

end module mci_base_ut
```

⟨mci_base_uti.f90⟩≡

```
⟨File header⟩

module mci_base_uti

⟨Use kinds⟩
  use io_units
  use diagnostics
  use phs_base
  use rng_base

  use mci_base

  use rng_base_ut, only: rng_test_t

⟨Standard module head⟩

⟨MCI base: public test auxiliary⟩

⟨MCI base: test declarations⟩

⟨MCI base: test types⟩

contains

⟨MCI base: tests⟩

end module mci_base_uti
```

API: driver for the unit tests below.

⟨*MCI base: public test*⟩≡

```
  public :: mci_base_test
```

⟨*MCI base: test driver*⟩≡

```
  subroutine mci_base_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨MCI base: execute tests⟩
  end subroutine mci_base_test
```

Test implementation of the configuration type

The concrete type contains the number of requested calls and the integral result, to be determined.

The `max_factor` entry is set for the actual test integration, where the integrand is not unity but some other constant value. This value should be set

here, such that the actual maximum of the integrand is known when vetoing unweighted events.

⟨*MCI base: public test auxiliary*⟩≡

```
public :: mci_test_t
```

⟨*MCI base: test types*⟩≡

```
type, extends (mci_t) :: mci_test_t
   integer :: divisions = 0
   integer :: tries = 0
   real(default) :: max_factor = 1
 contains
   procedure :: final => mci_test_final
   procedure :: write => mci_test_write
   procedure :: startup_message => mci_test_startup_message
   procedure :: write_log_entry => mci_test_write_log_entry
   procedure :: compute_md5sum => mci_test_compute_md5sum
   procedure :: declare_flat_dimensions => mci_test_ignore_flat_dimensions
   procedure :: declare_equivalences => mci_test_ignore_equivalences
   procedure :: set_divisions => mci_test_set_divisions
   procedure :: set_max_factor => mci_test_set_max_factor
   procedure :: allocate_instance => mci_test_allocate_instance
   procedure :: integrate => mci_test_integrate
   procedure :: prepare_simulation => mci_test_ignore_prepare_simulation
   procedure :: generate_weighted_event => mci_test_generate_weighted_event
   procedure :: generate_unweighted_event => &
        mci_test_generate_unweighted_event
   procedure :: rebuild_event => mci_test_rebuild_event
end type mci_test_t
```

Finalizer: base version is sufficient

⟨*MCI base: tests*⟩≡

```
subroutine mci_test_final (object)
  class(mci_test_t), intent(inout) :: object
  call object%base_final ()
end subroutine mci_test_final
```

Output: trivial

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_write (object, unit, pacify, md5sum_version)
  class(mci_test_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacify
  logical, intent(in), optional :: md5sum_version
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Test integrator:"
  call object%base_write (u, pacify, md5sum_version)
  if (object%divisions /= 0) then
     write (u, "(3x,A,I0)")  "Number of divisions  = ", object%divisions
  end if
  if (allocated (object%rng))  call object%rng%write (u)
end subroutine mci_test_write
```

Short version.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_startup_message (mci, unit, n_calls)
  class(mci_test_t), intent(in) :: mci
  integer, intent(in), optional :: unit, n_calls
  call mci%base_startup_message (unit = unit, n_calls = n_calls)
  write (msg_buffer, "(A,1x,I0,1x,A)") &
       "Integrator: Test:", mci%divisions, "divisions"
  call msg_message (unit = unit)
end subroutine mci_test_startup_message
```

Log entry: nothing.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_write_log_entry (mci, u)
  class(mci_test_t), intent(in) :: mci
  integer, intent(in) :: u
end subroutine mci_test_write_log_entry
```

Compute MD5 sum: nothing.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_compute_md5sum (mci, pacify)
  class(mci_test_t), intent(inout) :: mci
  logical, intent(in), optional :: pacify
end subroutine mci_test_compute_md5sum
```

This is a no-op for the test integrator.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_ignore_flat_dimensions (mci, dim_flat)
  class(mci_test_t), intent(inout) :: mci
  integer, dimension(:), intent(in) :: dim_flat
end subroutine mci_test_ignore_flat_dimensions
```

Ditto.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_ignore_equivalences (mci, channel, dim_offset)
  class(mci_test_t), intent(inout) :: mci
  type(phs_channel_t), dimension(:), intent(in) :: channel
  integer, intent(in) :: dim_offset
end subroutine mci_test_ignore_equivalences
```

Set the number of divisions to a nonzero value.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_set_divisions (object, divisions)
  class(mci_test_t), intent(inout) :: object
  integer, intent(in) :: divisions
  object%divisions = divisions
end subroutine mci_test_set_divisions
```

Set the maximum factor (default is 1).

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_set_max_factor (object, max_factor)
  class(mci_test_t), intent(inout) :: object
  real(default), intent(in) :: max_factor
  object%max_factor = max_factor
end subroutine mci_test_set_max_factor
```

Allocate instance with matching type.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_allocate_instance (mci, mci_instance)
  class(mci_test_t), intent(in) :: mci
  class(mci_instance_t), intent(out), pointer :: mci_instance
  allocate (mci_test_instance_t :: mci_instance)
end subroutine mci_test_allocate_instance
```

Integrate: sample at the midpoints of uniform bits and add the results. We implement this for one and for two dimensions. In the latter case, we scan over two channels and multiply with the channel weights.

The arguments `n_it` and `n_calls` are ignored in this implementations.

The test integrator does not set error or efficiency, so those will remain undefined.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_integrate (mci, instance, sampler, &
     n_it, n_calls, results, pacify)
  class(mci_test_t), intent(inout) :: mci
  class(mci_instance_t), intent(inout), target :: instance
  class(mci_sampler_t), intent(inout), target :: sampler
  integer, intent(in) :: n_it
  integer, intent(in) :: n_calls
  logical, intent(in), optional :: pacify
  class(mci_results_t), intent(inout), optional :: results
  real(default), dimension(:), allocatable :: integral
  real(default), dimension(:), allocatable :: x
  integer :: i, j, c
  select type (instance)
  type is (mci_test_instance_t)
     allocate (integral (mci%n_channel))
     integral = 0
     allocate (x (mci%n_dim))
     select case (mci%n_dim)
     case (1)
        do c = 1, mci%n_channel
           do i = 1, mci%divisions
              x(1) = (i - 0.5_default) / mci%divisions
              call instance%evaluate (sampler, c, x)
              integral(c) = integral(c) + instance%get_value ()
           end do
        end do
        mci%integral = dot_product (instance%w, integral) &
             / mci%divisions
        mci%integral_known = .true.
```

```
      case (2)
         do c = 1, mci%n_channel
            do i = 1, mci%divisions
               x(1) = (i - 0.5_default) / mci%divisions
               do j = 1, mci%divisions
                  x(2) = (j - 0.5_default) / mci%divisions
                  call instance%evaluate (sampler, c, x)
                  integral(c) = integral(c) + instance%get_value ()
               end do
            end do
         end do
         mci%integral = dot_product (instance%w, integral) &
              / mci%divisions / mci%divisions
         mci%integral_known = .true.
      end select
      if (present (results)) then
         call results%record (n_it, n_calls, &
              mci%integral, mci%error, &
              efficiency = 0._default)
      end if
   end select
end subroutine mci_test_integrate
```

Simulation initializer and finalizer: nothing to do here.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_ignore_prepare_simulation (mci)
  class(mci_test_t), intent(inout) :: mci
end subroutine mci_test_ignore_prepare_simulation
```

Event generator. We use mock random numbers for first selecting the channel and then setting the x values. The results reside in the state of `instance` and `sampler`.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_generate_weighted_event (mci, instance, sampler)
  class(mci_test_t), intent(inout) :: mci
  class(mci_instance_t), intent(inout), target :: instance
  class(mci_sampler_t), intent(inout), target :: sampler
  real(default) :: r
  real(default), dimension(:), allocatable :: x
  integer :: c
  select type (instance)
  type is (mci_test_instance_t)
     allocate (x (mci%n_dim))
     select case (mci%n_channel)
     case (1)
        c = 1
        call mci%rng%generate (x(1))
     case (2)
        call mci%rng%generate (r)
        if (r < instance%w(1)) then
           c = 1
        else
           c = 2
```

```
         end if
         call mci%rng%generate (x)
      end select
      call instance%evaluate (sampler, c, x)
   end select
end subroutine mci_test_generate_weighted_event
```

For unweighted events, we generate weighted events and apply a simple rejection step to the relative event weight, until an event passes.

(This might result in an endless loop if we happen to be in sync with the mock random generator cycle. Therefore, limit the number of tries.)

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_generate_unweighted_event (mci, instance, sampler)
  class(mci_test_t), intent(inout) :: mci
  class(mci_instance_t), intent(inout), target :: instance
  class(mci_sampler_t), intent(inout), target :: sampler
  real(default) :: r
  integer :: i
  select type (instance)
  type is (mci_test_instance_t)
     mci%tries = 0
     do i = 1, 10
        call mci%generate_weighted_event (instance, sampler)
        mci%tries = mci%tries + 1
        call mci%rng%generate (r)
        if (r < instance%rel_value)  exit
     end do
  end select
end subroutine mci_test_generate_unweighted_event
```

Here, we rebuild the event from the state without consulting the rng.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_rebuild_event (mci, instance, sampler, state)
  class(mci_test_t), intent(inout) :: mci
  class(mci_instance_t), intent(inout) :: instance
  class(mci_sampler_t), intent(inout) :: sampler
  class(mci_state_t), intent(in) :: state
  select type (instance)
  type is (mci_test_instance_t)
     call instance%recall (sampler, state)
  end select
end subroutine mci_test_rebuild_event
```

Instance of the test MCI type

This instance type simulates the VAMP approach. We implement the VAMP multi-channel formula, but keep the channel-specific probability functions g_i smooth and fixed. We also keep the weights fixed.

The setup is as follows: we have n mappings of the unit hypercube

$$x = x(x^{(k)}) \qquad \text{where } x = (x_1, \ldots). \tag{20.1}$$

The Jacobian factors are the determinants

$$f^{(k)}(x^{(k)}) = \left| \frac{\partial x}{\partial x^{(k)}} \right| \tag{20.2}$$

We introduce arbitrary probability functions

$$g^{(k)}(x^{(k)}) \qquad \text{with} \qquad \int dx^{(k)} g^{(k)}(x^{(k)}) = 1 \tag{20.3}$$

and weights

$$w_k \qquad \text{with} \qquad \sum_k w_k = 1 \tag{20.4}$$

and construct the joint probability function

$$g(x) = \sum_k w_k \frac{g^{(k)}(x^{(k)}(x))}{f^{(k)}(x^{(k)}(x))} \tag{20.5}$$

which also satisfies

$$\int g(x)\, dx = 1. \tag{20.6}$$

The algorithm implements a resolution of unity as follows

$$\begin{aligned} 1 &= \int dx = \int \frac{g(x)}{g(x)} dx \\ &= \sum w_k \int \frac{g^{(k)}(x^{(k)}(x))}{f^{(k)}(x^{(k)}(x))} \frac{dx}{g(x)} \\ &= \sum w_k \int g^{(k)}(x^{(k)}) \frac{dx^{(k)}}{g(x(x^{(k)}))} \end{aligned} \tag{20.7}$$

where each of the integrals in the sum is evaluated using the channel-specific variables $x^{(k)}$.

We provide two examples: (1) trivial with one channel, one dimension, and all functions unity and (2) two channels and two dimensions with

$$\begin{aligned} x(x^{(1)}) &= (x_1^{(1)}, x_2^{(1)}) \\ x(x^{(2)}) &= (x_1^{(2)2}, x_2^{(2)}) \end{aligned} \tag{20.8}$$

hence

$$f^{(1)} \equiv 1, \qquad\qquad f^{(2)}(x^{(2)}) = 2x_1^{(2)} \tag{20.9}$$

The probability functions are

$$g^{(1)} \equiv 1, \qquad\qquad g^{(2)}(x^{(2)}) = 2x_2^{(2)} \tag{20.10}$$

In the concrete implementation of the integrator instance we store values for the channel probabilities g_i and the accumulated probability g.

We also store the result (product of integrand and MCI weight), the expected maximum for the result in each channel.

⟨*XXX MCI base: public*⟩≡

```
public :: mci_test_instance_t
```

⟨*MCI base: test types*⟩+≡

```
type, extends (mci_instance_t) :: mci_test_instance_t
   type(mci_test_t), pointer :: mci => null ()
   real(default) :: g = 0
   real(default), dimension(:), allocatable :: gi
   real(default) :: value = 0
   real(default) :: rel_value = 0
   real(default), dimension(:), allocatable :: max
 contains
   procedure :: write => mci_test_instance_write
   procedure :: final => mci_test_instance_final
   procedure :: init => mci_test_instance_init
   procedure :: compute_weight => mci_test_instance_compute_weight
   procedure :: record_integrand => mci_test_instance_record_integrand
   procedure :: init_simulation => mci_test_instance_init_simulation
   procedure :: final_simulation => mci_test_instance_final_simulation
   procedure :: get_event_excess => mci_test_instance_get_event_excess
end type mci_test_instance_t
```

Output: trivial

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_instance_write (object, unit, pacify)
  class(mci_test_instance_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacify
  integer :: u, c
  u = given_output_unit (unit)
  write (u, "(1x,A,ES13.7)") "Result value = ", object%value
  write (u, "(1x,A,ES13.7)") "Rel. weight  = ", object%rel_value
  write (u, "(1x,A,ES13.7)") "Integrand    = ", object%integrand
  write (u, "(1x,A,ES13.7)") "MCI weight   = ", object%mci_weight
  write (u, "(3x,A,I0)")  "c = ", object%selected_channel
  write (u, "(3x,A,ES13.7)") "g = ", object%g
  write (u, "(1x,A)")  "Channel parameters:"
  do c = 1, object%mci%n_channel
     write (u, "(1x,I0,A,4(1x,ES13.7))")  c, ": w/f/g/m =", &
          object%w(c), object%f(c), object%gi(c), object%max(c)
     write (u, "(4x,A,9(1x,F9.7))")  "x =", object%x(:,c)
  end do
end subroutine mci_test_instance_write
```

The finalizer is empty.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_instance_final (object)
  class(mci_test_instance_t), intent(inout) :: object
end subroutine mci_test_instance_final
```

Initializer. We make use of the analytical result that the maximum of the weighted integrand, in each channel, is equal to 1 (one-dimensional case) and 2 (two-dimensional case), respectively.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_instance_init (mci_instance, mci)
```

```
    class(mci_test_instance_t), intent(out) :: mci_instance
    class(mci_t), intent(in), target :: mci
    call mci_instance%base_init (mci)
    select type (mci)
    type is (mci_test_t)
       mci_instance%mci => mci
    end select
    allocate (mci_instance%gi (mci%n_channel))
    mci_instance%gi = 0
    allocate (mci_instance%max (mci%n_channel))
    select case (mci%n_channel)
    case (1)
       mci_instance%max = 1._default
    case (2)
       mci_instance%max = 2._default
    end select
  end subroutine mci_test_instance_init
```

Compute weight: we implement the VAMP multi-channel formula. The channel probabilities `gi` are predefined functions.

⟨*MCI base: tests*⟩+≡

```
  subroutine mci_test_instance_compute_weight (mci, c)
    class(mci_test_instance_t), intent(inout) :: mci
    integer, intent(in) :: c
    integer :: i
    mci%selected_channel = c
    select case (mci%mci%n_dim)
    case (1)
       mci%gi(1) = 1
    case (2)
       mci%gi(1) = 1
       mci%gi(2) = 2 * mci%x(2,2)
    end select
    mci%g = 0
    do i = 1, mci%mci%n_channel
       mci%g = mci%g + mci%w(i) * mci%gi(i) / mci%f(i)
    end do
    mci%mci_weight = mci%gi(c) / mci%g
  end subroutine mci_test_instance_compute_weight
```

Record the integrand. Apply the Jacobian weight to get the absolute value. Divide by the channel maximum and by any overall factor to get the value relative to the maximum.

⟨*MCI base: tests*⟩+≡

```
  subroutine mci_test_instance_record_integrand (mci, integrand)
    class(mci_test_instance_t), intent(inout) :: mci
    real(default), intent(in) :: integrand
    mci%integrand = integrand
    mci%value = mci%integrand * mci%mci_weight
    mci%rel_value = mci%value / mci%max(mci%selected_channel) &
         / mci%mci%max_factor
  end subroutine mci_test_instance_record_integrand
```

Nothing to do here.

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_instance_init_simulation (instance, safety_factor)
  class(mci_test_instance_t), intent(inout) :: instance
  real(default), intent(in), optional :: safety_factor
end subroutine mci_test_instance_init_simulation

subroutine mci_test_instance_final_simulation (instance)
  class(mci_test_instance_t), intent(inout) :: instance
end subroutine mci_test_instance_final_simulation
```

Return always zero.

⟨*MCI base: tests*⟩+≡

```
function mci_test_instance_get_event_excess (mci) result (excess)
  class(mci_test_instance_t), intent(in) :: mci
  real(default) :: excess
  excess = 0
end function mci_test_instance_get_event_excess
```

Test sampler

The test sampler implements a fixed configuration, either trivial (one-channel, one-dimension), or slightly nontrivial (two-channel, two-dimension). In the second channel, the first parameter is mapped according to $x_1 = x_1^{(2)2}$, so we have $f^{(2)}(x^{(2)}) = 2x_1^{(2)}$.

For display purposes, we store the return values inside the object. This is not strictly necessary.

⟨*MCI base: test types*⟩+≡

```
type, extends (mci_sampler_t) :: test_sampler_t
   real(default) :: integrand = 0
   integer :: selected_channel = 0
   real(default), dimension(:,:), allocatable :: x
   real(default), dimension(:), allocatable :: f
 contains
   procedure :: init => test_sampler_init
   procedure :: write => test_sampler_write
   procedure :: compute => test_sampler_compute
   procedure :: is_valid => test_sampler_is_valid
   procedure :: evaluate => test_sampler_evaluate
   procedure :: rebuild => test_sampler_rebuild
   procedure :: fetch => test_sampler_fetch
end type test_sampler_t
```

⟨*MCI base: tests*⟩+≡

```
subroutine test_sampler_init (sampler, n)
  class(test_sampler_t), intent(out) :: sampler
  integer, intent(in) :: n
  allocate (sampler%x (n, n))
  allocate (sampler%f (n))
end subroutine test_sampler_init
```

Output

⟨*MCI base: tests*⟩+≡

```
subroutine test_sampler_write (object, unit, testflag)
  class(test_sampler_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u, c
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Test sampler:"
  write (u, "(3x,A,ES13.7)")  "Integrand = ", object%integrand
  write (u, "(3x,A,I0)")      "Channel   = ", object%selected_channel
  do c = 1, size (object%f)
     write (u, "(1x,I0,':',1x,A,ES13.7)") c, "f = ", object%f(c)
     write (u, "(4x,A,9(1x,F9.7))") "x =", object%x(:,c)
  end do
end subroutine test_sampler_write
```

Compute x and Jacobians, given the input parameter array. This is called both by `evaluate` and `rebuild`.

⟨*MCI base: tests*⟩+≡

```
subroutine test_sampler_compute (sampler, c, x_in)
  class(test_sampler_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  sampler%selected_channel = c
  select case (size (sampler%f))
  case (1)
     sampler%x(:,1) = x_in
     sampler%f = 1
  case (2)
     select case (c)
     case (1)
        sampler%x(:,1) = x_in
        sampler%x(1,2) = sqrt (x_in(1))
        sampler%x(2,2) = x_in(2)
     case (2)
        sampler%x(1,1) = x_in(1) ** 2
        sampler%x(2,1) = x_in(2)
        sampler%x(:,2) = x_in
     end select
     sampler%f(1) = 1
     sampler%f(2) = 2 * sampler%x(1,2)
  end select
end subroutine test_sampler_compute
```

The point is always valid.

⟨*MCI base: tests*⟩+≡

```
function test_sampler_is_valid (sampler) result (valid)
  class(test_sampler_t), intent(in) :: sampler
  logical :: valid
  valid = .true.
end function test_sampler_is_valid
```

The integrand is always equal to 1.

⟨*MCI base: tests*⟩+≡

```
subroutine test_sampler_evaluate (sampler, c, x_in, val, x, f)
  class(test_sampler_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  call sampler%compute (c, x_in)
  sampler%integrand = 1
  val = sampler%integrand
  x = sampler%x
  f = sampler%f
end subroutine test_sampler_evaluate
```

Construct kinematics from the input x array. Set the integrand instead of evaluating it.

⟨*MCI base: tests*⟩+≡

```
subroutine test_sampler_rebuild (sampler, c, x_in, val, x, f)
  class(test_sampler_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(in) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  call sampler%compute (c, x_in)
  sampler%integrand = val
  x = sampler%x
  f = sampler%f
end subroutine test_sampler_rebuild
```

Recall contents.

⟨*MCI base: tests*⟩+≡

```
subroutine test_sampler_fetch (sampler, val, x, f)
  class(test_sampler_t), intent(in) :: sampler
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  val = sampler%integrand
  x = sampler%x
  f = sampler%f
end subroutine test_sampler_fetch
```

Test results object

This mock object just stores and displays the current result.

⟨*MCI base: test types*⟩+≡

```
type, extends (mci_results_t) :: mci_test_results_t
   integer :: n_it = 0
   integer :: n_calls = 0
```

```
    real(default) :: integral = 0
    real(default) :: error = 0
    real(default) :: efficiency = 0
  contains
  ⟨MCI base: mci test results: TBP⟩
end type mci_test_results_t
```

Output.

⟨*MCI base: mci test results: TBP*⟩≡

```
procedure :: write => mci_test_results_write
```

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_results_write (object, unit, verbose, suppress)
  class(mci_test_results_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose, suppress
  integer :: u
  u = given_output_unit (unit)
  write (u, "(3x,A,1x,I0)") "Iterations = ", object%n_it
  write (u, "(3x,A,1x,I0)") "Calls      = ", object%n_calls
  write (u, "(3x,A,1x,F12.10)")  "Integral   = ", object%integral
  write (u, "(3x,A,1x,F12.10)")  "Error      = ", object%error
  write (u, "(3x,A,1x,F12.10)")  "Efficiency = ", object%efficiency
end subroutine mci_test_results_write
```

Record result.

⟨*MCI base: mci test results: TBP*⟩+≡

```
procedure :: record => mci_test_results_record
```

⟨*MCI base: tests*⟩+≡

```
subroutine mci_test_results_record (object, n_it, n_calls, &
     integral, error, efficiency, chain_weights, suppress)
  class(mci_test_results_t), intent(inout) :: object
  integer, intent(in) :: n_it
  integer, intent(in) :: n_calls
  real(default), intent(in) :: integral
  real(default), intent(in) :: error
  real(default), intent(in) :: efficiency
  real(default), dimension(:), intent(in), optional :: chain_weights
  logical, intent(in), optional :: suppress
  object%n_it = n_it
  object%n_calls = n_calls
  object%integral = integral
  object%error = error
  object%efficiency = efficiency
end subroutine mci_test_results_record
```

Integrator configuration data

Construct and display a test integrator configuration object.

⟨*MCI base: execute tests*⟩≡

```
call test (mci_base_1, "mci_base_1", &
```

```
         "integrator configuration", &
         u, results)
```

⟨*MCI base: test declarations*⟩≡

```
    public :: mci_base_1
```

⟨*MCI base: tests*⟩+≡

```
    subroutine mci_base_1 (u)
      integer, intent(in) :: u
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler

      real(default) :: integrand

      write (u, "(A)")  "* Test output: mci_base_1"
      write (u, "(A)")  "*   Purpose: initialize and display &
           &test integrator"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize integrator"
      write (u, "(A)")

      allocate (mci_test_t :: mci)
      call mci%set_dimensions (2, 2)

      call mci%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize instance"
      write (u, "(A)")

      call mci%allocate_instance (mci_instance)
      call mci_instance%init (mci)

      write (u, "(A)")  "* Initialize test sampler"
      write (u, "(A)")

      allocate (test_sampler_t :: sampler)
      select type (sampler)
      type is (test_sampler_t)
         call sampler%init (2)
      end select

      write (u, "(A)")  "* Evaluate sampler for given point and channel"
      write (u, "(A)")

      call sampler%evaluate (1, [0.25_default, 0.8_default], &
           integrand, mci_instance%x, mci_instance%f)

      call sampler%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Compute MCI weight"
      write (u, "(A)")
```

```
    call mci_instance%compute_weight (1)
    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Get integrand and compute weight for another point"
    write (u, "(A)")

    call mci_instance%evaluate (sampler, 2, [0.5_default, 0.6_default])
    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Recall results, again"
    write (u, "(A)")

    call mci_instance%final ()
    deallocate (mci_instance)

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    call mci_instance%fetch (sampler, 2)
    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Retrieve value"
    write (u, "(A)")

    write (u, "(1x,A,ES13.7)")  "Weighted integrand = ", &
         mci_instance%get_value ()

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_base_1"

  end subroutine mci_base_1
```

Trivial integral

Use the MCI approach to compute a trivial one-dimensional integral.

⟨*MCI base: execute tests*⟩+≡

```
  call test (mci_base_2, "mci_base_2", &
       "integration", &
       u, results)
```

⟨*MCI base: test declarations*⟩+≡

```
  public :: mci_base_2
```

⟨*MCI base: tests*⟩+≡

```
  subroutine mci_base_2 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
```

```
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler

  write (u, "(A)")  "* Test output: mci_base_2"
  write (u, "(A)")  "*   Purpose: perform a test integral"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize integrator"
  write (u, "(A)")

  allocate (mci_test_t :: mci)
  call mci%set_dimensions (1, 1)
  select type (mci)
  type is (mci_test_t)
     call mci%set_divisions (10)
  end select

  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize instance"
  write (u, "(A)")

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  write (u, "(A)")  "* Initialize test sampler"
  write (u, "(A)")

  allocate (test_sampler_t :: sampler)
  select type (sampler)
  type is (test_sampler_t)
     call sampler%init (1)
  end select

  write (u, "(A)")  "* Integrate"
  write (u, "(A)")

  call mci%integrate (mci_instance, sampler, 0, 0)

  call mci%write (u)

  call mci_instance%final ()
  call mci%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: mci_base_2"

end subroutine mci_base_2
```

Nontrivial integral

Use the MCI approach to compute a simple two-dimensional integral with two channels.

⟨*MCI base: execute tests*⟩+≡

```
call test (mci_base_3, "mci_base_3", &
     "integration (two channels)", &
     u, results)
```

⟨*MCI base: test declarations*⟩+≡

```
public :: mci_base_3
```

⟨*MCI base: tests*⟩+≡

```
subroutine mci_base_3 (u)
  integer, intent(in) :: u
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler

  write (u, "(A)")  "* Test output: mci_base_3"
  write (u, "(A)")  "*   Purpose: perform a nontrivial test integral"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize integrator"
  write (u, "(A)")

  allocate (mci_test_t :: mci)
  call mci%set_dimensions (2, 2)
  select type (mci)
  type is (mci_test_t)
     call mci%set_divisions (10)
  end select

  write (u, "(A)")  "* Initialize instance"
  write (u, "(A)")

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  write (u, "(A)")  "* Initialize test sampler"
  write (u, "(A)")

  allocate (test_sampler_t :: sampler)
  select type (sampler)
  type is (test_sampler_t)
     call sampler%init (2)
  end select

  write (u, "(A)")  "* Integrate"
  write (u, "(A)")

  call mci%integrate (mci_instance, sampler, 0, 0)
  call mci%write (u)

  write (u, "(A)")
```

```
    write (u, "(A)")  "* Integrate with higher resolution"
    write (u, "(A)")

    select type (mci)
    type is (mci_test_t)
       call mci%set_divisions (100)
    end select

    call mci%integrate (mci_instance, sampler, 0, 0)
    call mci%write (u)

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_base_3"

  end subroutine mci_base_3
```

Event generation

We generate "random" events, one weighted and one unweighted. The test implementation does not require an integration pass, we can generate events immediately.

⟨*MCI base: execute tests*⟩+≡

```
  call test (mci_base_4, "mci_base_4", &
       "event generation (two channels)", &
       u, results)
```

⟨*MCI base: test declarations*⟩+≡

```
  public :: mci_base_4
```

⟨*MCI base: tests*⟩+≡

```
  subroutine mci_base_4 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng

    write (u, "(A)")  "* Test output: mci_base_4"
    write (u, "(A)")  "*   Purpose: generate events"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator, instance, sampler"
    write (u, "(A)")

    allocate (mci_test_t :: mci)
    call mci%set_dimensions (2, 2)
    select type (mci)
    type is (mci_test_t)
       call mci%set_divisions (10)
    end select
```

```
    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    allocate (test_sampler_t :: sampler)
    select type (sampler)
    type is (test_sampler_t)
       call sampler%init (2)
    end select

    allocate (rng_test_t :: rng)
    call mci%import_rng (rng)

    write (u, "(A)")  "* Generate weighted event"
    write (u, "(A)")

    call mci%generate_weighted_event (mci_instance, sampler)

    call sampler%write (u)
    write (u, *)
    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Generate unweighted event"
    write (u, "(A)")

    call mci%generate_unweighted_event (mci_instance, sampler)

    select type (mci)
    type is (mci_test_t)
       write (u, "(A,I0)")  " Success in try ", mci%tries
       write (u, "(A)")
    end select

    call sampler%write (u)
    write (u, *)
    call mci_instance%write (u)

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_base_4"

  end subroutine mci_base_4
```

Store and recall data

We generate an event and store the relevant data, i.e., the input parameters and the result value for a particular channel. Then we use those data to recover the event, as far as the MCI record is concerned.

⟨*MCI base: execute tests*⟩+≡

```
call test (mci_base_5, "mci_base_5", &
     "store and recall", &
     u, results)
```

⟨*MCI base: test declarations*⟩+≡

```
public :: mci_base_5
```

⟨*MCI base: tests*⟩+≡

```
subroutine mci_base_5 (u)
  integer, intent(in) :: u
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng
  class(mci_state_t), allocatable :: state

  write (u, "(A)")  "* Test output: mci_base_5"
  write (u, "(A)")  "*   Purpose: store and recall an event"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize integrator, instance, sampler"
  write (u, "(A)")

  allocate (mci_test_t :: mci)
  call mci%set_dimensions (2, 2)
  select type (mci)
  type is (mci_test_t)
     call mci%set_divisions (10)
  end select

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  allocate (test_sampler_t :: sampler)
  select type (sampler)
  type is (test_sampler_t)
     call sampler%init (2)
  end select

  allocate (rng_test_t :: rng)
  call mci%import_rng (rng)

  write (u, "(A)")  "* Generate weighted event"
  write (u, "(A)")

  call mci%generate_weighted_event (mci_instance, sampler)

  call sampler%write (u)
  write (u, *)
  call mci_instance%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Store data"
  write (u, "(A)")
```

```
    allocate (state)
    call mci_instance%store (state)
    call mci_instance%final ()
    deallocate (mci_instance)

    call state%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Recall data and rebuild event"
    write (u, "(A)")

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)
    call mci%rebuild_event (mci_instance, sampler, state)

    call sampler%write (u)
    write (u, *)
    call mci_instance%write (u)

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_base_5"

  end subroutine mci_base_5
```

Chained channels

Chain channels together. In the base configuration, this just fills entries in an extra array (each channel may belong to a chain). In type implementations, this will be used for grouping equivalent channels by keeping their weights equal.

⟨*MCI base: execute tests*⟩+≡

```
  call test (mci_base_6, "mci_base_6", &
       "chained channels", &
       u, results)
```

⟨*MCI base: test declarations*⟩+≡

```
  public :: mci_base_6
```

⟨*MCI base: tests*⟩+≡

```
  subroutine mci_base_6 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci

    write (u, "(A)")  "* Test output: mci_base_6"
    write (u, "(A)")  "*   Purpose: initialize and display &
         &test integrator with chains"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")
```

```
    allocate (mci_test_t :: mci)
    call mci%set_dimensions (1, 5)

    write (u, "(A)")  "* Introduce chains"
    write (u, "(A)")

    call mci%declare_chains ([1, 2, 2, 1, 2])

    call mci%write (u)

    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_base_6"

  end subroutine mci_base_6
```

Recording results

Compute a simple two-dimensional integral and record the result.

⟨*MCI base: execute tests*⟩+≡

```
  call test (mci_base_7, "mci_base_7", &
       "recording results", &
       u, results)
```

⟨*MCI base: test declarations*⟩+≡

```
  public :: mci_base_7
```

⟨*MCI base: tests*⟩+≡

```
  subroutine mci_base_7 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(mci_results_t), allocatable :: results

    write (u, "(A)")  "* Test output: mci_base_7"
    write (u, "(A)")  "*   Purpose: perform a nontrivial test integral &
         &and record results"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")

    allocate (mci_test_t :: mci)
    call mci%set_dimensions (2, 2)
    select type (mci)
    type is (mci_test_t)
       call mci%set_divisions (10)
    end select

    write (u, "(A)")  "* Initialize instance"
    write (u, "(A)")
```

```
    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    write (u, "(A)")  "* Initialize test sampler"
    write (u, "(A)")

    allocate (test_sampler_t :: sampler)
    select type (sampler)
    type is (test_sampler_t)
       call sampler%init (2)
    end select

    allocate (mci_test_results_t :: results)

    write (u, "(A)")  "* Integrate"
    write (u, "(A)")

    call mci%integrate (mci_instance, sampler, 1, 1000, results)
    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Display results"
    write (u, "(A)")

    call results%write (u)

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_base_7"

  end subroutine mci_base_7
```

Timer

Simple checks for the embedded timer.

⟨*MCI base: execute tests*⟩+≡

```
  call test (mci_base_8, "mci_base_8", &
       "timer", &
       u, results)
```

⟨*MCI base: test declarations*⟩+≡

```
  public :: mci_base_8
```

⟨*MCI base: tests*⟩+≡

```
  subroutine mci_base_8 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci

    real(default) :: dummy

    write (u, "(A)")  "* Test output: mci_base_8"
    write (u, "(A)")  "*   Purpose: check timer availability"
```

```
  write (u, "(A)")

  write (u, "(A)")  "* Initialize integrator with timer"
  write (u, "(A)")

  allocate (mci_test_t :: mci)
  call mci%set_dimensions (2, 2)
  select type (mci)
  type is (mci_test_t)
     call mci%set_divisions (10)
  end select

  call mci%set_timer (active = .true.)
  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Start timer"
  write (u, "(A)")

  call mci%start_timer ()
  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Stop timer"
  write (u, "(A)")

  call mci%stop_timer ()
  write (u, "(A)")  " (ok)"

  write (u, "(A)")
  write (u, "(A)")  "* Readout"
  write (u, "(A)")

  dummy = mci%get_time ()
  write (u, "(A)")  " (ok)"

  write (u, "(A)")
  write (u, "(A)")  "* Deactivate timer"
  write (u, "(A)")

  call mci%set_timer (active = .false.)
  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call mci%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: mci_base_8"

end subroutine mci_base_8
```

20.2 Iterations

This module defines a container for the list of iterations and calls, to be submitted to integration.

⟨iterations.f90⟩≡
```
⟨File header⟩

module iterations

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use diagnostics

⟨Standard module head⟩

⟨Iterations: public⟩

⟨Iterations: types⟩

contains

⟨Iterations: procedures⟩

end module iterations
```

20.2.1 The iterations list

Each integration pass has a number of iterations and a number of calls per iteration. The last pass produces the end result; the previous passes are used for adaptation.

The flags adapt_grid and adapt_weight are used only if custom_adaptation is set. Otherwise, default settings are used that depend on the integration pass.

⟨Iterations: types⟩≡
```
type :: iterations_spec_t
   private
   integer :: n_it = 0
   integer :: n_calls = 0
   logical :: custom_adaptation = .false.
   logical :: adapt_grids = .false.
   logical :: adapt_weights = .false.
end type iterations_spec_t
```

We build up a list of iterations.

⟨Iterations: public⟩≡
```
public :: iterations_list_t
```

⟨Iterations: types⟩+≡
```
type :: iterations_list_t
   private
   integer :: n_pass = 0
   type(iterations_spec_t), dimension(:), allocatable :: pass
 contains
```

```
⟨Iterations: iterations list: TBP⟩
end type iterations_list_t
```

Initialize an iterations list. For each pass, we have to specify the number of iterations and calls. We may provide the adaption conventions explicitly, either as character codes or as logicals.

For passes where the adaptation conventions are not specified, we use the following default setting: adapt weights and grids for all passes except the last one.

⟨*Iterations: iterations list: TBP*⟩≡

```
procedure :: init => iterations_list_init
```

⟨*Iterations: procedures*⟩≡

```
subroutine iterations_list_init &
     (it_list, n_it, n_calls, adapt, adapt_code, adapt_grids, adapt_weights)
  class(iterations_list_t), intent(inout) :: it_list
  integer, dimension(:), intent(in) :: n_it, n_calls
  logical, dimension(:), intent(in), optional :: adapt
  type(string_t), dimension(:), intent(in), optional :: adapt_code
  logical, dimension(:), intent(in), optional :: adapt_grids, adapt_weights
  integer :: i
  it_list%n_pass = size (n_it)
  if (allocated (it_list%pass)) deallocate (it_list%pass)
  allocate (it_list%pass (it_list%n_pass))
  it_list%pass%n_it = n_it
  it_list%pass%n_calls = n_calls
  if (present (adapt)) then
     it_list%pass%custom_adaptation = adapt
     do i = 1, it_list%n_pass
        if (adapt(i)) then
           if (verify (adapt_code(i), "wg") /= 0) then
              call msg_error ("iteration specification: " &
                   // "adaptation code letters must be 'w' or 'g'")
           end if
           it_list%pass(i)%adapt_grids = scan (adapt_code(i), "g") /= 0
           it_list%pass(i)%adapt_weights = scan (adapt_code(i), "w") /= 0
        end if
     end do
  else if (present (adapt_grids) .and. present (adapt_weights)) then
     it_list%pass%custom_adaptation = .true.
     it_list%pass%adapt_grids = adapt_grids
     it_list%pass%adapt_weights = adapt_weights
  end if
  do i = 1, it_list%n_pass - 1
     if (.not. it_list%pass(i)%custom_adaptation) then
        it_list%pass(i)%adapt_grids = .true.
        it_list%pass(i)%adapt_weights = .true.
     end if
  end do
end subroutine iterations_list_init
```

⟨*Iterations: iterations list: TBP*⟩+≡

```
procedure :: clear => iterations_list_clear
```

⟨*Iterations: procedures*⟩+≡

```
subroutine iterations_list_clear (it_list)
  class(iterations_list_t), intent(inout) :: it_list
  it_list%n_pass = 0
  deallocate (it_list%pass)
end subroutine iterations_list_clear
```

Write the list of iterations.

⟨*Iterations: iterations list: TBP*⟩+≡

```
procedure :: write => iterations_list_write
```

⟨*Iterations: procedures*⟩+≡

```
subroutine iterations_list_write (it_list, unit)
  class(iterations_list_t), intent(in) :: it_list
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(A)")  char (it_list%to_string ())
end subroutine iterations_list_write
```

The output as a single-line string.

⟨*Iterations: iterations list: TBP*⟩+≡

```
procedure :: to_string => iterations_list_to_string
```

⟨*Iterations: procedures*⟩+≡

```
function iterations_list_to_string (it_list) result (buffer)
  class(iterations_list_t), intent(in) :: it_list
  type(string_t) :: buffer
  character(30) :: ibuf
  integer :: i
  buffer = "iterations = "
  if (it_list%n_pass > 0) then
     do i = 1, it_list%n_pass
        if (i > 1)  buffer = buffer // ", "
        write (ibuf, "(I0,':',I0)") &
             it_list%pass(i)%n_it, it_list%pass(i)%n_calls
        buffer = buffer // trim (ibuf)
        if (it_list%pass(i)%custom_adaptation &
             .or. it_list%pass(i)%adapt_grids &
             .or. it_list%pass(i)%adapt_weights) then
           buffer = buffer // ':"'
           if (it_list%pass(i)%adapt_grids)  buffer = buffer // "g"
           if (it_list%pass(i)%adapt_weights)  buffer = buffer // "w"
           buffer = buffer // '"'
        end if
     end do
  else
     buffer = buffer // "[undefined]"
  end if
end function iterations_list_to_string
```

20.2.2 Tools

Return the total number of passes.

⟨*Iterations: iterations list: TBP*⟩+≡

```
procedure :: get_n_pass => iterations_list_get_n_pass
```

⟨*Iterations: procedures*⟩+≡

```
function iterations_list_get_n_pass (it_list) result (n_pass)
  class(iterations_list_t), intent(in) :: it_list
  integer :: n_pass
  n_pass = it_list%n_pass
end function iterations_list_get_n_pass
```

Return the number of calls for a specific pass.

⟨*Iterations: iterations list: TBP*⟩+≡

```
procedure :: get_n_calls => iterations_list_get_n_calls
```

⟨*Iterations: procedures*⟩+≡

```
function iterations_list_get_n_calls (it_list, pass) result (n_calls)
  class(iterations_list_t), intent(in) :: it_list
  integer :: n_calls
  integer, intent(in) :: pass
  if (pass <= it_list%n_pass) then
     n_calls = it_list%pass(pass)%n_calls
  else
     n_calls = 0
  end if
end function iterations_list_get_n_calls
```

⟨*Iterations: iterations list: TBP*⟩+≡

```
procedure :: set_n_calls => iterations_list_set_n_calls
```

⟨*Iterations: procedures*⟩+≡

```
subroutine iterations_list_set_n_calls (it_list, pass, n_calls)
  class(iterations_list_t), intent(inout) :: it_list
  integer, intent(in) :: pass, n_calls
  it_list%pass(pass)%n_calls = n_calls
end subroutine iterations_list_set_n_calls
```

Get the adaptation mode (automatic/custom) and, for custom adaptation, the flags for a specific pass.

⟨*Iterations: iterations list: TBP*⟩+≡

```
procedure :: adapt_grids => iterations_list_adapt_grids
procedure :: adapt_weights => iterations_list_adapt_weights
```

⟨*Iterations: procedures*⟩+≡

```
function iterations_list_adapt_grids (it_list, pass) result (flag)
  logical :: flag
  class(iterations_list_t), intent(in) :: it_list
  integer, intent(in) :: pass
  if (pass <= it_list%n_pass) then
     flag = it_list%pass(pass)%adapt_grids
  else
     flag = .false.
```

```
    end if
  end function iterations_list_adapt_grids

  function iterations_list_adapt_weights (it_list, pass) result (flag)
    logical :: flag
    class(iterations_list_t), intent(in) :: it_list
    integer, intent(in) :: pass
    if (pass <= it_list%n_pass) then
       flag = it_list%pass(pass)%adapt_weights
    else
       flag = .false.
    end if
  end function iterations_list_adapt_weights
```

Return the total number of iterations / the iterations for a specific pass.

⟨*Iterations: iterations list: TBP*⟩+≡

```
  procedure :: get_n_it => iterations_list_get_n_it
```

⟨*Iterations: procedures*⟩+≡

```
  function iterations_list_get_n_it (it_list, pass) result (n_it)
    class(iterations_list_t), intent(in) :: it_list
    integer :: n_it
    integer, intent(in) :: pass
    if (pass <= it_list%n_pass) then
       n_it = it_list%pass(pass)%n_it
    else
       n_it = 0
    end if
  end function iterations_list_get_n_it
```

20.2.3 Iteration Multipliers

⟨*Iterations: public*⟩+≡

```
  public :: iteration_multipliers_t
```

⟨*Iterations: types*⟩+≡

```
  type :: iteration_multipliers_t
    real(default) :: mult_real = 1._default
    real(default) :: mult_virt = 1._default
    real(default) :: mult_dglap = 1._default
    real(default) :: mult_threshold = 1._default
    integer, dimension(:), allocatable :: n_calls0
  end type iteration_multipliers_t
```

20.2.4 Unit tests

Test module, followed by the corresponding implementation module.

⟨`iterations_ut.f90`⟩≡

⟨*File header*⟩

```
module iterations_ut
```

```
  use unit_tests
  use iterations_uti
```

⟨*Standard module head*⟩

⟨*Iterations: public test*⟩

```
contains
```

⟨*Iterations: test driver*⟩

```
end module iterations_ut
```

⟨iterations_uti.f90⟩≡

⟨*File header*⟩

```
module iterations_uti
```

⟨*Use strings*⟩

```
  use iterations
```

⟨*Standard module head*⟩

⟨*Iterations: test declarations*⟩

```
contains
```

⟨*Iterations: tests*⟩

```
end module iterations_uti
```

API: driver for the unit tests below.

⟨*Iterations: public test*⟩≡

```
  public :: iterations_test
```

⟨*Iterations: test driver*⟩≡

```
  subroutine iterations_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```

⟨*Iterations: execute tests*⟩

```
end subroutine iterations_test
```

Empty list

⟨*Iterations: execute tests*⟩≡

```
  call test (iterations_1, "iterations_1", &
       "empty iterations list", &
       u, results)
```

⟨*Iterations: test declarations*⟩≡

```
  public :: iterations_1
```

⟨*Iterations: tests*⟩≡

```
subroutine iterations_1 (u)
  integer, intent(in) :: u
  type(iterations_list_t) :: it_list

  write (u, "(A)")  "* Test output: iterations_1"
  write (u, "(A)")  "*   Purpose: display empty iterations list"
  write (u, "(A)")

  call it_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: iterations_1"

end subroutine iterations_1
```

Fill list

⟨*Iterations: execute tests*⟩+≡

```
call test (iterations_2, "iterations_2", &
     "create iterations list", &
     u, results)
```

⟨*Iterations: test declarations*⟩+≡

```
public :: iterations_2
```

⟨*Iterations: tests*⟩+≡

```
subroutine iterations_2 (u)
  integer, intent(in) :: u
  type(iterations_list_t) :: it_list

  write (u, "(A)")  "* Test output: iterations_2"
  write (u, "(A)")  "*   Purpose: fill and display iterations list"
  write (u, "(A)")

  write (u, "(A)")  "* Minimal setup (2 passes)"
  write (u, "(A)")

  call it_list%init ([2, 4], [5000, 20000])

  call it_list%write (u)
  call it_list%clear ()

  write (u, "(A)")
  write (u, "(A)")  "* Setup with flags (3 passes)"
  write (u, "(A)")

  call it_list%init ([2, 4, 5], [5000, 20000, 400], &
       [.false., .true., .true.], &
       [var_str (""), var_str ("g"), var_str ("wg")])

  call it_list%write (u)

  write (u, "(A)")
```

```
    write (u, "(A)")  "* Extract data"
    write (u, "(A)")

    write (u, "(A,I0)")  "n_pass = ", it_list%get_n_pass ()
    write (u, "(A)")
    write (u, "(A,I0)")  "n_calls(2) = ", it_list%get_n_calls (2)
    write (u, "(A)")
    write (u, "(A,I0)")  "n_it(3) = ", it_list%get_n_it (3)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: iterations_2"

  end subroutine iterations_2
```

20.3 Integration results

We record integration results and errors in a dedicated type. This allows us to do further statistics such as weighted average, chi-squared, grouping by integration passes, etc.

Note WHIZARD 2.2.0: This code is taken from the previous `processes` module essentially unchanged and converted into a separate module. It lacks an overhaul and, in particular, self-tests.

⟨`integration_results.f90`⟩≡

```
module integration_results

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_utils, only: mp_format, pac_fmt
  use format_defs, only: FMT_10, FMT_14
  use diagnostics
  use md5
  use os_interface
  use mci_base

⟨Standard module head⟩

⟨Integration results: public⟩

⟨Integration results: parameters⟩

⟨Integration results: types⟩

contains

⟨Integration results: procedures⟩

end module integration_results
```

20.3.1 Integration results entry

This object collects the results of an integration pass and makes them available to the outside.

The results object has to distinguish the process type:

We store the process type, the index of the integration pass and the absolute iteration index, the number of iterations contained in this result (for averages), and the integral (cross section or partial width), error estimate, efficiency.

For intermediate results, we set a flag if this result is an improvement w.r.t. previous ones.

The process type indicates decay or scattering. Dummy entries (skipped iterations) have a process type of PRC_UNKNOWN.

⟨*Integration results: types*⟩≡

```
type :: integration_entry_t
   private
   integer :: process_type = PRC_UNKNOWN
   integer :: pass = 0
   integer :: it = 0
   integer :: n_it = 0
   integer :: n_calls = 0
   logical :: improved = .false.
   real(default) :: integral = 0
   real(default) :: error = 0
   real(default) :: efficiency = 0
   real(default) :: chi2 = 0
   real(default), dimension(:), allocatable :: chain_weights
end type integration_entry_t
```

The possible values of the type indicator:

⟨*Integration results: parameters*⟩≡

```
integer, parameter, public :: PRC_UNKNOWN = 0
integer, parameter, public :: PRC_DECAY = 1
integer, parameter, public :: PRC_SCATTERING = 2
```

Initialize with all relevant data

⟨*Integration results: procedures*⟩≡

```
subroutine integration_entry_init (entry, &
     process_type, pass, it, n_it, n_calls, improved, &
     integral, error, efficiency, chi2, chain_weights)
  type(integration_entry_t), intent(out) :: entry
  integer, intent(in) :: process_type, pass, it, n_it, n_calls
  logical, intent(in) :: improved
  real(default), intent(in) :: integral, error, efficiency
  real(default), intent(in), optional :: chi2
  real(default), dimension(:), intent(in), optional :: chain_weights
  entry%process_type = process_type
  entry%pass = pass
  entry%it = it
  entry%n_it = n_it
  entry%n_calls = n_calls
  entry%improved = improved
  entry%integral = integral
```

```
    entry%error = error
    entry%efficiency = efficiency
    if (present (chi2)) &
         entry%chi2 = chi2
    if (present (chain_weights)) then
       allocate (entry%chain_weights (size (chain_weights)))
       entry%chain_weights = chain_weights
    end if
  end subroutine integration_entry_init
```

Access values, some of them computed on demand:

⟨*Integration results: procedures*⟩+≡

```
  elemental function integration_entry_get_pass (entry) result (n)
    integer :: n
    type(integration_entry_t), intent(in) :: entry
    n = entry%pass
  end function integration_entry_get_pass

  elemental function integration_entry_get_n_calls (entry) result (n)
    integer :: n
    type(integration_entry_t), intent(in) :: entry
    n = entry%n_calls
  end function integration_entry_get_n_calls

  elemental function integration_entry_get_integral (entry) result (int)
    real(default) :: int
    type(integration_entry_t), intent(in) :: entry
    int = entry%integral
  end function integration_entry_get_integral

  elemental function integration_entry_get_error (entry) result (err)
    real(default) :: err
    type(integration_entry_t), intent(in) :: entry
    err = entry%error
  end function integration_entry_get_error

  elemental function integration_entry_get_relative_error (entry) result (err)
    real(default) :: err
    type(integration_entry_t), intent(in) :: entry
    if (entry%integral /= 0) then
       err = entry%error / entry%integral
    else
       err = 0
    end if
  end function integration_entry_get_relative_error

  elemental function integration_entry_get_accuracy (entry) result (acc)
    real(default) :: acc
    type(integration_entry_t), intent(in) :: entry
    acc = accuracy (entry%integral, entry%error, entry%n_calls)
  end function integration_entry_get_accuracy

  elemental function accuracy (integral, error, n_calls) result (acc)
    real(default) :: acc
```

```
  real(default), intent(in) :: integral, error
  integer, intent(in) :: n_calls
  if (integral /= 0) then
     acc = error / integral * sqrt (real (n_calls, default))
  else
     acc = 0
  end if
end function accuracy

elemental function integration_entry_get_efficiency (entry) result (eff)
  real(default) :: eff
  type(integration_entry_t), intent(in) :: entry
  eff = entry%efficiency
end function integration_entry_get_efficiency

elemental function integration_entry_get_chi2 (entry) result (chi2)
  real(default) :: chi2
  type(integration_entry_t), intent(in) :: entry
  chi2 = entry%chi2
end function integration_entry_get_chi2

elemental function integration_entry_has_improved (entry) result (flag)
  logical :: flag
  type(integration_entry_t), intent(in) :: entry
  flag = entry%improved
end function integration_entry_has_improved

elemental function integration_entry_get_n_groves (entry) result (n_groves)
  integer :: n_groves
  type(integration_entry_t), intent(in) :: entry
  if (allocated (entry%chain_weights)) then
     n_groves = size (entry%chain_weights, 1)
  else
     n_groves = 0
  end if
end function integration_entry_get_n_groves
```

Output. This writes the header line for the result account below:

⟨*Integration results: procedures*⟩+≡

```
subroutine write_header (process_type, unit, logfile)
  integer, intent(in) :: process_type
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: logfile
  character(5) :: phys_unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  select case (process_type)
  case (PRC_DECAY);      phys_unit = "[GeV]"
  case (PRC_SCATTERING); phys_unit = "[fb] "
  case default
     phys_unit = ""
  end select
  write (msg_buffer, "(A)") &
       "It      Calls  Integral" // phys_unit // &
```

```
        " Error" // phys_unit // &
        "  Err[%]    Acc  Eff[%]   Chi2 N[It] |"
    call msg_message (unit=u, logfile=logfile)
  end subroutine write_header
```

This writes a separator for result display:

⟨*Integration results: procedures*⟩+≡

```
  subroutine write_hline (unit)
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(A)")  "|" // (repeat ("-", 77)) // "|"
    flush (u)
  end subroutine write_hline

  subroutine write_dline (unit)
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(A)")  "|" // (repeat ("=", 77)) // "|"
    flush (u)
  end subroutine write_dline
```

This writes the standard result account into one screen line. The verbose version uses multiple lines and prints the unabridged values. Dummy entries are not written.

⟨*Integration results: procedures*⟩+≡

```
  subroutine integration_entry_write (entry, unit, verbose, suppress)
    type(integration_entry_t), intent(in) :: entry
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    logical, intent(in), optional :: suppress
    integer :: u
    character(1) :: star
    character(12) :: fmt
    character(7) :: fmt2
    logical :: verb, supp
    u = given_output_unit (unit);  if (u < 0)  return
    verb = .false.;  if (present (verbose))  verb = verbose
    supp = .false.;  if (present (suppress)) supp = suppress
    if (verb)  then
       write (u, *)  "process_type = ", entry%process_type
       write (u, *)  "        pass = ", entry%pass
       write (u, *)  "          it = ", entry%it
       write (u, *)  "        n_it = ", entry%n_it
       write (u, *)  "     n_calls = ", entry%n_calls
       write (u, *)  "    improved = ", entry%improved
       write (u, *)  "    integral = ", entry%integral
       write (u, *)  "       error = ", entry%error
       write (u, *)  "  efficiency = ", entry%efficiency
       write (u, *)  "        chi2 = ", entry%chi2
       if (allocated (entry%chain_weights)) then
          write (u, *)  "    n_groves = ", size (entry%chain_weights)
```

```
         write (u, *)  "chain_weights = ", entry%chain_weights
      else
         write (u, *)  "    n_groves = 0"
      end if
   else if (entry%process_type /= PRC_UNKNOWN) then
      if (entry%improved .and. .not. supp) then
         star = "*"
      else
         star = " "
      end if
      call pac_fmt (fmt, FMT_14, "3x," // FMT_10 // ",1x", suppress)
      call pac_fmt (fmt2, "1x,F6.2", "2x,F5.1", suppress)
      if (entry%n_it /= 1) then
         write (u, "(1x,I3,1x,I10,1x," // fmt // ",1x,ES9.2,1x,F7.2," // &
           "1x,F7.2,A1," // fmt2 // ",1x,F7.2,1x,I3)") &
              entry%it, &
              entry%n_calls, &
              entry%integral, &
              abs(entry%error), &
              abs(integration_entry_get_relative_error (entry)) * 100, &
              abs(integration_entry_get_accuracy (entry)), &
              star, &
              entry%efficiency * 100, &
              entry%chi2, &
              entry%n_it
      else
         write (u, "(1x,I3,1x,I10,1x," // fmt // ",1x,ES9.2,1x,F7.2," // &
           "1x,F7.2,A1," // fmt2 // ",1x,F7.2,1x,I3)") &
              entry%it, &
              entry%n_calls, &
              entry%integral, &
              abs(entry%error), &
              abs(integration_entry_get_relative_error (entry)) * 100, &
              abs(integration_entry_get_accuracy (entry)), &
              star, &
              entry%efficiency * 100
      end if
   end if
   flush (u)
end subroutine integration_entry_write
```

Read the entry, assuming it has been written in verbose format.

⟨*Integration results: procedures*⟩+≡

```
subroutine integration_entry_read (entry, unit)
  type(integration_entry_t), intent(out) :: entry
  integer, intent(in) :: unit
  character(30) :: dummy
  character :: equals
  integer :: n_groves
  read (unit, *)  dummy, equals, entry%process_type
  read (unit, *)  dummy, equals, entry%pass
  read (unit, *)  dummy, equals, entry%it
  read (unit, *)  dummy, equals, entry%n_it
  read (unit, *)  dummy, equals, entry%n_calls
```

```
    read (unit, *)  dummy, equals, entry%improved
    read (unit, *)  dummy, equals, entry%integral
    read (unit, *)  dummy, equals, entry%error
    read (unit, *)  dummy, equals, entry%efficiency
    read (unit, *)  dummy, equals, entry%chi2
    read (unit, *)  dummy, equals, n_groves
    if (n_groves /= 0) then
       allocate (entry%chain_weights (n_groves))
       read (unit, *)  dummy, equals, entry%chain_weights
    end if
  end subroutine integration_entry_read
```

Write an account of the channel weights, accumulated by groves.

⟨*Integration results: procedures*⟩+≡

```
  subroutine integration_entry_write_chain_weights (entry, unit)
    type(integration_entry_t), intent(in) :: entry
    integer, intent(in), optional :: unit
    integer :: u, i
    u = given_output_unit (unit);  if (u < 0)  return
    if (allocated (entry%chain_weights)) then
       do i = 1, size (entry%chain_weights)
          write (u, "(1x,I3)", advance="no")  nint (entry%chain_weights(i) * 100)
       end do
       write (u, *)
    end if
  end subroutine integration_entry_write_chain_weights
```

Compute the average for all entries in the specified integration pass. The integrals are weighted w.r.t. their individual errors.

The quoted error of the result is the expected error, computed from the weighted average of the given individual errors.

This should be compared to the actual distribution of the results, from which we also can compute an error estimate if there is more than one iteration. The ratio of the distribution error and the averaged error, is the χ^2 value.

All error distributions are assumed Gaussian, of course. The χ^2 value is a partial check for this assumption. If it is significantly greater than unity, there is something wrong with the individual errors.

The efficiency returned is the one of the last entry in the integration pass.

If any error vanishes, averaging by this algorithm would fail. In this case, we simply average the entries and use the deviations from this average (if any) to estimate the error.

⟨*Integration results: procedures*⟩+≡

```
  function compute_average (entry, pass) result (result)
    type(integration_entry_t) :: result
    type(integration_entry_t), dimension(:), intent(in) :: entry
    integer, intent(in) :: pass
    integer :: i
    logical, dimension(size(entry)) :: mask
    real(default), dimension(size(entry)) :: ivar
    real(default) :: sum_ivar, variance
    result%process_type = entry(1)%process_type
    result%pass = pass
```

```
  mask = entry%pass == pass .and. entry%process_type /= PRC_UNKNOWN
  result%it = maxval (entry%it, mask)
  result%n_it = count (mask)
  result%n_calls = sum (entry%n_calls, mask)
  if (.not. any (mask .and. entry%error == 0)) then
     where (mask)
        ivar = 1 / entry%error ** 2
     elsewhere
        ivar = 0
     end where
     sum_ivar = sum (ivar, mask)
     if (sum_ivar /= 0) then
        variance = 1 / sum_ivar
     else
        variance = 0
     end if
     result%integral = sum (entry%integral * ivar, mask) * variance
     if (result%n_it > 1) then
        result%chi2 = &
             sum ((entry%integral - result%integral)**2 * ivar, mask) &
             / (result%n_it - 1)
     end if
  else if (result%n_it /= 0) then
     result%integral = sum (entry%integral, mask) / result%n_it
     if (result%n_it > 1) then
        variance = &
             sum ((entry%integral - result%integral)**2, mask) &
             / (result%n_it - 1)
        if (result%integral /= 0) then
           if (abs (variance / result%integral) &
                < 100 * epsilon (1._default)) then
              variance = 0
           end if
        end if
        result%chi2 = variance / result%n_it
     else
        variance = 0
     end if
  end if
  result%error = sqrt (variance)
  do i = size (entry), 1, -1
     if (mask(i)) then
        result%efficiency = entry(i)%efficiency
        exit
     end if
  end do
end function compute_average
```

20.3.2 Combined integration results

We collect a list of results which grows during the execution of the program. This is implemented as an array which grows if necessary; so we can easily

compute averages.

We implement this as an extension of the `mci_results_t` which is defined in `mci_base` as an abstract type. We thus decouple the implementation of the integrator from the implementation of the results display, but nevertheless can record intermediate results during integration. This implies that the present extension implements a `record` method.

⟨*Integration results: public*⟩≡
```
public :: integration_results_t
```

⟨*Integration results: types*⟩+≡
```
type, extends (mci_results_t) :: integration_results_t
   private
   integer :: process_type = PRC_UNKNOWN
   integer :: current_pass = 0
   integer :: n_pass = 0
   integer :: n_it = 0
   logical :: screen = .false.
   integer :: unit = 0
   real(default) :: error_threshold = 0
   type(integration_entry_t), dimension(:), allocatable :: entry
   type(integration_entry_t), dimension(:), allocatable :: average
 contains
 ⟨Integration results: integration results: TBP⟩
end type integration_results_t
```

The array is extended in chunks of 10 entries.

⟨*Integration results: parameters*⟩+≡
```
integer, parameter :: RESULTS_CHUNK_SIZE = 10
```

The standard does not require to explicitly initialize the integers; however, some gfortran version has a bug here and misses the default initialization in the type definition.

⟨*Integration results: integration results: TBP*⟩≡
```
procedure :: init => integration_results_init
```

⟨*Integration results: procedures*⟩+≡
```
subroutine integration_results_init (results, process_type)
  class(integration_results_t), intent(out) :: results
  integer, intent(in) :: process_type
  results%process_type = process_type
  results%n_pass = 0
  results%n_it = 0
  allocate (results%entry (RESULTS_CHUNK_SIZE))
  allocate (results%average (RESULTS_CHUNK_SIZE))
end subroutine integration_results_init
```

Set addititional parameters: the `error_threshold` declares that any error value (in absolute numbers) smaller than this is to be considered zero.

⟨*Integration results: integration results: TBP*⟩+≡
```
procedure :: set_error_threshold => integration_results_set_error_threshold
```

⟨*Integration results: procedures*⟩+≡

```
subroutine integration_results_set_error_threshold (results, error_threshold)
  class(integration_results_t), intent(inout) :: results
  real(default), intent(in) :: error_threshold
  results%error_threshold = error_threshold
end subroutine integration_results_set_error_threshold
```

Output (ASCII format). The `verbose` format is used for writing the header in grid files.

⟨*Integration results: integration results: TBP*⟩+≡

```
procedure :: write => integration_results_write
```

⟨*Integration results: procedures*⟩+≡

```
subroutine integration_results_write (object, unit, verbose, suppress)
  class(integration_results_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  logical, intent(in), optional :: suppress
  logical :: verb
  integer :: u, n
  u = given_output_unit (unit);  if (u < 0)  return
  verb = .false.;  if (present (verbose))  verb = verbose
  if (.not. verb) then
     call write_dline (unit)
     if (object%n_it /= 0) then
        call write_header (object%entry(1)%process_type, unit, &
             logfile=.false.)
        call write_dline (unit)
        do n = 1, object%n_it
           if (n > 1) then
              if (object%entry(n)%pass /= object%entry(n-1)%pass) then
                 call write_hline (unit)
                 call integration_entry_write &
                      (object%average(object%entry(n-1)%pass), &
                         unit, suppress = suppress)
                 call write_hline (unit)
              end if
           end if
           call integration_entry_write (object%entry(n), unit, &
                   suppress = suppress)
        end do
        call write_hline(unit)
        call integration_entry_write (object%average(object%n_pass), &
                 unit, suppress = suppress)
     else
        call msg_message ("[WHIZARD integration results: empty]", unit)
     end if
     call write_dline (unit)
  else
     write (u, *)  "begin(integration_results)"
     write (u, *)  "  n_pass = ", object%n_pass
     write (u, *)  "    n_it = ", object%n_it
     if (object%n_it > 0) then
        write (u, *)  "begin(integration_pass)"
```

```
      do n = 1, object%n_it
         if (n > 1) then
            if (object%entry(n)%pass /= object%entry(n-1)%pass) then
               write (u, *)  "end(integration_pass)"
               write (u, *)  "begin(integration_pass)"
            end if
         end if
         write (u, *)  "begin(iteration)"
         call integration_entry_write (object%entry(n), unit, &
                  verbose = verb, suppress = suppress)
         write (u, *)  "end(iteration)"
      end do
      write (u, *)  "end(integration_pass)"
   end if
   write (u, *)  "end(integration_results)"
  end if
  flush (u)
end subroutine integration_results_write
```

During integration, we do not want to print all results at once, but each intermediate result as soon as we get it. Thus, the previous procedure is chopped in pieces. First piece: store the output unit and a flag whether we want to print to standard output as well. Then write the header if the results are still empty, i.e., before integration has started. The second piece writes a single result to the saved output channels. We call this from the `record` method, which can be called from the integrator directly. The third piece writes the average result, once a pass has been completed. The fourth piece writes a footer (if any), assuming that this is the final result.

⟨*Integration results: integration results: TBP*⟩+≡

```
procedure :: display_init => integration_results_display_init
procedure :: display_current => integration_results_display_current
procedure :: display_pass => integration_results_display_pass
procedure :: display_final => integration_results_display_final
```

⟨*Integration results: procedures*⟩+≡

```
subroutine integration_results_display_init &
     (results, process_type, screen, unit)
  class(integration_results_t), intent(inout) :: results
  integer, intent(in) :: process_type
  logical, intent(in) :: screen
  integer, intent(in), optional :: unit
  integer :: u
  if (present (unit))  results%unit = unit
  u = given_output_unit ()
  results%screen = screen
  if (results%n_it == 0) then
     if (results%screen) then
        call write_dline (u)
        call write_header (process_type, u, &
             logfile=.false.)
        call write_dline (u)
     end if
     if (results%unit /= 0) then
```

```
         call write_dline (results%unit)
         call write_header (process_type, results%unit, &
              logfile=.false.)
         call write_dline (results%unit)
      end if
   else
      if (results%screen) then
         call write_hline (u)
      end if
      if (results%unit /= 0) then
         call write_hline (results%unit)
      end if
   end if
end subroutine integration_results_display_init

subroutine integration_results_display_current (results, pacify)
   class(integration_results_t), intent(in) :: results
   integer :: u
   logical, intent(in), optional :: pacify
   u = given_output_unit ()
   if (results%screen) then
      call integration_entry_write (results%entry(results%n_it), u, &
           suppress = pacify)
   end if
   if (results%unit /= 0) then
      call integration_entry_write (results%entry(results%n_it), &
           results%unit, suppress = pacify)
   end if
end subroutine integration_results_display_current

subroutine integration_results_display_pass (results, pacify)
   class(integration_results_t), intent(in) :: results
   logical, intent(in), optional :: pacify
   integer :: u
   u = given_output_unit ()
   if (results%screen) then
      call write_hline (u)
      call integration_entry_write &
           (results%average(results%entry(results%n_it)%pass), &
               u, suppress = pacify)
   end if
   if (results%unit /= 0) then
      call write_hline (results%unit)
      call integration_entry_write &
           (results%average(results%entry(results%n_it)%pass), &
               results%unit, suppress = pacify)
   end if
end subroutine integration_results_display_pass

subroutine integration_results_display_final (results)
   class(integration_results_t), intent(inout) :: results
   integer :: u
   u = given_output_unit ()
   if (results%screen) then
```

```
      call write_dline (u)
    end if
    if (results%unit /= 0) then
      call write_dline (results%unit)
    end if
    results%screen = .false.
    results%unit = 0
  end subroutine integration_results_display_final
```

Write a concise table of chain weights, i.e., the channel history where channels are collected by chains.

⟨*Integration results: integration results: TBP*⟩+≡

```
  procedure :: write_chain_weights => &
      integration_results_write_chain_weights
```

⟨*Integration results: procedures*⟩+≡

```
  subroutine integration_results_write_chain_weights (results, unit)
    class(integration_results_t), intent(in) :: results
    integer, intent(in), optional :: unit
    integer :: u, i, n
    u = given_output_unit (unit);  if (u < 0)  return
    if (allocated (results%entry(1)%chain_weights) .and. results%n_it /= 0) then
      call msg_message ("Phase-space chain (grove) weight history: " &
          // "(numbers in %)", unit)
      write (u, "(A9)", advance="no")  "| chain |"
    do i = 1, integration_entry_get_n_groves (results%entry(1))
        write (u, "(1x,I3)", advance="no")  i
      end do
      write (u, *)
      call write_dline (unit)
      do n = 1, results%n_it
        if (n > 1) then
          if (results%entry(n)%pass /= results%entry(n-1)%pass) then
            call write_hline (unit)
          end if
        end if
        write (u, "(1x,I6,1x,A1)", advance="no")  n, "|"
        call integration_entry_write_chain_weights (results%entry(n), unit)
      end do
      flush (u)
      call write_dline(unit)
    end if
  end subroutine integration_results_write_chain_weights
```

Read the list from file. The file must be written using the `verbose` option of the writing routine.

⟨*Integration results: procedures*⟩+≡

```
  subroutine integration_results_read (results, unit)
    type(integration_results_t), intent(out) :: results
    integer, intent(in) :: unit
    character(80) :: buffer
    character :: equals
    integer :: pass, it
```

```
  read (unit, *)  buffer
  if (trim (adjustl (buffer)) /= "begin(integration_results)") then
     call read_err ();  return
  end if
  read (unit, *)  buffer, equals, results%n_pass
  read (unit, *)  buffer, equals, results%n_it
  allocate (results%entry (results%n_it + RESULTS_CHUNK_SIZE))
  allocate (results%average (results%n_it + RESULTS_CHUNK_SIZE))
  it = 0
  do pass = 1, results%n_pass
     read (unit, *)  buffer
     if (trim (adjustl (buffer)) /= "begin(integration_pass)") then
        call read_err ();  return
     end if
     READ_ENTRIES: do
        read (unit, *)  buffer
        if (trim (adjustl (buffer)) /= "begin(iteration)") then
           exit READ_ENTRIES
        end if
        it = it + 1
        call integration_entry_read (results%entry(it), unit)
        read (unit, *)  buffer
        if (trim (adjustl (buffer)) /= "end(iteration)") then
           call read_err (); return
        end if
     end do READ_ENTRIES
     if (trim (adjustl (buffer)) /= "end(integration_pass)") then
        call read_err (); return
     end if
     results%average(pass) = compute_average (results%entry, pass)
  end do
  read (unit, *)  buffer
  if (trim (adjustl (buffer)) /= "end(integration_results)") then
     call read_err (); return
  end if
contains
  subroutine read_err ()
    call msg_fatal ("Reading integration results from file: syntax error")
  end subroutine read_err
end subroutine integration_results_read
```

Check integration results for consistency. We compare against an array of pass indices and call numbers. If there is a difference, up to the number of iterations done so far, we return failure. Dummy entries (where pass = 0) are ignored.

⟨*Integration results: procedures*⟩+≡

```
function integration_results_iterations_are_consistent &
     (results, pass, n_calls) result (flag)
  logical :: flag
  type(integration_results_t), intent(in) :: results
  integer, dimension(:), intent(in) :: pass, n_calls
  integer :: n_it
  n_it = results%n_it
  flag = size (pass) >= n_it .and. size (n_calls) >= n_it
```

```
  if (flag) then
     flag = all (results%entry(:n_it)%pass == pass(:n_it) &
                 .and. &
                 (results%entry(:n_it)%n_calls == n_calls(:n_it) &
                  .or. &
                  results%entry(:n_it)%process_type == PRC_UNKNOWN))
  end if
end function integration_results_iterations_are_consistent
```

Discard all results starting from the specified iteration.

⟨*Integration results: procedures*⟩+≡

```
subroutine integration_results_discard (results, it)
  type(integration_results_t), intent(inout) :: results
  integer, intent(in) :: it
  if (it <= results%n_it) then
     select case (it)
     case (:1)
        results%n_it = 0
        results%n_pass = 0
        results%current_pass = 0
     case default
        results%n_it = it - 1
        results%n_pass = maxval (results%entry(1:results%n_it)%pass)
        results%current_pass = results%n_pass
     end select
  end if
end subroutine integration_results_discard
```

Expand the list of entries if the limit has been reached:

⟨*Integration results: integration results: TBP*⟩+≡

```
procedure :: expand => integration_results_expand
```

⟨*Integration results: procedures*⟩+≡

```
subroutine integration_results_expand (results)
  class(integration_results_t), intent(inout) :: results
  type(integration_entry_t), dimension(:), allocatable :: entry_tmp
  if (results%n_it == size (results%entry)) then
     allocate (entry_tmp (results%n_it))
     entry_tmp = results%entry
     deallocate (results%entry)
     allocate (results%entry (results%n_it + RESULTS_CHUNK_SIZE))
     results%entry(:results%n_it) = entry_tmp
     deallocate (entry_tmp)
  end if
  if (results%n_pass == size (results%average)) then
     allocate (entry_tmp (results%n_pass))
     entry_tmp = results%average
     deallocate (results%average)
     allocate (results%average (results%n_it + RESULTS_CHUNK_SIZE))
     results%average(:results%n_pass) = entry_tmp
     deallocate (entry_tmp)
  end if
end subroutine integration_results_expand
```

Increment the current_pass counter. Can be done before integration; after integration, the recording method may use the value of this counter to define the entry.

⟨*Integration results: integration results: TBP*⟩+≡
```
procedure :: new_pass => integration_results_new_pass
```

⟨*Integration results: procedures*⟩+≡
```
subroutine integration_results_new_pass (results)
  class(integration_results_t), intent(inout) :: results
  results%current_pass = results%current_pass + 1
end subroutine integration_results_new_pass
```

Append a new entry to the list and, if appropriate, compute the average.

⟨*Integration results: integration results: TBP*⟩+≡
```
procedure :: append_entry => integration_results_append_entry
```

⟨*Integration results: procedures*⟩+≡
```
subroutine integration_results_append_entry (results, entry)
  class(integration_results_t), intent(inout) :: results
  type(integration_entry_t), intent(in), optional :: entry
  if (results%n_it == 0) then
     results%n_it = 1
     results%n_pass = 1
  else
     call results%expand ()
     if (present (entry)) then
        if (entry%pass /= results%entry(results%n_it)%pass) &
             results%n_pass = results%n_pass + 1
     end if
     results%n_it = results%n_it + 1
  end if
  if (present (entry)) then
     results%entry(results%n_it) = entry
     results%average(results%n_pass) = &
          compute_average (results%entry, entry%pass)
  end if
end subroutine integration_results_append_entry
```

Enter results into the results list. For the error value, we may compare them with a given threshold. This guards against numerical noise, if the exact error would be zero.

⟨*Integration results: integration results: TBP*⟩+≡
```
procedure :: append => integration_results_append
```

⟨*Integration results: procedures*⟩+≡
```
subroutine integration_results_append (results, &
     n_it, n_calls, &
     integral, error, efficiency, &
     chain_weights)
  class(integration_results_t), intent(inout) :: results
  integer, intent(in) :: n_it, n_calls
  real(default), intent(in) :: integral, error, efficiency
  real(default), dimension(:), intent(in), optional :: chain_weights
  logical :: improved
```

```
    type(integration_entry_t) :: entry
    real(default) :: err_checked
    if (results%n_it /= 0) then
       improved = abs(accuracy (integral, error, n_calls)) &
            < abs(integration_entry_get_accuracy (results%entry(results%n_it)))
    else
       improved = .true.
    end if
    if (abs (error) >= results%error_threshold) then
       err_checked = error
    else
       err_checked = 0
    end if
    call integration_entry_init (entry, &
         results%process_type, results%current_pass, &
         results%n_it+1, n_it, n_calls, improved, &
         integral, err_checked, efficiency, &
         chain_weights=chain_weights)
    call results%append_entry (entry)
  end subroutine integration_results_append
```

Enter an empty result into the results list.

⟨*Integration results: public*⟩+≡

```
  public :: integration_results_append_null
```

⟨*Integration results: procedures*⟩+≡

```
  subroutine integration_results_append_null (results, pass, n_it)
    type(integration_results_t), intent(inout) :: results
    integer, intent(in) :: pass, n_it
    type(integration_entry_t) :: entry
    call integration_entry_init (entry, &
         PRC_UNKNOWN, results%current_pass, n_it, 1, 0, .false., &
         0._default, 0._default, 0._default)
    call results%append_entry (entry)
  end subroutine integration_results_append_null
```

Record an integration pass executed by an `mci` integrator object.

There is a tolerance below we treat an error (relative to the integral) as zero.

⟨*Integration results: parameters*⟩+≡

```
  real(default), parameter, public :: INTEGRATION_ERROR_TOLERANCE = 1e-10
```

⟨*Integration results: integration results: TBP*⟩+≡

```
  procedure :: record => integration_results_record
```

⟨*Integration results: procedures*⟩+≡

```
  subroutine integration_results_record &
       (object, n_it, n_calls, integral, error, efficiency, &
        chain_weights, suppress)
    class(integration_results_t), intent(inout) :: object
    integer, intent(in) :: n_it, n_calls
    real(default), intent(in) :: integral, error, efficiency
    real(default), dimension(:), intent(in), optional :: chain_weights
    real(default) :: err
    logical, intent(in), optional :: suppress
```

```
    if (abs (error) >= abs (integral) * INTEGRATION_ERROR_TOLERANCE) then
       err = error
    else
       err = 0
    end if
    call object%append (n_it, n_calls, integral, err, efficiency, chain_weights)
    call object%display_current (suppress)
  end subroutine integration_results_record
```

20.3.3 Access results

Return true if the results object has entries.

⟨*Integration results: integration results: TBP*⟩+≡

```
  procedure :: exist => integration_results_exist
```

⟨*Integration results: procedures*⟩+≡

```
  function integration_results_exist (results) result (flag)
    logical :: flag
    class(integration_results_t), intent(in) :: results
    flag = results%n_pass > 0
  end function integration_results_exist
```

Retrieve information from the results record. If `last` is set and true, take the last iteration. If `it` is set instead, take this iteration. If `pass` is set, take this average. If none is set, take the final average.

If the result would be invalid, the entry is not assigned. Due to default initialization, this returns a null entry.

⟨*Integration results: integration results: TBP*⟩+≡

```
  procedure :: get_entry => results_get_entry
```

⟨*Integration results: procedures*⟩+≡

```
  function results_get_entry (results, last, it, pass) result (entry)
    class(integration_results_t), intent(in) :: results
    type(integration_entry_t) :: entry
    logical, intent(in), optional :: last
    integer, intent(in), optional :: it, pass
    if (present (last)) then
       if (allocated (results%entry) .and. results%n_it > 0) then
          entry = results%entry(results%n_it)
       else
          call error ()
       end if
    else if (present (it)) then
       if (allocated (results%entry) .and. it > 0 .and. it <= results%n_it) then
          entry = results%entry(it)
       else
          call error ()
       end if
    else if (present (pass)) then
       if (allocated (results%average) &
            .and. pass > 0 .and. pass <= results%n_pass) then
```

```
        entry = results%average (pass)
      else
        call error ()
      end if
    else
      if (allocated (results%average) .and. results%n_pass > 0) then
        entry = results%average (results%n_pass)
      else
        call error ()
      end if
    end if
  contains
    subroutine error ()
      call msg_fatal ("Requested integration result is not available")
    end subroutine error
  end function results_get_entry
```

The individual procedures. The `results` record should have the `target` attribute, but only locally within the function.

⟨*Integration results: integration results: TBP*⟩+≡

```
  procedure :: get_n_calls => integration_results_get_n_calls
  procedure :: get_integral => integration_results_get_integral
  procedure :: get_error => integration_results_get_error
  procedure :: get_accuracy => integration_results_get_accuracy
  procedure :: get_chi2 => integration_results_get_chi2
  procedure :: get_efficiency => integration_results_get_efficiency
```

⟨*Integration results: procedures*⟩+≡

```
  function integration_results_get_n_calls (results, last, it, pass) &
      result (n_calls)
    class(integration_results_t), intent(in), target :: results
    integer :: n_calls
    logical, intent(in), optional :: last
    integer, intent(in), optional :: it, pass
    n_calls = integration_entry_get_n_calls &
        (results%get_entry (last, it, pass))
  end function integration_results_get_n_calls

  function integration_results_get_integral (results, last, it, pass) &
      result (integral)
    class(integration_results_t), intent(in), target :: results
    real(default) :: integral
    logical, intent(in), optional :: last
    integer, intent(in), optional :: it, pass
    integral = integration_entry_get_integral &
        (results%get_entry (last, it, pass))
  end function integration_results_get_integral

  function integration_results_get_error (results, last, it, pass) &
      result (error)
    class(integration_results_t), intent(in), target :: results
    real(default) :: error
    logical, intent(in), optional :: last
    integer, intent(in), optional :: it, pass
```

```
  error = integration_entry_get_error &
        (results%get_entry (last, it, pass))
end function integration_results_get_error

function integration_results_get_accuracy (results, last, it, pass) &
     result (accuracy)
  class(integration_results_t), intent(in), target :: results
  real(default) :: accuracy
  logical, intent(in), optional :: last
  integer, intent(in), optional :: it, pass
  accuracy = integration_entry_get_accuracy &
        (results%get_entry (last, it, pass))
end function integration_results_get_accuracy

function integration_results_get_chi2 (results, last, it, pass) &
     result (chi2)
  class(integration_results_t), intent(in), target :: results
  real(default) :: chi2
  logical, intent(in), optional :: last
  integer, intent(in), optional :: it, pass
  chi2 = integration_entry_get_chi2 &
        (results%get_entry (last, it, pass))
end function integration_results_get_chi2

function integration_results_get_efficiency (results, last, it, pass) &
     result (efficiency)
  class(integration_results_t), intent(in), target :: results
  real(default) :: efficiency
  logical, intent(in), optional :: last
  integer, intent(in), optional :: it, pass
  efficiency = integration_entry_get_efficiency &
        (results%get_entry (last, it, pass))
end function integration_results_get_efficiency
```

Return the last pass index and the index of the last iteration *within* the last pass. The third routine returns the absolute index of the last iteration.

⟨*Integration results: procedures*⟩+≡

```
function integration_results_get_current_pass (results) result (pass)
  integer :: pass
  type(integration_results_t), intent(in) :: results
  pass = results%n_pass
end function integration_results_get_current_pass

function integration_results_get_current_it (results) result (it)
  integer :: it
  type(integration_results_t), intent(in) :: results
  if (allocated (results%entry)) then
     it = count (results%entry(1:results%n_it)%pass == results%n_pass)
  else
     it = 0
  end if
end function integration_results_get_current_it
```

```
function integration_results_get_last_it (results) result (it)
  integer :: it
  type(integration_results_t), intent(in) :: results
  it = results%n_it
end function integration_results_get_last_it
```

Return the index of the best iteration (lowest accuracy value) within the current pass. If none qualifies, return zero.

⟨*Integration results: procedures*⟩+≡

```
function integration_results_get_best_it (results) result (it)
  integer :: it
  type(integration_results_t), intent(in) :: results
  integer :: i
  real(default) :: acc, acc_best
  acc_best = -1
  it = 0
  do i = 1, results%n_it
     if (results%entry(i)%pass == results%n_pass) then
        acc = integration_entry_get_accuracy (results%entry(i))
        if (acc_best < 0 .or. acc <= acc_best) then
           acc_best = acc
           it = i
        end if
     end if
  end do
end function integration_results_get_best_it
```

Compute the MD5 sum by printing everything and checksumming the resulting file.

⟨*Integration results: procedures*⟩+≡

```
function integration_results_get_md5sum (results) result (md5sum_results)
  character(32) :: md5sum_results
  type(integration_results_t), intent(in) :: results
  integer :: u
  u = free_unit ()
  open (unit = u, status = "scratch", action = "readwrite")
  call integration_results_write (results, u, verbose=.true.)
  rewind (u)
  md5sum_results = md5sum (u)
  close (u)
end function integration_results_get_md5sum
```

This is (ab)used to suppress numerical noise when integrating constant matrix elements.

⟨*Integration results: integration results: TBP*⟩+≡

```
procedure :: pacify => integration_results_pacify
```

⟨*Integration results: procedures*⟩+≡

```
subroutine integration_results_pacify (results, efficiency_reset)
  class(integration_results_t), intent(inout) :: results
  logical, intent(in), optional :: efficiency_reset
  integer :: i
  logical :: reset
```

```
    reset = .false.
    if (present (efficiency_reset))  reset = efficiency_reset
    if (allocated (results%entry)) then
       do i = 1, size (results%entry)
          call pacify (results%entry(i)%error, &
               results%entry(i)%integral * 1.E-9_default)
          if (reset)  results%entry(i)%efficiency = 1
       end do
    end if
    if (allocated (results%average)) then
       do i = 1, size (results%average)
          call pacify (results%average(i)%error, &
               results%average(i)%integral * 1.E-9_default)
          if (reset)  results%average(i)%efficiency = 1
       end do
    end if
  end subroutine integration_results_pacify
```

⟨*Integration results: integration results: TBP*⟩+≡

```
  procedure :: record_correction => integration_results_record_correction
```

⟨*Integration results: procedures*⟩+≡

```
  subroutine integration_results_record_correction (object, corr, err)
    class(integration_results_t), intent(inout) :: object
    real(default), intent(in) :: corr, err
    integer :: u
    u = given_output_unit ()
    if (object%screen) then
      call write_hline (u)
      call msg_message ("NLO Correction: [O(alpha_s+1)/O(alpha_s)]")
      write(msg_buffer,'(1X,A1,F8.4,A4,F9.5,1X,A3)') '(', corr, ' +- ', err, ') %'
      call msg_message ()
    end if
  end subroutine integration_results_record_correction
```

20.3.4 Results display

Write a driver file for history visualization.

The ratio of y range over y value must not become too small, otherwise we run into an arithmetic overflow in GAMELAN. 2% appears to be safe.

⟨*Integration results: parameters*⟩+≡

```
  real, parameter, public :: GML_MIN_RANGE_RATIO = 0.02
```

⟨*Integration results: public*⟩+≡

```
  public :: integration_results_write_driver
```

⟨*Integration results: procedures*⟩+≡

```
  subroutine integration_results_write_driver (results, filename, eff_reset)
    type(integration_results_t), intent(inout) :: results
    type(string_t), intent(in) :: filename
    logical, intent(in), optional :: eff_reset
    type(string_t) :: file_tex
    integer :: unit
```

```
integer :: n, i, n_pass, pass
integer, dimension(:), allocatable :: ipass
real(default) :: ymin, ymax, yavg, ydif, y0, y1
logical :: reset
file_tex = filename // ".tex"
unit = free_unit ()
open (unit=unit, file=char(file_tex), action="write", status="replace")
reset = .false.; if (present (eff_reset))  reset = eff_reset
n = results%n_it
n_pass = results%n_pass
allocate (ipass (results%n_pass))
ipass(1) = 0
pass = 2
do i = 1, n-1
   if (integration_entry_get_pass (results%entry(i)) &
       /= integration_entry_get_pass (results%entry(i+1))) then
      ipass(pass) = i
      pass = pass + 1
   end if
end do
ymin = minval (integration_entry_get_integral (results%entry(:n)) &
               - integration_entry_get_error (results%entry(:n)))
ymax = maxval (integration_entry_get_integral (results%entry(:n)) &
               + integration_entry_get_error (results%entry(:n)))
yavg = (ymax + ymin) / 2
ydif = (ymax - ymin)
if (ydif * 1.5 > GML_MIN_RANGE_RATIO * yavg) then
   y0 = yavg - ydif * 0.75
   y1 = yavg + ydif * 0.75
else
   y0 = yavg * (1 - GML_MIN_RANGE_RATIO / 2)
   y1 = yavg * (1 + GML_MIN_RANGE_RATIO / 2)
end if
write (unit, "(A)") "\documentclass{article}"
write (unit, "(A)") "\usepackage{a4wide}"
write (unit, "(A)") "\usepackage{gamelan}"
write (unit, "(A)") "\usepackage{amsmath}"
write (unit, "(A)") ""
write (unit, "(A)") "\begin{document}"
write (unit, "(A)") "\begin{gmlfile}"
write (unit, "(A)") "\section*{Integration Results Display}"
write (unit, "(A)") ""
write (unit, "(A)") "Process: \verb|" // char (filename) // "|"
write (unit, "(A)") ""
write (unit, "(A)") "\vspace*{2\baselineskip}"
write (unit, "(A)") "\unitlength 1mm"
write (unit, "(A)") "\begin{gmlcode}"
write (unit, "(A)") "  picture sym;  sym = fshape (circle scaled 1mm)();"
write (unit, "(A)") "  color col.band;  col.band = 0.9white;"
write (unit, "(A)") "  color col.eband;  col.eband = 0.98white;"
write (unit, "(A)") "\end{gmlcode}"
write (unit, "(A)") "\begin{gmlgraph*}(130,180)[history]"
write (unit, "(A)") "  setup (linear, linear);"
write (unit, "(A,I0,A)") "  history.n_pass = ", n_pass, ";"
```

```
write (unit, "(A,I0,A)") "  history.n_it   = ", n, ";"
write (unit, "(A,A,A)")  "  history.y0 = #""", char (mp_format (y0)), """;"
write (unit, "(A,A,A)")  "  history.y1 = #""", char (mp_format (y1)), """;"
write (unit, "(A)") &
     "  graphrange (#0.5, history.y0), (#(n+0.5), history.y1);"
do pass = 1, n_pass
   write (unit, "(A,I0,A,I0,A)") &
        "  history.pass[", pass, "] = ", ipass(pass), ";"
   write (unit, "(A,I0,A,A,A)") &
        "  history.avg[", pass, "] = #""", &
        char (mp_format &
           (integration_entry_get_integral (results%average(pass)))), &
        """;"
   write (unit, "(A,I0,A,A,A)") &
        "  history.err[", pass, "] = #""", &
        char (mp_format &
           (integration_entry_get_error (results%average(pass)))), &
        """;"
   write (unit, "(A,I0,A,A,A)") &
        "  history.chi[", pass, "] = #""", &
        char (mp_format &
           (integration_entry_get_chi2 (results%average(pass)))), &
        """;"
end do
write (unit, "(A,I0,A,I0,A)") &
     "  history.pass[", n_pass + 1, "] = ", n, ";"
write (unit, "(A)")  "  for i = 1 upto history.n_pass:"
write (unit, "(A)")  "    if history.chi[i] greater one:"
write (unit, "(A)")  "    fill plot ("
write (unit, "(A)")  &
     "      (#(history.pass[i]  +.5), " &
     // "history.avg[i] minus history.err[i] times history.chi[i]),"
write (unit, "(A)")  &
     "      (#(history.pass[i+1]+.5), " &
     // "history.avg[i] minus history.err[i] times history.chi[i]),"
write (unit, "(A)")  &
     "      (#(history.pass[i+1]+.5), " &
     // "history.avg[i] plus history.err[i] times history.chi[i]),"
write (unit, "(A)")  &
     "      (#(history.pass[i]  +.5), " &
     // "history.avg[i] plus history.err[i] times history.chi[i])"
write (unit, "(A)")  "    ) withcolor col.eband fi;"
write (unit, "(A)")  "    fill plot ("
write (unit, "(A)")  &
     "      (#(history.pass[i]  +.5), history.avg[i] minus history.err[i]),"
write (unit, "(A)")  &
     "      (#(history.pass[i+1]+.5), history.avg[i] minus history.err[i]),"
write (unit, "(A)")  &
     "      (#(history.pass[i+1]+.5), history.avg[i] plus history.err[i]),"
write (unit, "(A)")  &
     "      (#(history.pass[i]  +.5), history.avg[i] plus history.err[i])"
write (unit, "(A)")  "    ) withcolor col.band;"
write (unit, "(A)")  "    draw plot ("
write (unit, "(A)")  &
```

```
     "      (#(history.pass[i]  +.5), history.avg[i]),"
  write (unit, "(A)")  &
     "      (#(history.pass[i+1]+.5), history.avg[i])"
  write (unit, "(A)")  "      ) dashed evenly;"
  write (unit, "(A)")  "  endfor"
  write (unit, "(A)")  "  for i = 1 upto history.n_pass + 1:"
  write (unit, "(A)")  "    draw plot ("
  write (unit, "(A)")  &
     "      (#(history.pass[i]+.5), history.y0),"
  write (unit, "(A)")  &
     "      (#(history.pass[i]+.5), history.y1)"
  write (unit, "(A)")  "      ) dashed withdots;"
  write (unit, "(A)")  "  endfor"
  do i = 1, n
     write (unit, "(A,I0,A,A,A,A,A)") "  plot (history) (#", &
       i, ", #""", &
  char (mp_format (integration_entry_get_integral (results%entry(i)))),&
       """) vbar #""", &
       char (mp_format (integration_entry_get_error (results%entry(i)))), &
       """;"
  end do
  write (unit, "(A)") "  draw piecewise from (history) " &
    // "withsymbol sym;"
  write (unit, "(A)") "  fullgrid.lr (5,20);"
  write (unit, "(A)") "  standardgrid.bt (n);"
  write (unit, "(A)")  "  begingmleps ""Whizard-Logo.eps"";"
  write (unit, "(A)")  "    base := (120*unitlength,170*unitlength);"
  write (unit, "(A)")  "    height := 9.6*unitlength;"
  write (unit, "(A)")  "    width := 11.2*unitlength;"
  write (unit, "(A)")  "  endgmleps;"
  write (unit, "(A)") "\end{gmlgraph*}"
  write (unit, "(A)") "\end{gmlfile}"
  write (unit, "(A)") "\clearpage"
  write (unit, "(A)") "\begin{verbatim}"
  if (reset) then
    call results%pacify (reset)
  end if
  call integration_results_write (results, unit)
  write (unit, "(A)") "\end{verbatim}"
  write (unit, "(A)") "\end{document}"
  close (unit)
end subroutine integration_results_write_driver
```

Call LaTeX and Metapost for the history driver file, and convert to PS and PDF.

⟨*Integration results: public*⟩+≡

```
public :: integration_results_compile_driver
```

⟨*Integration results: procedures*⟩+≡

```
subroutine integration_results_compile_driver (results, filename, os_data)
  type(integration_results_t), intent(in) :: results
  type(string_t), intent(in) :: filename
  type(os_data_t), intent(in) :: os_data
  integer :: unit_dev, status
  type(string_t) :: file_tex, file_dvi, file_ps, file_pdf, file_mp
```

```
type(string_t) :: setenv_tex, setenv_mp, pipe, pipe_dvi
if (.not. os_data%event_analysis) then
   call msg_warning ("Skipping integration history display " &
       // "because latex or mpost is not available")
   return
end if
file_tex = filename // ".tex"
file_dvi = filename // ".dvi"
file_ps = filename // ".ps"
file_pdf = filename // ".pdf"
file_mp = filename // ".mp"
call msg_message ("Creating integration history display "&
     // char (file_ps) // " and " // char (file_pdf))
BLOCK: do
   unit_dev = free_unit ()
   open (file = "/dev/null", unit = unit_dev, &
          action = "write", iostat = status)
   if (status /= 0) then
      pipe = ""
      pipe_dvi = ""
   else
      pipe = " > /dev/null"
      pipe_dvi = " 2>/dev/null 1>/dev/null"
   end if
   close (unit_dev)
   if (os_data%whizard_texpath /= "") then
      setenv_tex = &
           "TEXINPUTS=" // os_data%whizard_texpath // ":$TEXINPUTS "
      setenv_mp = &
           "MPINPUTS=" // os_data%whizard_texpath // ":$MPINPUTS "
   else
      setenv_tex = ""
      setenv_mp = ""
   end if
   call os_system_call (setenv_tex // os_data%latex // " " // &
        file_tex // pipe, status)
   if (status /= 0)  exit BLOCK
   if (os_data%gml /= "") then
      call os_system_call (setenv_mp // os_data%gml // " " // &
           file_mp // pipe, status)
   else
      call msg_error ("Could not use GAMELAN/MetaPOST.")
      exit BLOCK
   end if
   if (status /= 0)  exit BLOCK
   call os_system_call (setenv_tex // os_data%latex // " " // &
         file_tex // pipe, status)
   if (status /= 0)  exit BLOCK
   if (os_data%event_analysis_ps) then
      call os_system_call (os_data%dvips // " " // &
         file_dvi // pipe_dvi, status)
      if (status /= 0)  exit BLOCK
   else
      call msg_warning ("Skipping PostScript generation because dvips " &
```

```
            // "is not available")
         exit BLOCK
      end if
      if (os_data%event_analysis_pdf) then
         call os_system_call (os_data%ps2pdf // " " // &
                 file_ps, status)
         if (status /= 0)  exit BLOCK
      else
         call msg_warning ("Skipping PDF generation because ps2pdf " &
            // "is not available")
         exit BLOCK
      end if
      exit BLOCK
   end do BLOCK
   if (status /= 0) then
      call msg_error ("Unable to compile integration history display")
   end if
end subroutine integration_results_compile_driver
```

20.4 Simple midpoint integration

This is a most simple implementation of an integrator. The algorithm is the straightforward multi-dimensional midpoint rule, i.e., the integration hypercube is binned uniformly, the integrand is evaluated at the midpoints of each bin, and the result is the average. The binning is equivalent for all integration dimensions.

This rule is accurate to the order h^2, where h is the bin width. Given that $h = N^{-1/d}$, where d is the integration dimension and N is the total number of sampling points, we get a relative error of order $N^{-2/d}$. This is superior to MC integration if $d < 4$, and equivalent if $d = 4$. It is not worse than higher-order formulas (such as Gauss integration) if the integrand is not smooth, e.g., if it contains cuts.

The integrator is specifically single-channel. However, we do not limit the dimension.

⟨`mci_midpoint.f90`⟩≡

```
⟨File header⟩

module mci_midpoint

⟨Use kinds⟩
  use io_units
  use diagnostics
  use phs_base

  use mci_base

⟨Standard module head⟩

⟨MCI midpoint: public⟩

⟨MCI midpoint: types⟩

contains

⟨MCI midpoint: procedures⟩

end module mci_midpoint
```

20.4.1 Integrator

The object contains the methods for integration and event generation. For the actual work and data storage, it spawns an instance object.

After an integration pass, we update the `max` parameter to indicate the maximum absolute value of the integrand that the integrator encountered. This is required for event generation.

⟨*MCI midpoint: public*⟩≡

```
public :: mci_midpoint_t
```

⟨*MCI midpoint: types*⟩≡

```
type, extends (mci_t) :: mci_midpoint_t
   integer :: n_dim_binned = 0
   logical, dimension(:), allocatable :: dim_is_binned
```

```
      logical :: calls_known = .false.
      integer :: n_calls = 0
      integer :: n_calls_pos = 0
      integer :: n_calls_nul = 0
      integer :: n_calls_neg = 0
      real(default) :: integral_pos = 0
      real(default) :: integral_neg = 0
      integer, dimension(:), allocatable :: n_bin
      logical :: max_known = .false.
      real(default) :: max = 0
      real(default) :: min = 0
      real(default) :: max_abs = 0
      real(default) :: min_abs = 0
    contains
    ⟨MCI midpoint: mci midpoint: TBP⟩
  end type mci_midpoint_t
```

Finalizer: base version is sufficient

⟨*MCI midpoint: mci midpoint: TBP*⟩≡

```
    procedure :: final => mci_midpoint_final
```

⟨*MCI midpoint: procedures*⟩≡

```
    subroutine mci_midpoint_final (object)
      class(mci_midpoint_t), intent(inout) :: object
      call object%base_final ()
    end subroutine mci_midpoint_final
```

Output.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
    procedure :: write => mci_midpoint_write
```

⟨*MCI midpoint: procedures*⟩+≡

```
    subroutine mci_midpoint_write (object, unit, pacify, md5sum_version)
      class(mci_midpoint_t), intent(in) :: object
      integer, intent(in), optional :: unit
      logical, intent(in), optional :: pacify
      logical, intent(in), optional :: md5sum_version
      integer :: u, i
      u = given_output_unit (unit)
      write (u, "(1x,A)") "Single-channel midpoint rule integrator:"
      call object%base_write (u, pacify, md5sum_version)
      if (object%n_dim_binned < object%n_dim) then
         write (u, "(3x,A,99(1x,I0))")  "Flat dimensions      =", &
              pack ([(i, i = 1, object%n_dim)], mask = .not. object%dim_is_binned)
         write (u, "(3x,A,I0)")  "Number of binned dim = ", object%n_dim_binned
      end if
      if (object%calls_known) then
         write (u, "(3x,A,99(1x,I0))")  "Number of bins       =", object%n_bin
         write (u, "(3x,A,I0)")  "Number of calls      = ", object%n_calls
         if (object%n_calls_pos /= object%n_calls) then
            write (u, "(3x,A,I0)")  "  positive value     = ", object%n_calls_pos
            write (u, "(3x,A,I0)")  "  zero value         = ", object%n_calls_nul
            write (u, "(3x,A,I0)")  "  negative value     = ", object%n_calls_neg
            write (u, "(3x,A,ES17.10)") &
```

```
            "Integral (pos. part) = ", object%integral_pos
         write (u, "(3x,A,ES17.10)") &
            "Integral (neg. part) = ", object%integral_neg
      end if
   end if
   if (object%max_known) then
      write (u, "(3x,A,ES17.10)")  "Maximum of integrand = ", object%max
      write (u, "(3x,A,ES17.10)")  "Minimum of integrand = ", object%min
      if (object%min /= object%min_abs) then
         write (u, "(3x,A,ES17.10)")  "Maximum (abs. value) = ", object%max_abs
         write (u, "(3x,A,ES17.10)")  "Minimum (abs. value) = ", object%min_abs
      end if
   end if
   if (allocated (object%rng))  call object%rng%write (u)
end subroutine mci_midpoint_write
```

Startup message: short version.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
procedure :: startup_message => mci_midpoint_startup_message
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_startup_message (mci, unit, n_calls)
  class(mci_midpoint_t), intent(in) :: mci
  integer, intent(in), optional :: unit, n_calls
  call mci%base_startup_message (unit = unit, n_calls = n_calls)
  if (mci%n_dim_binned < mci%n_dim) then
     write (msg_buffer, "(A,2(1x,I0,1x,A))") &
          "Integrator: Midpoint rule:", &
          mci%n_dim_binned, "binned dimensions"
  else
     write (msg_buffer, "(A,2(1x,I0,1x,A))") &
          "Integrator: Midpoint rule"
  end if
  call msg_message (unit = unit)
end subroutine mci_midpoint_startup_message
```

Log entry: just headline.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
procedure :: write_log_entry => mci_midpoint_write_log_entry
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_write_log_entry (mci, u)
  class(mci_midpoint_t), intent(in) :: mci
  integer, intent(in) :: u
  write (u, "(1x,A)")  "MC Integrator is Midpoint rule"
end subroutine mci_midpoint_write_log_entry
```

MD5 sum: nothing.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
procedure :: compute_md5sum => mci_midpoint_compute_md5sum
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_compute_md5sum (mci, pacify)
  class(mci_midpoint_t), intent(inout) :: mci
```

```
    logical, intent(in), optional :: pacify
  end subroutine mci_midpoint_compute_md5sum
```

The number of channels must be one.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
  procedure :: set_dimensions => mci_midpoint_set_dimensions
```

⟨*MCI midpoint: procedures*⟩+≡

```
  subroutine mci_midpoint_set_dimensions (mci, n_dim, n_channel)
    class(mci_midpoint_t), intent(inout) :: mci
    integer, intent(in) :: n_dim
    integer, intent(in) :: n_channel
    if (n_channel == 1) then
       mci%n_channel = n_channel
       mci%n_dim = n_dim
       allocate (mci%dim_is_binned (mci%n_dim))
       mci%dim_is_binned = .true.
       mci%n_dim_binned = count (mci%dim_is_binned)
       allocate (mci%n_bin (mci%n_dim))
       mci%n_bin = 0
    else
       call msg_fatal ("Attempt to initialize single-channel integrator &
            &for multiple channels")
    end if
  end subroutine mci_midpoint_set_dimensions
```

Declare particular dimensions as flat. These dimensions will not be binned.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
  procedure :: declare_flat_dimensions => mci_midpoint_declare_flat_dimensions
```

⟨*MCI midpoint: procedures*⟩+≡

```
  subroutine mci_midpoint_declare_flat_dimensions (mci, dim_flat)
    class(mci_midpoint_t), intent(inout) :: mci
    integer, dimension(:), intent(in) :: dim_flat
    integer :: d
    mci%n_dim_binned = mci%n_dim - size (dim_flat)
    do d = 1, size (dim_flat)
       mci%dim_is_binned(dim_flat(d)) = .false.
    end do
    mci%n_dim_binned = count (mci%dim_is_binned)
  end subroutine mci_midpoint_declare_flat_dimensions
```

Declare particular channels as equivalent. This has no effect.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
  procedure :: declare_equivalences => mci_midpoint_ignore_equivalences
```

⟨*MCI midpoint: procedures*⟩+≡

```
  subroutine mci_midpoint_ignore_equivalences (mci, channel, dim_offset)
    class(mci_midpoint_t), intent(inout) :: mci
    type(phs_channel_t), dimension(:), intent(in) :: channel
    integer, intent(in) :: dim_offset
  end subroutine mci_midpoint_ignore_equivalences
```

Allocate instance with matching type.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
procedure :: allocate_instance => mci_midpoint_allocate_instance
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_allocate_instance (mci, mci_instance)
  class(mci_midpoint_t), intent(in) :: mci
  class(mci_instance_t), intent(out), pointer :: mci_instance
  allocate (mci_midpoint_instance_t :: mci_instance)
end subroutine mci_midpoint_allocate_instance
```

Integrate. The number of dimensions is arbitrary. We make sure that the number of calls is evenly distributed among the dimensions. The actual number of calls will typically be smaller than the requested number, but never smaller than 1.

The sampling over a variable number of dimensions implies a variable number of nested loops. We implement this by a recursive subroutine, one loop in each recursion level.

The number of iterations `n_it` is ignored. Also, the error is set to zero in the current implementation.

With this integrator, we allow the calculation to abort immediately when forced by a signal. There is no state that we can save, hence we do not catch an interrupt.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
procedure :: integrate => mci_midpoint_integrate
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_integrate (mci, instance, sampler, n_it, n_calls, &
     results, pacify)
  class(mci_midpoint_t), intent(inout) :: mci
  class(mci_instance_t), intent(inout), target :: instance
  class(mci_sampler_t), intent(inout), target :: sampler
  integer, intent(in) :: n_it
  integer, intent(in) :: n_calls
  logical, intent(in), optional :: pacify
  class(mci_results_t), intent(inout), optional :: results
  real(default), dimension(:), allocatable :: x
  real(default) :: integral, integral_pos, integral_neg
  integer :: n_bin
  select type (instance)
  type is (mci_midpoint_instance_t)
     allocate (x (mci%n_dim))
     integral = 0
     integral_pos = 0
     integral_neg = 0
     select case (mci%n_dim_binned)
     case (1)
        n_bin = n_calls
     case (2:)
        n_bin = max (int (n_calls ** (1. / mci%n_dim_binned)), 1)
     end select
     where (mci%dim_is_binned)
        mci%n_bin = n_bin
```

```
      elsewhere
         mci%n_bin = 1
      end where
      mci%n_calls = product (mci%n_bin)
      mci%n_calls_pos = 0
      mci%n_calls_nul = 0
      mci%n_calls_neg = 0
      mci%calls_known = .true.
      call sample_dim (mci%n_dim)
      mci%integral = integral / mci%n_calls
      mci%integral_pos = integral_pos / mci%n_calls
      mci%integral_neg = integral_neg / mci%n_calls
      mci%integral_known = .true.
      call instance%set_max ()
      if (present (results)) then
         call results%record (1, mci%n_calls, &
              mci%integral, mci%error, mci%efficiency)
      end if
   end select
contains
   recursive subroutine sample_dim (d)
     integer, intent(in) :: d
     integer :: i
     real(default) :: value
     do i = 1, mci%n_bin(d)
        x(d) = (i - 0.5_default) / mci%n_bin(d)
        if (d > 1) then
           call sample_dim (d - 1)
        else
           if (signal_is_pending ())  return
           call instance%evaluate (sampler, 1, x)
           value = instance%get_value ()
           if (value > 0) then
              mci%n_calls_pos = mci%n_calls_pos + 1
              integral = integral + value
              integral_pos = integral_pos + value
           else if (value == 0) then
              mci%n_calls_nul = mci%n_calls_nul + 1
           else
              mci%n_calls_neg = mci%n_calls_neg + 1
              integral = integral + value
              integral_neg = integral_neg + value
           end if
        end if
     end do
   end subroutine sample_dim
end subroutine mci_midpoint_integrate
```

Simulation initializer and finalizer: nothing to do here.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
procedure :: prepare_simulation => mci_midpoint_ignore_prepare_simulation
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_ignore_prepare_simulation (mci)
```

```
    class(mci_midpoint_t), intent(inout) :: mci
  end subroutine mci_midpoint_ignore_prepare_simulation
```

Generate weighted event.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
  procedure :: generate_weighted_event => mci_midpoint_generate_weighted_event
```

⟨*MCI midpoint: procedures*⟩+≡

```
  subroutine mci_midpoint_generate_weighted_event (mci, instance, sampler)
    class(mci_midpoint_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout), target :: instance
    class(mci_sampler_t), intent(inout), target :: sampler
    real(default), dimension(mci%n_dim) :: x
    select type (instance)
    type is (mci_midpoint_instance_t)
       call mci%rng%generate (x)
       call instance%evaluate (sampler, 1, x)
       instance%excess_weight = 0
    end select
  end subroutine mci_midpoint_generate_weighted_event
```

For unweighted events, we generate weighted events and apply a simple rejection step to the relative event weight, until an event passes.

Note that we use the `max_abs` value stored in the configuration record, not the one stored in the instance. The latter may change during event generation. After an event generation pass is over, we may update the value for a subsequent pass.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
  procedure :: generate_unweighted_event => &
       mci_midpoint_generate_unweighted_event
```

⟨*MCI midpoint: procedures*⟩+≡

```
  subroutine mci_midpoint_generate_unweighted_event (mci, instance, sampler)
    class(mci_midpoint_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout), target :: instance
    class(mci_sampler_t), intent(inout), target :: sampler
    real(default) :: x, norm, int
    select type (instance)
    type is (mci_midpoint_instance_t)
       if (mci%max_known .and. mci%max_abs > 0) then
          norm = abs (mci%max_abs * instance%safety_factor)
          REJECTION: do
             call mci%generate_weighted_event (instance, sampler)
             if (sampler%is_valid ()) then
                call mci%rng%generate (x)
                int = abs (instance%integrand)
                if (x * norm <= int) then
                   if (norm > 0 .and. norm < int) then
                      instance%excess_weight = int / norm - 1
                   end if
                   exit REJECTION
                end if
             end if
```

```
            if (signal_is_pending ())  return
         end do REJECTION
      else
         call msg_fatal ("Unweighted event generation: &
              &maximum of integrand is zero or unknown")
      end if
   end select
end subroutine mci_midpoint_generate_unweighted_event
```

Rebuild an event, using the `state` input.

⟨*MCI midpoint: mci midpoint: TBP*⟩+≡

```
procedure :: rebuild_event => mci_midpoint_rebuild_event
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_rebuild_event (mci, instance, sampler, state)
  class(mci_midpoint_t), intent(inout) :: mci
  class(mci_instance_t), intent(inout) :: instance
  class(mci_sampler_t), intent(inout) :: sampler
  class(mci_state_t), intent(in) :: state
  select type (instance)
  type is (mci_midpoint_instance_t)
     call instance%recall (sampler, state)
  end select
end subroutine mci_midpoint_rebuild_event
```

20.4.2 Integrator instance

Covering the case of flat dimensions, we store a complete `x` array. This is filled when generating events.

⟨*MCI midpoint: public*⟩+≡

```
public :: mci_midpoint_instance_t
```

⟨*MCI midpoint: types*⟩+≡

```
type, extends (mci_instance_t) :: mci_midpoint_instance_t
   type(mci_midpoint_t), pointer :: mci => null ()
   logical :: max_known = .false.
   real(default) :: max = 0
   real(default) :: min = 0
   real(default) :: max_abs = 0
   real(default) :: min_abs = 0
   real(default) :: safety_factor = 1
   real(default) :: excess_weight = 0
 contains
 ⟨MCI midpoint: mci midpoint instance: TBP⟩
end type mci_midpoint_instance_t
```

Output.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩≡

```
procedure :: write => mci_midpoint_instance_write
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_instance_write (object, unit, pacify)
  class(mci_midpoint_instance_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacify
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A,9(1x,F12.10))")  "x =", object%x(:,1)
  write (u, "(1x,A,ES19.12)") "Integrand = ", object%integrand
  write (u, "(1x,A,ES19.12)") "Weight    = ", object%mci_weight
  if (object%safety_factor /= 1) then
     write (u, "(1x,A,ES19.12)") "Safety f  = ", object%safety_factor
  end if
  if (object%excess_weight /= 0) then
     write (u, "(1x,A,ES19.12)") "Excess    = ", object%excess_weight
  end if
  if (object%max_known) then
     write (u, "(1x,A,ES19.12)") "Maximum   = ", object%max
     write (u, "(1x,A,ES19.12)") "Minimum   = ", object%min
     if (object%min /= object%min_abs) then
        write (u, "(1x,A,ES19.12)") "Max.(abs) = ", object%max_abs
        write (u, "(1x,A,ES19.12)") "Min.(abs) = ", object%min_abs
     end if
  end if
end subroutine mci_midpoint_instance_write
```

The finalizer is empty.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡

```
procedure :: final => mci_midpoint_instance_final
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_instance_final (object)
  class(mci_midpoint_instance_t), intent(inout) :: object
end subroutine mci_midpoint_instance_final
```

Initializer.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡

```
procedure :: init => mci_midpoint_instance_init
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_instance_init (mci_instance, mci)
  class(mci_midpoint_instance_t), intent(out) :: mci_instance
  class(mci_t), intent(in), target :: mci
  call mci_instance%base_init (mci)
  select type (mci)
  type is (mci_midpoint_t)
     mci_instance%mci => mci
     call mci_instance%get_max ()
     mci_instance%selected_channel = 1
  end select
end subroutine mci_midpoint_instance_init
```

Copy the stored extrema of the integrand in the instance record.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡

```
procedure :: get_max => mci_midpoint_instance_get_max
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_instance_get_max (instance)
  class(mci_midpoint_instance_t), intent(inout) :: instance
  associate (mci => instance%mci)
    if (mci%max_known) then
       instance%max_known = .true.
       instance%max = mci%max
       instance%min = mci%min
       instance%max_abs = mci%max_abs
       instance%min_abs = mci%min_abs
    end if
  end associate
end subroutine mci_midpoint_instance_get_max
```

Reverse operations: recall the extrema, but only if they are wider than the extrema already stored in the configuration. Also recalculate the efficiency value.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡

```
procedure :: set_max => mci_midpoint_instance_set_max
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_instance_set_max (instance)
  class(mci_midpoint_instance_t), intent(inout) :: instance
  associate (mci => instance%mci)
    if (instance%max_known) then
       if (mci%max_known) then
          mci%max = max (mci%max, instance%max)
          mci%min = min (mci%min, instance%min)
          mci%max_abs = max (mci%max_abs, instance%max_abs)
          mci%min_abs = min (mci%min_abs, instance%min_abs)
       else
          mci%max = instance%max
          mci%min = instance%min
          mci%max_abs = instance%max_abs
          mci%min_abs = instance%min_abs
          mci%max_known = .true.
       end if
       if (mci%max_abs /= 0) then
          if (mci%integral_neg == 0) then
             mci%efficiency = mci%integral / mci%max_abs
             mci%efficiency_known = .true.
          else if (mci%n_calls /= 0) then
             mci%efficiency = &
                  (mci%integral_pos - mci%integral_neg) / mci%max_abs
             mci%efficiency_known = .true.
          end if
       end if
    end if
  end associate
end subroutine mci_midpoint_instance_set_max
```

The weight is the Jacobian of the mapping for the only channel.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡

```
procedure :: compute_weight => mci_midpoint_instance_compute_weight
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_instance_compute_weight (mci, c)
  class(mci_midpoint_instance_t), intent(inout) :: mci
  integer, intent(in) :: c
  select case (c)
  case (1)
     mci%mci_weight = mci%f(1)
  case default
     call msg_fatal ("MCI midpoint integrator: only single channel supported")
  end select
end subroutine mci_midpoint_instance_compute_weight
```

Record the integrand. Update stored values for maximum and minimum.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡

```
procedure :: record_integrand => mci_midpoint_instance_record_integrand
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_instance_record_integrand (mci, integrand)
  class(mci_midpoint_instance_t), intent(inout) :: mci
  real(default), intent(in) :: integrand
  mci%integrand = integrand
  if (mci%max_known) then
     mci%max = max (mci%max, integrand)
     mci%min = min (mci%min, integrand)
     mci%max_abs = max (mci%max_abs, abs (integrand))
     mci%min_abs = min (mci%min_abs, abs (integrand))
  else
     mci%max = integrand
     mci%min = integrand
     mci%max_abs = abs (integrand)
     mci%min_abs = abs (integrand)
     mci%max_known = .true.
  end if
end subroutine mci_midpoint_instance_record_integrand
```

We store the safety factor, otherwise nothing to do here.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡

```
procedure :: init_simulation => mci_midpoint_instance_init_simulation
procedure :: final_simulation => mci_midpoint_instance_final_simulation
```

⟨*MCI midpoint: procedures*⟩+≡

```
subroutine mci_midpoint_instance_init_simulation (instance, safety_factor)
  class(mci_midpoint_instance_t), intent(inout) :: instance
  real(default), intent(in), optional :: safety_factor
  if (present (safety_factor))  instance%safety_factor = safety_factor
end subroutine mci_midpoint_instance_init_simulation

subroutine mci_midpoint_instance_final_simulation (instance)
  class(mci_midpoint_instance_t), intent(inout) :: instance
end subroutine mci_midpoint_instance_final_simulation
```

Return excess weight for the current event.

⟨*MCI midpoint: mci midpoint instance: TBP*⟩+≡
```
procedure :: get_event_excess => mci_midpoint_instance_get_event_excess
```

⟨*MCI midpoint: procedures*⟩+≡
```
function mci_midpoint_instance_get_event_excess (mci) result (excess)
  class(mci_midpoint_instance_t), intent(in) :: mci
  real(default) :: excess
  excess = mci%excess_weight
end function mci_midpoint_instance_get_event_excess
```

20.4.3 Unit tests

Test module, followed by the corresponding implementation module.

⟨mci_midpoint_ut.f90⟩≡
```
⟨File header⟩

module mci_midpoint_ut
  use unit_tests
  use mci_midpoint_uti

⟨Standard module head⟩

⟨MCI midpoint: public test⟩

contains

⟨MCI midpoint: test driver⟩

end module mci_midpoint_ut
```

⟨mci_midpoint_uti.f90⟩≡
```
⟨File header⟩

module mci_midpoint_uti

⟨Use kinds⟩
  use io_units
  use rng_base
  use mci_base

  use mci_midpoint

  use rng_base_ut, only: rng_test_t

⟨Standard module head⟩

⟨MCI midpoint: test declarations⟩

⟨MCI midpoint: test types⟩

contains
```

⟨*MCI midpoint: tests*⟩

```
end module mci_midpoint_uti
```

API: driver for the unit tests below.

⟨*MCI midpoint: public test*⟩≡

```
public :: mci_midpoint_test
```

⟨*MCI midpoint: test driver*⟩≡

```
subroutine mci_midpoint_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨MCI midpoint: execute tests⟩
end subroutine mci_midpoint_test
```

Test sampler

A test sampler object should implement a function with known integral that we can use to check the integrator.

This is the function $f(x) = 3x^2$ with integral $\int_0^1 f(x)\,dx = 1$ and maximum $f(1) = 3$. If the integration dimension is greater than one, the function is extended as a constant in the other dimension(s).

Mimicking the behavior of a process object, we store the argument and result inside the sampler, so we can `fetch` results.

⟨*MCI midpoint: test types*⟩≡

```
type, extends (mci_sampler_t) :: test_sampler_1_t
   real(default), dimension(:), allocatable :: x
   real(default) :: val
 contains
 ⟨MCI midpoint: test sampler 1: TBP⟩
end type test_sampler_1_t
```

Output: There is nothing stored inside, so just print an informative line.

⟨*MCI midpoint: test sampler 1: TBP*⟩≡

```
procedure :: write => test_sampler_1_write
```

⟨*MCI midpoint: tests*⟩≡

```
subroutine test_sampler_1_write (object, unit, testflag)
  class(test_sampler_1_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)") "Test sampler: f(x) = 3 x^2"
end subroutine test_sampler_1_write
```

Evaluation: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI midpoint: test sampler 1: TBP*⟩+≡

```
procedure :: evaluate => test_sampler_1_evaluate
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine test_sampler_1_evaluate (sampler, c, x_in, val, x, f)
  class(test_sampler_1_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  if (allocated (sampler%x))  deallocate (sampler%x)
  allocate (sampler%x (size (x_in)))
  sampler%x = x_in
  sampler%val = 3 * x_in(1) ** 2
  call sampler%fetch (val, x, f)
end subroutine test_sampler_1_evaluate
```

The point is always valid.

⟨*MCI midpoint: test sampler 1: TBP*⟩+≡

```
procedure :: is_valid => test_sampler_1_is_valid
```

⟨*MCI midpoint: tests*⟩+≡

```
function test_sampler_1_is_valid (sampler) result (valid)
  class(test_sampler_1_t), intent(in) :: sampler
  logical :: valid
  valid = .true.
end function test_sampler_1_is_valid
```

Rebuild: compute all but the function value.

⟨*MCI midpoint: test sampler 1: TBP*⟩+≡

```
procedure :: rebuild => test_sampler_1_rebuild
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine test_sampler_1_rebuild (sampler, c, x_in, val, x, f)
  class(test_sampler_1_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(in) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  if (allocated (sampler%x))  deallocate (sampler%x)
  allocate (sampler%x (size (x_in)))
  sampler%x = x_in
  sampler%val = val
  x(:,1) = sampler%x
  f = 1
end subroutine test_sampler_1_rebuild
```

Extract the results.

⟨*MCI midpoint: test sampler 1: TBP*⟩+≡

```
procedure :: fetch => test_sampler_1_fetch
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine test_sampler_1_fetch (sampler, val, x, f)
  class(test_sampler_1_t), intent(in) :: sampler
  real(default), intent(out) :: val
```

```
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    val = sampler%val
    x(:,1) = sampler%x
    f = 1
  end subroutine test_sampler_1_fetch
```

This is the function $f(x) = 3x^2 + 2y$ with integral $\int_0^1 f(x,y)\,dx\,dy = 2$ and maximum $f(1) = 5$.

⟨*MCI midpoint: test types*⟩+≡

```
  type, extends (mci_sampler_t) :: test_sampler_2_t
     real(default) :: val
     real(default), dimension(2) :: x
   contains
   ⟨MCI midpoint: test sampler 2: TBP⟩
  end type test_sampler_2_t
```

Output: There is nothing stored inside, so just print an informative line.

⟨*MCI midpoint: test sampler 2: TBP*⟩≡

```
  procedure :: write => test_sampler_2_write
```

⟨*MCI midpoint: tests*⟩+≡

```
  subroutine test_sampler_2_write (object, unit, testflag)
    class(test_sampler_2_t), intent(in) :: object
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: testflag
    integer :: u
    u = given_output_unit (unit)
    write (u, "(1x,A)") "Test sampler: f(x) = 3 x^2 + 2 y"
  end subroutine test_sampler_2_write
```

Evaluate: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI midpoint: test sampler 2: TBP*⟩+≡

```
  procedure :: evaluate => test_sampler_2_evaluate
```

⟨*MCI midpoint: tests*⟩+≡

```
  subroutine test_sampler_2_evaluate (sampler, c, x_in, val, x, f)
    class(test_sampler_2_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    sampler%x = x_in
    sampler%val = 3 * x_in(1) ** 2 + 2 * x_in(2)
    call sampler%fetch (val, x, f)
  end subroutine test_sampler_2_evaluate
```

The point is always valid.

⟨*MCI midpoint: test sampler 2: TBP*⟩+≡

```
  procedure :: is_valid => test_sampler_2_is_valid
```

⟨*MCI midpoint: tests*⟩+≡

```
function test_sampler_2_is_valid (sampler) result (valid)
  class(test_sampler_2_t), intent(in) :: sampler
  logical :: valid
  valid = .true.
end function test_sampler_2_is_valid
```

Rebuild: compute all but the function value.

⟨*MCI midpoint: test sampler 2: TBP*⟩+≡

```
procedure :: rebuild => test_sampler_2_rebuild
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine test_sampler_2_rebuild (sampler, c, x_in, val, x, f)
  class(test_sampler_2_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(in) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  sampler%x = x_in
  sampler%val = val
  x(:,1) = sampler%x
  f = 1
end subroutine test_sampler_2_rebuild
```

⟨*MCI midpoint: test sampler 2: TBP*⟩+≡

```
procedure :: fetch => test_sampler_2_fetch
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine test_sampler_2_fetch (sampler, val, x, f)
  class(test_sampler_2_t), intent(in) :: sampler
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  val = sampler%val
  x(:,1) = sampler%x
  f = 1
end subroutine test_sampler_2_fetch
```

This is the function $f(x) = (1 - 3x^2)\,\theta(x - 1/2)$ with integral $\int_0^1 f(x)\,dx = -3/8$, minimum $f(1) = -2$ and maximum $f(1/2) = 1/4$. If the integration dimension is greater than one, the function is extended as a constant in the other dimension(s).

⟨*MCI midpoint: test types*⟩+≡

```
type, extends (mci_sampler_t) :: test_sampler_4_t
   real(default) :: val
   real(default), dimension(:), allocatable :: x
 contains
 ⟨MCI midpoint: test sampler 4: TBP⟩
end type test_sampler_4_t
```

Output: There is nothing stored inside, so just print an informative line.

⟨*MCI midpoint: test sampler 4: TBP*⟩≡

```
procedure :: write => test_sampler_4_write
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine test_sampler_4_write (object, unit, testflag)
  class(test_sampler_4_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)") "Test sampler: f(x) = 1 - 3 x^2"
end subroutine test_sampler_4_write
```

Evaluation: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI midpoint: test sampler 4: TBP*⟩+≡

```
procedure :: evaluate => test_sampler_4_evaluate
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine test_sampler_4_evaluate (sampler, c, x_in, val, x, f)
  class(test_sampler_4_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  if (x_in(1) >= .5_default) then
     sampler%val = 1 - 3 * x_in(1) ** 2
  else
     sampler%val = 0
  end if
  if (.not. allocated (sampler%x))  allocate (sampler%x (size (x_in)))
  sampler%x = x_in
  call sampler%fetch (val, x, f)
end subroutine test_sampler_4_evaluate
```

The point is always valid.

⟨*MCI midpoint: test sampler 4: TBP*⟩+≡

```
procedure :: is_valid => test_sampler_4_is_valid
```

⟨*MCI midpoint: tests*⟩+≡

```
function test_sampler_4_is_valid (sampler) result (valid)
  class(test_sampler_4_t), intent(in) :: sampler
  logical :: valid
  valid = .true.
end function test_sampler_4_is_valid
```

Rebuild: compute all but the function value.

⟨*MCI midpoint: test sampler 4: TBP*⟩+≡

```
procedure :: rebuild => test_sampler_4_rebuild
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine test_sampler_4_rebuild (sampler, c, x_in, val, x, f)
  class(test_sampler_4_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(in) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  sampler%x = x_in
  sampler%val = val
  x(:,1) = sampler%x
  f = 1
end subroutine test_sampler_4_rebuild
```

⟨*MCI midpoint: test sampler 4: TBP*⟩+≡

```
procedure :: fetch => test_sampler_4_fetch
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine test_sampler_4_fetch (sampler, val, x, f)
  class(test_sampler_4_t), intent(in) :: sampler
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  val = sampler%val
  x(:,1) = sampler%x
  f = 1
end subroutine test_sampler_4_fetch
```

One-dimensional integration

Construct an integrator and use it for a one-dimensional sampler.

⟨*MCI midpoint: execute tests*⟩≡

```
call test (mci_midpoint_1, "mci_midpoint_1", &
     "one-dimensional integral", &
     u, results)
```

⟨*MCI midpoint: test declarations*⟩≡

```
public :: mci_midpoint_1
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine mci_midpoint_1 (u)
  integer, intent(in) :: u
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler

  write (u, "(A)")  "* Test output: mci_midpoint_1"
  write (u, "(A)")  "*   Purpose: integrate function in one dimension"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize integrator"
  write (u, "(A)")

  allocate (mci_midpoint_t :: mci)
```

```
  call mci%set_dimensions (1, 1)

  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize instance"
  write (u, "(A)")

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  write (u, "(A)")  "* Initialize test sampler"
  write (u, "(A)")

  allocate (test_sampler_1_t :: sampler)
  call sampler%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate for x = 0.8"
  write (u, "(A)")

  call mci_instance%evaluate (sampler, 1, [0.8_default])
  call mci_instance%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate for x = 0.7"
  write (u, "(A)")

  call mci_instance%evaluate (sampler, 1, [0.7_default])
  call mci_instance%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate for x = 0.9"
  write (u, "(A)")

  call mci_instance%evaluate (sampler, 1, [0.9_default])
  call mci_instance%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Integrate with n_calls = 1000"
  write (u, "(A)")

  call mci%integrate (mci_instance, sampler, 1, 1000)
  call mci%write (u)

  call mci_instance%final ()
  call mci%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: mci_midpoint_1"

end subroutine mci_midpoint_1
```

Two-dimensional integration

Construct an integrator and use it for a two-dimensional sampler.

⟨*MCI midpoint: execute tests*⟩+≡

```
call test (mci_midpoint_2, "mci_midpoint_2", &
     "two-dimensional integral", &
     u, results)
```

⟨*MCI midpoint: test declarations*⟩+≡

```
public :: mci_midpoint_2
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine mci_midpoint_2 (u)
  integer, intent(in) :: u
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler

  write (u, "(A)")  "* Test output: mci_midpoint_2"
  write (u, "(A)")  "*   Purpose: integrate function in two dimensions"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize integrator"
  write (u, "(A)")

  allocate (mci_midpoint_t :: mci)
  call mci%set_dimensions (2, 1)

  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize instance"
  write (u, "(A)")

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  write (u, "(A)")  "* Initialize test sampler"
  write (u, "(A)")

  allocate (test_sampler_2_t :: sampler)
  call sampler%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate for x = 0.8, y = 0.2"
  write (u, "(A)")

  call mci_instance%evaluate (sampler, 1, [0.8_default, 0.2_default])
  call mci_instance%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Integrate with n_calls = 1000"
  write (u, "(A)")

  call mci%integrate (mci_instance, sampler, 1, 1000)
```

```
    call mci%write (u)

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_midpoint_2"

  end subroutine mci_midpoint_2
```

Two-dimensional integration with flat dimension

Construct an integrator and use it for a two-dimensional sampler, where the function is constant in the second dimension.

⟨*MCI midpoint: execute tests*⟩+≡

```
  call test (mci_midpoint_3, "mci_midpoint_3", &
      "two-dimensional integral with flat dimension", &
      u, results)
```

⟨*MCI midpoint: test declarations*⟩+≡

```
  public :: mci_midpoint_3
```

⟨*MCI midpoint: tests*⟩+≡

```
  subroutine mci_midpoint_3 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler

    write (u, "(A)")  "* Test output: mci_midpoint_3"
    write (u, "(A)")  "*   Purpose: integrate function with one flat dimension"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")

    allocate (mci_midpoint_t :: mci)
    select type (mci)
    type is (mci_midpoint_t)
       call mci%set_dimensions (2, 1)
       call mci%declare_flat_dimensions ([2])
    end select

    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize instance"
    write (u, "(A)")

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    write (u, "(A)")  "* Initialize test sampler"
```

```
    write (u, "(A)")

    allocate (test_sampler_1_t :: sampler)
    call sampler%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Evaluate for x = 0.8, y = 0.2"
    write (u, "(A)")

    call mci_instance%evaluate (sampler, 1, [0.8_default, 0.2_default])
    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Integrate with n_calls = 1000"
    write (u, "(A)")

    call mci%integrate (mci_instance, sampler, 1, 1000)
    call mci%write (u)

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_midpoint_3"

  end subroutine mci_midpoint_3
```

Integrand with sign flip

Construct an integrator and use it for a one-dimensional sampler.

⟨*MCI midpoint: execute tests*⟩+≡

```
  call test (mci_midpoint_4, "mci_midpoint_4", &
       "integrand with sign flip", &
       u, results)
```

⟨*MCI midpoint: test declarations*⟩+≡

```
  public :: mci_midpoint_4
```

⟨*MCI midpoint: tests*⟩+≡

```
  subroutine mci_midpoint_4 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler

    write (u, "(A)")  "* Test output: mci_midpoint_4"
    write (u, "(A)")  "*   Purpose: integrate function with sign flip &
         &in one dimension"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")

    allocate (mci_midpoint_t :: mci)
```

```
    call mci%set_dimensions (1, 1)

    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize instance"
    write (u, "(A)")

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    write (u, "(A)")  "* Initialize test sampler"
    write (u, "(A)")

    allocate (test_sampler_4_t :: sampler)
    call sampler%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Evaluate for x = 0.8"
    write (u, "(A)")

    call mci_instance%evaluate (sampler, 1, [0.8_default])
    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Integrate with n_calls = 1000"
    write (u, "(A)")

    call mci%integrate (mci_instance, sampler, 1, 1000)
    call mci%write (u)

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_midpoint_4"

  end subroutine mci_midpoint_4
```

Weighted events

Generate weighted events. Without rejection, we do not need to know maxima and minima, so we can start generating events immediately. We have two dimensions.

⟨*MCI midpoint: execute tests*⟩+≡

```
  call test (mci_midpoint_5, "mci_midpoint_5", &
       "weighted events", &
       u, results)
```

⟨*MCI midpoint: test declarations*⟩+≡

```
  public :: mci_midpoint_5
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine mci_midpoint_5 (u)
  integer, intent(in) :: u
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng
  class(mci_state_t), allocatable :: state

  write (u, "(A)")  "* Test output: mci_midpoint_5"
  write (u, "(A)")  "*   Purpose: generate weighted events"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize integrator"
  write (u, "(A)")

  allocate (mci_midpoint_t :: mci)
  call mci%set_dimensions (2, 1)

  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize instance"
  write (u, "(A)")

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  write (u, "(A)")  "* Initialize test sampler"
  write (u, "(A)")

  allocate (test_sampler_2_t :: sampler)

  write (u, "(A)")  "* Initialize random-number generator"
  write (u, "(A)")

  allocate (rng_test_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  write (u, "(A)")  "* Generate weighted event"
  write (u, "(A)")

  call mci%generate_weighted_event (mci_instance, sampler)
  call mci_instance%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Generate weighted event"
  write (u, "(A)")

  call mci%generate_weighted_event (mci_instance, sampler)
  call mci_instance%write (u)

  write (u, "(A)")
```

```
    write (u, "(A)")  "* Store data"
    write (u, "(A)")

    allocate (state)
    call mci_instance%store (state)
    call mci_instance%final ()
    deallocate (mci_instance)

    call state%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Recall data and rebuild event"
    write (u, "(A)")

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)
    call mci%rebuild_event (mci_instance, sampler, state)

    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call mci_instance%final ()
    deallocate (mci_instance)
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_midpoint_5"

  end subroutine mci_midpoint_5
```

Unweighted events

Generate unweighted events. The integrand has a sign flip in it.

⟨*MCI midpoint: execute tests*⟩+≡

```
  call test (mci_midpoint_6, "mci_midpoint_6", &
       "unweighted events", &
       u, results)
```

⟨*MCI midpoint: test declarations*⟩+≡

```
  public :: mci_midpoint_6
```

⟨*MCI midpoint: tests*⟩+≡

```
  subroutine mci_midpoint_6 (u)
    integer, intent(in) :: u
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng

    write (u, "(A)")  "* Test output: mci_midpoint_6"
    write (u, "(A)")  "*   Purpose: generate unweighted events"
    write (u, "(A)")
```

```
    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")

    allocate (mci_midpoint_t :: mci)
    call mci%set_dimensions (1, 1)

    write (u, "(A)")  "* Initialize instance"
    write (u, "(A)")

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    write (u, "(A)")  "* Initialize test sampler"
    write (u, "(A)")

    allocate (test_sampler_4_t :: sampler)

    write (u, "(A)")  "* Initialize random-number generator"
    write (u, "(A)")

    allocate (rng_test_t :: rng)
    call rng%init ()
    call mci%import_rng (rng)

    write (u, "(A)")  "* Integrate (determine maximum of integrand"
    write (u, "(A)")
    call mci%integrate (mci_instance, sampler, 1, 1000)
    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Generate unweighted event"
    write (u, "(A)")

    call mci%generate_unweighted_event (mci_instance, sampler)
    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call mci_instance%final ()
    deallocate (mci_instance)
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_midpoint_6"

  end subroutine mci_midpoint_6
```

Excess weight

Generate unweighted events. With only 2 points for integration, the maximum of the integrand is too low, and we produce excess weight.

⟨*MCI midpoint: execute tests*⟩+≡

```
call test (mci_midpoint_7, "mci_midpoint_7", &
     "excess weight", &
     u, results)
```

⟨*MCI midpoint: test declarations*⟩+≡

```
public :: mci_midpoint_7
```

⟨*MCI midpoint: tests*⟩+≡

```
subroutine mci_midpoint_7 (u)
  integer, intent(in) :: u
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng

  write (u, "(A)")  "* Test output: mci_midpoint_7"
  write (u, "(A)")  "*   Purpose: generate unweighted event &
       &with excess weight"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize integrator"
  write (u, "(A)")

  allocate (mci_midpoint_t :: mci)
  call mci%set_dimensions (1, 1)

  write (u, "(A)")  "* Initialize instance"
  write (u, "(A)")

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  write (u, "(A)")  "* Initialize test sampler"
  write (u, "(A)")

  allocate (test_sampler_4_t :: sampler)

  write (u, "(A)")  "* Initialize random-number generator"
  write (u, "(A)")

  allocate (rng_test_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  write (u, "(A)")  "* Integrate (determine maximum of integrand"
  write (u, "(A)")
  call mci%integrate (mci_instance, sampler, 1, 2)
  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Generate unweighted event"
  write (u, "(A)")

  call mci_instance%init_simulation ()
```

```
  call mci%generate_unweighted_event (mci_instance, sampler)
  call mci_instance%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Use getter methods"
  write (u, "(A)")

  write (u, "(1x,A,1x,ES19.12)")  "weight =", mci_instance%get_event_weight ()
  write (u, "(1x,A,1x,ES19.12)")  "excess =", mci_instance%get_event_excess ()

  write (u, "(A)")
  write (u, "(A)")  "* Apply safety factor"
  write (u, "(A)")

  call mci_instance%init_simulation (safety_factor = 2.1_default)

  write (u, "(A)")  "* Generate unweighted event"
  write (u, "(A)")

  call mci%generate_unweighted_event (mci_instance, sampler)
  call mci_instance%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Use getter methods"
  write (u, "(A)")

  write (u, "(1x,A,1x,ES19.12)")  "weight =", mci_instance%get_event_weight ()
  write (u, "(1x,A,1x,ES19.12)")  "excess =", mci_instance%get_event_excess ()

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call mci_instance%final ()
  deallocate (mci_instance)
  call mci%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: mci_midpoint_7"

end subroutine mci_midpoint_7
```

20.5 VAMP interface

The standard method for integration is VAMP: the multi-channel version of the VEGAS algorithm. Each parameterization (channel) of the hypercube is binned in each dimension. The binning is equally equidistant, but an iteration of the integration procedure, the binning is updated for each dimension, according to the variance distribution of the integrand, summed over all other dimension. In the next iteration, the binning approximates (hopefully) follows the integrand more closely, and the accuracy of the result is increased. Furthermore, the relative weight of the individual channels is also updated after an iteration.

The bin distribution is denoted as the grid for a channel, which we can write to file and reuse later.

In our implementation we specify the generic VAMP algorithm more tightly: the number of bins is equal for all dimensions, the initial weights are all equal. The user controls whether to update bins and/or weights after each iteration. The integration is organized in passes, each one consisting of several iterations with a common number of calls to the integrand. The first passes are intended as warmup, so the results are displayed but otherwise discarded. In the final pass, the integration estimates for the individual iterations are averaged for the final result.

⟨mci_vamp.f90⟩≡

```
⟨File header⟩

module mci_vamp

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use constants, only: zero
  use format_utils, only: pac_fmt
  use format_utils, only: write_separator
  use format_defs, only: FMT_12, FMT_14, FMT_17, FMT_19
  use diagnostics
  use md5
  use phs_base
  use rng_base
  use rng_tao
  use vamp !NODEP!
  use exceptions !NODEP!

  use mci_base

⟨Standard module head⟩

⟨MCI vamp: public⟩

⟨MCI vamp: types⟩

⟨MCI vamp: interfaces⟩

contains
```

⟨*MCI vamp: procedures*⟩

```
end module mci_vamp
```

20.5.1 Grid parameters

This is a transparent container. It holds the parameters that are stored in grid files, and are checked when grid files are read.

⟨*MCI vamp: public*⟩≡

```
public :: grid_parameters_t
```

⟨*MCI vamp: types*⟩≡

```
type :: grid_parameters_t
   integer :: threshold_calls = 0
   integer :: min_calls_per_channel = 10
   integer :: min_calls_per_bin = 10
   integer :: min_bins = 3
   integer :: max_bins = 20
   logical :: stratified = .true.
   logical :: use_vamp_equivalences = .true.
   real(default) :: channel_weights_power = 0.25_default
   real(default) :: accuracy_goal = 0
   real(default) :: error_goal = 0
   real(default) :: rel_error_goal = 0
 contains
 ⟨MCI vamp: grid parameters: TBP⟩
end type grid_parameters_t
```

I/O:

⟨*MCI vamp: grid parameters: TBP*⟩≡

```
procedure :: write => grid_parameters_write
```

⟨*MCI vamp: procedures*⟩≡

```
subroutine grid_parameters_write (object, unit)
  class(grid_parameters_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(3x,A,I0)") "threshold_calls       = ", &
       object%threshold_calls
  write (u, "(3x,A,I0)") "min_calls_per_channel = ", &
       object%min_calls_per_channel
  write (u, "(3x,A,I0)") "min_calls_per_bin     = ", &
       object%min_calls_per_bin
  write (u, "(3x,A,I0)") "min_bins              = ", &
       object%min_bins
  write (u, "(3x,A,I0)") "max_bins              = ", &
       object%max_bins
  write (u, "(3x,A,L1)") "stratified            = ", &
       object%stratified
  write (u, "(3x,A,L1)") "use_vamp_equivalences = ", &
       object%use_vamp_equivalences
  write (u, "(3x,A,F10.7)") "channel_weights_power = ", &
       object%channel_weights_power
```

```
    if (object%accuracy_goal > 0) then
       write (u, "(3x,A,F10.7)") "accuracy_goal         = ", &
            object%accuracy_goal
    end if
    if (object%error_goal > 0) then
       write (u, "(3x,A,F10.7)") "error_goal            = ", &
            object%error_goal
    end if
    if (object%rel_error_goal > 0) then
       write (u, "(3x,A,F10.7)") "rel_error_goal        = ", &
            object%rel_error_goal
    end if
  end subroutine grid_parameters_write
```

20.5.2 History parameters

The history parameters are also stored in a transparent container. This is not a part of the grid definition, and should not be included in the MD5 sum.

⟨*MCI vamp: public*⟩+≡

```
  public :: history_parameters_t
```

⟨*MCI vamp: types*⟩+≡

```
  type :: history_parameters_t
     logical :: global = .true.
     logical :: global_verbose = .false.
     logical :: channel = .false.
     logical :: channel_verbose = .false.
   contains
   ⟨MCI vamp: history parameters: TBP⟩
  end type history_parameters_t
```

I/O:

⟨*MCI vamp: history parameters: TBP*⟩≡

```
  procedure :: write => history_parameters_write
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine history_parameters_write (object, unit)
    class(history_parameters_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(3x,A,L1)") "history(global)       = ", object%global
    write (u, "(3x,A,L1)") "history(global) verb. = ", object%global_verbose
    write (u, "(3x,A,L1)") "history(channels)     = ", object%channel
    write (u, "(3x,A,L1)") "history(chann.) verb. = ", object%channel_verbose
  end subroutine history_parameters_write
```

20.5.3 Integration pass

We store the parameters for each integration pass in a linked list.

⟨*MCI vamp: types*⟩+≡

```
type :: pass_t
   integer :: i_pass = 0
   integer :: i_first_it = 0
   integer :: n_it = 0
   integer :: n_calls = 0
   integer :: n_bins = 0
   logical :: adapt_grids = .false.
   logical :: adapt_weights = .false.
   logical :: is_final_pass = .false.
   logical :: integral_defined = .false.
   integer, dimension(:), allocatable :: calls
   real(default), dimension(:), allocatable :: integral
   real(default), dimension(:), allocatable :: error
   real(default), dimension(:), allocatable :: efficiency
   type(vamp_history), dimension(:), allocatable :: v_history
   type(vamp_history), dimension(:,:), allocatable :: v_histories
   type(pass_t), pointer :: next => null ()
 contains
 ⟨MCI vamp: pass: TBP⟩
end type pass_t
```

Finalizer. The VAMP histories contain a pointer array.

⟨*MCI vamp: pass: TBP*⟩≡

```
procedure :: final => pass_final
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine pass_final (object)
  class(pass_t), intent(inout) :: object
  if (allocated (object%v_history)) then
     call vamp_delete_history (object%v_history)
  end if
  if (allocated (object%v_histories)) then
     call vamp_delete_history (object%v_histories)
  end if
end subroutine pass_final
```

Output. Note that the precision of the numerical values should match the precision for comparing output from file with data.

⟨*MCI vamp: pass: TBP*⟩+≡

```
procedure :: write => pass_write
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine pass_write (object, unit, pacify)
  class(pass_t), intent(in) :: object
  integer, intent(in) :: unit
  logical, intent(in), optional :: pacify
  integer :: u, i
  character(len=7) :: fmt
  call pac_fmt (fmt, FMT_17, FMT_14, pacify)
  u = given_output_unit (unit)
  write (u, "(3x,A,I0)")  "n_it          = ", object%n_it
  write (u, "(3x,A,I0)")  "n_calls       = ", object%n_calls
  write (u, "(3x,A,I0)")  "n_bins        = ", object%n_bins
  write (u, "(3x,A,L1)")  "adapt grids   = ", object%adapt_grids
```

```
    write (u, "(3x,A,L1)")  "adapt weights = ", object%adapt_weights
    if (object%integral_defined) then
       write (u, "(3x,A)")  "Results:  [it, calls, integral, error, efficiency]"
       do i = 1, object%n_it
          write (u, "(5x,I0,1x,I0,3(1x," // fmt // "))") &
               i, object%calls(i), object%integral(i), object%error(i), &
               object%efficiency(i)
       end do
    else
       write (u, "(3x,A)")  "Results: [undefined]"
    end if
  end subroutine pass_write
```

Read and reconstruct the pass.

⟨*MCI vamp: pass: TBP*⟩+≡

```
  procedure :: read => pass_read
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine pass_read (object, u, n_pass, n_it)
    class(pass_t), intent(out) :: object
    integer, intent(in) :: u, n_pass, n_it
    integer :: i, j
    character(80) :: buffer
    object%i_pass = n_pass + 1
    object%i_first_it = n_it + 1
    call read_ival (u, object%n_it)
    call read_ival (u, object%n_calls)
    call read_ival (u, object%n_bins)
    call read_lval (u, object%adapt_grids)
    call read_lval (u, object%adapt_weights)
    allocate (object%calls (object%n_it), source = 0)
    allocate (object%integral (object%n_it), source = 0._default)
    allocate (object%error (object%n_it), source = 0._default)
    allocate (object%efficiency (object%n_it), source = 0._default)
    read (u, "(A)")  buffer
    select case (trim (adjustl (buffer)))
    case ("Results:  [it, calls, integral, error, efficiency]")
       do i = 1, object%n_it
          read (u, *) &
               j, object%calls(i), object%integral(i), object%error(i), &
               object%efficiency(i)
       end do
       object%integral_defined = .true.
    case ("Results: [undefined]")
       object%integral_defined = .false.
    case default
       call msg_fatal ("Reading integration pass: corrupted file")
    end select
  end subroutine pass_read
```

Write the VAMP history for this pass. (The subroutine writes the whole array at once.)

⟨*MCI vamp: pass: TBP*⟩+≡

```
  procedure :: write_history => pass_write_history
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine pass_write_history (pass, unit)
  class(pass_t), intent(in) :: pass
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  if (allocated (pass%v_history)) then
     call vamp_write_history (u, pass%v_history)
  else
     write (u, "(1x,A)")  "Global history: [undefined]"
  end if
  if (allocated (pass%v_histories)) then
     write (u, "(1x,A)")  "Channel histories:"
     call vamp_write_history (u, pass%v_histories)
  else
     write (u, "(1x,A)")  "Channel histories: [undefined]"
  end if
end subroutine pass_write_history
```

Given a number of calls and iterations, compute remaining data.

⟨*MCI vamp: pass: TBP*⟩+≡

```
procedure :: configure => pass_configure
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine pass_configure (pass, n_it, n_calls, min_calls, &
     min_bins, max_bins, min_channel_calls)
  class(pass_t), intent(inout) :: pass
  integer, intent(in) :: n_it, n_calls, min_channel_calls
  integer, intent(in) :: min_calls, min_bins, max_bins
  pass%n_it = n_it
  if (min_calls /= 0) then
     pass%n_bins =  max (min_bins, &
          min (n_calls / min_calls, max_bins))
  else
     pass%n_bins = max_bins
  end if
  pass%n_calls = max (n_calls, max (min_calls, min_channel_calls))
  if (pass%n_calls /= n_calls) then
     write (msg_buffer, "(A,I0)")  "VAMP: too few calls, resetting " &
          // "n_calls to ", pass%n_calls
     call msg_warning ()
  end if
  allocate (pass%calls (n_it), source = 0)
  allocate (pass%integral (n_it), source = 0._default)
  allocate (pass%error (n_it), source = 0._default)
  allocate (pass%efficiency (n_it), source = 0._default)
end subroutine pass_configure
```

Allocate the VAMP history and give options. We assume that the `configure` routine above has been executed, so the number of iterations is known.

⟨*MCI vamp: pass: TBP*⟩+≡

```
procedure :: configure_history => pass_configure_history
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine pass_configure_history (pass, n_channels, par)
  class(pass_t), intent(inout) :: pass
  integer, intent(in) :: n_channels
  type(history_parameters_t), intent(in) :: par
  if (par%global) then
     allocate (pass%v_history (pass%n_it))
     call vamp_create_history (pass%v_history, &
          verbose = par%global_verbose)
  end if
  if (par%channel) then
     allocate (pass%v_histories (pass%n_it, n_channels))
     call vamp_create_history (pass%v_histories, &
          verbose = par%channel_verbose)
  end if
end subroutine pass_configure_history
```

Given two pass objects, compare them. All parameters must match. Where integrations are done in both (number of calls nonzero), the results must be equal (up to numerical noise).

The allocated array sizes might be different, but should match up to the common `n_it` value.

⟨*MCI vamp: interfaces*⟩≡

```
interface operator (.matches.)
   module procedure pass_matches
end interface operator (.matches.)
```

⟨*MCI vamp: procedures*⟩+≡

```
function pass_matches (pass, ref) result (ok)
  type(pass_t), intent(in) :: pass, ref
  integer :: n
  logical :: ok
  ok = .true.
  if (ok)  ok = pass%i_pass == ref%i_pass
  if (ok)  ok = pass%i_first_it == ref%i_first_it
  if (ok)  ok = pass%n_it == ref%n_it
  if (ok)  ok = pass%n_calls == ref%n_calls
  if (ok)  ok = pass%n_bins == ref%n_bins
  if (ok)  ok = pass%adapt_grids .eqv. ref%adapt_grids
  if (ok)  ok = pass%adapt_weights .eqv. ref%adapt_weights
  if (ok)  ok = pass%integral_defined .eqv. ref%integral_defined
  if (pass%integral_defined) then
     n = pass%n_it
     if (ok)  ok = all (pass%calls(:n) == ref%calls(:n))
     if (ok)  ok = all (pass%integral(:n) .matches. ref%integral(:n))
     if (ok)  ok = all (pass%error(:n) .matches. ref%error(:n))
     if (ok)  ok = all (pass%efficiency(:n) .matches. ref%efficiency(:n))
  end if
end function pass_matches
```

Update a pass object, given a reference. The parameters must match, except for the `n_it` entry. The number of complete iterations must be less or equal to the reference, and the number of complete iterations in the reference must be no

larger than n_it. Where results are present in both passes, they must match. Where results are present in the reference only, the pass is updated accordingly.

⟨*MCI vamp: pass: TBP*⟩+≡

```
procedure :: update => pass_update
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine pass_update (pass, ref, ok)
  class(pass_t), intent(inout) :: pass
  type(pass_t), intent(in) :: ref
  logical, intent(out) :: ok
  integer :: n, n_ref
  ok = .true.
  if (ok)  ok = pass%i_pass == ref%i_pass
  if (ok)  ok = pass%i_first_it == ref%i_first_it
  if (ok)  ok = pass%n_calls == ref%n_calls
  if (ok)  ok = pass%n_bins == ref%n_bins
  if (ok)  ok = pass%adapt_grids .eqv. ref%adapt_grids
  if (ok)  ok = pass%adapt_weights .eqv. ref%adapt_weights
  if (ok) then
     if (ref%integral_defined) then
        if (.not. allocated (pass%calls)) then
           allocate (pass%calls (pass%n_it), source = 0)
           allocate (pass%integral (pass%n_it), source = 0._default)
           allocate (pass%error (pass%n_it), source = 0._default)
           allocate (pass%efficiency (pass%n_it), source = 0._default)
        end if
        n = count (pass%calls /= 0)
        n_ref = count (ref%calls /= 0)
        ok = n <= n_ref .and. n_ref <= pass%n_it
        if (ok)  ok = all (pass%calls(:n) == ref%calls(:n))
        if (ok)  ok = all (pass%integral(:n) .matches. ref%integral(:n))
        if (ok)  ok = all (pass%error(:n) .matches. ref%error(:n))
        if (ok)  ok = all (pass%efficiency(:n) .matches. ref%efficiency(:n))
        if (ok) then
           pass%calls(n+1:n_ref) = ref%calls(n+1:n_ref)
           pass%integral(n+1:n_ref) = ref%integral(n+1:n_ref)
           pass%error(n+1:n_ref) = ref%error(n+1:n_ref)
           pass%efficiency(n+1:n_ref) = ref%efficiency(n+1:n_ref)
           pass%integral_defined = any (pass%calls /= 0)
        end if
     end if
  end if
end subroutine pass_update
```

Match two real numbers: they are equal up to a tolerance, which is 10^{-8}, matching the number of digits that are output by pass_write. In particular, if one number is exactly zero, the other one must also be zero.

⟨*MCI vamp: interfaces*⟩+≡

```
interface operator (.matches.)
   module procedure real_matches
end interface operator (.matches.)
```

⟨*MCI vamp: procedures*⟩+≡

```
elemental function real_matches (x, y) result (ok)
```

```
    real(default), intent(in) :: x, y
    logical :: ok
    real(default), parameter :: tolerance = 1.e-8_default
    ok = abs (x - y) <= tolerance * max (abs (x), abs (y))
  end function real_matches
```

Return the index of the most recent complete integration. If there is none, return zero.

⟨*MCI vamp: pass: TBP*⟩+≡

```
  procedure :: get_integration_index => pass_get_integration_index
```

⟨*MCI vamp: procedures*⟩+≡

```
  function pass_get_integration_index (pass) result (n)
    class (pass_t), intent(in) :: pass
    integer :: n
    integer :: i
    n = 0
    if (allocated (pass%calls)) then
       do i = 1, pass%n_it
          if (pass%calls(i) == 0)  exit
          n = i
       end do
    end if
  end function pass_get_integration_index
```

Return the most recent integral and error, if available.

⟨*MCI vamp: pass: TBP*⟩+≡

```
  procedure :: get_calls => pass_get_calls
  procedure :: get_integral => pass_get_integral
  procedure :: get_error => pass_get_error
  procedure :: get_efficiency => pass_get_efficiency
```

⟨*MCI vamp: procedures*⟩+≡

```
  function pass_get_calls (pass) result (calls)
    class(pass_t), intent(in) :: pass
    integer :: calls
    integer :: n
    n = pass%get_integration_index ()
    if (n /= 0) then
       calls = pass%calls(n)
    else
       calls = 0
    end if
  end function pass_get_calls

  function pass_get_integral (pass) result (integral)
    class(pass_t), intent(in) :: pass
    real(default) :: integral
    integer :: n
    n = pass%get_integration_index ()
    if (n /= 0) then
       integral = pass%integral(n)
    else
       integral = 0
```

```
  end if
end function pass_get_integral

function pass_get_error (pass) result (error)
  class(pass_t), intent(in) :: pass
  real(default) :: error
  integer :: n
  n = pass%get_integration_index ()
  if (n /= 0) then
     error = pass%error(n)
  else
     error = 0
  end if
end function pass_get_error

function pass_get_efficiency (pass) result (efficiency)
  class(pass_t), intent(in) :: pass
  real(default) :: efficiency
  integer :: n
  n = pass%get_integration_index ()
  if (n /= 0) then
     efficiency = pass%efficiency(n)
  else
     efficiency = 0
  end if
end function pass_get_efficiency
```

20.5.4 Integrator

⟨*MCI vamp: public*⟩+≡

```
public :: mci_vamp_t
```

⟨*MCI vamp: types*⟩+≡

```
type, extends (mci_t) :: mci_vamp_t
   logical, dimension(:), allocatable :: dim_is_flat
   type(grid_parameters_t) :: grid_par
   type(history_parameters_t) :: history_par
   integer :: min_calls = 0
   type(pass_t), pointer :: first_pass => null ()
   type(pass_t), pointer :: current_pass => null ()
   type(vamp_equivalences_t) :: equivalences
   logical :: rebuild = .true.
   logical :: check_grid_file = .true.
   logical :: grid_filename_set = .false.
   logical :: negative_weights = .false.
   logical :: verbose = .false.
   type(string_t) :: grid_filename
   character(32) :: md5sum_adapted = ""
 contains
 ⟨MCI vamp: mci vamp: TBP⟩
end type mci_vamp_t
```

Reset: delete integration-pass entries.

⟨*MCI vamp: mci vamp: TBP*⟩≡
```
procedure :: reset => mci_vamp_reset
```

⟨*MCI vamp: procedures*⟩+≡
```
subroutine mci_vamp_reset (object)
  class(mci_vamp_t), intent(inout) :: object
  type(pass_t), pointer :: current_pass
  do while (associated (object%first_pass))
     current_pass => object%first_pass
     object%first_pass => current_pass%next
     call current_pass%final ()
     deallocate (current_pass)
  end do
  object%current_pass => null ()
end subroutine mci_vamp_reset
```

Finalizer: reset and finalize the equivalences list.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
procedure :: final => mci_vamp_final
```

⟨*MCI vamp: procedures*⟩+≡
```
subroutine mci_vamp_final (object)
  class(mci_vamp_t), intent(inout) :: object
  call object%reset ()
  call vamp_equivalences_final (object%equivalences)
  call object%base_final ()
end subroutine mci_vamp_final
```

Output. Do not output the grids themselves, this may result in tons of data.

⟨*MCI vamp: mci vamp: TBP*⟩+≡
```
procedure :: write => mci_vamp_write
```

⟨*MCI vamp: procedures*⟩+≡
```
subroutine mci_vamp_write (object, unit, pacify, md5sum_version)
  class(mci_vamp_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacify
  logical, intent(in), optional :: md5sum_version
  type(pass_t), pointer :: current_pass
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "VAMP integrator:"
  call object%base_write (u, pacify, md5sum_version)
  if (allocated (object%dim_is_flat)) then
     write (u, "(3x,A,999(1x,I0))")  "Flat dimensions    =", &
          pack ([(i, i = 1, object%n_dim)], object%dim_is_flat)
  end if
  write (u, "(1x,A)")  "Grid parameters:"
  call object%grid_par%write (u)
  write (u, "(3x,A,I0)") "min_calls             = ", object%min_calls
  write (u, "(3x,A,L1)") "negative weights      = ", &
       object%negative_weights
  write (u, "(3x,A,L1)") "verbose               = ", &
```

```
        object%verbose
    if (object%grid_par%use_vamp_equivalences) then
       call vamp_equivalences_write (object%equivalences, u)
    end if
    current_pass => object%first_pass
    do while (associated (current_pass))
       write (u, "(1x,A,I0,A)")  "Integration pass:"
       call current_pass%write (u, pacify)
       current_pass => current_pass%next
    end do
    if (object%md5sum_adapted /= "") then
       write (u, "(1x,A,A,A)")  "MD5 sum (including results) = '", &
            object%md5sum_adapted, "'"
    end if
  end subroutine mci_vamp_write
```

Write the history parameters.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
  procedure :: write_history_parameters => mci_vamp_write_history_parameters
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_write_history_parameters (mci, unit)
    class(mci_vamp_t), intent(in) :: mci
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(1x,A)")  "VAMP history parameters:"
    call mci%history_par%write (unit)
  end subroutine mci_vamp_write_history_parameters
```

Write the history, iterating over passes. We keep this separate from the generic `write` routine.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
  procedure :: write_history => mci_vamp_write_history
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_write_history (mci, unit)
    class(mci_vamp_t), intent(in) :: mci
    integer, intent(in), optional :: unit
    type(pass_t), pointer :: current_pass
    integer :: i_pass
    integer :: u
    u = given_output_unit (unit)
    if (associated (mci%first_pass)) then
       write (u, "(1x,A)")  "VAMP history (global):"
       i_pass = 0
       current_pass => mci%first_pass
       do while (associated (current_pass))
          i_pass = i_pass + 1
          write (u, "(1x,A,I0,':')")  "Pass #", i_pass
          call current_pass%write_history (u)
          current_pass => current_pass%next
       end do
    end if
```

```
end subroutine mci_vamp_write_history
```

Compute the MD5 sum, including the configuration MD5 sum and the printout, which incorporates the current results.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: compute_md5sum => mci_vamp_compute_md5sum
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_compute_md5sum (mci, pacify)
  class(mci_vamp_t), intent(inout) :: mci
  logical, intent(in), optional :: pacify
  integer :: u
  mci%md5sum_adapted = ""
  u = free_unit ()
  open (u, status = "scratch", action = "readwrite")
  write (u, "(A)")  mci%md5sum
  call mci%write (u, pacify, md5sum_version = .true.)
  rewind (u)
  mci%md5sum_adapted = md5sum (u)
  close (u)
end subroutine mci_vamp_compute_md5sum
```

Return the MD5 sum: If available, return the adapted one.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: get_md5sum => mci_vamp_get_md5sum
```

⟨*MCI vamp: procedures*⟩+≡

```
pure function mci_vamp_get_md5sum (mci) result (md5sum)
  class(mci_vamp_t), intent(in) :: mci
  character(32) :: md5sum
  if (mci%md5sum_adapted /= "") then
     md5sum = mci%md5sum_adapted
  else
     md5sum = mci%md5sum
  end if
end function mci_vamp_get_md5sum
```

Startup message: short version.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: startup_message => mci_vamp_startup_message
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_startup_message (mci, unit, n_calls)
  class(mci_vamp_t), intent(in) :: mci
  integer, intent(in), optional :: unit, n_calls
  integer :: num_calls, n_bins
  if (present (n_calls)) then
     num_calls = n_calls
  else
     num_calls = 0
  end if
  if (mci%min_calls /= 0) then
     n_bins =  max (mci%grid_par%min_bins, &
```

```
            min (num_calls / mci%min_calls, &
            mci%grid_par%max_bins))
      else
         n_bins = mci%grid_par%max_bins
      end if
      call mci%base_startup_message (unit = unit, n_calls = n_calls)
      if (mci%grid_par%use_vamp_equivalences) then
         write (msg_buffer, "(A,2(1x,I0,1x,A))") &
              "Integrator: Using VAMP channel equivalences"
         call msg_message (unit = unit)
      end if
      write (msg_buffer, "(A,2(1x,I0,1x,A),L1)") &
           "Integrator:", num_calls, &
           "initial calls,", n_bins, &
           "bins, stratified = ", &
           mci%grid_par%stratified
      call msg_message (unit = unit)
      write (msg_buffer, "(A,2(1x,I0,1x,A))") &
           "Integrator: VAMP"
      call msg_message (unit = unit)
    end subroutine mci_vamp_startup_message
```

Log entry: just headline.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
    procedure :: write_log_entry => mci_vamp_write_log_entry
```

⟨*MCI vamp: procedures*⟩+≡

```
    subroutine mci_vamp_write_log_entry (mci, u)
      class(mci_vamp_t), intent(in) :: mci
      integer, intent(in) :: u
      write (u, "(1x,A)")  "MC Integrator is VAMP"
      call write_separator (u)
      call mci%write_history (u)
      call write_separator (u)
      if (mci%grid_par%use_vamp_equivalences) then
         call vamp_equivalences_write (mci%equivalences, u)
      else
         write (u, "(3x,A)") "No VAMP equivalences have been used"
      end if
      call write_separator (u)
      call mci%write_chain_weights (u)
    end subroutine mci_vamp_write_log_entry
```

Set the MCI index (necessary for processes with multiple components). We append the index to the grid filename, just before the final dotted suffix.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
    procedure :: record_index => mci_vamp_record_index
```

⟨*MCI vamp: procedures*⟩+≡

```
    subroutine mci_vamp_record_index (mci, i_mci)
      class(mci_vamp_t), intent(inout) :: mci
      integer, intent(in) :: i_mci
      type(string_t) :: basename, suffix
      character(32) :: buffer
```

```
    if (mci%grid_filename_set) then
       basename = mci%grid_filename
       call split (basename, suffix, ".", back=.true.)
       write (buffer, "(I0)")  i_mci
       if (basename /= "") then
          mci%grid_filename = basename // "_m" // trim (buffer) // "." // suffix
       else
          mci%grid_filename = suffix // "_m" // trim (buffer) // ".vg"
       end if
    end if
  end subroutine mci_vamp_record_index
```

Set the grid parameters.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
  procedure :: set_grid_parameters => mci_vamp_set_grid_parameters
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_set_grid_parameters (mci, grid_par)
    class(mci_vamp_t), intent(inout) :: mci
    type(grid_parameters_t), intent(in) :: grid_par
    mci%grid_par = grid_par
    mci%min_calls = grid_par%min_calls_per_bin * mci%n_channel
  end subroutine mci_vamp_set_grid_parameters
```

Set the history parameters.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
  procedure :: set_history_parameters => mci_vamp_set_history_parameters
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_set_history_parameters (mci, history_par)
    class(mci_vamp_t), intent(inout) :: mci
    type(history_parameters_t), intent(in) :: history_par
    mci%history_par = history_par
  end subroutine mci_vamp_set_history_parameters
```

Set the rebuild flag, also the flag for checking the grid file.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
  procedure :: set_rebuild_flag => mci_vamp_set_rebuild_flag
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_set_rebuild_flag (mci, rebuild, check_grid_file)
    class(mci_vamp_t), intent(inout) :: mci
    logical, intent(in) :: rebuild
    logical, intent(in) :: check_grid_file
    mci%rebuild = rebuild
    mci%check_grid_file = check_grid_file
  end subroutine mci_vamp_set_rebuild_flag
```

Set the filename.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
  procedure :: set_grid_filename => mci_vamp_set_grid_filename
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_set_grid_filename (mci, name, run_id)
  class(mci_vamp_t), intent(inout) :: mci
  type(string_t), intent(in) :: name
  type(string_t), intent(in), optional :: run_id
  if (present (run_id)) then
     mci%grid_filename = name // "." // run_id // ".vg"
  else
     mci%grid_filename = name // ".vg"
  end if
  mci%grid_filename_set = .true.
end subroutine mci_vamp_set_grid_filename
```

To simplify the interface, we prepend a grid path in a separate subroutine.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: prepend_grid_path => mci_vamp_prepend_grid_path
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_prepend_grid_path (mci, prefix)
  class(mci_vamp_t), intent(inout) :: mci
  type(string_t), intent(in) :: prefix
  if (mci%grid_filename_set) then
     mci%grid_filename = prefix // "/" // mci%grid_filename
  else
     call msg_warning ("Cannot add prefix to invalid grid filename!")
  end if
end subroutine mci_vamp_prepend_grid_path
```

Declare particular dimensions as flat.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: declare_flat_dimensions => mci_vamp_declare_flat_dimensions
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_declare_flat_dimensions (mci, dim_flat)
  class(mci_vamp_t), intent(inout) :: mci
  integer, dimension(:), intent(in) :: dim_flat
  integer :: d
  allocate (mci%dim_is_flat (mci%n_dim), source = .false.)
  do d = 1, size (dim_flat)
     mci%dim_is_flat(dim_flat(d)) = .true.
  end do
end subroutine mci_vamp_declare_flat_dimensions
```

Declare equivalences. We have an array of channel equivalences, provided by the phase-space module. Here, we translate this into the `vamp_equivalences` array.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: declare_equivalences => mci_vamp_declare_equivalences
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_declare_equivalences (mci, channel, dim_offset)
  class(mci_vamp_t), intent(inout) :: mci
  type(phs_channel_t), dimension(:), intent(in) :: channel
```

```
    integer, intent(in) :: dim_offset
    integer, dimension(:), allocatable :: perm, mode
    integer :: n_channels, n_dim, n_equivalences
    integer :: c, i, j, left, right
    n_channels = mci%n_channel
    n_dim = mci%n_dim
    n_equivalences = 0
    do c = 1, n_channels
       n_equivalences = n_equivalences + size (channel(c)%eq)
    end do
    call vamp_equivalences_init (mci%equivalences, &
         n_equivalences, n_channels, n_dim)
    allocate (perm (n_dim))
    allocate (mode (n_dim))
    perm(1:dim_offset) = [(i, i = 1, dim_offset)]
    mode(1:dim_offset) = VEQ_IDENTITY
    c = 1
    j = 0
    do i = 1, n_equivalences
       if (j < size (channel(c)%eq)) then
          j = j + 1
       else
          c = c + 1
          j = 1
       end if
       associate (eq => channel(c)%eq(j))
         left = c
         right = eq%c
         perm(dim_offset+1:) = eq%perm + dim_offset
         mode(dim_offset+1:) = eq%mode
         call vamp_equivalence_set (mci%equivalences, &
              i, left, right, perm, mode)
       end associate
    end do
    call vamp_equivalences_complete (mci%equivalences)
  end subroutine mci_vamp_declare_equivalences
```

Allocate instance with matching type.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
  procedure :: allocate_instance => mci_vamp_allocate_instance
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_allocate_instance (mci, mci_instance)
    class(mci_vamp_t), intent(in) :: mci
    class(mci_instance_t), intent(out), pointer :: mci_instance
    allocate (mci_vamp_instance_t :: mci_instance)
  end subroutine mci_vamp_allocate_instance
```

Allocate a new integration pass. We can preset everything that does not depend on the number of iterations and calls. This is postponed to the `integrate` method.

In the final pass, we do not check accuracy goal etc., since we can assume that the user wants to perform and average all iterations in this pass.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: add_pass => mci_vamp_add_pass
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_add_pass (mci, adapt_grids, adapt_weights, final_pass)
  class(mci_vamp_t), intent(inout) :: mci
  logical, intent(in), optional :: adapt_grids, adapt_weights, final_pass
  integer :: i_pass, i_it
  type(pass_t), pointer :: new
  allocate (new)
  if (associated (mci%current_pass)) then
     i_pass = mci%current_pass%i_pass + 1
     i_it   = mci%current_pass%i_first_it + mci%current_pass%n_it
     mci%current_pass%next => new
  else
     i_pass = 1
     i_it = 1
     mci%first_pass => new
  end if
  mci%current_pass => new
  new%i_pass = i_pass
  new%i_first_it = i_it
  if (present (adapt_grids)) then
     new%adapt_grids = adapt_grids
  else
     new%adapt_grids = .false.
  end if
  if (present (adapt_weights)) then
     new%adapt_weights = adapt_weights
  else
     new%adapt_weights = .false.
  end if
  if (present (final_pass)) then
     new%is_final_pass = final_pass
  else
     new%is_final_pass = .false.
  end if
end subroutine mci_vamp_add_pass
```

Update the list of integration passes. All passes except for the last one must match exactly. For the last one, integration results are updated. The reference output may contain extra passes, these are ignored.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: update_from_ref => mci_vamp_update_from_ref
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_update_from_ref (mci, mci_ref, success)
  class(mci_vamp_t), intent(inout) :: mci
  class(mci_t), intent(in) :: mci_ref
  logical, intent(out) :: success
  type(pass_t), pointer :: current_pass, ref_pass
  select type (mci_ref)
  type is (mci_vamp_t)
     current_pass => mci%first_pass
```

```
      ref_pass => mci_ref%first_pass
      success = .true.
      do while (success .and. associated (current_pass))
         if (associated (ref_pass)) then
            if (associated (current_pass%next)) then
               success = current_pass .matches. ref_pass
            else
               call current_pass%update (ref_pass, success)
               if (current_pass%integral_defined) then
                  mci%integral = current_pass%get_integral ()
                  mci%error = current_pass%get_error ()
                  mci%efficiency = current_pass%get_efficiency ()
                  mci%integral_known = .true.
                  mci%error_known = .true.
                  mci%efficiency_known = .true.
               end if
            end if
            current_pass => current_pass%next
            ref_pass => ref_pass%next
         else
            success = .false.
         end if
      end do
   end select
end subroutine mci_vamp_update_from_ref
```

Update the MCI record (i.e., the integration passes) by reading from input stream. The stream should contain a `write` output from a previous run. We first check the MD5 sum of the configuration parameters. If that matches, we proceed directly to the stored integration passes. If successful, we may continue to read the file; the position will be after a blank line that must follow the MCI record.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: update => mci_vamp_update
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_update (mci, u, success)
  class(mci_vamp_t), intent(inout) :: mci
  integer, intent(in) :: u
  logical, intent(out) :: success
  character(80) :: buffer
  character(32) :: md5sum_file
  type(mci_vamp_t) :: mci_file
  integer :: n_pass, n_it
  call read_sval (u, md5sum_file)
  if (mci%check_grid_file) then
     success = md5sum_file == mci%md5sum
  else
     success = .true.
  end if
  if (success) then
     read (u, *)
     read (u, "(A)")  buffer
     if (trim (adjustl (buffer)) == "VAMP integrator:") then
```

```
         n_pass = 0
         n_it = 0
         do
            read (u, "(A)")  buffer
            select case (trim (adjustl (buffer)))
            case ("")
               exit
            case ("Integration pass:")
               call mci_file%add_pass ()
               call mci_file%current_pass%read (u, n_pass, n_it)
               n_pass = n_pass + 1
               n_it = n_it + mci_file%current_pass%n_it
            end select
         end do
         call mci%update_from_ref (mci_file, success)
         call mci_file%final ()
      else
         call msg_fatal ("VAMP: reading grid file: corrupted data")
      end if
   end if
end subroutine mci_vamp_update
```

Read / write grids from / to file.

Bug fix for 2.2.5: after reading grids from file, channel weights must be copied back to the `mci_instance` record.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: write_grids => mci_vamp_write_grids
procedure :: read_grids_header => mci_vamp_read_grids_header
procedure :: read_grids_data => mci_vamp_read_grids_data
procedure :: read_grids => mci_vamp_read_grids
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_write_grids (mci, instance)
  class(mci_vamp_t), intent(in) :: mci
  class(mci_instance_t), intent(inout) :: instance
  integer :: u
  select type (instance)
  type is (mci_vamp_instance_t)
     if (mci%grid_filename_set) then
        if (instance%grids_defined) then
           u = free_unit ()
           open (u, file = char (mci%grid_filename), &
                action = "write", status = "replace")
           write (u, "(1x,A,A,A)")  "MD5sum = '", mci%md5sum, "'"
           write (u, *)
           call mci%write (u)
           write (u, *)
           write (u, "(1x,A)")  "VAMP grids:"
           call vamp_write_grids (instance%grids, u, &
                write_integrals = .true.)
           close (u)
        else
           call msg_bug ("VAMP: write grids: grids undefined")
        end if
```

```
      else
         call msg_bug ("VAMP: write grids: filename undefined")
      end if
   end select
end subroutine mci_vamp_write_grids

subroutine mci_vamp_read_grids_header (mci, success)
   class(mci_vamp_t), intent(inout) :: mci
   logical, intent(out) :: success
   logical :: exist
   integer :: u
   success = .false.
   if (mci%grid_filename_set) then
      inquire (file = char (mci%grid_filename), exist = exist)
      if (exist) then
         u = free_unit ()
         open (u, file = char (mci%grid_filename), &
              action = "read", status = "old")
         call mci%update (u, success)
         close (u)
         if (.not. success) then
            write (msg_buffer, "(A,A,A)") &
                 "VAMP: parameter mismatch, discarding grid file '", &
                 char (mci%grid_filename), "'"
            call msg_message ()
         end if
      end if
   else
      call msg_bug ("VAMP: read grids: filename undefined")
   end if
end subroutine mci_vamp_read_grids_header

subroutine mci_vamp_read_grids_data (mci, instance, read_integrals)
   class(mci_vamp_t), intent(in) :: mci
   class(mci_instance_t), intent(inout) :: instance
   logical, intent(in), optional :: read_integrals
   integer :: u
   character(80) :: buffer
   select type (instance)
   type is (mci_vamp_instance_t)
      if (.not. instance%grids_defined) then
         u = free_unit ()
         open (u, file = char (mci%grid_filename), &
              action = "read", status = "old")
         do
            read (u, "(A)")  buffer
            if (trim (adjustl (buffer)) == "VAMP grids:")  exit
         end do
         call vamp_read_grids (instance%grids, u, read_integrals)
         close (u)
         call instance%set_channel_weights (instance%grids%weights)
         instance%grids_defined = .true.
      else
         call msg_bug ("VAMP: read grids: grids already defined")
```

```
      end if
    end select
  end subroutine mci_vamp_read_grids_data

  subroutine mci_vamp_read_grids (mci, instance, success)
    class(mci_vamp_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout) :: instance
    logical, intent(out) :: success
    logical :: exist
    integer :: u
    character(80) :: buffer
    select type (instance)
    type is (mci_vamp_instance_t)
       success = .false.
       if (mci%grid_filename_set) then
          if (.not. instance%grids_defined) then
             inquire (file = char (mci%grid_filename), exist = exist)
             if (exist) then
                u = free_unit ()
                open (u, file = char (mci%grid_filename), &
                     action = "read", status = "old")
                call mci%update (u, success)
                if (success) then
                   read (u, "(A)")  buffer
                   if (trim (adjustl (buffer)) == "VAMP grids:") then
                      call vamp_read_grids (instance%grids, u)
                   else
                      call msg_fatal ("VAMP: reading grid file: &
                           &corrupted grid data")
                   end if
                else
                   write (msg_buffer, "(A,A,A)") &
                        "VAMP: parameter mismatch, discarding grid file '", &
                        char (mci%grid_filename), "'"
                   call msg_message ()
                end if
                close (u)
                instance%grids_defined = success
             end if
          else
             call msg_bug ("VAMP: read grids: grids already defined")
          end if
       else
          call msg_bug ("VAMP: read grids: filename undefined")
       end if
    end select
  end subroutine mci_vamp_read_grids
```

Auxiliary: Read real, integer, string value. We search for an equals sign, the value must follow.

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine read_rval (u, rval)
    integer, intent(in) :: u
```

```
    real(default), intent(out) :: rval
    character(80) :: buffer
    read (u, "(A)")  buffer
    buffer = adjustl (buffer(scan (buffer, "=") + 1:))
    read (buffer, *)  rval
  end subroutine read_rval

  subroutine read_ival (u, ival)
    integer, intent(in) :: u
    integer, intent(out) :: ival
    character(80) :: buffer
    read (u, "(A)")  buffer
    buffer = adjustl (buffer(scan (buffer, "=") + 1:))
    read (buffer, *)  ival
  end subroutine read_ival

  subroutine read_sval (u, sval)
    integer, intent(in) :: u
    character(*), intent(out) :: sval
    character(80) :: buffer
    read (u, "(A)")  buffer
    buffer = adjustl (buffer(scan (buffer, "=") + 1:))
    read (buffer, *)  sval
  end subroutine read_sval

  subroutine read_lval (u, lval)
    integer, intent(in) :: u
    logical, intent(out) :: lval
    character(80) :: buffer
    read (u, "(A)")  buffer
    buffer = adjustl (buffer(scan (buffer, "=") + 1:))
    read (buffer, *)  lval
  end subroutine read_lval
```

Integrate. Perform a new integration pass (possibly reusing previous results), which may consist of several iterations.

Note: we record the integral once per iteration. The integral stored in the `mci` record itself is the last integral of the current iteration, no averaging done. The `results` record may average results.

Note: recording the efficiency is not supported yet.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
  procedure :: integrate => mci_vamp_integrate
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_integrate (mci, instance, sampler, &
      n_it, n_calls, results, pacify)
    class(mci_vamp_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout), target :: instance
    class(mci_sampler_t), intent(inout), target :: sampler
    integer, intent(in) :: n_it
    integer, intent(in) :: n_calls
    class(mci_results_t), intent(inout), optional :: results
    logical, intent(in), optional :: pacify
    integer :: it
```

```
logical :: reshape, from_file, success
select type (instance)
type is (mci_vamp_instance_t)
   if (associated (mci%current_pass)) then
      mci%current_pass%integral_defined = .false.
      call mci%current_pass%configure (n_it, n_calls, &
           mci%min_calls, mci%grid_par%min_bins, &
           mci%grid_par%max_bins, &
           mci%grid_par%min_calls_per_channel * mci%n_channel)
      call mci%current_pass%configure_history &
           (mci%n_channel, mci%history_par)
      instance%pass_complete = .false.
      instance%it_complete = .false.
      call instance%new_pass (reshape)
      if (.not. instance%grids_defined .or. instance%grids_from_file) then
         if (mci%grid_filename_set .and. .not. mci%rebuild) then
            call mci%read_grids_header (success)
            from_file = success
            if (.not. instance%grids_defined .and. success) then
               call mci%read_grids_data (instance)
            end if
         else
            from_file = .false.
         end if
      else
         from_file = .false.
      end if
      if (from_file) then
         if (.not. mci%check_grid_file) &
              call msg_warning ("Reading grid file: MD5 sum check disabled")
         call msg_message ("VAMP: " &
              // "using grids and results from file '" &
              // char (mci%grid_filename) // "'")
      else if (.not. instance%grids_defined) then
         call instance%create_grids ()
      end if
      do it = 1, instance%n_it
         if (signal_is_pending ())  return
         instance%grids_from_file = from_file .and. &
              it <= mci%current_pass%get_integration_index ()
         if (.not. instance%grids_from_file) then
            instance%it_complete = .false.
            call instance%adapt_grids ()
            if (signal_is_pending ())  return
            call instance%adapt_weights ()
            if (signal_is_pending ())  return
            call instance%discard_integrals (reshape)
            if (mci%grid_par%use_vamp_equivalences) then
               call instance%sample_grids (mci%rng, sampler, &
                    mci%equivalences)
            else
               call instance%sample_grids (mci%rng, sampler)
            end if
            if (signal_is_pending ())  return
```

```
         instance%it_complete = .true.
         if (instance%integral /= 0) then
            mci%current_pass%calls(it) = instance%calls
            mci%current_pass%integral(it) = instance%integral
            if (abs (instance%error / instance%integral) &
                 > epsilon (1._default)) then
               mci%current_pass%error(it) = instance%error
            end if
            mci%current_pass%efficiency(it) = instance%efficiency
         end if
         mci%current_pass%integral_defined = .true.
      end if
      if (present (results)) then
         if (mci%has_chains ()) then
            call mci%collect_chain_weights (instance%w)
            call results%record (1, &
                 n_calls    = mci%current_pass%calls(it), &
                 integral   = mci%current_pass%integral(it), &
                 error      = mci%current_pass%error(it), &
                 efficiency = mci%current_pass%efficiency(it), &
                 chain_weights = mci%chain_weights, &
                 suppress = pacify)
         else
            call results%record (1, &
                 n_calls    = mci%current_pass%calls(it), &
                 integral   = mci%current_pass%integral(it), &
                 error      = mci%current_pass%error(it), &
                 efficiency = mci%current_pass%efficiency(it), &
                 suppress = pacify)
         end if
      end if
      if (.not. instance%grids_from_file &
           .and. mci%grid_filename_set) then
         call mci%write_grids (instance)
      end if
      call instance%allow_adaptation ()
      reshape = .false.
      if (.not. mci%current_pass%is_final_pass) then
         call mci%check_goals (it, success)
         if (success)  exit
      end if
   end do
   if (signal_is_pending ())  return
   instance%pass_complete = .true.
   mci%integral = mci%current_pass%get_integral()
   mci%error = mci%current_pass%get_error()
   mci%efficiency = mci%current_pass%get_efficiency()
   mci%integral_known = .true.
   mci%error_known = .true.
   mci%efficiency_known = .true.
   call mci%compute_md5sum (pacify)
else
   call msg_bug ("MCI integrate: current_pass object not allocated")
end if
```

```
    end select
  end subroutine mci_vamp_integrate
```

Check whether we are already finished with this pass.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
  procedure :: check_goals => mci_vamp_check_goals
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_check_goals (mci, it, success)
    class(mci_vamp_t), intent(inout) :: mci
    integer, intent(in) :: it
    logical, intent(out) :: success
    success = .false.
    if (mci%error_reached (it)) then
       mci%current_pass%n_it = it
       call msg_message ("VAMP: error goal reached; &
            &skipping iterations")
       success = .true.
       return
    end if
    if (mci%rel_error_reached (it)) then
       mci%current_pass%n_it = it
       call msg_message ("VAMP: relative error goal reached; &
            &skipping iterations")
       success = .true.
       return
    end if
    if (mci%accuracy_reached (it)) then
       mci%current_pass%n_it = it
       call msg_message ("VAMP: accuracy goal reached; &
            &skipping iterations")
       success = .true.
       return
    end if
  end subroutine mci_vamp_check_goals
```

Return true if the error, relative error, or accuracy goal has been reached, if any.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
  procedure :: error_reached => mci_vamp_error_reached
  procedure :: rel_error_reached => mci_vamp_rel_error_reached
  procedure :: accuracy_reached => mci_vamp_accuracy_reached
```

⟨*MCI vamp: procedures*⟩+≡

```
  function mci_vamp_error_reached (mci, it) result (flag)
    class(mci_vamp_t), intent(in) :: mci
    integer, intent(in) :: it
    logical :: flag
    real(default) :: error_goal, error
    error_goal = mci%grid_par%error_goal
    if (error_goal > 0) then
       associate (pass => mci%current_pass)
         if (pass%integral_defined) then
            error = abs (pass%error(it))
```

```
          flag = error < error_goal
       else
          flag = .false.
       end if
     end associate
  else
     flag = .false.
  end if
end function mci_vamp_error_reached

function mci_vamp_rel_error_reached (mci, it) result (flag)
  class(mci_vamp_t), intent(in) :: mci
  integer, intent(in) :: it
  logical :: flag
  real(default) :: rel_error_goal, rel_error
  rel_error_goal = mci%grid_par%rel_error_goal
  if (rel_error_goal > 0) then
     associate (pass => mci%current_pass)
       if (pass%integral_defined) then
          if (pass%integral(it) /= 0) then
             rel_error = abs (pass%error(it) / pass%integral(it))
             flag = rel_error < rel_error_goal
          else
             flag = .true.
          end if
       else
          flag = .false.
       end if
     end associate
  else
     flag = .false.
  end if
end function mci_vamp_rel_error_reached

function mci_vamp_accuracy_reached (mci, it) result (flag)
  class(mci_vamp_t), intent(in) :: mci
  integer, intent(in) :: it
  logical :: flag
  real(default) :: accuracy_goal, accuracy
  accuracy_goal = mci%grid_par%accuracy_goal
  if (accuracy_goal > 0) then
     associate (pass => mci%current_pass)
       if (pass%integral_defined) then
          if (pass%integral(it) /= 0) then
             accuracy = abs (pass%error(it) / pass%integral(it)) &
                  * sqrt (real (pass%calls(it), default))
             flag = accuracy < accuracy_goal
          else
             flag = .true.
          end if
       else
          flag = .false.
       end if
     end associate
```

```
    else
       flag = .false.
    end if
  end function mci_vamp_accuracy_reached
```

Prepare an event generation pass. Should be called before a sequence of events is generated, then we should call the corresponding finalizer.

The pass-specific data of the previous integration pass are retained, but we reset the number of iterations and calls to zero. The latter now counts the number of events (calls to the sampling function, actually).

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
  procedure :: prepare_simulation => mci_vamp_prepare_simulation
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_prepare_simulation (mci)
    class(mci_vamp_t), intent(inout) :: mci
    logical :: success
    if (mci%grid_filename_set) then
       call mci%read_grids_header (success)
       call mci%compute_md5sum ()
       if (.not. success) then
          call msg_fatal ("Simulate: " &
               // "reading integration grids from file '" &
               // char (mci%grid_filename) // "' failed")
       end if
    else
       call msg_bug ("VAMP: simulation: no grids, no grid filename")
    end if
  end subroutine mci_vamp_prepare_simulation
```

Generate weighted event. Note that the event weight (`vamp_weight`) is not just the MCI weight. `vamp_next_event` selects a channel based on the channel weights multiplied by the (previously recorded) maximum integrand value of the channel. The MCI weight is renormalized accordingly, to cancel this effect on the result.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
  procedure :: generate_weighted_event => mci_vamp_generate_weighted_event
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_generate_weighted_event (mci, instance, sampler)
    class(mci_vamp_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout), target :: instance
    class(mci_sampler_t), intent(inout), target :: sampler
    class(vamp_data_t), allocatable :: data
    type(exception) :: vamp_exception
    select type (instance)
    type is (mci_vamp_instance_t)
       instance%vamp_weight_set = .false.
       allocate (mci_workspace_t :: data)
       select type (data)
       type is (mci_workspace_t)
          data%sampler => sampler
          data%instance => instance
```

```
      end select
      select type (rng => mci%rng)
      type is (rng_tao_t)
         if (instance%grids_defined) then
            call vamp_next_event ( &
                 instance%vamp_x, &
                 rng%state, &
                 instance%grids, &
                 vamp_sampling_function, &
                 data, &
                 phi = phi_trivial, &
                 weight = instance%vamp_weight, &
                 exc = vamp_exception)
            call handle_vamp_exception (vamp_exception, mci%verbose)
            instance%vamp_excess = 0
            instance%vamp_weight_set = .true.
         else
            call msg_bug ("VAMP: generate event: grids undefined")
         end if
      class default
        call msg_fatal ("VAMP event generation: &
              &random-number generator must be TAO")
      end select
    end select
  end subroutine mci_vamp_generate_weighted_event
```

Generate unweighted event.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
  procedure :: generate_unweighted_event => &
      mci_vamp_generate_unweighted_event
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_generate_unweighted_event (mci, instance, sampler)
    class(mci_vamp_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout), target :: instance
    class(mci_sampler_t), intent(inout), target :: sampler
    class(vamp_data_t), allocatable :: data
    logical :: positive
    type(exception) :: vamp_exception
    select type (instance)
    type is (mci_vamp_instance_t)
       instance%vamp_weight_set = .false.
       allocate (mci_workspace_t :: data)
       select type (data)
       type is (mci_workspace_t)
          data%sampler => sampler
          data%instance => instance
       end select
       select type (rng => mci%rng)
       type is (rng_tao_t)
          if (instance%grids_defined) then
             REJECTION: do
                call vamp_next_event ( &
                     instance%vamp_x, &
```

```
                rng%state, &
                instance%grids, &
                vamp_sampling_function, &
                data, &
                phi = phi_trivial, &
                excess = instance%vamp_excess, &
                positive = positive, &
                exc = vamp_exception)
            if (signal_is_pending ())  return
            if (sampler%is_valid ())  exit REJECTION
         end do REJECTION
         call handle_vamp_exception (vamp_exception, mci%verbose)
         if (positive) then
            instance%vamp_weight = 1
         else if (instance%negative_weights) then
            instance%vamp_weight = -1
         else
            call msg_fatal ("VAMP: event with negative weight generated")
            instance%vamp_weight = 0
         end if
         instance%vamp_weight_set = .true.
      else
         call msg_bug ("VAMP: generate event: grids undefined")
      end if
   class default
     call msg_fatal ("VAMP event generation: &
           &random-number generator must be TAO")
   end select
  end select
end subroutine mci_vamp_generate_unweighted_event
```

Rebuild an event, using the `state` input.

Note: This feature is currently unused.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: rebuild_event => mci_vamp_rebuild_event
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_rebuild_event (mci, instance, sampler, state)
  class(mci_vamp_t), intent(inout) :: mci
  class(mci_instance_t), intent(inout) :: instance
  class(mci_sampler_t), intent(inout) :: sampler
  class(mci_state_t), intent(in) :: state
  call msg_bug ("MCI vamp rebuild event not implemented yet")
end subroutine mci_vamp_rebuild_event
```

Pacify: override the default no-op, since VAMP numerics might need some massage.

⟨*MCI vamp: mci vamp: TBP*⟩+≡

```
procedure :: pacify => mci_vamp_pacify
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_pacify (object, efficiency_reset, error_reset)
  class(mci_vamp_t), intent(inout) :: object
  logical, intent(in), optional :: efficiency_reset, error_reset
```

```
    logical :: err_reset
    type(pass_t), pointer :: current_pass
    err_reset = .false.
    if (present (error_reset))  err_reset = error_reset
    current_pass => object%first_pass
    do while (associated (current_pass))
       if (allocated (current_pass%error) .and. err_reset) then
          current_pass%error = 0
       end if
       if (allocated (current_pass%efficiency) .and. err_reset) then
          current_pass%efficiency = 1
       end if
       current_pass => current_pass%next
    end do
  end subroutine mci_vamp_pacify
```

20.5.5 Sampler as Workspace

In the full setup, the sampling function requires the process instance object as workspace. We implement this by (i) implementing the process instance as a type extension of the abstract `sampler_t` object used by the MCI implementation and (ii) providing such an object as an extra argument to the sampling function that VAMP can call. To minimize cross-package dependencies, we use an abstract type `vamp_workspace` that VAMP declares and extend this by including a pointer to the `sampler` and `instance` objects. In the body of the sampling function, we dereference this pointer and can then work with the contents.

⟨*MCI vamp: types*⟩+≡

```
  type, extends (vamp_data_t) :: mci_workspace_t
     class(mci_sampler_t), pointer :: sampler => null ()
     class(mci_vamp_instance_t), pointer :: instance => null ()
  end type mci_workspace_t
```

20.5.6 Integrator instance

The history entries should point to the corresponding history entry in the `pass_t` object. If there is none, we may allocate a local history, which is then just transient.

⟨*MCI vamp: public*⟩+≡

```
  public :: mci_vamp_instance_t
```

⟨*MCI vamp: types*⟩+≡

```
  type, extends (mci_instance_t) :: mci_vamp_instance_t
     type(mci_vamp_t), pointer :: mci => null ()
     logical :: grids_defined = .false.
     logical :: grids_from_file = .false.
     integer :: n_it = 0
     integer :: it = 0
     logical :: pass_complete = .false.
     integer :: n_calls = 0
```

```
    integer :: calls = 0
    logical :: it_complete = .false.
    logical :: enable_adapt_grids = .false.
    logical :: enable_adapt_weights = .false.
    logical :: allow_adapt_grids = .false.
    logical :: allow_adapt_weights = .false.
    integer :: n_adapt_grids = 0
    integer :: n_adapt_weights = 0
    logical :: generating_events = .false.
    real(default) :: safety_factor = 1
    type(vamp_grids) :: grids
    real(default) :: g = 0
    real(default), dimension(:), allocatable :: gi
    real(default) :: integral = 0
    real(default) :: error = 0
    real(default) :: efficiency = 0
    real(default), dimension(:), allocatable :: vamp_x
    logical :: vamp_weight_set = .false.
    real(default) :: vamp_weight = 0
    real(default) :: vamp_excess = 0
    logical :: allocate_global_history = .false.
    type(vamp_history), dimension(:), pointer :: v_history => null ()
    logical :: allocate_channel_history = .false.
    type(vamp_history), dimension(:,:), pointer :: v_histories => null ()
  contains
  ⟨MCI vamp: mci vamp instance: TBP⟩
end type mci_vamp_instance_t
```

Output.

⟨*MCI vamp: mci vamp instance: TBP*⟩≡

```
procedure :: write => mci_vamp_instance_write
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_instance_write (object, unit, pacify)
  class(mci_vamp_instance_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacify
  integer :: u, i
  character(len=7) :: fmt
  call pac_fmt (fmt, FMT_17, FMT_14, pacify)
  u = given_output_unit (unit)
  write (u, "(3x,A," // FMT_19 // ")") "Integrand = ", object%integrand
  write (u, "(3x,A," // FMT_19 // ")") "Weight    = ", object%mci_weight
  if (object%vamp_weight_set) then
     write (u, "(3x,A," // FMT_19 // ")") "VAMP wgt  = ", object%vamp_weight
     if (object%vamp_excess /= 0) then
        write (u, "(3x,A," // FMT_19 // ")") "VAMP exc  = ", &
              object%vamp_excess
     end if
  end if
  write (u, "(3x,A,L1)")  "adapt grids   = ", object%enable_adapt_grids
  write (u, "(3x,A,L1)")  "adapt weights = ", object%enable_adapt_weights
  if (object%grids_defined) then
     if (object%grids_from_file) then
```

```
      write (u, "(3x,A)")  "VAMP grids: read from file"
   else
      write (u, "(3x,A)")  "VAMP grids: defined"
   end if
else
   write (u, "(3x,A)")  "VAMP grids: [undefined]"
end if
write (u, "(3x,A,I0)")  "n_it          = ", object%n_it
write (u, "(3x,A,I0)")  "it            = ", object%it
write (u, "(3x,A,L1)")  "pass complete = ", object%it_complete
write (u, "(3x,A,I0)")  "n_calls       = ", object%n_calls
write (u, "(3x,A,I0)")  "calls         = ", object%calls
write (u, "(3x,A,L1)")  "it complete   = ", object%it_complete
write (u, "(3x,A,I0)")  "n adapt.(g)   = ", object%n_adapt_grids
write (u, "(3x,A,I0)")  "n adapt.(w)   = ", object%n_adapt_weights
write (u, "(3x,A,L1)")  "gen. events   = ", object%generating_events
write (u, "(3x,A,L1)")  "neg. weights  = ", object%negative_weights
if (object%safety_factor /= 1)  write &
      (u, "(3x,A," // fmt // ")")  "safety f = ", object%safety_factor
write (u, "(3x,A," // fmt // ")")  "integral = ", object%integral
write (u, "(3x,A," // fmt // ")")  "error    = ", object%error
write (u, "(3x,A," // fmt // ")")  "eff.     = ", object%efficiency
write (u, "(3x,A)")  "weights:"
do i = 1, size (object%w)
   write (u, "(5x,I0,1x," // FMT_12 // ")")  i, object%w(i)
end do
end subroutine mci_vamp_instance_write
```

Write the grids to the specified unit.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
procedure :: write_grids => mci_vamp_instance_write_grids
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_instance_write_grids (object, unit)
  class(mci_vamp_instance_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  if (object%grids_defined) then
     call vamp_write_grids (object%grids, u, write_integrals = .true.)
  end if
end subroutine mci_vamp_instance_write_grids
```

Finalizer: the history arrays are pointer arrays and need finalization.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
procedure :: final => mci_vamp_instance_final
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_instance_final (object)
  class(mci_vamp_instance_t), intent(inout) :: object
  if (object%allocate_global_history) then
     if (associated (object%v_history)) then
        call vamp_delete_history (object%v_history)
        deallocate (object%v_history)
```

```
      end if
    end if
    if (object%allocate_channel_history) then
      if (associated (object%v_histories)) then
        call vamp_delete_history (object%v_histories)
        deallocate (object%v_histories)
      end if
    end if
    if (object%grids_defined) then
      call vamp_delete_grids (object%grids)
      object%grids_defined = .false.
    end if
  end subroutine mci_vamp_instance_final
```

Initializer.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
  procedure :: init => mci_vamp_instance_init
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_instance_init (mci_instance, mci)
    class(mci_vamp_instance_t), intent(out) :: mci_instance
    class(mci_t), intent(in), target :: mci
    call mci_instance%base_init (mci)
    select type (mci)
    type is (mci_vamp_t)
      mci_instance%mci => mci
      allocate (mci_instance%gi (mci%n_channel))
      mci_instance%allocate_global_history = .not. mci%history_par%global
      mci_instance%allocate_channel_history = .not. mci%history_par%channel
      mci_instance%negative_weights = mci%negative_weights
    end select
  end subroutine mci_vamp_instance_init
```

Prepare a new integration pass: write the pass-specific settings to the `instance` object. This should be called initially, together with the `create_grids` procedure, and whenever we start a new integration pass.

Set `reshape` if the number of calls is different than previously (unless it was zero, indicating the first pass).

We link VAMP histories to the allocated histories in the current pass object, so the recorded results are persistent. However, if there are no histories present there, we allocate them locally. In that case, the histories will disappear together with the MCI instance object.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
  procedure :: new_pass => mci_vamp_instance_new_pass
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_instance_new_pass (instance, reshape)
    class(mci_vamp_instance_t), intent(inout) :: instance
    logical, intent(out) :: reshape
    type(pass_t), pointer :: current
    associate (mci => instance%mci)
      current => mci%current_pass
      instance%n_it = current%n_it
```

```
      if (instance%n_calls == 0) then
         reshape = .false.
         instance%n_calls = current%n_calls
      else if (instance%n_calls == current%n_calls) then
         reshape = .false.
      else
         reshape = .true.
         instance%n_calls = current%n_calls
      end if
      instance%it = 0
      instance%calls = 0
      instance%enable_adapt_grids = current%adapt_grids
      instance%enable_adapt_weights = current%adapt_weights
      instance%generating_events = .false.
      if (instance%allocate_global_history) then
         if (associated (instance%v_history)) then
            call vamp_delete_history (instance%v_history)
            deallocate (instance%v_history)
         end if
         allocate (instance%v_history (instance%n_it))
         call vamp_create_history (instance%v_history, verbose = .false.)
      else
         instance%v_history => current%v_history
      end if
      if (instance%allocate_channel_history) then
         if (associated (instance%v_histories)) then
            call vamp_delete_history (instance%v_histories)
            deallocate (instance%v_histories)
         end if
         allocate (instance%v_histories (instance%n_it, mci%n_channel))
         call vamp_create_history (instance%v_histories, verbose = .false.)
      else
         instance%v_histories => current%v_histories
      end if
    end associate
  end subroutine mci_vamp_instance_new_pass
```

Create a grid set within the `instance` object, using the data of the current integration pass. Also reset counters that track this grid set.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
  procedure :: create_grids => mci_vamp_instance_create_grids
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_instance_create_grids (instance)
    class(mci_vamp_instance_t), intent(inout) :: instance
    type (pass_t), pointer :: current
    integer, dimension(:), allocatable :: num_div
    real(default), dimension(:,:), allocatable :: region
    associate (mci => instance%mci)
      current => mci%current_pass
      allocate (num_div (mci%n_dim))
      allocate (region (2, mci%n_dim))
      region(1,:) = 0
      region(2,:) = 1
```

```
      num_div = current%n_bins
      instance%n_adapt_grids = 0
      instance%n_adapt_weights = 0
      if (.not. instance%grids_defined) then
         call vamp_create_grids (instance%grids, &
              region, &
              current%n_calls, &
              weights = instance%w, &
              num_div = num_div, &
              stratified = mci%grid_par%stratified)
         instance%grids_defined = .true.
      else
         call msg_bug ("VAMP: create grids: grids already defined")
      end if
    end associate
  end subroutine mci_vamp_instance_create_grids
```

Reset a grid set, so we can start a fresh integration pass. In effect, we delete results of previous integrations, but keep the grid shapes, weights, and variance arrays, so adaptation is still possible. The grids are prepared for a specific number of calls (per iteration) and sampling mode (stratified/importance).

The `vamp_discard_integrals` implementation will reshape the grids only if the argument `num_calls` is present.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
  procedure :: discard_integrals => mci_vamp_instance_discard_integrals
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_instance_discard_integrals (instance, reshape)
    class(mci_vamp_instance_t), intent(inout) :: instance
    logical, intent(in) :: reshape
    instance%calls = 0
    instance%integral = 0
    instance%error = 0
    instance%efficiency = 0
    associate (mci => instance%mci)
      if (instance%grids_defined) then
         if (mci%grid_par%use_vamp_equivalences) then
            if (reshape) then
               call vamp_discard_integrals (instance%grids, &
                    num_calls = instance%n_calls, &
                    stratified = mci%grid_par%stratified, &
                    eq = mci%equivalences)
            else
               call vamp_discard_integrals (instance%grids, &
                    stratified = mci%grid_par%stratified, &
                    eq = mci%equivalences)
            end if
         else
            if (reshape) then
               call vamp_discard_integrals (instance%grids, &
                    num_calls = instance%n_calls, &
                    stratified = mci%grid_par%stratified)
            else
               call vamp_discard_integrals (instance%grids, &
```

```
                    stratified = mci%grid_par%stratified)
            end if
         end if
      else
         call msg_bug ("VAMP: discard integrals: grids undefined")
      end if
    end associate
  end subroutine mci_vamp_instance_discard_integrals
```

After grids are created (with equidistant binning and equal weight), adaptation is redundant. Therefore, we should allow it only after a complete integration step has been performed, calling this.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
  procedure :: allow_adaptation => mci_vamp_instance_allow_adaptation
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_instance_allow_adaptation (instance)
    class(mci_vamp_instance_t), intent(inout) :: instance
    instance%allow_adapt_grids = .true.
    instance%allow_adapt_weights = .true.
  end subroutine mci_vamp_instance_allow_adaptation
```

Adapt grids.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
  procedure :: adapt_grids => mci_vamp_instance_adapt_grids
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_instance_adapt_grids (instance)
    class(mci_vamp_instance_t), intent(inout) :: instance
    if (instance%enable_adapt_grids .and. instance%allow_adapt_grids) then
       if (instance%grids_defined) then
          call vamp_refine_grids (instance%grids)
          instance%n_adapt_grids = instance%n_adapt_grids + 1
      else
         call msg_bug ("VAMP: adapt grids: grids undefined")
      end if
    end if
  end subroutine mci_vamp_instance_adapt_grids
```

Adapt weights. Use the variance array returned by `VAMP` for recalculating the weight array. The parameter `channel_weights_power` dampens fluctuations.

If the number of calls in a given channel falls below a user-defined threshold, the weight is not lowered further but kept at this threshold. The other channel weights are reduced accordingly.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
  procedure :: adapt_weights => mci_vamp_instance_adapt_weights
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_instance_adapt_weights (instance)
    class(mci_vamp_instance_t), intent(inout) :: instance
    real(default) :: w_sum, w_avg_ch, sum_w_underflow, w_min
    real(default), dimension(:), allocatable :: weights
    integer :: n_ch, ch, n_underflow
```

```
  logical, dimension(:), allocatable :: mask, underflow
  type(exception) :: vamp_exception
  logical :: wsum_non_zero
  if (instance%enable_adapt_weights .and. instance%allow_adapt_weights) then
     associate (mci => instance%mci)
       if (instance%grids_defined) then
          allocate (weights (size (instance%grids%weights)))
          weights = instance%grids%weights &
               * vamp_get_variance (instance%grids%grids) &
               ** mci%grid_par%channel_weights_power
          w_sum = sum (weights)
          if (w_sum /= 0) then
             weights = weights / w_sum
             if (mci%n_chain /= 0) then
                allocate (mask (mci%n_channel))
                do ch = 1, mci%n_chain
                   mask = mci%chain == ch
                   n_ch = count (mask)
                   if (n_ch /= 0) then
                      w_avg_ch = sum (weights, mask) / n_ch
                      where (mask)  weights = w_avg_ch
                   end if
                end do
             end if
             if (mci%grid_par%threshold_calls /= 0) then
                w_min = &
                     real (mci%grid_par%threshold_calls, default) &
                     / instance%n_calls
                allocate (underflow (mci%n_channel))
                underflow = weights /= 0 .and. abs (weights) < w_min
                n_underflow = count (underflow)
                sum_w_underflow = sum (weights, mask=underflow)
                if (sum_w_underflow /= 1) then
                   where (underflow)
                      weights = w_min
                   elsewhere
                      weights = weights &
                           * (1 - n_underflow * w_min) / (1 - sum_w_underflow)
                   end where
                end if
             end if
          end if
          call instance%set_channel_weights (weights, wsum_non_zero)
          if (wsum_non_zero) call vamp_update_weights &
             (instance%grids, weights, exc = vamp_exception)
          call handle_vamp_exception (vamp_exception, mci%verbose)
       else
          call msg_bug ("VAMP: adapt weights: grids undefined")
       end if
     end associate
     instance%n_adapt_weights = instance%n_adapt_weights + 1
  end if
end subroutine mci_vamp_instance_adapt_weights
```

Integration: sample the VAMP grids. The number of calls etc. are already stored inside the grids. We provide the random-number generator, the sampling function, and a link to the workspace object, which happens to contain a pointer to the sampler object. The sampler object thus becomes the workspace of the sampling function.

Note: in the current implementation, the random-number generator must be the TAO generator. This explicit dependence should be removed from the VAMP implementation.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
procedure :: sample_grids => mci_vamp_instance_sample_grids
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_instance_sample_grids (instance, rng, sampler, eq)
  class(mci_vamp_instance_t), intent(inout), target :: instance
  class(rng_t), intent(inout) :: rng
  class(mci_sampler_t), intent(inout), target :: sampler
  type(vamp_equivalences_t), intent(in), optional :: eq
  class(vamp_data_t), allocatable :: data
  type(exception) :: vamp_exception
  allocate (mci_workspace_t :: data)
  select type (data)
  type is (mci_workspace_t)
     data%sampler => sampler
     data%instance => instance
  end select
  select type (rng)
  type is (rng_tao_t)
     instance%it = instance%it + 1
     instance%calls = 0
     if (instance%grids_defined) then
        call vamp_sample_grids ( &
             rng%state, &
             instance%grids, &
             vamp_sampling_function, &
             data, &
             1, &
             eq = eq, &
             history = instance%v_history(instance%it:), &
             histories = instance%v_histories(instance%it:,:), &
             integral = instance%integral, &
             std_dev = instance%error, &
             exc = vamp_exception, &
             negative_weights = instance%negative_weights)
        call handle_vamp_exception (vamp_exception, instance%mci%verbose)
        instance%efficiency = instance%get_efficiency ()
     else
        call msg_bug ("VAMP: sample grids: grids undefined")
     end if
  class default
     call msg_fatal ("VAMP integration: random-number generator must be TAO")
  end select
end subroutine mci_vamp_instance_sample_grids
```

Compute the reweighting efficiency for the current grids, suitable averaged over all active channels.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
procedure :: get_efficiency_array => mci_vamp_instance_get_efficiency_array
procedure :: get_efficiency => mci_vamp_instance_get_efficiency
```

⟨*MCI vamp: procedures*⟩+≡

```
function mci_vamp_instance_get_efficiency_array (mci) result (efficiency)
  class(mci_vamp_instance_t), intent(in) :: mci
  real(default), dimension(:), allocatable :: efficiency
  allocate (efficiency (mci%mci%n_channel))
  if (.not. mci%negative_weights) then
     where (mci%grids%grids%f_max /= 0)
        efficiency = mci%grids%grids%mu(1) / abs (mci%grids%grids%f_max)
     elsewhere
        efficiency = 0
     end where
  else
     where (mci%grids%grids%f_max /= 0)
        efficiency = &
             (mci%grids%grids%mu_plus(1) - mci%grids%grids%mu_minus(1)) &
             / abs (mci%grids%grids%f_max)
     elsewhere
        efficiency = 0
     end where
  end if
end function mci_vamp_instance_get_efficiency_array

function mci_vamp_instance_get_efficiency (mci) result (efficiency)
  class(mci_vamp_instance_t), intent(in) :: mci
  real(default) :: efficiency
  real(default), dimension(:), allocatable :: weight
  real(default) :: norm
  allocate (weight (mci%mci%n_channel))
  weight = mci%grids%weights * abs (mci%grids%grids%f_max)
  norm = sum (weight)
  if (norm /= 0) then
     efficiency = dot_product (mci%get_efficiency_array (), weight) / norm
  else
     efficiency = 1
  end if
end function mci_vamp_instance_get_efficiency
```

Prepare an event generation pass. Should be called before a sequence of events is generated, then we should call the corresponding finalizer.

The pass-specific data of the previous integration pass are retained, but we reset the number of iterations and calls to zero. The latter now counts the number of events (calls to the sampling function, actually).

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
procedure :: init_simulation => mci_vamp_instance_init_simulation
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_instance_init_simulation (instance, safety_factor)
  class(mci_vamp_instance_t), intent(inout) :: instance
```

```
    real(default), intent(in), optional :: safety_factor
    associate (mci => instance%mci)
      allocate (instance%vamp_x (mci%n_dim))
      instance%it = 0
      instance%calls = 0
      instance%generating_events = .true.
      if (present (safety_factor))  instance%safety_factor = safety_factor
      if (.not. instance%grids_defined) then
         if (mci%grid_filename_set) then
            if (.not. mci%check_grid_file) &
                 call msg_warning ("Reading grid file: MD5 sum check disabled")
            call msg_message ("Simulate: " &
                 // "using integration grids from file '" &
                 // char (mci%grid_filename) // "'")
            call mci%read_grids_data (instance)
            if (instance%safety_factor /= 1) then
               write (msg_buffer, "(A,ES10.3,A)")  "Simulate: &
                    &applying safety factor", instance%safety_factor, &
                    " to event rejection"
               call msg_message ()
               instance%grids%grids%f_max = &
                    instance%grids%grids%f_max * instance%safety_factor
            end if
         else
            call msg_bug ("VAMP: simulation: no grids, no grid filename")
         end if
      end if
    end associate
  end subroutine mci_vamp_instance_init_simulation
```

Finalize an event generation pass. Should be called before a sequence of events is generated, then we should call the corresponding finalizer.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
  procedure :: final_simulation => mci_vamp_instance_final_simulation
```

⟨*MCI vamp: procedures*⟩+≡

```
  subroutine mci_vamp_instance_final_simulation (instance)
    class(mci_vamp_instance_t), intent(inout) :: instance
    if (allocated (instance%vamp_x))  deallocate (instance%vamp_x)
  end subroutine mci_vamp_instance_final_simulation
```

20.5.7 Sampling function

The VAMP sampling function has a well-defined interface which we have to implement. The `data` argument allows us to pass pointers to the `sampler` and `instance` objects, so we can access configuration data and fill point-dependent contents within these objects.

The `weights` and `channel` argument must be present in the call.

Note: we would normally declare the `instance` pointer with the concrete type, or just use the `data` component directly. Unfortunately, gfortran 4.6 forgets the inherited base-type methods in that case.

Note: this is the place where we must look for external signals, i.e., interrupt from the OS. We would like to raise a `VAMP` exception which is then caught by `vamp_sample_grids` as the caller, so it dumps its current state and returns (with the signal still pending). `WHIZARD` will then terminate gracefully. Of course, VAMP should be able to resume from the dump.

In the current implementation, we handle the exception in place and terminate immediately. The incomplete current integration pass is lost.

⟨*MCI vamp: procedures*⟩+≡

```
function vamp_sampling_function &
     (xi, data, weights, channel, grids) result (f)
  real(default) :: f
  real(default), dimension(:), intent(in) :: xi
  class(vamp_data_t), intent(in) :: data
  real(default), dimension(:), intent(in), optional :: weights
  integer, intent(in), optional :: channel
  type(vamp_grid), dimension(:), intent(in), optional :: grids
  type(exception) :: exc
  class(mci_instance_t), pointer :: instance
  logical :: verbose
  character(*), parameter :: FN = "WHIZARD sampling function"
  select type (data)
  type is (mci_workspace_t)
     instance => data%instance
     select type (instance)
     class is (mci_vamp_instance_t)
        instance%calls = instance%calls + 1
        verbose = instance%mci%verbose
     end select
     call instance%evaluate (data%sampler, channel, xi)
     if (signal_is_pending ()) then
        call raise_exception (exc, EXC_FATAL, FN, "signal received")
        call handle_vamp_exception (exc, verbose)
        call terminate_now_if_signal ()
     end if
     f = instance%get_value ()
     call terminate_now_if_single_event ()
  end select
end function vamp_sampling_function
```

This is supposed to be the mapping between integration channels. The VAMP event generating procedures technically require it, but it is meaningless in our setup where all transformations happen inside the sampler object. So, this implementation is trivial:

⟨*MCI vamp: procedures*⟩+≡

```
pure function phi_trivial (xi, channel_dummy) result (x)
  real(default), dimension(:), intent(in) :: xi
  integer, intent(in) :: channel_dummy
  real(default), dimension(size(xi)) :: x
  x = xi
end function phi_trivial
```

20.5.8 Integrator instance: evaluation

Here, we compute the multi-channel reweighting factor for the current channel, that accounts for the Jacobians of the transformations from/to all other channels.

The computation of the VAMP probabilities may consume considerable time, therefore we enable parallel evaluation. (Collecting the contributions to `mci%g` is a reduction, which we should also implement via OpenMP.)

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
procedure :: compute_weight => mci_vamp_instance_compute_weight
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_instance_compute_weight (mci, c)
  class(mci_vamp_instance_t), intent(inout) :: mci
  integer, intent(in) :: c
  integer :: i
  mci%selected_channel = c
  !$OMP PARALLEL PRIVATE(i) SHARED(mci)
  !$OMP DO
  do i = 1, mci%mci%n_channel
     if (mci%w(i) /= 0) then
        mci%gi(i) = vamp_probability (mci%grids%grids(i), mci%x(:,i))
     else
        mci%gi(i) = 0
     end if
  end do
  !$OMP END DO
  !$OMP END PARALLEL
  mci%g = 0
  if (mci%gi(c) /= 0) then
     do i = 1, mci%mci%n_channel
        if (mci%w(i) /= 0 .and. mci%f(i) /= 0) then
           mci%g = mci%g + mci%w(i) * mci%gi(i) / mci%f(i)
        end if
     end do
  end if
  if (mci%g /= 0) then
     mci%mci_weight = mci%gi(c) / mci%g
  else
     mci%mci_weight = 0
  end if
end subroutine mci_vamp_instance_compute_weight
```

Record the integrand.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
procedure :: record_integrand => mci_vamp_instance_record_integrand
```

⟨*MCI vamp: procedures*⟩+≡

```
subroutine mci_vamp_instance_record_integrand (mci, integrand)
  class(mci_vamp_instance_t), intent(inout) :: mci
  real(default), intent(in) :: integrand
  mci%integrand = integrand
end subroutine mci_vamp_instance_record_integrand
```

Get the event weight. The default routine returns the same value that we would use for integration. This is correct if we select the integration channel according to the channel weight. `vamp_next_event` does differently, so we should rather rely on the weight that VAMP returns. This is the value stored in `vamp_weight`. We override the default TBP accordingly.

⟨*MCI vamp: mci vamp instance: TBP*⟩+≡

```
procedure :: get_event_weight => mci_vamp_instance_get_event_weight
procedure :: get_event_excess => mci_vamp_instance_get_event_excess
```

⟨*MCI vamp: procedures*⟩+≡

```
function mci_vamp_instance_get_event_weight (mci) result (value)
  class(mci_vamp_instance_t), intent(in) :: mci
  real(default) :: value
  if (mci%vamp_weight_set) then
     value = mci%vamp_weight
  else
     call msg_bug ("VAMP: attempt to read undefined event weight")
  end if
end function mci_vamp_instance_get_event_weight

function mci_vamp_instance_get_event_excess (mci) result (value)
  class(mci_vamp_instance_t), intent(in) :: mci
  real(default) :: value
  if (mci%vamp_weight_set) then
     value = mci%vamp_excess
  else
     call msg_bug ("VAMP: attempt to read undefined event excess weight")
  end if
end function mci_vamp_instance_get_event_excess
```

20.5.9 VAMP exceptions

A VAMP routine may have raised an exception. Turn this into a WHIZARD error message.

An external signal could raise a fatal exception, but this should be delayed and handled by the correct termination routine.

⟨*MCI vamp: procedures*⟩+≡

```
subroutine handle_vamp_exception (exc, verbose)
  type(exception), intent(in) :: exc
  logical, intent(in) :: verbose
  integer :: exc_level
  if (verbose) then
     exc_level = EXC_INFO
  else
     exc_level = EXC_ERROR
  end if
  if (exc%level >= exc_level) then
     write (msg_buffer, "(A,':',1x,A)")  trim (exc%origin), trim (exc%message)
     select case (exc%level)
     case (EXC_INFO);  call msg_message ()
     case (EXC_WARN);  call msg_warning ()
     case (EXC_ERROR); call msg_error ()
```

```
      case (EXC_FATAL)
         if (signal_is_pending ()) then
            call msg_message ()
         else
            call msg_fatal ()
         end if
      end select
   end if
end subroutine handle_vamp_exception
```

20.5.10 Unit tests

Test module, followed by the corresponding implementation module.

⟨mci_vamp_ut.f90⟩≡

```
⟨File header⟩

module mci_vamp_ut
  use unit_tests
  use mci_vamp_uti

⟨Standard module head⟩

⟨MCI vamp: public test⟩

contains

⟨MCI vamp: test driver⟩

end module mci_vamp_ut
```

⟨mci_vamp_uti.f90⟩≡

```
⟨File header⟩

module mci_vamp_uti

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use constants, only: PI, TWOPI
  use rng_base
  use rng_tao
  use phs_base
  use mci_base
  use vamp, only: vamp_write_grids !NODEP!

  use mci_vamp

⟨Standard module head⟩

⟨MCI vamp: test declarations⟩

⟨MCI vamp: test types⟩
```

```
contains

⟨MCI vamp: tests⟩

end module mci_vamp_uti
```

API: driver for the unit tests below.

⟨*MCI vamp: public test*⟩≡

```
public :: mci_vamp_test
```

⟨*MCI vamp: test driver*⟩≡

```
subroutine mci_vamp_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨MCI vamp: execute tests⟩
end subroutine mci_vamp_test
```

Test sampler

A test sampler object should implement a function with known integral that we can use to check the integrator.

In mode 1, the function is $f(x) = 3x^2$ with integral $\int_0^1 f(x)\,dx = 1$ and maximum $f(1) = 3$. If the integration dimension is greater than one, the function is extended as a constant in the other dimension(s).

In mode 2, the function is $11x^{10}$, also with integral 1.

Mode 4 includes ranges of zero and negative function value, the integral is negative. The results should be identical to the results of `mci_midpoint_4`, where the same function is evaluated. The function is $f(x) = (1-3x^2)\,\theta(x-1/2)$ with integral $\int_0^1 f(x)\,dx = -3/8$, minimum $f(1) = -2$ and maximum $f(1/2) = 1/4$.

⟨*MCI vamp: test types*⟩≡

```
type, extends (mci_sampler_t) :: test_sampler_1_t
   real(default), dimension(:), allocatable :: x
   real(default) :: val
   integer :: mode = 1
 contains
 ⟨MCI vamp: test sampler 1: TBP⟩
end type test_sampler_1_t
```

Output: There is nothing stored inside, so just print an informative line.

⟨*MCI vamp: test sampler 1: TBP*⟩≡

```
procedure :: write => test_sampler_1_write
```

⟨*MCI vamp: tests*⟩≡

```
subroutine test_sampler_1_write (object, unit, testflag)
  class(test_sampler_1_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  select case (object%mode)
  case (1)
     write (u, "(1x,A)") "Test sampler: f(x) = 3 x^2"
```

```
    case (2)
       write (u, "(1x,A)") "Test sampler: f(x) = 11 x^10"
    case (3)
       write (u, "(1x,A)") "Test sampler: f(x) = 11 x^10 * 2 * cos^2 (2 pi y)"
    case (4)
       write (u, "(1x,A)") "Test sampler: f(x) = (1 - 3 x^2) theta(x - 1/2)"
    end select
  end subroutine test_sampler_1_write
```

Evaluation: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI vamp: test sampler 1: TBP*⟩+≡

```
  procedure :: evaluate => test_sampler_1_evaluate
```

⟨*MCI vamp: tests*⟩+≡

```
  subroutine test_sampler_1_evaluate (sampler, c, x_in, val, x, f)
    class(test_sampler_1_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    if (allocated (sampler%x))  deallocate (sampler%x)
    allocate (sampler%x (size (x_in)))
    sampler%x = x_in
    select case (sampler%mode)
    case (1)
       sampler%val = 3 * x_in(1) ** 2
    case (2)
       sampler%val = 11 * x_in(1) ** 10
    case (3)
       sampler%val = 11 * x_in(1) ** 10 * 2 * cos (twopi * x_in(2)) ** 2
    case (4)
       if (x_in(1) >= .5_default) then
          sampler%val = 1 - 3 * x_in(1) ** 2
       else
          sampler%val = 0
       end if
    end select
    call sampler%fetch (val, x, f)
  end subroutine test_sampler_1_evaluate
```

The point is always valid.

⟨*MCI vamp: test sampler 1: TBP*⟩+≡

```
  procedure :: is_valid => test_sampler_1_is_valid
```

⟨*MCI vamp: tests*⟩+≡

```
  function test_sampler_1_is_valid (sampler) result (valid)
    class(test_sampler_1_t), intent(in) :: sampler
    logical :: valid
    valid = .true.
  end function test_sampler_1_is_valid
```

Rebuild: compute all but the function value.

⟨*MCI vamp: test sampler 1: TBP*⟩+≡
```
procedure :: rebuild => test_sampler_1_rebuild
```

⟨*MCI vamp: tests*⟩+≡
```
subroutine test_sampler_1_rebuild (sampler, c, x_in, val, x, f)
  class(test_sampler_1_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(in) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  if (allocated (sampler%x))  deallocate (sampler%x)
  allocate (sampler%x (size (x_in)))
  sampler%x = x_in
  sampler%val = val
  x(:,1) = sampler%x
  f = 1
end subroutine test_sampler_1_rebuild
```

Extract the results.

⟨*MCI vamp: test sampler 1: TBP*⟩+≡
```
procedure :: fetch => test_sampler_1_fetch
```

⟨*MCI vamp: tests*⟩+≡
```
subroutine test_sampler_1_fetch (sampler, val, x, f)
  class(test_sampler_1_t), intent(in) :: sampler
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  val = sampler%val
  x(:,1) = sampler%x
  f = 1
end subroutine test_sampler_1_fetch
```

Two-channel, two dimension test sampler

This sampler implements the function

$$f(x,y) = 4\sin^2(\pi x)\sin^2(\pi y) + 2\sin^2(\pi v) \tag{20.11}$$

where

$$x = u^v \qquad u = xy \tag{20.12}$$

$$y = u^{(1-v)} \qquad v = \frac{1}{2}\left(1 + \frac{\log(x/y)}{\log xy}\right) \tag{20.13}$$

Each term contributes 1 to the integral. The first term in the function is peaked along a cross aligned to the coordinates x and y, while the second term is peaked along the diagonal $x = y$.

The Jacobian is

$$\frac{\partial(x,y)}{\partial(u,v)} = |\log u| \tag{20.14}$$

⟨*MCI vamp: test types*⟩+≡

```
type, extends (mci_sampler_t) :: test_sampler_2_t
   real(default), dimension(:,:), allocatable :: x
   real(default), dimension(:), allocatable :: f
   real(default) :: val
 contains
 ⟨MCI vamp: test sampler 2: TBP⟩
end type test_sampler_2_t
```

Output: There is nothing stored inside, so just print an informative line.

⟨*MCI vamp: test sampler 2: TBP*⟩≡

```
procedure :: write => test_sampler_2_write
```

⟨*MCI vamp: tests*⟩+≡

```
subroutine test_sampler_2_write (object, unit, testflag)
  class(test_sampler_2_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)") "Two-channel test sampler 2"
end subroutine test_sampler_2_write
```

Kinematics: compute x and Jacobians, given the input parameter array.

⟨*MCI vamp: test sampler 2: TBP*⟩+≡

```
procedure :: compute => test_sampler_2_compute
```

⟨*MCI vamp: tests*⟩+≡

```
subroutine test_sampler_2_compute (sampler, c, x_in)
  class(test_sampler_2_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default) :: xx, yy, uu, vv
  if (.not. allocated (sampler%x)) &
       allocate (sampler%x (size (x_in), 2))
  if (.not. allocated (sampler%f)) &
       allocate (sampler%f (2))
  select case (c)
  case (1)
     xx = x_in(1)
     yy = x_in(2)
     uu = xx * yy
     vv = (1 + log (xx/yy) / log (xx*yy)) / 2
  case (2)
     uu = x_in(1)
     vv = x_in(2)
     xx = uu ** vv
     yy = uu ** (1 - vv)
  end select
  sampler%val = (2 * sin (pi * xx) * sin (pi * yy)) ** 2 &
       + 2 * sin (pi * vv) ** 2
  sampler%f(1) = 1
  sampler%f(2) = abs (log (uu))
  sampler%x(:,1) = [xx, yy]
```

```
    sampler%x(:,2) = [uu, vv]
  end subroutine test_sampler_2_compute
```

Evaluation: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI vamp: test sampler 2: TBP*⟩+≡

```
  procedure :: evaluate => test_sampler_2_evaluate
```

⟨*MCI vamp: tests*⟩+≡

```
  subroutine test_sampler_2_evaluate (sampler, c, x_in, val, x, f)
    class(test_sampler_2_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    call sampler%compute (c, x_in)
    call sampler%fetch (val, x, f)
  end subroutine test_sampler_2_evaluate
```

The point is always valid.

⟨*MCI vamp: test sampler 2: TBP*⟩+≡

```
  procedure :: is_valid => test_sampler_2_is_valid
```

⟨*MCI vamp: tests*⟩+≡

```
  function test_sampler_2_is_valid (sampler) result (valid)
    class(test_sampler_2_t), intent(in) :: sampler
    logical :: valid
    valid = .true.
  end function test_sampler_2_is_valid
```

Rebuild: compute all but the function value.

⟨*MCI vamp: test sampler 2: TBP*⟩+≡

```
  procedure :: rebuild => test_sampler_2_rebuild
```

⟨*MCI vamp: tests*⟩+≡

```
  subroutine test_sampler_2_rebuild (sampler, c, x_in, val, x, f)
    class(test_sampler_2_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(in) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    call sampler%compute (c, x_in)
    x = sampler%x
    f = sampler%f
  end subroutine test_sampler_2_rebuild
```

Extract the results.

⟨*MCI vamp: test sampler 2: TBP*⟩+≡

```
  procedure :: fetch => test_sampler_2_fetch
```

⟨*MCI vamp: tests*⟩+≡

```
subroutine test_sampler_2_fetch (sampler, val, x, f)
  class(test_sampler_2_t), intent(in) :: sampler
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  val = sampler%val
  x = sampler%x
  f = sampler%f
end subroutine test_sampler_2_fetch
```

Two-channel, one dimension test sampler

This sampler implements the function

$$f(x, y) = a * 5x^4 + b * 5(1 - x)^4 \tag{20.15}$$

Each term contributes 1 to the integral, multiplied by a or b, respectively. The first term is peaked at $x = 1$, the second one at $x = 0$..

We implement the two mappings

$$x = u^{1/5} \quad \text{and} \quad x = 1 - v^{1/5}, \tag{20.16}$$

with Jacobians

$$\frac{\partial(x)}{\partial(u)} = u^{-4/5}/5 \quad \text{and} \quad v^{-4/5}/5, \tag{20.17}$$

respectively. The first mapping concentrates points near $x = 1$, the second one near $x = 0$.

⟨*MCI vamp: test types*⟩+≡

```
type, extends (mci_sampler_t) :: test_sampler_3_t
   real(default), dimension(:,:), allocatable :: x
   real(default), dimension(:), allocatable :: f
   real(default) :: val
   real(default) :: a = 1
   real(default) :: b = 1
 contains
 ⟨MCI vamp: test sampler 3: TBP⟩
end type test_sampler_3_t
```

Output: display a and b

⟨*MCI vamp: test sampler 3: TBP*⟩≡

```
procedure :: write => test_sampler_3_write
```

⟨*MCI vamp: tests*⟩+≡

```
subroutine test_sampler_3_write (object, unit, testflag)
  class(test_sampler_3_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)") "Two-channel test sampler 3"
  write (u, "(3x,A,F5.2)")  "a = ", object%a
```

```
    write (u, "(3x,A,F5.2)")  "b = ", object%b
  end subroutine test_sampler_3_write
```

Kinematics: compute x and Jacobians, given the input parameter array.

⟨*MCI vamp: test sampler 3: TBP*⟩+≡

```
  procedure :: compute => test_sampler_3_compute
```

⟨*MCI vamp: tests*⟩+≡

```
  subroutine test_sampler_3_compute (sampler, c, x_in)
    class(test_sampler_3_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default) :: u, v, xx
    if (.not. allocated (sampler%x)) &
         allocate (sampler%x (size (x_in), 2))
    if (.not. allocated (sampler%f)) &
         allocate (sampler%f (2))
    select case (c)
    case (1)
       u = x_in(1)
       xx = u ** 0.2_default
       v = (1 - xx) ** 5._default
    case (2)
       v = x_in(1)
       xx = 1 - v ** 0.2_default
       u = xx ** 5._default
    end select
    sampler%val = sampler%a * 5 * xx ** 4 + sampler%b * 5 * (1 - xx) ** 4
    sampler%f(1) = 0.2_default * u ** (-0.8_default)
    sampler%f(2) = 0.2_default * v ** (-0.8_default)
    sampler%x(:,1) = [u]
    sampler%x(:,2) = [v]
  end subroutine test_sampler_3_compute
```

Evaluation: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI vamp: test sampler 3: TBP*⟩+≡

```
  procedure :: evaluate => test_sampler_3_evaluate
```

⟨*MCI vamp: tests*⟩+≡

```
  subroutine test_sampler_3_evaluate (sampler, c, x_in, val, x, f)
    class(test_sampler_3_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    call sampler%compute (c, x_in)
    call sampler%fetch (val, x, f)
  end subroutine test_sampler_3_evaluate
```

The point is always valid.

⟨*MCI vamp: test sampler 3: TBP*⟩+≡

```
  procedure :: is_valid => test_sampler_3_is_valid
```

⟨*MCI vamp: tests*⟩+≡
```
function test_sampler_3_is_valid (sampler) result (valid)
  class(test_sampler_3_t), intent(in) :: sampler
  logical :: valid
  valid = .true.
end function test_sampler_3_is_valid
```

Rebuild: compute all but the function value.

⟨*MCI vamp: test sampler 3: TBP*⟩+≡
```
procedure :: rebuild => test_sampler_3_rebuild
```

⟨*MCI vamp: tests*⟩+≡
```
subroutine test_sampler_3_rebuild (sampler, c, x_in, val, x, f)
  class(test_sampler_3_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(in) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  call sampler%compute (c, x_in)
  x = sampler%x
  f = sampler%f
end subroutine test_sampler_3_rebuild
```

Extract the results.

⟨*MCI vamp: test sampler 3: TBP*⟩+≡
```
procedure :: fetch => test_sampler_3_fetch
```

⟨*MCI vamp: tests*⟩+≡
```
subroutine test_sampler_3_fetch (sampler, val, x, f)
  class(test_sampler_3_t), intent(in) :: sampler
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  val = sampler%val
  x = sampler%x
  f = sampler%f
end subroutine test_sampler_3_fetch
```

One-dimensional integration

Construct an integrator and use it for a one-dimensional sampler.

Note: We would like to check the precise contents of the grid allocated during integration, but the output format for reals is very long (for good reasons), so the last digits in the grid content display are numerical noise. So, we just check the integration results.

⟨*MCI vamp: execute tests*⟩≡
```
call test (mci_vamp_1, "mci_vamp_1", &
     "one-dimensional integral", &
     u, results)
```

⟨*MCI vamp: test declarations*⟩≡
```
public :: mci_vamp_1
```

⟨*MCI vamp: tests*⟩+≡

```
subroutine mci_vamp_1 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng

  write (u, "(A)")  "* Test output: mci_vamp_1"
  write (u, "(A)")  "*   Purpose: integrate function in one dimension &
       &(single channel)"

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator"
  write (u, "(A)")

  allocate (mci_vamp_t :: mci)
  call mci%set_dimensions (1, 1)
  select type (mci)
  type is (mci_vamp_t)
     grid_par%use_vamp_equivalences = .false.
     call mci%set_grid_parameters (grid_par)
  end select

  allocate (rng_tao_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize instance"
  write (u, "(A)")

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  write (u, "(A)")  "* Initialize test sampler"
  write (u, "(A)")

  allocate (test_sampler_1_t :: sampler)
  call sampler%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Integrate with n_calls = 1000"
  write (u, "(A)")  "   (lower precision to avoid"
  write (u, "(A)")  "      numerical noise)"
  write (u, "(A)")

  select type (mci)
  type is (mci_vamp_t)
     call mci%add_pass ()
  end select
```

```
    call mci%integrate (mci_instance, sampler, 1, 1000, pacify = .true.)
    call mci%write (u, .true.)

    write (u, "(A)")
    write (u, "(A)")  "* Contents of mci_instance:"
    write (u, "(A)")

    call mci_instance%write (u, .true.)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp_1"

  end subroutine mci_vamp_1
```

Multiple iterations

Construct an integrator and use it for a one-dimensional sampler. Integrate with five iterations without grid adaptation.

⟨*MCI vamp: execute tests*⟩+≡

```
  call test (mci_vamp_2, "mci_vamp_2", &
       "multiple iterations", &
       u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
  public :: mci_vamp_2
```

⟨*MCI vamp: tests*⟩+≡

```
  subroutine mci_vamp_2 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng

    write (u, "(A)")  "* Test output: mci_vamp_2"
    write (u, "(A)")  "*   Purpose: integrate function in one dimension &
         &(single channel)"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator, sampler, instance"
    write (u, "(A)")

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (1, 1)
    select type (mci)
    type is (mci_vamp_t)
```

```
      grid_par%use_vamp_equivalences = .false.
      call mci%set_grid_parameters (grid_par)
   end select

   allocate (rng_tao_t :: rng)
   call rng%init ()
   call mci%import_rng (rng)

   call mci%allocate_instance (mci_instance)
   call mci_instance%init (mci)

   allocate (test_sampler_1_t :: sampler)
   select type (sampler)
   type is (test_sampler_1_t)
      sampler%mode = 2
   end select
   call sampler%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 100"
   write (u, "(A)")

   select type (mci)
   type is (mci_vamp_t)
      call mci%add_pass (adapt_grids = .false.)
   end select
   call mci%integrate (mci_instance, sampler, 3, 100)
   call mci%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Contents of mci_instance:"
   write (u, "(A)")

   call mci_instance%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call mci_instance%final ()
   call mci%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: mci_vamp_2"

end subroutine mci_vamp_2
```

Grid adaptation

Construct an integrator and use it for a one-dimensional sampler. Integrate with three iterations and in-between grid adaptations.

⟨*MCI vamp: execute tests*⟩+≡

```
   call test (mci_vamp_3, "mci_vamp_3", &
```

```
         "grid adaptation", &
         u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
    public :: mci_vamp_3
```

⟨*MCI vamp: tests*⟩+≡

```
    subroutine mci_vamp_3 (u)
      integer, intent(in) :: u
      type(grid_parameters_t) :: grid_par
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      class(rng_t), allocatable :: rng

      write (u, "(A)")  "* Test output: mci_vamp_3"
      write (u, "(A)")  "*   Purpose: integrate function in one dimension &
           &(single channel)"
      write (u, "(A)")  "*            and adapt grid"

      write (u, "(A)")
      write (u, "(A)")  "* Initialize integrator, sampler, instance"
      write (u, "(A)")

      allocate (mci_vamp_t :: mci)
      call mci%set_dimensions (1, 1)
      select type (mci)
      type is (mci_vamp_t)
         grid_par%use_vamp_equivalences = .false.
         call mci%set_grid_parameters (grid_par)
      end select

      allocate (rng_tao_t :: rng)
      call rng%init ()
      call mci%import_rng (rng)

      call mci%allocate_instance (mci_instance)
      call mci_instance%init (mci)

      allocate (test_sampler_1_t :: sampler)
      select type (sampler)
      type is (test_sampler_1_t)
         sampler%mode = 2
      end select
      call sampler%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 100"
      write (u, "(A)")

      select type (mci)
      type is (mci_vamp_t)
         call mci%add_pass (adapt_grids = .true.)
      end select
      call mci%integrate (mci_instance, sampler, 3, 100)
```

```
    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Contents of mci_instance:"
    write (u, "(A)")

    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp_3"

  end subroutine mci_vamp_3
```

Two-dimensional integral

Construct an integrator and use it for a two-dimensional sampler. Integrate with three iterations and in-between grid adaptations.

⟨*MCI vamp: execute tests*⟩+≡

```
  call test (mci_vamp_4, "mci_vamp_4", &
       "two-dimensional integration", &
       u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
  public :: mci_vamp_4
```

⟨*MCI vamp: tests*⟩+≡

```
  subroutine mci_vamp_4 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng

    write (u, "(A)")  "* Test output: mci_vamp_4"
    write (u, "(A)")  "*   Purpose: integrate function in two dimensions &
         &(single channel)"
    write (u, "(A)")  "*            and adapt grid"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator, sampler, instance"
    write (u, "(A)")

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (2, 1)
    select type (mci)
    type is (mci_vamp_t)
```

```
      grid_par%use_vamp_equivalences = .false.
      call mci%set_grid_parameters (grid_par)
   end select

   allocate (rng_tao_t :: rng)
   call rng%init ()
   call mci%import_rng (rng)

   call mci%allocate_instance (mci_instance)
   call mci_instance%init (mci)

   allocate (test_sampler_1_t :: sampler)
   select type (sampler)
   type is (test_sampler_1_t)
      sampler%mode = 3
   end select
   call sampler%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
   write (u, "(A)")

   select type (mci)
   type is (mci_vamp_t)
      call mci%add_pass (adapt_grids = .true.)
   end select
   call mci%integrate (mci_instance, sampler, 3, 1000)
   call mci%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Contents of mci_instance:"
   write (u, "(A)")

   call mci_instance%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call mci_instance%final ()
   call mci%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: mci_vamp_4"

end subroutine mci_vamp_4
```

Two-channel integral

Construct an integrator and use it for a two-dimensional sampler with two channels.

Integrate with three iterations and in-between grid adaptations.

⟨*MCI vamp: execute tests*⟩+≡

```
    call test (mci_vamp_5, "mci_vamp_5", &
         "two-dimensional integration", &
         u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
    public :: mci_vamp_5
```

⟨*MCI vamp: tests*⟩+≡

```
    subroutine mci_vamp_5 (u)
      integer, intent(in) :: u
      type(grid_parameters_t) :: grid_par
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      class(rng_t), allocatable :: rng

      write (u, "(A)")  "* Test output: mci_vamp_5"
      write (u, "(A)")  "*   Purpose: integrate function in two dimensions &
           &(two channels)"
      write (u, "(A)")  "*            and adapt grid"

      write (u, "(A)")
      write (u, "(A)")  "* Initialize integrator, sampler, instance"
      write (u, "(A)")

      allocate (mci_vamp_t :: mci)
      call mci%set_dimensions (2, 2)
      select type (mci)
      type is (mci_vamp_t)
         grid_par%stratified = .false.
         grid_par%use_vamp_equivalences = .false.
         call mci%set_grid_parameters (grid_par)
      end select

      allocate (rng_tao_t :: rng)
      call rng%init ()
      call mci%import_rng (rng)

      call mci%allocate_instance (mci_instance)
      call mci_instance%init (mci)

      allocate (test_sampler_2_t :: sampler)
      call sampler%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
      write (u, "(A)")

      select type (mci)
      type is (mci_vamp_t)
         call mci%add_pass (adapt_grids = .true.)
      end select
      call mci%integrate (mci_instance, sampler, 3, 1000)
      call mci%write (u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Contents of mci_instance:"
    write (u, "(A)")

    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp_5"

  end subroutine mci_vamp_5
```

Weight adaptation

Construct an integrator and use it for a one-dimensional sampler with two channels.

Integrate with three iterations and in-between weight adaptations.

⟨*MCI vamp: execute tests*⟩+≡

```
  call test (mci_vamp_6, "mci_vamp_6", &
       "weight adaptation", &
       u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
  public :: mci_vamp_6
```

⟨*MCI vamp: tests*⟩+≡

```
  subroutine mci_vamp_6 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng

    write (u, "(A)")  "* Test output: mci_vamp_6"
    write (u, "(A)")  "*   Purpose: integrate function in one dimension &
         &(two channels)"
    write (u, "(A)")  "*            and adapt weights"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator, sampler, instance"
    write (u, "(A)")

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (1, 2)
    select type (mci)
    type is (mci_vamp_t)
       grid_par%stratified = .false.
```

```
   grid_par%use_vamp_equivalences = .false.
   call mci%set_grid_parameters (grid_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_3_t :: sampler)
select type (sampler)
type is (test_sampler_3_t)
   sampler%a = 0.9_default
   sampler%b = 0.1_default
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_weights = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
call mci%final ()
deallocate (mci_instance)
deallocate (mci)

write (u, "(A)")
write (u, "(A)")  "* Re-initialize with chained channels"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (1, 2)
call mci%declare_chains ([1,1])
select type (mci)
type is (mci_vamp_t)
   grid_par%stratified = .false.
```

```
      grid_par%use_vamp_equivalences = .false.
      call mci%set_grid_parameters (grid_par)
   end select

   allocate (rng_tao_t :: rng)
   call rng%init ()
   call mci%import_rng (rng)

   call mci%allocate_instance (mci_instance)
   call mci_instance%init (mci)

   write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
   write (u, "(A)")

   select type (mci)
   type is (mci_vamp_t)
      call mci%add_pass (adapt_weights = .true.)
   end select
   call mci%integrate (mci_instance, sampler, 3, 1000)
   call mci%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Contents of mci_instance:"
   write (u, "(A)")

   call mci_instance%write (u)

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call mci_instance%final ()
   call mci%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: mci_vamp_6"

end subroutine mci_vamp_6
```

Equivalences

Construct an integrator and use it for a one-dimensional sampler with two channels.

Integrate with three iterations and in-between grid adaptations. Apply an equivalence between the two channels, so the binning of the two channels is forced to coincide. Compare this with the behavior without equivalences.

⟨*MCI vamp: execute tests*⟩+≡

```
call test (mci_vamp_7, "mci_vamp_7", &
     "use channel equivalences", &
     u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
public :: mci_vamp_7
```

⟨*MCI vamp: tests*⟩+≡

```
subroutine mci_vamp_7 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  type(phs_channel_t), dimension(:), allocatable :: channel
  class(rng_t), allocatable :: rng
  real(default), dimension(:,:), allocatable :: x
  integer :: u_grid, iostat, i, div, ch
  character(16) :: buffer

  write (u, "(A)")  "* Test output: mci_vamp_7"
  write (u, "(A)")  "*   Purpose: check effect of channel equivalences"

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator, sampler, instance"
  write (u, "(A)")

  allocate (mci_vamp_t :: mci)
  call mci%set_dimensions (1, 2)
  select type (mci)
  type is (mci_vamp_t)
     grid_par%stratified = .false.
     grid_par%use_vamp_equivalences = .false.
     call mci%set_grid_parameters (grid_par)
  end select

  allocate (rng_tao_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  allocate (test_sampler_3_t :: sampler)
  select type (sampler)
  type is (test_sampler_3_t)
     sampler%a = 0.7_default
     sampler%b = 0.3_default
  end select
  call sampler%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Integrate with n_it = 2 and n_calls = 1000, &
       &adapt grids"
  write (u, "(A)")

  select type (mci)
  type is (mci_vamp_t)
     call mci%add_pass (adapt_grids = .true.)
  end select
  call mci%integrate (mci_instance, sampler, 2, 1000)
```

```
call mci%write (u)

write (u, "(A)")
write (u, "(A)") "* Write grids and extract binning"
write (u, "(A)")

u_grid = free_unit ()
open (u_grid, status = "scratch", action = "readwrite")
select type (mci_instance)
type is (mci_vamp_instance_t)
   call vamp_write_grids (mci_instance%grids, u_grid)
end select
rewind (u_grid)
allocate (x (0:20, 2))
do div = 1, 2
   FIND_BINS1: do
      read (u_grid, "(A)")  buffer
      if (trim (adjustl (buffer)) == "begin d%x") then
         do
            read (u_grid, *, iostat = iostat)  i, x(i,div)
            if (iostat /= 0)  exit FIND_BINS1
         end do
      end if
   end do FIND_BINS1
end do
close (u_grid)

write (u, "(1x,A,L1)")  "Equal binning in both channels = ", &
     all (x(:,1) == x(:,2))
deallocate (x)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
call mci%final ()
deallocate (mci_instance)
deallocate (mci)

write (u, "(A)")
write (u, "(A)")  "* Re-initialize integrator, instance"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (1, 2)
select type (mci)
type is (mci_vamp_t)
   grid_par%stratified = .false.
   grid_par%use_vamp_equivalences = .true.
   call mci%set_grid_parameters (grid_par)
end select

write (u, "(A)")  "* Define equivalences"
```

```
write (u, "(A)")

allocate (channel (2))
do ch = 1, 2
   allocate (channel(ch)%eq (2))
   do i = 1, 2
      associate (eq => channel(ch)%eq(i))
        call eq%init (1)
        eq%c = i
        eq%perm = [1]
        eq%mode = [0]
      end associate
   end do
   write (u, "(1x,I0,':')", advance = "no")  ch
   call channel(ch)%write (u)
end do
call mci%declare_equivalences (channel, dim_offset = 0)

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 2 and n_calls = 1000, &
     &adapt grids"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_grids = .true.)
end select
call mci%integrate (mci_instance, sampler, 2, 1000)

call mci%write (u)

write (u, "(A)")
write (u, "(A)") "* Write grids and extract binning"
write (u, "(A)")

u_grid = free_unit ()
open (u_grid, status = "scratch", action = "readwrite")
select type (mci_instance)
type is (mci_vamp_instance_t)
   call vamp_write_grids (mci_instance%grids, u_grid)
end select
rewind (u_grid)
allocate (x (0:20, 2))
do div = 1, 2
   FIND_BINS2: do
      read (u_grid, "(A)")  buffer
      if (trim (adjustl (buffer)) == "begin d%x") then
```

```
            do
               read (u_grid, *, iostat = iostat)  i, x(i,div)
               if (iostat /= 0)  exit FIND_BINS2
            end do
         end if
      end do FIND_BINS2
   end do
   close (u_grid)

   write (u, "(1x,A,L1)")  "Equal binning in both channels = ", &
        all (x(:,1) == x(:,2))
   deallocate (x)


   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call mci_instance%final ()
   call mci%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: mci_vamp_7"

end subroutine mci_vamp_7
```

Multiple passes

Integrate with three passes and different settings for weight and grid adaptation.

⟨*MCI vamp: execute tests*⟩+≡

```
call test (mci_vamp_8, "mci_vamp_8", &
     "integration passes", &
     u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
public :: mci_vamp_8
```

⟨*MCI vamp: tests*⟩+≡

```
subroutine mci_vamp_8 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng

  write (u, "(A)")  "* Test output: mci_vamp_8"
  write (u, "(A)")  "*   Purpose: integrate function in one dimension &
       &(two channels)"
  write (u, "(A)")  "*            in three passes"

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator, sampler, instance"
  write (u, "(A)")
```

```
allocate (mci_vamp_t :: mci)
call mci%set_dimensions (1, 2)
select type (mci)
type is (mci_vamp_t)
   grid_par%stratified = .false.
   grid_par%use_vamp_equivalences = .false.
   call mci%set_grid_parameters (grid_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_3_t :: sampler)
select type (sampler)
type is (test_sampler_3_t)
   sampler%a = 0.9_default
   sampler%b = 0.1_default
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with grid and weight adaptation"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_grids = .true., adapt_weights = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with grid adaptation"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_grids = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
```

```
    write (u, "(A)")

    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Integrate without adaptation"
    write (u, "(A)")

    select type (mci)
    type is (mci_vamp_t)
       call mci%add_pass ()
    end select
    call mci%integrate (mci_instance, sampler, 3, 1000)
    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Contents of mci_instance:"
    write (u, "(A)")

    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp_8"

  end subroutine mci_vamp_8
```

Weighted events

Construct an integrator and use it for a two-dimensional sampler with two channels. Integrate and generate a weighted event.

⟨*MCI vamp: execute tests*⟩+≡

```
  call test (mci_vamp_9, "mci_vamp_9", &
       "weighted event", &
       u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
  public :: mci_vamp_9
```

⟨*MCI vamp: tests*⟩+≡

```
  subroutine mci_vamp_9 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng
```

```
write (u, "(A)")  "* Test output: mci_vamp_9"
write (u, "(A)")  "*   Purpose: integrate function in two dimensions &
     &(two channels)"
write (u, "(A)")  "*            and generate a weighted event"

write (u, "(A)")
write (u, "(A)")  "* Initialize integrator, sampler, instance"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (2, 2)
select type (mci)
type is (mci_vamp_t)
   grid_par%stratified = .false.
   grid_par%use_vamp_equivalences = .false.
   call mci%set_grid_parameters (grid_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_2_t :: sampler)
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
write (u, "(A)")

call mci%add_pass ()
call mci%integrate (mci_instance, sampler, 1, 1000)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate a weighted event"
write (u, "(A)")

call mci_instance%init_simulation ()
call mci%generate_weighted_event (mci_instance, sampler)

write (u, "(1x,A)")  "MCI instance:"
call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final_simulation ()
call mci_instance%final ()
call mci%final ()

write (u, "(A)")
```

```
    write (u, "(A)")  "* Test output end: mci_vamp_9"

  end subroutine mci_vamp_9
```

Grids I/O

Construct an integrator and allocate grids. Write grids to file, read them in again and compare.

⟨*MCI vamp: execute tests*⟩+≡

```
  call test (mci_vamp_10, "mci_vamp_10", &
       "grids I/O", &
       u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
  public :: mci_vamp_10
```

⟨*MCI vamp: tests*⟩+≡

```
  subroutine mci_vamp_10 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng
    type(string_t) :: file1, file2
    character(80) :: buffer1, buffer2
    integer :: u1, u2, iostat1, iostat2
    logical :: equal, success

    write (u, "(A)")  "* Test output: mci_vamp_10"
    write (u, "(A)")  "*   Purpose: write and read VAMP grids"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator, sampler, instance"
    write (u, "(A)")

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (2, 2)
    select type (mci)
    type is (mci_vamp_t)
       grid_par%stratified = .false.
       grid_par%use_vamp_equivalences = .false.
       call mci%set_grid_parameters (grid_par)
    end select

    allocate (rng_tao_t :: rng)
    call rng%init ()
    call mci%import_rng (rng)

    mci%md5sum = "1234567890abcdef1234567890abcdef"

    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)
```

```
allocate (test_sampler_2_t :: sampler)
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
write (u, "(A)")

call mci%add_pass ()
call mci%integrate (mci_instance, sampler, 1, 1000)

write (u, "(A)")  "* Write grids to file"
write (u, "(A)")

file1 = "mci_vamp_10.1"
select type (mci)
type is (mci_vamp_t)
   call mci%set_grid_filename (file1)
   call mci%write_grids (mci_instance)
end select

call mci_instance%final ()
call mci%final ()
deallocate (mci)

write (u, "(A)")  "* Read grids from file"
write (u, "(A)")

allocate (mci_vamp_t :: mci)
call mci%set_dimensions (2, 2)
select type (mci)
type is (mci_vamp_t)
   call mci%set_grid_parameters (grid_par)
end select

allocate (rng_tao_t :: rng)
call rng%init ()
call mci%import_rng (rng)

mci%md5sum = "1234567890abcdef1234567890abcdef"

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

select type (mci)
type is (mci_vamp_t)
   call mci%set_grid_filename (file1)
   call mci%add_pass ()
   call mci%current_pass%configure (1, 1000, &
        mci%min_calls, &
        mci%grid_par%min_bins, mci%grid_par%max_bins, &
        mci%grid_par%min_calls_per_channel * mci%n_channel)
   call mci%read_grids_header (success)
   call mci%compute_md5sum ()
```

```
      call mci%read_grids_data (mci_instance, read_integrals = .true.)
   end select
   write (u, "(1x,A,L1)")  "success = ", success

   write (u, "(A)")
   write (u, "(A)")  "* Write grids again"
   write (u, "(A)")

   file2 = "mci_vamp_10.2"
   select type (mci)
   type is (mci_vamp_t)
      call mci%set_grid_filename (file2)
      call mci%write_grids (mci_instance)
   end select

   u1 = free_unit ()
   open (u1, file = char (file1) // ".vg", action = "read", status = "old")
   u2 = free_unit ()
   open (u2, file = char (file2) // ".vg", action = "read", status = "old")

   equal = .true.
   iostat1 = 0
   iostat2 = 0
   do while (equal .and. iostat1 == 0 .and. iostat2 == 0)
      read (u1, "(A)", iostat = iostat1)  buffer1
      read (u2, "(A)", iostat = iostat2)  buffer2
      equal = buffer1 == buffer2 .and. iostat1 == iostat2
   end do
   close (u1)
   close (u2)

   if (equal) then
      write (u, "(1x,A)")  "Success: grid files are identical"
   else
      write (u, "(1x,A)")  "Failure: grid files differ"
   end if

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call mci_instance%final ()
   call mci%final ()

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: mci_vamp_10"

end subroutine mci_vamp_10
```

Weighted events with grid I/O

Construct an integrator and use it for a two-dimensional sampler with two channels. Integrate, write grids, and generate a weighted event using the grids

from file.

⟨*MCI vamp: execute tests*⟩+≡

```
call test (mci_vamp_11, "mci_vamp_11", &
     "weighted events with grid I/O", &
     u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
public :: mci_vamp_11
```

⟨*MCI vamp: tests*⟩+≡

```
subroutine mci_vamp_11 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng

  write (u, "(A)")  "* Test output: mci_vamp_11"
  write (u, "(A)")  "*   Purpose: integrate function in two dimensions &
       &(two channels)"
  write (u, "(A)")  "*            and generate a weighted event"

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator, sampler, instance"
  write (u, "(A)")

  allocate (mci_vamp_t :: mci)
  call mci%set_dimensions (2, 2)
  select type (mci)
  type is (mci_vamp_t)
     grid_par%stratified = .false.
     grid_par%use_vamp_equivalences = .false.
     call mci%set_grid_parameters (grid_par)
     call mci%set_grid_filename (var_str ("mci_vamp_11"))
  end select

  allocate (rng_tao_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  allocate (test_sampler_2_t :: sampler)

  write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
  write (u, "(A)")

  call mci%add_pass ()
  call mci%integrate (mci_instance, sampler, 1, 1000)

  write (u, "(A)")  "* Reset instance"
  write (u, "(A)")
```

```
    call mci_instance%final ()
    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    write (u, "(A)")  "* Generate a weighted event"
    write (u, "(A)")

    call mci_instance%init_simulation ()
    call mci%generate_weighted_event (mci_instance, sampler)

    write (u, "(A)")  "* Cleanup"

    call mci_instance%final_simulation ()
    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp_11"

  end subroutine mci_vamp_11
```

Unweighted events with grid I/O

Construct an integrator and use it for a two-dimensional sampler with two channels.

⟨*MCI vamp: execute tests*⟩+≡

```
  call test (mci_vamp_12, "mci_vamp_12", &
       "unweighted events with grid I/O", &
       u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
  public :: mci_vamp_12
```

⟨*MCI vamp: tests*⟩+≡

```
  subroutine mci_vamp_12 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng

    write (u, "(A)")  "* Test output: mci_vamp_12"
    write (u, "(A)")  "*   Purpose: integrate function in two dimensions &
         &(two channels)"
    write (u, "(A)")  "*            and generate an unweighted event"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator, sampler, instance"
    write (u, "(A)")

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (2, 2)
```

```
  select type (mci)
  type is (mci_vamp_t)
     grid_par%stratified = .false.
     grid_par%use_vamp_equivalences = .false.
     call mci%set_grid_parameters (grid_par)
     call mci%set_grid_filename (var_str ("mci_vamp_12"))
  end select

  allocate (rng_tao_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  allocate (test_sampler_2_t :: sampler)

  write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 1000"
  write (u, "(A)")

  call mci%add_pass ()
  call mci%integrate (mci_instance, sampler, 1, 1000)

  write (u, "(A)")  "* Reset instance"
  write (u, "(A)")

  call mci_instance%final ()
  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  write (u, "(A)")  "* Generate an unweighted event"
  write (u, "(A)")

  call mci_instance%init_simulation ()
  call mci%generate_unweighted_event (mci_instance, sampler)

  write (u, "(1x,A)")  "MCI instance:"
  call mci_instance%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call mci_instance%final_simulation ()
  call mci_instance%final ()
  call mci%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: mci_vamp_12"

end subroutine mci_vamp_12
```

Update integration results

Compare two `mci` objects; match the two and update the first if successful.

⟨*MCI vamp: execute tests*⟩+≡

```
call test (mci_vamp_13, "mci_vamp_13", &
     "updating integration results", &
     u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
public :: mci_vamp_13
```

⟨*MCI vamp: tests*⟩+≡

```
subroutine mci_vamp_13 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  class(mci_t), allocatable, target :: mci, mci_ref
  logical :: success

  write (u, "(A)")  "* Test output: mci_vamp_13"
  write (u, "(A)")  "*   Purpose: match and update integrators"

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator with no passes"
  write (u, "(A)")

  allocate (mci_vamp_t :: mci)
  call mci%set_dimensions (2, 2)
  select type (mci)
  type is (mci_vamp_t)
     grid_par%stratified = .false.
     grid_par%use_vamp_equivalences = .false.
     call mci%set_grid_parameters (grid_par)
  end select
  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize reference"
  write (u, "(A)")

  allocate (mci_vamp_t :: mci_ref)
  call mci_ref%set_dimensions (2, 2)
  select type (mci_ref)
  type is (mci_vamp_t)
     call mci_ref%set_grid_parameters (grid_par)
  end select

  select type (mci_ref)
  type is (mci_vamp_t)
     call mci_ref%add_pass (adapt_grids = .true.)
     call mci_ref%current_pass%configure (2, 1000, 0, 1, 5, 0)
     mci_ref%current_pass%calls = [77, 77]
     mci_ref%current_pass%integral = [1.23_default, 3.45_default]
     mci_ref%current_pass%error = [0.23_default, 0.45_default]
     mci_ref%current_pass%efficiency = [0.1_default, 0.6_default]
     mci_ref%current_pass%integral_defined = .true.
```

```
   call mci_ref%add_pass ()
   call mci_ref%current_pass%configure (2, 2000, 0, 1, 7, 0)
   mci_ref%current_pass%calls = [99, 0]
   mci_ref%current_pass%integral = [7.89_default, 0._default]
   mci_ref%current_pass%error = [0.89_default, 0._default]
   mci_ref%current_pass%efficiency = [0.86_default, 0._default]
   mci_ref%current_pass%integral_defined = .true.
end select

call mci_ref%write (u)

write (u, "(A)")
write (u, "(A)")  "* Update integrator (no-op, should succeed)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Add pass to integrator"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_grids = .true.)
   call mci%current_pass%configure (2, 1000, 0, 1, 5, 0)
   mci%current_pass%calls = [77, 77]
   mci%current_pass%integral = [1.23_default, 3.45_default]
   mci%current_pass%error = [0.23_default, 0.45_default]
   mci%current_pass%efficiency = [0.1_default, 0.6_default]
   mci%current_pass%integral_defined = .true.
end select

write (u, "(A)")  "* Update integrator (no-op, should succeed)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Add pass to integrator, wrong parameters"
```

```
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass ()
   call mci%current_pass%configure (2, 1000, 0, 1, 7, 0)
end select

write (u, "(A)")  "* Update integrator (should fail)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Reset and add passes to integrator"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%reset ()
   call mci%add_pass (adapt_grids = .true.)
   call mci%current_pass%configure (2, 1000, 0, 1, 5, 0)
   mci%current_pass%calls = [77, 77]
   mci%current_pass%integral = [1.23_default, 3.45_default]
   mci%current_pass%error = [0.23_default, 0.45_default]
   mci%current_pass%efficiency = [0.1_default, 0.6_default]
   mci%current_pass%integral_defined = .true.

   call mci%add_pass ()
   call mci%current_pass%configure (2, 2000, 0, 1, 7, 0)
end select

write (u, "(A)")  "* Update integrator (should succeed)"
write (u, "(A)")

select type (mci)
type is (mci_vamp_t)
   call mci%update_from_ref (mci_ref, success)
end select

write (u, "(1x,A,L1)")  "success = ", success
write (u, "(A)")
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Update again (no-op, should succeed)"
write (u, "(A)")
```

```
    select type (mci)
    type is (mci_vamp_t)
       call mci%update_from_ref (mci_ref, success)
    end select

    write (u, "(1x,A,L1)")  "success = ", success
    write (u, "(A)")
    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Add extra result to integrator"
    write (u, "(A)")

    select type (mci)
    type is (mci_vamp_t)
       mci%current_pass%calls(2) = 1234
    end select

    write (u, "(A)")  "* Update integrator (should fail)"
    write (u, "(A)")

    select type (mci)
    type is (mci_vamp_t)
       call mci%update_from_ref (mci_ref, success)
    end select

    write (u, "(1x,A,L1)")  "success = ", success
    write (u, "(A)")
    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call mci%final ()
    call mci_ref%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp_13"

  end subroutine mci_vamp_13
```

Accuracy Goal

Integrate with multiple iterations. Skip iterations once an accuracy goal has been reached.

⟨*MCI vamp: execute tests*⟩+≡

```
  call test (mci_vamp_14, "mci_vamp_14", &
       "accuracy goal", &
       u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
  public :: mci_vamp_14
```

⟨*MCI vamp: tests*⟩+≡

```
subroutine mci_vamp_14 (u)
  integer, intent(in) :: u
  type(grid_parameters_t) :: grid_par
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng

  write (u, "(A)")  "* Test output: mci_vamp_14"
  write (u, "(A)")  "*   Purpose: integrate function in one dimension &
       &(single channel)"
  write (u, "(A)")  "*            and check accuracy goal"

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator, sampler, instance"
  write (u, "(A)")

  allocate (mci_vamp_t :: mci)
  call mci%set_dimensions (1, 1)
  select type (mci)
  type is (mci_vamp_t)
     grid_par%use_vamp_equivalences = .false.
     grid_par%accuracy_goal = 5E-2_default
     call mci%set_grid_parameters (grid_par)
  end select

  allocate (rng_tao_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  allocate (test_sampler_1_t :: sampler)
  select type (sampler)
  type is (test_sampler_1_t)
     sampler%mode = 2
  end select
  call sampler%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Integrate with n_it = 5 and n_calls = 100"
  write (u, "(A)")

  select type (mci)
  type is (mci_vamp_t)
     call mci%add_pass (adapt_grids = .true.)
  end select
  call mci%integrate (mci_instance, sampler, 5, 100)
  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"
```

```
    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp_14"

  end subroutine mci_vamp_14
```

VAMP history

Integrate with three passes and different settings for weight and grid adaptation. Then show the VAMP history.

⟨*MCI vamp: execute tests*⟩+≡

```
  call test (mci_vamp_15, "mci_vamp_15", &
       "VAMP history", &
       u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
  public :: mci_vamp_15
```

⟨*MCI vamp: tests*⟩+≡

```
  subroutine mci_vamp_15 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    type(history_parameters_t) :: history_par
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng

    write (u, "(A)")  "* Test output: mci_vamp_15"
    write (u, "(A)")  "*   Purpose: integrate function in one dimension &
         &(two channels)"
    write (u, "(A)")  "*            in three passes, show history"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator, sampler, instance"
    write (u, "(A)")

    history_par%channel = .true.

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (1, 2)
    select type (mci)
    type is (mci_vamp_t)
       grid_par%stratified = .false.
       grid_par%use_vamp_equivalences = .false.
       call mci%set_grid_parameters (grid_par)
       call mci%set_history_parameters (history_par)
    end select

    allocate (rng_tao_t :: rng)
```

```
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_3_t :: sampler)
select type (sampler)
type is (test_sampler_3_t)
   sampler%a = 0.9_default
   sampler%b = 0.1_default
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Pass 1: grid and weight adaptation"

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_grids = .true., adapt_weights = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)

write (u, "(A)")
write (u, "(A)")  "* Pass 2: grid adaptation"

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass (adapt_grids = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 1000)

write (u, "(A)")
write (u, "(A)")  "* Pass 3: without adaptation"

select type (mci)
type is (mci_vamp_t)
   call mci%add_pass ()
end select
call mci%integrate (mci_instance, sampler, 3, 1000)

write (u, "(A)")
write (u, "(A)")  "* Contents of MCI record, with history"
write (u, "(A)")

call mci%write (u)
select type (mci)
type is (mci_vamp_t)
   call mci%write_history (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Cleanup"
```

```
    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp_15"

  end subroutine mci_vamp_15
```

One-dimensional integration with sign change

Construct an integrator and use it for a one-dimensional sampler.

⟨*MCI vamp: execute tests*⟩+≡

```
  call test (mci_vamp_16, "mci_vamp_16", &
       "1-D integral with sign change", &
       u, results)
```

⟨*MCI vamp: test declarations*⟩+≡

```
  public :: mci_vamp_16
```

⟨*MCI vamp: tests*⟩+≡

```
  subroutine mci_vamp_16 (u)
    integer, intent(in) :: u
    type(grid_parameters_t) :: grid_par
    class(mci_t), allocatable, target :: mci
    class(mci_instance_t), pointer :: mci_instance => null ()
    class(mci_sampler_t), allocatable :: sampler
    class(rng_t), allocatable :: rng

    write (u, "(A)")  "* Test output: mci_vamp_16"
    write (u, "(A)")  "*   Purpose: integrate function in one dimension &
         &(single channel)"

    write (u, "(A)")
    write (u, "(A)")  "* Initialize integrator"
    write (u, "(A)")

    allocate (mci_vamp_t :: mci)
    call mci%set_dimensions (1, 1)
    select type (mci)
    type is (mci_vamp_t)
       grid_par%use_vamp_equivalences = .false.
       call mci%set_grid_parameters (grid_par)
       mci%negative_weights = .true.
    end select

    allocate (rng_tao_t :: rng)
    call rng%init ()
    call mci%import_rng (rng)

    call mci%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize instance"
    write (u, "(A)")
```

```
    call mci%allocate_instance (mci_instance)
    call mci_instance%init (mci)

    write (u, "(A)")  "* Initialize test sampler"
    write (u, "(A)")

    allocate (test_sampler_1_t :: sampler)
    select type (sampler)
    type is (test_sampler_1_t)
       sampler%mode = 4
    end select
    call sampler%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Integrate with n_calls = 1000"
    write (u, "(A)")  "   (lower precision to avoid"
    write (u, "(A)")  "      numerical noise)"
    write (u, "(A)")

    select type (mci)
    type is (mci_vamp_t)
       call mci%add_pass ()
    end select
    call mci%integrate (mci_instance, sampler, 1, 1000, pacify = .true.)
    call mci%write (u, .true.)

    write (u, "(A)")
    write (u, "(A)")  "* Contents of mci_instance:"
    write (u, "(A)")

    call mci_instance%write (u, .true.)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp_16"

  end subroutine mci_vamp_16
```

20.6 Multi-channel integration with VAMP2

The multi-channel integration uses VEGAS as backbone integrator. The base interface for the multi-channel integration is given by `mci_base` module.

We interface the VAMP2 interface given by `vamp2` module.

⟨`mci_vamp2.f90`⟩≡
  ⟨*File header*⟩

```
module mci_vamp2

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_utils, only: pac_fmt
  use format_utils, only: write_separator, write_indent
  use format_defs, only: FMT_12, FMT_14, FMT_17, FMT_19
  use diagnostics
  use md5
  use phs_base
  use rng_base
  use rng_tao

  use mci_base

  use vegas, only: VEGAS_MODE_IMPORTANCE, VEGAS_MODE_IMPORTANCE_ONLY
  use vamp2

⟨Standard module head⟩

⟨MCI vamp2: public⟩

⟨MCI vamp2: parameters⟩

⟨MCI vamp2: types⟩

⟨MCI vamp2: interfaces⟩

contains

⟨MCI vamp2: procedures⟩

end module mci_vamp2
```

20.6.1 Type: mci_vamp2_func_t

⟨*MCI vamp2: types*⟩≡

```
type, extends (vamp2_func_t) :: mci_vamp2_func_t
   private
   real(default) :: integrand = 0.
   class(mci_sampler_t), pointer :: sampler => null ()
   class(mci_vamp2_instance_t), pointer :: instance => null ()
 contains
 ⟨MCI vamp2: mci vamp2 func: TBP⟩
end type mci_vamp2_func_t
```

Set instance and sampler aka workspace. Also, reset number of `n_calls`.

⟨*MCI vamp2: mci vamp2 func: TBP*⟩≡

```
procedure, public :: set_workspace => mci_vamp2_func_set_workspace
```

⟨*MCI vamp2: procedures*⟩≡

```
subroutine mci_vamp2_func_set_workspace (self, instance, sampler)
  class(mci_vamp2_func_t), intent(inout) :: self
  class(mci_vamp2_instance_t), intent(inout), target :: instance
  class(mci_sampler_t), intent(inout), target :: sampler
  self%instance => instance
  self%sampler => sampler
end subroutine mci_vamp2_func_set_workspace
```

Get the different channel probabilities.

⟨*MCI vamp2: mci vamp2 func: TBP*⟩+≡

```
procedure, public :: get_probabilities => mci_vamp2_func_get_probabilities
```

⟨*MCI vamp2: procedures*⟩+≡

```
function mci_vamp2_func_get_probabilities (self) result (gi)
  class(mci_vamp2_func_t), intent(inout) :: self
  real(default), dimension(self%n_channel) :: gi
  gi = self%gi
end function mci_vamp2_func_get_probabilities
```

Get multi-channel weight.

⟨*MCI vamp2: mci vamp2 func: TBP*⟩+≡

```
procedure, public :: get_weight => mci_vamp2_func_get_weight
```

⟨*MCI vamp2: procedures*⟩+≡

```
real(default) function mci_vamp2_func_get_weight (self) result (g)
  class(mci_vamp2_func_t), intent(in) :: self
  g = self%g
end function mci_vamp2_func_get_weight
```

Set integrand.

⟨*MCI vamp2: mci vamp2 func: TBP*⟩+≡

```
procedure, public :: set_integrand => mci_vamp2_func_set_integrand
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_func_set_integrand (self, integrand)
  class(mci_vamp2_func_t), intent(inout) :: self
  real(default), intent(in) :: integrand
  self%integrand = integrand
end subroutine mci_vamp2_func_set_integrand
```

Evaluate the mappings.

⟨*MCI vamp2: mci vamp2 func: TBP*⟩+≡

```
procedure, public :: evaluate_maps => mci_vamp2_func_evaluate_maps
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_func_evaluate_maps (self, x)
  class(mci_vamp2_func_t), intent(inout) :: self
  real(default), dimension(:), intent(in) :: x
  select type (self)
  type is (mci_vamp2_func_t)
     call self%instance%evaluate (self%sampler, self%current_channel, x)
  end select
  self%valid_x = self%instance%valid
```

```
    self%xi = self%instance%x
    self%det = self%instance%f
  end subroutine mci_vamp2_func_evaluate_maps
```

Evaluate the function, more or less.

⟨*MCI vamp2: mci vamp2 func: TBP*⟩+≡

```
  procedure, public :: evaluate_func => mci_vamp2_func_evaluate_func
```

⟨*MCI vamp2: procedures*⟩+≡

```
  real(default) function mci_vamp2_func_evaluate_func (self, x) result (f)
    class(mci_vamp2_func_t), intent(in) :: self
    real(default), dimension(:), intent(in) :: x
    f = self%integrand
  end function mci_vamp2_func_evaluate_func
```

20.6.2 Type: mci_vamp2_config_t

We extend `vamp2_config_t`.

⟨*MCI vamp2: public*⟩≡

```
  public :: mci_vamp2_config_t
```

⟨*MCI vamp2: types*⟩+≡

```
  type, extends (vamp2_config_t) :: mci_vamp2_config_t
     logical :: use_channel_equivalences = .false.
   contains
   ⟨MCI vamp2: mci vamp2 config: TBP⟩
  end type mci_vamp2_config_t
```

20.6.3 Integration pass

The list of passes is organized in a separate container. We store the parameters and results for each integration pass in `pass_t` and the linked list is stored in `list_pass_t`.

⟨*MCI vamp2: types*⟩+≡

```
  type :: list_pass_t
     type(pass_t), pointer :: first => null ()
     type(pass_t), pointer :: current => null ()
   contains
   ⟨MCI vamp2: list pass: TBP⟩
  end type list_pass_t
```

Finalizer. Deallocate each element of the list beginning by the first.

⟨*MCI vamp2: list pass: TBP*⟩≡

```
  procedure :: final => list_pass_final
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine list_pass_final (self)
    class(list_pass_t), intent(inout) :: self
    type(pass_t), pointer :: current
    current => self%first
```

```
    do while (associated (current))
       self%first => current%next
       deallocate (current)
       current => self%first
    end do
  end subroutine list_pass_final
```

Add a new pass.

⟨*MCI vamp2: list pass: TBP*⟩+≡

```
  procedure :: add => list_pass_add
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine list_pass_add (self, adapt_grids, adapt_weights, final_pass)
    class(list_pass_t), intent(inout) :: self
    logical, intent(in), optional :: adapt_grids, adapt_weights, final_pass
    type(pass_t), pointer :: new_pass
    allocate (new_pass)
    new_pass%i_pass = 1
    new_pass%i_first_it = 1
    new_pass%adapt_grids = .false.; if (present (adapt_grids)) &
         & new_pass%adapt_grids = adapt_grids
    new_pass%adapt_weights = .false.; if (present (adapt_weights)) &
         & new_pass%adapt_weights = adapt_weights
    new_pass%is_final_pass = .false.; if (present (final_pass)) &
         & new_pass%is_final_pass = final_pass
    if (.not. associated (self%first)) then
       self%first => new_pass
    else
       new_pass%i_pass = new_pass%i_pass + self%current%i_pass
       new_pass%i_first_it = self%current%i_first_it + self%current%n_it
       self%current%next => new_pass
    end if
    self%current => new_pass
  end subroutine list_pass_add
```

Update list from a reference. All passes except for the last one must match exactly. For the last one, integration results are updated. The reference output may contain extra passes, these are ignored.

⟨*MCI vamp2: list pass: TBP*⟩+≡

```
  procedure :: update_from_ref => list_pass_update_from_ref
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine list_pass_update_from_ref (self, ref, success)
    class(list_pass_t), intent(inout) :: self
    type(list_pass_t), intent(in) :: ref
    logical, intent(out) :: success
    type(pass_t), pointer :: current, ref_current
    current => self%first
    ref_current => ref%first
    success = .true.
    do while (success .and. associated (current))
       if (associated (ref_current)) then
          if (associated (current%next)) then
             success = current .matches. ref_current
```

```
         else
            call current%update (ref_current, success)
         end if
         current => current%next
         ref_current => ref_current%next
      else
         success = .false.
      end if
   end do
end subroutine list_pass_update_from_ref
```

Output. Write the complete linked list to the specified unit.

⟨*MCI vamp2: list pass: TBP*⟩+≡

```
procedure :: write => list_pass_write
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine list_pass_write (self, unit, pacify)
  class(list_pass_t), intent(in) :: self
  integer, intent(in) :: unit
  logical, intent(in), optional :: pacify
  type(pass_t), pointer :: current
  current => self%first
  do while (associated (current))
     write (unit, descr_fmt) "Integration pass:"
     call current%write (unit, pacify)
     current => current%next
  end do
end subroutine list_pass_write
```

The parameters and results are stored in the nodes `pass_t` of the linked list.

⟨*MCI vamp2: types*⟩+≡

```
type :: pass_t
   integer :: i_pass = 0
   integer :: i_first_it = 0
   integer :: n_it = 0
   integer :: n_calls = 0
   logical :: adapt_grids = .false.
   logical :: adapt_weights = .false.
   logical :: is_final_pass = .false.
   logical :: integral_defined = .false.
   integer, dimension(:), allocatable :: calls
   real(default), dimension(:), allocatable :: integral
   real(default), dimension(:), allocatable :: error
   real(default), dimension(:), allocatable :: efficiency
   type(pass_t), pointer :: next => null ()
 contains
 ⟨MCI vamp2: pass: TBP⟩
end type pass_t
```

Output. Note that the precision of the numerical values should match the precision for comparing output from file with data.

⟨*MCI vamp2: pass: TBP*⟩≡

```
procedure :: write => pass_write
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine pass_write (self, unit, pacify)
  class(pass_t), intent(in) :: self
  integer, intent(in) :: unit
  logical, intent(in), optional :: pacify
  integer :: u, i
  character(len=7) :: fmt
  call pac_fmt (fmt, FMT_17, FMT_14, pacify)
  u = given_output_unit (unit)
  write (u, integer_fmt) "n_it = ", self%n_it
  write (u, integer_fmt) "n_calls = ", self%n_calls
  write (u, logical_fmt) "adapt grids = ", self%adapt_grids
  write (u, logical_fmt) "adapt weights = ", self%adapt_weights
  if (self%integral_defined) then
     write (u, descr_fmt) "Results:  [it, calls, integral, error, efficiency]"
     do i = 1, self%n_it
        write (u, "(2x,I0,1x,I0,3(1x," // fmt // "))") &
             i, self%calls(i), self%integral(i), self%error(i), &
             self%efficiency(i)
     end do
  else
     write (u, "(3x,A)")  "Results: [undefined]"
  end if
end subroutine pass_write
```

Read and reconstruct the pass.

⟨*MCI vamp2: pass: TBP*⟩+≡

```
procedure :: read => pass_read
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine pass_read (self, u, n_pass, n_it)
  class(pass_t), intent(out) :: self
  integer, intent(in) :: u, n_pass, n_it
  integer :: i, j
  character(80) :: buffer
  self%i_pass = n_pass + 1
  self%i_first_it = n_it + 1
  read (u, integer_fmt) buffer, self%n_it
  read (u, integer_fmt) buffer, self%n_calls
  read (u, logical_fmt) buffer, self%adapt_grids
  read (u, logical_fmt) buffer, self%adapt_weights
  allocate (self%calls (self%n_it), source = 0)
  allocate (self%integral (self%n_it), source = 0._default)
  allocate (self%error (self%n_it), source = 0._default)
  allocate (self%efficiency (self%n_it), source = 0._default)
  read (u, descr_fmt)  buffer
  select case (trim (adjustl (buffer)))
  case ("Results:  [it, calls, integral, error, efficiency]")
     do i = 1, self%n_it
        read (u, *) &
             j, self%calls(i), self%integral(i), self%error(i), &
             self%efficiency(i)
     end do
     self%integral_defined = .true.
```

```
    case ("Results: [undefined]")
       self%integral_defined = .false.
    case default
       call msg_fatal ("Reading integration pass: corrupted file")
    end select
  end subroutine pass_read
```

Configure. We adjust the number of `n_calls`, if it is lower than `n_calls_min_per_channel` times `b_channel`, and print a warning message.

⟨*MCI vamp2: pass: TBP*⟩+≡

```
  procedure :: configure => pass_configure
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine pass_configure (pass, n_it, n_calls, n_calls_min)
    class(pass_t), intent(inout) :: pass
    integer, intent(in) :: n_it
    integer, intent(in) :: n_calls
    integer, intent(in) :: n_calls_min
    pass%n_it = n_it
    pass%n_calls = max (n_calls, n_calls_min)
    if (pass%n_calls /= n_calls) then
       write (msg_buffer, "(A,I0)")  "VAMP2: too few calls, resetting " &
            // "n_calls to ", pass%n_calls
       call msg_warning ()
    end if
    allocate (pass%calls (n_it), source = 0)
    allocate (pass%integral (n_it), source = 0._default)
    allocate (pass%error (n_it), source = 0._default)
    allocate (pass%efficiency (n_it), source = 0._default)
  end subroutine pass_configure
```

Given two pass objects, compare them. All parameters must match. Where integrations are done in both (number of calls nonzero), the results must be equal (up to numerical noise).

The allocated array sizes might be different, but should match up to the common `n_it` value.

⟨*MCI vamp2: interfaces*⟩≡

```
  interface operator (.matches.)
     module procedure pass_matches
  end interface operator (.matches.)
```

⟨*MCI vamp2: procedures*⟩+≡

```
  function pass_matches (pass, ref) result (ok)
    type(pass_t), intent(in) :: pass, ref
    integer :: n
    logical :: ok
    ok = .true.
    if (ok)  ok = pass%i_pass == ref%i_pass
    if (ok)  ok = pass%i_first_it == ref%i_first_it
    if (ok)  ok = pass%n_it == ref%n_it
    if (ok)  ok = pass%n_calls == ref%n_calls
    if (ok)  ok = pass%adapt_grids .eqv. ref%adapt_grids
    if (ok)  ok = pass%adapt_weights .eqv. ref%adapt_weights
```

```
    if (ok)  ok = pass%integral_defined .eqv. ref%integral_defined
    if (pass%integral_defined) then
       n = pass%n_it
       if (ok)  ok = all (pass%calls(:n) == ref%calls(:n))
       if (ok)  ok = all (pass%integral(:n) .matches. ref%integral(:n))
       if (ok)  ok = all (pass%error(:n) .matches. ref%error(:n))
       if (ok)  ok = all (pass%efficiency(:n) .matches. ref%efficiency(:n))
    end if
  end function pass_matches
```

Update a pass object, given a reference. The parameters must match, except for the `n_it` entry. The number of complete iterations must be less or equal to the reference, and the number of complete iterations in the reference must be no larger than `n_it`. Where results are present in both passes, they must match. Where results are present in the reference only, the pass is updated accordingly.

⟨*MCI vamp2: pass: TBP*⟩+≡

```
  procedure :: update => pass_update
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine pass_update (pass, ref, ok)
    class(pass_t), intent(inout) :: pass
    type(pass_t), intent(in) :: ref
    logical, intent(out) :: ok
    integer :: n, n_ref
    ok = .true.
    if (ok)  ok = pass%i_pass == ref%i_pass
    if (ok)  ok = pass%i_first_it == ref%i_first_it
    if (ok)  ok = pass%n_calls == ref%n_calls
    if (ok)  ok = pass%adapt_grids .eqv. ref%adapt_grids
    if (ok)  ok = pass%adapt_weights .eqv. ref%adapt_weights
    if (ok) then
       if (ref%integral_defined) then
          if (.not. allocated (pass%calls)) then
             allocate (pass%calls (pass%n_it), source = 0)
             allocate (pass%integral (pass%n_it), source = 0._default)
             allocate (pass%error (pass%n_it), source = 0._default)
             allocate (pass%efficiency (pass%n_it), source = 0._default)
          end if
          n = count (pass%calls /= 0)
          n_ref = count (ref%calls /= 0)
          ok = n <= n_ref .and. n_ref <= pass%n_it
          if (ok)  ok = all (pass%calls(:n) == ref%calls(:n))
          if (ok)  ok = all (pass%integral(:n) .matches. ref%integral(:n))
          if (ok)  ok = all (pass%error(:n) .matches. ref%error(:n))
          if (ok)  ok = all (pass%efficiency(:n) .matches. ref%efficiency(:n))
          if (ok) then
             pass%calls(n+1:n_ref) = ref%calls(n+1:n_ref)
             pass%integral(n+1:n_ref) = ref%integral(n+1:n_ref)
             pass%error(n+1:n_ref) = ref%error(n+1:n_ref)
             pass%efficiency(n+1:n_ref) = ref%efficiency(n+1:n_ref)
             pass%integral_defined = any (pass%calls /= 0)
          end if
       end if
    end if
  end subroutine pass_update
```

Match two real numbers: they are equal up to a tolerance, which is 10^{-8}, matching the number of digits that are output by `pass_write`. In particular, if one number is exactly zero, the other one must also be zero.

⟨*MCI vamp2: interfaces*⟩+≡
```
interface operator (.matches.)
   module procedure real_matches
end interface operator (.matches.)
```

⟨*MCI vamp2: procedures*⟩+≡
```
elemental function real_matches (x, y) result (ok)
  real(default), intent(in) :: x, y
  logical :: ok
  real(default), parameter :: tolerance = 1.e-8_default
  ok = abs (x - y) <= tolerance * max (abs (x), abs (y))
end function real_matches
```

Return the index of the most recent complete integration. If there is none, return zero.

⟨*MCI vamp2: pass: TBP*⟩+≡
```
procedure :: get_integration_index => pass_get_integration_index
```

⟨*MCI vamp2: procedures*⟩+≡
```
function pass_get_integration_index (pass) result (n)
  class (pass_t), intent(in) :: pass
  integer :: n
  integer :: i
  n = 0
  if (allocated (pass%calls)) then
     do i = 1, pass%n_it
        if (pass%calls(i) == 0)  exit
        n = i
     end do
  end if
end function pass_get_integration_index
```

Return the most recent integral and error, if available.

⟨*MCI vamp2: pass: TBP*⟩+≡
```
procedure :: get_calls => pass_get_calls
procedure :: get_integral => pass_get_integral
procedure :: get_error => pass_get_error
procedure :: get_efficiency => pass_get_efficiency
```

⟨*MCI vamp2: procedures*⟩+≡
```
function pass_get_calls (pass) result (calls)
  class(pass_t), intent(in) :: pass
  integer :: calls
  integer :: n
  n = pass%get_integration_index ()
  calls = 0
  if (n /= 0) then
     calls = pass%calls(n)
  end if
```

```
end function pass_get_calls

function pass_get_integral (pass) result (integral)
  class(pass_t), intent(in) :: pass
  real(default) :: integral
  integer :: n
  n = pass%get_integration_index ()
  integral = 0
  if (n /= 0) then
     integral = pass%integral(n)
  end if
end function pass_get_integral

function pass_get_error (pass) result (error)
  class(pass_t), intent(in) :: pass
  real(default) :: error
  integer :: n
  n = pass%get_integration_index ()
  error = 0
  if (n /= 0) then
     error = pass%error(n)
  end if
end function pass_get_error

function pass_get_efficiency (pass) result (efficiency)
  class(pass_t), intent(in) :: pass
  real(default) :: efficiency
  integer :: n
  n = pass%get_integration_index ()
  efficiency = 0
  if (n /= 0) then
     efficiency = pass%efficiency(n)
  end if
end function pass_get_efficiency
```

20.6.4 Integrator

We store the different passes of integration, adaptation and actual sampling, in a linked list.

We store the total number of calls `n_calls` and the minimal number of calls `n_calls_min`. The latter is calculated based on `n_channel` and `min_calls_per_channel`. If `n_calls` is smaller than `n_calls_min`, then we replace `n_calls` with `n_min_calls`.

⟨*MCI vamp2: public*⟩+≡

```
public :: mci_vamp2_t
```

⟨*MCI vamp2: types*⟩+≡

```
type, extends(mci_t) :: mci_vamp2_t
   type(mci_vamp2_config_t) :: config
   type(vamp2_t) :: integrator
   logical :: integrator_defined = .false.
   logical :: integrator_from_file = .false.
   logical :: adapt_grids = .false.
```

```
    logical :: adapt_weights = .false.
    integer :: n_adapt_grids = 0
    integer :: n_adapt_weights = 0
    integer :: n_calls = 0
    type(list_pass_t) :: list_pass
    logical :: rebuild = .true.
    logical :: check_grid_file = .true.
    logical :: integrator_filename_set = .false.
    logical :: verbose = .false.
    logical :: pass_complete = .false.
    logical :: it_complete = .false.
    type(string_t) :: integrator_filename
    character(32) :: md5sum_adapted = ""
  contains
  ⟨MCI vamp2: mci vamp2: TBP⟩
end type mci_vamp2_t
```

Finalizer: call to base and list finalizer.

⟨*MCI vamp2: mci vamp2: TBP*⟩≡

```
procedure, public :: final => mci_vamp2_final
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_final (object)
  class(mci_vamp2_t), intent(inout) :: object
  call object%list_pass%final ()
  call object%base_final ()
end subroutine mci_vamp2_final
```

Output. Do not output the grids themselves, this may result in tons of data.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
procedure, public :: write => mci_vamp2_write
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_write (object, unit, pacify, md5sum_version)
  class(mci_vamp2_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacify
  logical, intent(in), optional :: md5sum_version
  integer :: u, i
  u = given_output_unit (unit)
  write (u, descr_fmt) "VAMP2 integrator:"
  call object%base_write (u, pacify, md5sum_version)
  write (u, descr_fmt) "Grid config:"
  call object%config%write (u)
  write (u, logical_fmt) "integrator_defined = ", object%integrator_defined
  write (u, logical_fmt) "integrator_from_file = ", object%integrator_from_file
  write (u, logical_fmt) "adapt_grids = ", object%adapt_grids
  write (u, logical_fmt) "adapt_weights = ", object%adapt_weights
  write (u, integer_fmt) "n_adapt_grids = ", object%n_adapt_grids
  write (u, integer_fmt) "n_adapt_weights = ", object%n_adapt_weights
  write (u, logical_fmt) "verbose = ", object%verbose
  call object%list_pass%write (u, pacify)
  if (object%md5sum_adapted /= "") then
     write (u, md5_fmt)  "MD5 sum (including results) = ", object%md5sum_adapted
```

```
    end if
  end subroutine mci_vamp2_write
```

Compute the (adapted) MD5 sum, including the configuration MD5 sum and the printout, which incorporates the current results.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, public :: compute_md5sum => mci_vamp2_compute_md5sum
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_compute_md5sum (mci, pacify)
    class(mci_vamp2_t), intent(inout) :: mci
    logical, intent(in), optional :: pacify
    integer :: u
    mci%md5sum_adapted = ""
    u = free_unit ()
    open (u, status = "scratch", action = "readwrite")
    write (u, "(A)")  mci%md5sum
    call mci%write (u, pacify, md5sum_version = .true.)
    rewind (u)
    mci%md5sum_adapted = md5sum (u)
    close (u)
  end subroutine mci_vamp2_compute_md5sum
```

Return the MD5 sum: If available, return the adapted one.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, public :: get_md5sum => mci_vamp2_get_md5sum
```

⟨*MCI vamp2: procedures*⟩+≡

```
  pure function mci_vamp2_get_md5sum (mci) result (md5sum)
    class(mci_vamp2_t), intent(in) :: mci
    character(32) :: md5sum
    if (mci%md5sum_adapted /= "") then
       md5sum = mci%md5sum_adapted
    else
       md5sum = mci%md5sum
    end if
  end function mci_vamp2_get_md5sum
```

Startup message: short version. Make a call to the base function and print additional information about the multi-channel parameters.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, public :: startup_message => mci_vamp2_startup_message
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_startup_message (mci, unit, n_calls)
    class(mci_vamp2_t), intent(in) :: mci
    integer, intent(in), optional :: unit, n_calls
    integer :: num_calls, n_bins
    num_calls = 0; if (present (n_calls)) num_calls = n_calls
    n_bins = mci%config%n_bins_max
    call mci%base_startup_message (unit = unit, n_calls = n_calls)
    if (mci%config%use_channel_equivalences) then
       write (msg_buffer, "(A,2(1x,I0,1x,A))") &
```

```
            "Integrator: Using channel equivalences"
         call msg_message (unit = unit)
      end if
      write (msg_buffer, "(A,2(1x,I0,1x,A),L1)") &
           "Integrator:", num_calls, &
           "initial calls,", n_bins, &
           "max. bins, stratified = ", &
           mci%config%stratified
      call msg_message (unit = unit)
      write (msg_buffer, "(A,2(1x,I0,1x,A))") &
           "Integrator: VAMP2"
      call msg_message (unit = unit)
   end subroutine mci_vamp2_startup_message
```

Log entry: just headline.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
   procedure, public :: write_log_entry => mci_vamp2_write_log_entry
```

⟨*MCI vamp2: procedures*⟩+≡

```
   subroutine mci_vamp2_write_log_entry (mci, u)
      class(mci_vamp2_t), intent(in) :: mci
      integer, intent(in) :: u
      write (u, "(1x,A)")  "MC Integrator is VAMP2"
      call write_separator (u)
      ! TODO
      ! call mci%history%write (u)
      call write_separator (u)
      if (mci%config%use_channel_equivalences) then
         ! TODO
         write (u, "(3x,A)") "No channel equivalences have not been implemented yet."
      else
         write (u, "(3x,A)") "No channel equivalences have been used."
      end if
      call write_separator (u)
      call mci%write_chain_weights (u)
   end subroutine mci_vamp2_write_log_entry
```

Set the MCI index (necessary for processes with multiple components). We append the index to the grid filename, just before the final dotted suffix.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
   procedure, public :: record_index => mci_vamp2_record_index
```

⟨*MCI vamp2: procedures*⟩+≡

```
   subroutine mci_vamp2_record_index (mci, i_mci)
      class(mci_vamp2_t), intent(inout) :: mci
      integer, intent(in) :: i_mci
      type(string_t) :: basename, suffix
      character(32) :: buffer
      if (mci%integrator_filename_set) then
         basename = mci%integrator_filename
         call split (basename, suffix, ".", back=.true.)
         write (buffer, "(I0)") i_mci
         if (basename /= "") then
            mci%integrator_filename = basename // "_m" // trim (buffer) // "." // suffix
```

```
      else
         mci%integrator_filename = suffix // "_m" // trim (buffer) // ".vg2"
      end if
   end if
end subroutine mci_vamp2_record_index
```

Set the configuration object. We adjust the maximum number of bins `n_bins_max` according to `n_calls`

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
procedure, public :: set_config => mci_vamp2_set_config
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_set_config (mci, config)
  class(mci_vamp2_t), intent(inout) :: mci
  type(mci_vamp2_config_t), intent(in) :: config
  mci%config = config
end subroutine mci_vamp2_set_config
```

Set the the rebuild flag, also the for checking the grid.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
procedure, public :: set_rebuild_flag => mci_vamp2_set_rebuild_flag
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_set_rebuild_flag (mci, rebuild, check_grid_file)
  class(mci_vamp2_t), intent(inout) :: mci
  logical, intent(in) :: rebuild
  logical, intent(in) :: check_grid_file
  mci%rebuild = rebuild
  mci%check_grid_file = check_grid_file
end subroutine mci_vamp2_set_rebuild_flag
```

Set the filename.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
procedure, public :: set_integrator_filename => mci_vamp2_set_integrator_filename
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_set_integrator_filename (mci, name, run_id)
  class(mci_vamp2_t), intent(inout) :: mci
  type(string_t), intent(in) :: name
  type(string_t), intent(in), optional :: run_id
  mci%integrator_filename = name // ".vg2"
  if (present (run_id)) then
     mci%integrator_filename = name // "." // run_id // ".vg2"
  end if
  mci%integrator_filename_set = .true.
end subroutine mci_vamp2_set_integrator_filename
```

To simplify the interface, we prepend a grid path in a separate subroutine.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
procedure :: prepend_integrator_path => mci_vamp2_prepend_integrator_path
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_prepend_integrator_path (mci, prefix)
  class(mci_vamp2_t), intent(inout) :: mci
  type(string_t), intent(in) :: prefix
  if (.not. mci%integrator_filename_set) then
     call msg_warning ("Cannot add prefix to invalid integrator filename!")
  end if
  mci%integrator_filename = prefix // "/" // mci%integrator_filename
end subroutine mci_vamp2_prepend_integrator_path
```

TODO: Not implemented.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
procedure, public :: declare_flat_dimensions => mci_vamp2_declare_flat_dimensions
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_declare_flat_dimensions (mci, dim_flat)
  class(mci_vamp2_t), intent(inout) :: mci
  integer, dimension(:), intent(in) :: dim_flat
end subroutine mci_vamp2_declare_flat_dimensions
```

TODO: Not implemented.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
procedure, public :: declare_equivalences => mci_vamp2_declare_equivalences
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_declare_equivalences (mci, channel, dim_offset)
  class(mci_vamp2_t), intent(inout) :: mci
  type(phs_channel_t), dimension(:), intent(in) :: channel
  integer, intent(in) :: dim_offset
end subroutine mci_vamp2_declare_equivalences
```

Allocate instance with matching type.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
procedure, public :: allocate_instance => mci_vamp2_allocate_instance
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_allocate_instance (mci, mci_instance)
  class(mci_vamp2_t), intent(in) :: mci
  class(mci_instance_t), intent(out), pointer :: mci_instance
  allocate (mci_vamp2_instance_t :: mci_instance)
end subroutine mci_vamp2_allocate_instance
```

Allocate a new integration pass. We can preset everything that does not depend on the number of iterations and calls. This is postponed to the integrate method.

In the final pass, we do not check accuracy goal etc., since we can assume that the user wants to perform and average all iterations in this pass.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
procedure, public :: add_pass => mci_vamp2_add_pass
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_add_pass (mci, adapt_grids, adapt_weights, final_pass)
  class(mci_vamp2_t), intent(inout) :: mci
  logical, intent(in), optional :: adapt_grids, adapt_weights, final_pass
```

```
    call mci%list_pass%add (adapt_grids, adapt_weights, final_pass)
  end subroutine mci_vamp2_add_pass
```

Update the list of integration passes.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, public :: update_from_ref => mci_vamp2_update_from_ref
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_update_from_ref (mci, mci_ref, success)
    class(mci_vamp2_t), intent(inout) :: mci
    class(mci_t), intent(in) :: mci_ref
    logical, intent(out) :: success
    select type (mci_ref)
    type is (mci_vamp2_t)
       call mci%list_pass%update_from_ref (mci_ref%list_pass, success)
       if (mci%list_pass%current%integral_defined) then
          mci%integral = mci%list_pass%current%get_integral ()
          mci%error = mci%list_pass%current%get_error ()
          mci%efficiency = mci%list_pass%current%get_efficiency ()
          mci%integral_known = .true.
          mci%error_known = .true.
          mci%efficiency_known = .true.
       end if
    end select
  end subroutine mci_vamp2_update_from_ref
```

Update the MCI record (i.e., the integration passes) by reading from input stream. The stream should contain a write output from a previous run. We first check the MD5 sum of the configuration parameters. If that matches, we proceed directly to the stored integration passes. If successful, we may continue to read the file; the position will be after a blank line that must follow the MCI record.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, public :: update => mci_vamp2_update
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_update (mci, u, success)
    class(mci_vamp2_t), intent(inout) :: mci
    integer, intent(in) :: u
    logical, intent(out) :: success
    character(80) :: buffer
    character(32) :: md5sum_file
    type(mci_vamp2_t) :: mci_file
    integer :: n_pass, n_it
    read (u, md5_fmt) buffer, md5sum_file
    success = .true.; if (mci%check_grid_file) &
       & success = (md5sum_file == mci%md5sum)
    if (success) then
       read (u, *)
       read (u, descr_fmt) buffer
       if (trim (adjustl (buffer)) /= "VAMP2 integrator:") then
          call msg_fatal ("VAMP2: reading grid file: corrupted data")
       end if
```

```
      n_pass = 0
      n_it = 0
      do
         read (u, descr_fmt) buffer
         select case (trim (adjustl (buffer)))
         case ("")
            exit
         case ("Integration pass:")
            call mci_file%list_pass%add ()
            call mci_file%list_pass%current%read (u, n_pass, n_it)
            n_pass = n_pass + 1
            n_it = n_it + mci_file%list_pass%current%n_it
         end select
      end do
      call mci%update_from_ref (mci_file, success)
      call mci_file%final ()
   end if
end subroutine mci_vamp2_update
```

Read / write grids from / to file.

⟨*MCI vamp2: parameters*⟩≡

```
character(len=*), parameter, private :: &
     descr_fmt   = "(1X,A)", &
     md5_fmt     = "(1X,A25,1X,A32)", &
     integer_fmt = "(1X,A25,1X,I15)", &
     logical_fmt = "(1X,A25,1X,L1)", &
     double_fmt  = "(1X,A25,1X," // FMT_14 // ")", &
     double_array_fmt = "(1X,I25,1X," // FMT_14 // ")", &
     double_array2_fmt =  "(1X,2(1X,I8),1X," // FMT_14 // ")"
```

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
procedure :: write_grids => mci_vamp2_write_grids
procedure :: read_grids_header => mci_vamp2_read_grids_header
procedure :: read_grids_data => mci_vamp2_read_grids_data
procedure :: read_grids => mci_vamp2_read_grids
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_write_grids (mci)
  class(mci_vamp2_t), intent(in) :: mci
  integer :: u
  if (.not. mci%integrator_filename_set) then
     call msg_bug ("VAMP2: write grids: filename undefined")
  end if
  if (.not. mci%integrator_defined) then
     call msg_bug ("VAMP2: write grids: grids undefined")
  end if
  u = free_unit ()
  open (u, file = char (mci%integrator_filename), &
       action = "write", status = "replace")
  write (u, md5_fmt) "MD5sum =", mci%md5sum
  write (u, *)
  call mci%write (u)
  write (u, *)
```

```
  write (u, descr_fmt) "VAMP2 grids:"
  call mci%integrator%write_grids (u)
  close (u)
end subroutine mci_vamp2_write_grids

subroutine mci_vamp2_read_grids_header (mci, success)
  class(mci_vamp2_t), intent(inout) :: mci
  logical, intent(out) :: success
  logical :: exist
  integer :: u
  success = .false.
  if (.not. mci%integrator_filename_set) then
     call msg_bug ("VAMP2: read grids: filename undefined")
  end if
  inquire (file = char (mci%integrator_filename), exist = exist)
  if (exist) then
     u = free_unit ()
     open (u, file = char (mci%integrator_filename), &
          action = "read", status = "old")
     call mci%update (u, success)
     close (u)
     if (.not. success) then
        write (msg_buffer, "(A,A,A)") &
             "VAMP2: header: parameter mismatch, discarding grid file '", &
             char (mci%integrator_filename), "'"
        call msg_message ()
     end if
  end if
end subroutine mci_vamp2_read_grids_header

subroutine mci_vamp2_read_grids_data (mci)
  class(mci_vamp2_t), intent(inout) :: mci
  integer :: u
  character(80) :: buffer
  if (mci%integrator_defined) then
     call msg_bug ("VAMP2: read grids: grids already defined")
  end if
  u = free_unit ()
  open (u, file = char (mci%integrator_filename), &
       action = "read", status = "old")
  do
     read (u, descr_fmt)  buffer
     if (trim (adjustl (buffer)) == "VAMP2 grids:")  exit
  end do
  call mci%integrator%read_grids (u)
  close (u)
  mci%integrator_defined = .true.
end subroutine mci_vamp2_read_grids_data

subroutine mci_vamp2_read_grids (mci, success)
  class(mci_vamp2_t), intent(inout) :: mci
  logical, intent(out) :: success
  logical :: exist
  integer :: u
```

```
    character(80) :: buffer
    success = .false.
    if (.not. mci%integrator_filename_set) then
       call msg_bug ("VAMP2: read grids: filename undefined")
    end if
    if (mci%integrator_defined) then
       call msg_bug ("VAMP2: read grids: grids already defined")
    end if
    inquire (file = char (mci%integrator_filename), exist = exist)
    if (exist) then
       u = free_unit ()
       open (u, file = char (mci%integrator_filename), &
            action = "read", status = "old")
       call mci%update (u, success)
       if (success) then
          read (u, descr_fmt)  buffer
          if (trim (adjustl (buffer)) /= "VAMP2 grids:") then
             call msg_fatal ("VAMP2: reading grid file: &
                  &corrupted grid data")
          end if
          call mci%integrator%read_grids (u)
       else
          write (msg_buffer, "(A,A,A)") &
               "VAMP2: read grids: parameter mismatch, discarding grid file '", &
               char (mci%integrator_filename), "'"
          call msg_message ()
       end if
       close (u)
       mci%integrator_defined = success
    end if
  end subroutine mci_vamp2_read_grids
```

Interface: VAMP2

We define the interfacing procedures, as such, initialising the VAMP2 integrator or resetting the results.

Initialise the VAMP2 integrator which is stored within the `mci` object, using the data of the current integration pass. Furthermore, reset the counters that track this set of integrator.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, public :: init_integrator => mci_vamp2_init_integrator
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_init_integrator (mci)
    class(mci_vamp2_t), intent(inout) :: mci
    type (pass_t), pointer :: current
    integer :: ch, vegas_mode
    current => mci%list_pass%current
    vegas_mode = merge (VEGAS_MODE_IMPORTANCE, VEGAS_MODE_IMPORTANCE_ONLY,&
         & mci%config%stratified)
    mci%n_adapt_grids = 0
    mci%n_adapt_weights = 0
    if (mci%integrator_defined) then
       call msg_bug ("[MCI VAMP2]: init integrator: &
```

```
          & integrator is already initialised.")
    end if
    mci%integrator = vamp2_t (mci%n_channel, mci%n_dim, &
         & n_bins_max = mci%config%n_bins_max, &
         & iterations = 1, &
         & mode = vegas_mode)
    call mci%integrator%set_config (mci%config)
    mci%integrator_defined = .true.
  end subroutine mci_vamp2_init_integrator
```

Reset a grid set. Purge the accumulated results.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, public :: reset_result => mci_vamp2_reset_result
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_reset_result (mci)
    class(mci_vamp2_t), intent(inout) :: mci
    if (.not. mci%integrator_defined) then
       call msg_bug ("[MCI VAMP2] reset results: integrator undefined")
    end if
    call mci%integrator%reset_result ()
  end subroutine mci_vamp2_reset_result
```

Set calls per channel. The number of calls to each channel is defined by the channel weight

$$\alpha_i = \frac{N_i}{\sum N_i}. \tag{20.18}$$

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, public :: set_calls => mci_vamp2_set_calls
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_set_calls (mci, n_calls)
    class(mci_vamp2_t), intent(inout) :: mci
    integer :: n_calls
    if (.not. mci%integrator_defined) then
       call msg_bug ("[MCI VAMP2] set calls: grids undefined")
    end if
    call mci%integrator%set_calls (n_calls)
  end subroutine mci_vamp2_set_calls
```

Integration

Initialize. We prepare the integrator from a previous pass, or from file, or with new objects.

At the emd, set the number of calls for the current, if the integrator is not read from file.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, private :: init_integration => mci_vamp2_init_integration
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_init_integration (mci, n_it, n_calls, instance)
  class(mci_vamp2_t), intent(inout) :: mci
  integer, intent(in) :: n_it
  integer, intent(in) :: n_calls
  class(mci_instance_t), intent(inout) :: instance
  logical :: from_file, success
  if (.not. associated (mci%list_pass%current)) then
     call msg_bug ("MCI integrate: current_pass object not allocated")
  end if
  associate (current_pass => mci%list_pass%current)
    current_pass%integral_defined = .false.
    mci%config%n_calls_min = mci%config%n_calls_min_per_channel * mci%config%n_channel
    call current_pass%configure (n_it, n_calls, mci%config%n_calls_min)
    mci%adapt_grids = current_pass%adapt_grids
    mci%adapt_weights = current_pass%adapt_weights
    mci%pass_complete = .false.
    mci%it_complete = .false.
    from_file = .false.
    if (.not. mci%integrator_defined .or. mci%integrator_from_file) then
       if (mci%integrator_filename_set .and. .not. mci%rebuild) then
          call mci%read_grids_header (success)
          from_file = success
          if (.not. mci%integrator_defined .and. success) &
               & call mci%read_grids_data ()
       end if
    end if
    if (from_file) then
       if (.not. mci%check_grid_file) &
            & call msg_warning ("Reading grid file: MD5 sum check disabled")
       call msg_message ("VAMP2: " &
            // "using grids and results from file " &
            // char (mci%integrator_filename) // "")
    else if (.not. mci%integrator_defined) then
       call mci%init_integrator ()
    end if
    mci%integrator_from_file = from_file
    if (.not. mci%integrator_from_file) then
       call mci%integrator%set_calls (current_pass%n_calls)
    end if
  end associate
end subroutine mci_vamp2_init_integration
```

Integrate. Perform a new integration pass (possibly reusing previous results), which may consist of several iterations. We reinitialise the sampling new each time and set the workspace again. Note: we record the integral once per iteration. The integral stored in the mci record itself is the last integral of the current iteration, no averaging done. The results record may average results. Note: recording the efficiency is not supported yet.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
procedure, public :: integrate => mci_vamp2_integrate
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_integrate (mci, instance, sampler, &
```

```
   n_it, n_calls, results, pacify)
class(mci_vamp2_t), intent(inout) :: mci
class(mci_instance_t), intent(inout), target :: instance
class(mci_sampler_t), intent(inout), target :: sampler
integer, intent(in) :: n_it
integer, intent(in) :: n_calls
class(mci_results_t), intent(inout), optional :: results
logical, intent(in), optional :: pacify
integer :: it
logical :: from_file, success
call mci%init_integration (n_it, n_calls, instance)
from_file = mci%integrator_from_file
select type (instance)
type is (mci_vamp2_instance_t)
   call instance%set_workspace (sampler)
end select
associate (current_pass => mci%list_pass%current)
  do it = 1, current_pass%n_it
     if (signal_is_pending ()) return
     mci%integrator_from_file = from_file .and. &
          it <= current_pass%get_integration_index ()
     if (.not. mci%integrator_from_file) then
        mci%it_complete = .false.
        select type (instance)
        type is (mci_vamp2_instance_t)
           call mci%integrator%integrate (instance%func, mci%rng, &
                & iterations = 1, &
                & opt_reset_result = .true., &
                & opt_refine_grid = mci%adapt_grids, &
                & opt_adapt_weight = mci%adapt_weights, &
                & opt_verbose = mci%verbose)
        end select
        if (signal_is_pending ()) return
        mci%it_complete = .true.
        if (mci%integrator%get_integral () /= 0) then
           select type (instance)
           type is (mci_vamp2_instance_t)
              current_pass%calls(it) = instance%func%get_n_calls ()
              call instance%func%reset_n_calls ()
           end select
           current_pass%integral(it) = mci%integrator%get_integral ()
           current_pass%error(it) = sqrt (mci%integrator%get_variance ())
           current_pass%efficiency(it) = mci%integrator%get_efficiency ()
        end if
        current_pass%integral_defined = .true.
     end if
     if (present (results)) then
        if (mci%has_chains ()) then
           call mci%collect_chain_weights (instance%w)
           call results%record (1, &
                n_calls = current_pass%calls(it), &
                integral = current_pass%integral(it), &
                error = current_pass%error(it), &
                efficiency = current_pass%efficiency(it), &
```

```
                chain_weights = mci%chain_weights, &
                suppress = pacify)
         else
            call results%record (1, &
                n_calls = current_pass%calls(it), &
                integral = current_pass%integral(it), &
                error = current_pass%error(it), &
                efficiency = current_pass%efficiency(it), &
                suppress = pacify)
         end if
      end if
      if (.not. mci%integrator_from_file &
           .and. mci%integrator_filename_set) then
         call mci%write_grids ()
      end if
      if (.not. current_pass%is_final_pass) then
         call mci%check_goals (it, success)
         if (success) exit
      end if
   end do
   if (signal_is_pending ()) return
   mci%pass_complete = .true.
   mci%integral = current_pass%get_integral()
   mci%error = current_pass%get_error()
   mci%efficiency = current_pass%get_efficiency()
   mci%integral_known = .true.
   mci%error_known = .true.
   mci%efficiency_known = .true.
   call mci%compute_md5sum (pacify)
 end associate
end subroutine mci_vamp2_integrate
```

Check whether we are already finished with this pass.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
procedure, private :: check_goals => mci_vamp2_check_goals
```

⟨*MCI vamp2: procedures*⟩+≡

```
subroutine mci_vamp2_check_goals (mci, it, success)
  class(mci_vamp2_t), intent(inout) :: mci
  integer, intent(in) :: it
  logical, intent(out) :: success
  success = .false.
  associate (current_pass => mci%list_pass%current)
    if (mci%error_reached (it)) then
       current_pass%n_it = it
       call msg_message ("[MCI VAMP2] error goal reached; &
            &skipping iterations")
       success = .true.
       return
    end if
    if (mci%rel_error_reached (it)) then
       current_pass%n_it = it
       call msg_message ("[MCI VAMP2] relative error goal reached; &
            &skipping iterations")
       success = .true.
```

```
        return
      end if
      if (mci%accuracy_reached (it)) then
         current_pass%n_it = it
         call msg_message ("[MCI VAMP2] accuracy goal reached; &
              &skipping iterations")
         success = .true.
         return
      end if
    end associate
  end subroutine mci_vamp2_check_goals
```

Return true if the error, relative error or accurary goals hase been reached, if any.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, private :: error_reached => mci_vamp2_error_reached
  procedure, private :: rel_error_reached => mci_vamp2_rel_error_reached
  procedure, private :: accuracy_reached => mci_vamp2_accuracy_reached
```

⟨*MCI vamp2: procedures*⟩+≡

```
  function mci_vamp2_error_reached (mci, it) result (flag)
    class(mci_vamp2_t), intent(in) :: mci
    integer, intent(in) :: it
    logical :: flag
    real(default) :: error_goal, error
    error_goal = mci%config%error_goal
    flag = .false.
    associate (current_pass => mci%list_pass%current)
      if (error_goal > 0 .and. current_pass%integral_defined) then
         error = abs (current_pass%error(it))
         flag = error < error_goal
      end if
    end associate
  end function mci_vamp2_error_reached

  function mci_vamp2_rel_error_reached (mci, it) result (flag)
    class(mci_vamp2_t), intent(in) :: mci
    integer, intent(in) :: it
    logical :: flag
    real(default) :: rel_error_goal, rel_error
    rel_error_goal = mci%config%rel_error_goal
    flag = .false.
    associate (current_pass => mci%list_pass%current)
      if (rel_error_goal > 0 .and. current_pass%integral_defined) then
         rel_error = abs (current_pass%error(it) / current_pass%integral(it))
         flag = rel_error < rel_error_goal
      end if
    end associate
  end function mci_vamp2_rel_error_reached

  function mci_vamp2_accuracy_reached (mci, it) result (flag)
    class(mci_vamp2_t), intent(in) :: mci
    integer, intent(in) :: it
    logical :: flag
```

```
    real(default) :: accuracy_goal, accuracy
    accuracy_goal = mci%config%accuracy_goal
    flag = .false.
    associate (current_pass => mci%list_pass%current)
      if (accuracy_goal > 0 .and. current_pass%integral_defined) then
         if (current_pass%integral(it) /= 0) then
            accuracy = abs (current_pass%error(it) / current_pass%integral(it)) &
                 * sqrt (real (current_pass%calls(it), default))
            flag = accuracy < accuracy_goal
         else
            flag = .true.
         end if
      end if
    end associate
  end function mci_vamp2_accuracy_reached
```

20.6.5 Event generation

Prepare simulation. We check the grids and reread them from file, if necessary.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, public :: prepare_simulation => mci_vamp2_prepare_simulation
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_prepare_simulation (mci)
    class(mci_vamp2_t), intent(inout) :: mci
    logical :: success
    if (.not. mci%integrator_filename_set) then
       call msg_bug ("VAMP2: preapre simulation: integrator filename not set.")
    end if
    call mci%read_grids_header (success)
    call mci%compute_md5sum ()
    if (.not. success) then
       call msg_fatal ("Simulate: " &
            // "reading integration grids from file " &
            // char (mci%integrator_filename) // " failed")
    end if
    if (.not. mci%integrator_defined) then
       call mci%read_grids_data ()
    end if
  end subroutine mci_vamp2_prepare_simulation
```

Generate an unweighted event. We only set the workspace again before generating an event.

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, public :: generate_weighted_event => mci_vamp2_generate_weighted_event
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_generate_weighted_event (mci, instance, sampler)
    class(mci_vamp2_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout), target :: instance
    class(mci_sampler_t), intent(inout), target :: sampler
    if (.not. mci%integrator_defined) then
       call msg_bug ("VAMP2: generate weighted event: undefined integrator")
```

```
    end if
    select type (instance)
    type is (mci_vamp2_instance_t)
       instance%event_generated = .false.
       call instance%set_workspace (sampler)
       call mci%integrator%generate_event (&
            & instance%func, mci%rng, instance%event_x, &
            opt_event_weight = instance%event_weight)
       instance%event_excess = 0
       instance%n_events = instance%n_events + 1
       instance%event_generated = .true.
    end select
  end subroutine mci_vamp2_generate_weighted_event
```

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, public :: generate_unweighted_event => mci_vamp2_generate_unweighted_event
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_generate_unweighted_event (mci, instance, sampler)
    class(mci_vamp2_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout), target :: instance
    class(mci_sampler_t), intent(inout), target :: sampler
    if (.not. mci%integrator_defined) then
       call msg_bug ("VAMP2: generate unweighted event: undefined integrator")
    end if
    select type (instance)
    type is (mci_vamp2_instance_t)
       instance%event_generated = .false.
       call instance%set_workspace (sampler)
       generate: do
          call mci%integrator%generate_event (&
               & instance%func, mci%rng, instance%event_x, &
               opt_event_excess = instance%event_excess)
          if (signal_is_pending ()) return
          if (sampler%is_valid ()) exit generate
       end do generate
       instance%event_weight = 1
       instance%n_events = instance%n_events + 1
       instance%event_generated = .true.
    end select
  end subroutine mci_vamp2_generate_unweighted_event
```

⟨*MCI vamp2: mci vamp2: TBP*⟩+≡

```
  procedure, public :: rebuild_event => mci_vamp2_rebuild_event
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_rebuild_event (mci, instance, sampler, state)
    class(mci_vamp2_t), intent(inout) :: mci
    class(mci_instance_t), intent(inout) :: instance
    class(mci_sampler_t), intent(inout) :: sampler
    class(mci_state_t), intent(in) :: state
    call msg_bug ("MCI VAMP2 rebuild event not implemented yet.")
  end subroutine mci_vamp2_rebuild_event
```

20.6.6 Integrator instance

We store all information relevant for simulation. The event weight is stored, when a weighted event is generated, and the event excess, when a larger weight occurs than actual stored max. weight.

⟨*MCI vamp2: public*⟩+≡
```
public :: mci_vamp2_instance_t
```

⟨*MCI vamp2: types*⟩+≡
```
type, extends (mci_instance_t) :: mci_vamp2_instance_t
   class(mci_vamp2_func_t), allocatable :: func
   real(default), dimension(:), allocatable :: gi
   integer :: n_events = 0
   logical :: event_generated = .false.
   real(default) :: event_weight = 0.
   real(default) :: event_excess = 0.
   real(default), dimension(:), allocatable :: event_x
 contains
 ⟨MCI vamp2: mci vamp2 instance: TBP⟩
end type mci_vamp2_instance_t
```

Output.

⟨*MCI vamp2: mci vamp2 instance: TBP*⟩≡
```
procedure, public :: write => mci_vamp2_instance_write
```

⟨*MCI vamp2: procedures*⟩+≡
```
subroutine mci_vamp2_instance_write (object, unit, pacify)
  class(mci_vamp2_instance_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacify
  integer :: u, ch, j
  character(len=7) :: fmt
  call pac_fmt (fmt, FMT_17, FMT_14, pacify)
  u = given_output_unit (unit)
  write (u, descr_fmt) "MCI VAMP2 instance"
  write (u, integer_fmt) "Selected channel =", object%selected_channel
  write (u, double_fmt) "Integrand =", object%integrand
  write (u, double_fmt) "MCI weight =", object%mci_weight
  write (u, logical_fmt) "Valid =", object%valid
  write (u, descr_fmt) "MCI a-priori weight"
  do ch = 1, size (object%w)
     write (u, double_array_fmt) ch, object%w(ch)
  end do
  write (u, descr_fmt) "MCI jacobian"
  do ch = 1, size (object%w)
     write (u, double_array_fmt) ch, object%f(ch)
  end do
  write (u, descr_fmt) "MCI mapped x"
  do ch = 1, size (object%w)
     do j = 1, size (object%x, 1)
        write (u, double_array2_fmt) j, ch, object%x(j, ch)
     end do
  end do
  write (u, descr_fmt) "MCI channel weight"
```

```
    do ch = 1, size (object%w)
       write (u, double_array_fmt) ch, object%gi(ch)
    end do
    write (u, integer_fmt) "Number of event =", object%n_events
    write (u, logical_fmt) "Event generated =", object%event_generated
    write (u, double_fmt) "Event weight =", object%event_weight
    write (u, double_fmt) "Event excess =", object%event_excess
    write (u, logical_fmt) "Negative (event) weight = ", object%negative_weights
    write (u, descr_fmt) "MCI event"
    do j = 1, size (object%event_x)
       write (u, double_array_fmt) j, object%event_x(j)
    end do
  end subroutine mci_vamp2_instance_write
```

Finalizer. We are only using allocatable, so there is nothing to do here.

⟨*MCI vamp2: mci vamp2 instance: TBP*⟩+≡

```
  procedure, public :: final => mci_vamp2_instance_final
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_instance_final (object)
    class(mci_vamp2_instance_t), intent(inout) :: object
    !
  end subroutine mci_vamp2_instance_final
```

Initializer.

⟨*MCI vamp2: mci vamp2 instance: TBP*⟩+≡

```
  procedure, public :: init => mci_vamp2_instance_init
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_instance_init (mci_instance, mci)
    class(mci_vamp2_instance_t), intent(out) :: mci_instance
    class(mci_t), intent(in), target :: mci
    call mci_instance%base_init (mci)
    allocate (mci_instance%gi(mci%n_channel), source=0._default)
    allocate (mci_instance%event_x(mci%n_dim), source=0._default)
    allocate (mci_vamp2_func_t :: mci_instance%func)
    call mci_instance%func%init (n_dim = mci%n_dim, n_channel = mci%n_channel)
  end subroutine mci_vamp2_instance_init
```

Set workspace for `mci_vamp2_func_t`.

⟨*MCI vamp2: mci vamp2 instance: TBP*⟩+≡

```
  procedure, public :: set_workspace => mci_vamp2_instance_set_workspace
```

⟨*MCI vamp2: procedures*⟩+≡

```
  subroutine mci_vamp2_instance_set_workspace (instance, sampler)
    class(mci_vamp2_instance_t), intent(inout), target :: instance
    class(mci_sampler_t), intent(inout), target :: sampler
    !call instance%func%set_workspace (instance, sampler)
    instance%func%instance => instance
    instance%func%sampler => sampler
  end subroutine mci_vamp2_instance_set_workspace
```

Evaluation

Compute multi-channel weight. The computation of the multi-channel weight is done by the VAMP2 function. We retrieve the information.

⟨*MCI vamp2: mci vamp2 instance: TBP*⟩+≡
```
procedure, public :: compute_weight => mci_vamp2_instance_compute_weight
```

⟨*MCI vamp2: procedures*⟩+≡
```
subroutine mci_vamp2_instance_compute_weight (mci, c)
  class(mci_vamp2_instance_t), intent(inout) :: mci
  integer, intent(in) :: c
  mci%gi = mci%func%get_probabilities ()
  mci%mci_weight = mci%func%get_weight ()
end subroutine mci_vamp2_instance_compute_weight
```

Record the integrand.

⟨*MCI vamp2: mci vamp2 instance: TBP*⟩+≡
```
procedure, public :: record_integrand => mci_vamp2_instance_record_integrand
```

⟨*MCI vamp2: procedures*⟩+≡
```
subroutine mci_vamp2_instance_record_integrand (mci, integrand)
  class(mci_vamp2_instance_t), intent(inout) :: mci
  real(default), intent(in) :: integrand
  mci%integrand = integrand
  call mci%func%set_integrand (integrand)
end subroutine mci_vamp2_instance_record_integrand
```

Event simulation

Base skeleton for event simulation.

⟨*MCI vamp2: mci vamp2 instance: TBP*⟩+≡
```
procedure, public :: init_simulation => mci_vamp2_instance_init_simulation
```

⟨*MCI vamp2: procedures*⟩+≡
```
subroutine mci_vamp2_instance_init_simulation (instance, safety_factor)
  class(mci_vamp2_instance_t), intent(inout) :: instance
  real(default), intent(in), optional :: safety_factor
  if (present (safety_factor)) then
     call msg_message ("MCI VAMP2: init simulation: safety factor is not implemented yet.")
  end if
  instance%n_events = 0
  instance%event_generated = .false.
end subroutine mci_vamp2_instance_init_simulation
```

⟨*MCI vamp2: mci vamp2 instance: TBP*⟩+≡
```
procedure, public :: final_simulation => mci_vamp2_instance_final_simulation
```

⟨*MCI vamp2: procedures*⟩+≡
```
subroutine mci_vamp2_instance_final_simulation (instance)
  class(mci_vamp2_instance_t), intent(inout) :: instance
  !
end subroutine mci_vamp2_instance_final_simulation
```

⟨*MCI vamp2: mci vamp2 instance: TBP*⟩+≡

```
procedure, public :: get_event_weight => mci_vamp2_instance_get_event_weight
```

⟨*MCI vamp2: procedures*⟩+≡

```
function mci_vamp2_instance_get_event_weight (mci) result (weight)
  class(mci_vamp2_instance_t), intent(in) :: mci
  real(default) :: weight
  if (.not. mci%event_generated) then
     call msg_bug ("MCI VAMP2: get event weight: no event generated")
  end if
  weight = mci%event_weight
end function mci_vamp2_instance_get_event_weight
```

⟨*MCI vamp2: mci vamp2 instance: TBP*⟩+≡

```
procedure, public :: get_event_excess => mci_vamp2_instance_get_event_excess
```

⟨*MCI vamp2: procedures*⟩+≡

```
function mci_vamp2_instance_get_event_excess (mci) result (excess)
  class(mci_vamp2_instance_t), intent(in) :: mci
  real(default) :: excess
  if (.not. mci%event_generated) then
     call msg_bug ("MCI VAMP2: get event excess: no event generated")
  end if
  excess = mci%event_excess
end function mci_vamp2_instance_get_event_excess
```

20.6.7 Unit tests

Test module, followed by the corresponding implementation module.

⟨mci_vamp2_ut.f90⟩≡

```
⟨File header⟩

module mci_vamp2_ut
  use unit_tests
  use mci_vamp2_uti

⟨Standard module head⟩

⟨MCI vamp2: public test⟩

contains

⟨MCI vamp2: test driver⟩

end module mci_vamp2_ut
```

⟨mci_vamp2_uti.f90⟩≡

```
⟨File header⟩

module mci_vamp2_uti

⟨Use kinds⟩
⟨Use strings⟩

  use io_units
  use constants, only: PI, TWOPI
  use rng_base
  use rng_tao
  use rng_stream
  use mci_base

  use mci_vamp2

⟨Standard module head⟩

⟨MCI vamp2: test declarations⟩

⟨MCI vamp2: test types⟩

contains

⟨MCI vamp2: tests⟩

end module mci_vamp2_uti
```

API: driver for the unit tests below.

⟨*MCI vamp2: public test*⟩≡

```
  public :: mci_vamp2_test
```

⟨*MCI vamp2: test driver*⟩≡

```
  subroutine mci_vamp2_test (u, results)
    integer, intent(in) :: u
```

```
  type(test_results_t), intent(inout) :: results
⟨MCI vamp2: execute tests⟩
end subroutine mci_vamp2_test
```

Test sampler

A test sampler object should implement a function with known integral that we can use to check the integrator.

In mode 1, the function is $f(x) = 3x^2$ with integral $\int_0^1 f(x)\,dx = 1$ and maximum $f(1) = 3$. If the integration dimension is greater than one, the function is extended as a constant in the other dimension(s).

In mode 2, the function is $11x^{10}$, also with integral 1.

Mode 4 includes ranges of zero and negative function value, the integral is negative. The results should be identical to the results of `mci_midpoint_4`, where the same function is evaluated. The function is $f(x) = (1-3x^2)\,\theta(x-1/2)$ with integral $\int_0^1 f(x)\,dx = -3/8$, minimum $f(1) = -2$ and maximum $f(1/2) = 1/4$.

⟨*MCI vamp2: test types*⟩≡

```
type, extends (mci_sampler_t) :: test_sampler_1_t
   real(default), dimension(:), allocatable :: x
   real(default) :: val
   integer :: mode = 1
 contains
 ⟨MCI vamp2: test sampler 1: TBP⟩
end type test_sampler_1_t
```

Output: There is nothing stored inside, so just print an informative line.

⟨*MCI vamp2: test sampler 1: TBP*⟩≡

```
procedure, public :: write => test_sampler_1_write
```

⟨*MCI vamp2: tests*⟩≡

```
subroutine test_sampler_1_write (object, unit, testflag)
  class(test_sampler_1_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  select case (object%mode)
  case (1)
     write (u, "(1x,A)") "Test sampler: f(x) = 3 x^2"
  case (2)
     write (u, "(1x,A)") "Test sampler: f(x) = 11 x^10"
  case (3)
     write (u, "(1x,A)") "Test sampler: f(x) = 11 x^10 * 2 * cos^2 (2 pi y)"
  case (4)
     write (u, "(1x,A)") "Test sampler: f(x) = (1 - 3 x^2) theta(x - 1/2)"
  end select
end subroutine test_sampler_1_write
```

Evaluation: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI vamp2: test sampler 1: TBP*⟩+≡

```
procedure, public :: evaluate => test_sampler_1_evaluate
```

⟨*MCI vamp2: tests*⟩+≡

```
subroutine test_sampler_1_evaluate (sampler, c, x_in, val, x, f)
  class(test_sampler_1_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  if (allocated (sampler%x))  deallocate (sampler%x)
  allocate (sampler%x (size (x_in)))
  sampler%x = x_in
  select case (sampler%mode)
  case (1)
     sampler%val = 3 * x_in(1) ** 2
  case (2)
     sampler%val = 11 * x_in(1) ** 10
  case (3)
     sampler%val = 11 * x_in(1) ** 10 * 2 * cos (twopi * x_in(2)) ** 2
  case (4)
     if (x_in(1) >= .5_default) then
        sampler%val = 1 - 3 * x_in(1) ** 2
     else
        sampler%val = 0
     end if
  end select
  call sampler%fetch (val, x, f)
end subroutine test_sampler_1_evaluate
```

The point is always valid.

⟨*MCI vamp2: test sampler 1: TBP*⟩+≡

```
procedure, public :: is_valid => test_sampler_1_is_valid
```

⟨*MCI vamp2: tests*⟩+≡

```
function test_sampler_1_is_valid (sampler) result (valid)
  class(test_sampler_1_t), intent(in) :: sampler
  logical :: valid
  valid = .true.
end function test_sampler_1_is_valid
```

Rebuild: compute all but the function value.

⟨*MCI vamp2: test sampler 1: TBP*⟩+≡

```
procedure, public :: rebuild => test_sampler_1_rebuild
```

⟨*MCI vamp2: tests*⟩+≡

```
subroutine test_sampler_1_rebuild (sampler, c, x_in, val, x, f)
  class(test_sampler_1_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(in) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  if (allocated (sampler%x))  deallocate (sampler%x)
```

```
    allocate (sampler%x (size (x_in)))
    sampler%x = x_in
    sampler%val = val
    x(:,1) = sampler%x
    f = 1
  end subroutine test_sampler_1_rebuild
```

Extract the results.

⟨*MCI vamp2: test sampler 1: TBP*⟩+≡

```
  procedure, public :: fetch => test_sampler_1_fetch
```

⟨*MCI vamp2: tests*⟩+≡

```
  subroutine test_sampler_1_fetch (sampler, val, x, f)
    class(test_sampler_1_t), intent(in) :: sampler
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    val = sampler%val
    x(:,1) = sampler%x
    f = 1
  end subroutine test_sampler_1_fetch
```

Two-channel, two dimension test sampler

This sampler implements the function

$$f(x,y) = 4\sin^2(\pi x)\sin^2(\pi y) + 2\sin^2(\pi v) \tag{20.19}$$

where

$$x = u^v \qquad u = xy \tag{20.20}$$

$$y = u^{(1-v)} \qquad v = \frac{1}{2}\left(1 + \frac{\log(x/y)}{\log xy}\right) \tag{20.21}$$

Each term contributes 1 to the integral. The first term in the function is peaked along a cross aligned to the coordinates x and y, while the second term is peaked along the diagonal $x = y$.

The Jacobian is

$$\frac{\partial(x,y)}{\partial(u,v)} = |\log u| \tag{20.22}$$

⟨*MCI vamp2: test types*⟩+≡

```
  type, extends (mci_sampler_t) :: test_sampler_2_t
     real(default), dimension(:,:), allocatable :: x
     real(default), dimension(:), allocatable :: f
     real(default) :: val
   contains
   ⟨MCI vamp2: test sampler 2: TBP⟩
  end type test_sampler_2_t
```

Output: There is nothing stored inside, so just print an informative line.

⟨*MCI vamp2: test sampler 2: TBP*⟩≡

```
  procedure, public :: write => test_sampler_2_write
```

⟨*MCI vamp2: tests*⟩+≡

```
subroutine test_sampler_2_write (object, unit, testflag)
  class(test_sampler_2_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)") "Two-channel test sampler 2"
end subroutine test_sampler_2_write
```

Kinematics: compute x and Jacobians, given the input parameter array.

⟨*MCI vamp2: test sampler 2: TBP*⟩+≡

```
procedure, public :: compute => test_sampler_2_compute
```

⟨*MCI vamp2: tests*⟩+≡

```
subroutine test_sampler_2_compute (sampler, c, x_in)
  class(test_sampler_2_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default) :: xx, yy, uu, vv
  if (.not. allocated (sampler%x)) &
       allocate (sampler%x (size (x_in), 2))
  if (.not. allocated (sampler%f)) &
       allocate (sampler%f (2))
  select case (c)
  case (1)
     xx = x_in(1)
     yy = x_in(2)
     uu = xx * yy
     vv = (1 + log (xx/yy) / log (xx*yy)) / 2
  case (2)
     uu = x_in(1)
     vv = x_in(2)
     xx = uu ** vv
     yy = uu ** (1 - vv)
  end select
  sampler%val = (2 * sin (pi * xx) * sin (pi * yy)) ** 2 &
       + 2 * sin (pi * vv) ** 2
  sampler%f(1) = 1
  sampler%f(2) = abs (log (uu))
  sampler%x(:,1) = [xx, yy]
  sampler%x(:,2) = [uu, vv]
end subroutine test_sampler_2_compute
```

Evaluation: compute the function value. The output x parameter (only one channel) is identical to the input x, and the Jacobian is 1.

⟨*MCI vamp2: test sampler 2: TBP*⟩+≡

```
procedure, public :: evaluate => test_sampler_2_evaluate
```

⟨*MCI vamp2: tests*⟩+≡

```
subroutine test_sampler_2_evaluate (sampler, c, x_in, val, x, f)
  class(test_sampler_2_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
```

```
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    call sampler%compute (c, x_in)
    call sampler%fetch (val, x, f)
  end subroutine test_sampler_2_evaluate
```

The point is always valid.

⟨*MCI vamp2: test sampler 2: TBP*⟩+≡

```
  procedure, public :: is_valid => test_sampler_2_is_valid
```

⟨*MCI vamp2: tests*⟩+≡

```
  function test_sampler_2_is_valid (sampler) result (valid)
    class(test_sampler_2_t), intent(in) :: sampler
    logical :: valid
    valid = .true.
  end function test_sampler_2_is_valid
```

Rebuild: compute all but the function value.

⟨*MCI vamp2: test sampler 2: TBP*⟩+≡

```
  procedure, public :: rebuild => test_sampler_2_rebuild
```

⟨*MCI vamp2: tests*⟩+≡

```
  subroutine test_sampler_2_rebuild (sampler, c, x_in, val, x, f)
    class(test_sampler_2_t), intent(inout) :: sampler
    integer, intent(in) :: c
    real(default), dimension(:), intent(in) :: x_in
    real(default), intent(in) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    call sampler%compute (c, x_in)
    x = sampler%x
    f = sampler%f
  end subroutine test_sampler_2_rebuild
```

Extract the results.

⟨*MCI vamp2: test sampler 2: TBP*⟩+≡

```
  procedure, public :: fetch => test_sampler_2_fetch
```

⟨*MCI vamp2: tests*⟩+≡

```
  subroutine test_sampler_2_fetch (sampler, val, x, f)
    class(test_sampler_2_t), intent(in) :: sampler
    real(default), intent(out) :: val
    real(default), dimension(:,:), intent(out) :: x
    real(default), dimension(:), intent(out) :: f
    val = sampler%val
    x = sampler%x
    f = sampler%f
  end subroutine test_sampler_2_fetch
```

One-dimensional integration

Construct an integrator and use it for a one-dimensional sampler.

⟨*MCI vamp2: execute tests*⟩≡

```
call test (mci_vamp2_1, "mci_vamp2_1", "one-dimensional integral", u, results)
```

⟨*MCI vamp2: test declarations*⟩≡

```
public :: mci_vamp2_1
```

⟨*MCI vamp2: tests*⟩+≡

```
subroutine mci_vamp2_1 (u)
  integer, intent(in) :: u
  type(mci_vamp2_config_t) :: config
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable, target :: mci_sampler
  class(rng_t), allocatable :: rng
  type(string_t) :: filename

  write (u, "(A)") "* Test output: mci_vamp2_1"
  write (u, "(A)") "*   Purpose: integrate function in one dimension (single channel)"

  write (u, "(A)")
  write (u, "(A)") "* Initialise integrator"
  write (u, "(A)")

  allocate (mci_vamp2_t :: mci)
  call mci%set_dimensions (1, 1)

  filename = "mci_vamp2_1"
  select type (mci)
  type is (mci_vamp2_t)
     call mci%set_config (config)
     call mci%set_integrator_filename (filename)
  end select

  allocate (rng_stream_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)") "* Initialise instance"
  write (u, "(A)")

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  write (u, "(A)")
  write (u, "(A)") "* Initialise test sampler"
  write (u, "(A)")

  allocate (test_sampler_1_t :: mci_sampler)
  call mci_sampler%write (u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Integrate with n_calls = 1000"
    write (u, "(A)")  "   (lower precision to avoid"
    write (u, "(A)")  "      numerical noise)"
    write (u, "(A)")

    select type (mci)
    type is (mci_vamp2_t)
       call mci%add_pass ()
    end select
    call mci%integrate (mci_instance, mci_sampler, 1, 1000, pacify = .true.)
    call mci%write (u, .true.)

    write (u, "(A)")
    write (u, "(A)")  "* Contents of mci_instance:"
    write (u, "(A)")

    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)") "* Dump channel weights and grids"
    write (u, "(A)")

   select type (mci)
   type is (mci_vamp2_t)
      call mci%write_grids ()
   end select

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp2_1"

  end subroutine mci_vamp2_1
```

Multiple iterations

Construct an integrator and use it for a one-dimensional sampler. Integrate with five iterations without grid adaptation.

⟨*MCI vamp2: execute tests*⟩+≡

```
    call test (mci_vamp2_2, "mci_vamp2_2", &
         "multiple iterations", &
         u, results)
```

⟨*MCI vamp2: test declarations*⟩+≡

```
    public :: mci_vamp2_2
```

⟨*MCI vamp2: tests*⟩+≡

```
subroutine mci_vamp2_2 (u)
  type(mci_vamp2_config_t) :: config
  integer, intent(in) :: u
  class(mci_t), allocatable, target :: mci
  class(mci_instance_t), pointer :: mci_instance => null ()
  class(mci_sampler_t), allocatable :: sampler
  class(rng_t), allocatable :: rng
  type(string_t) :: filename

  write (u, "(A)")  "* Test output: mci_vamp2_2"
  write (u, "(A)")  "*   Purpose: integrate function in one dimension &
       &(single channel), but multiple iterations."

  write (u, "(A)")
  write (u, "(A)")  "* Initialize integrator, sampler, instance"
  write (u, "(A)")

  allocate (mci_vamp2_t :: mci)
  call mci%set_dimensions (1, 1)
  filename = "mci_vamp2_2"
  select type (mci)
  type is (mci_vamp2_t)
     call mci%set_config (config)
     call mci%set_integrator_filename (filename)
  end select

  allocate (rng_stream_t :: rng)
  call rng%init ()
  call mci%import_rng (rng)

  call mci%allocate_instance (mci_instance)
  call mci_instance%init (mci)

  allocate (test_sampler_1_t :: sampler)
  select type (sampler)
  type is (test_sampler_1_t)
     sampler%mode = 2
  end select
  call sampler%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 100"
  write (u, "(A)")

  select type (mci)
  type is (mci_vamp2_t)
     call mci%add_pass (adapt_grids = .false.)
  end select
  call mci%integrate (mci_instance, sampler, 3, 100)
  call mci%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Contents of mci_instance:"
  write (u, "(A)")
```

```
    call mci_instance%write (u)

    write (u, "(A)")
    write (u, "(A)") "* Dump channel weights and grids"
    write (u, "(A)")

   select type (mci)
   type is (mci_vamp2_t)
      call mci%write_grids ()
   end select

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call mci_instance%final ()
    call mci%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: mci_vamp2_2"

  end subroutine mci_vamp2_2
```

Grid adaptation

Construct an integrator and use it for a one-dimensional sampler. Integrate with three iterations and in-between grid adaptations.

⟨*MCI vamp2: execute tests*⟩+≡

```
    call test (mci_vamp2_3, "mci_vamp2_3", &
         "grid adaptation", &
         u, results)
```

⟨*MCI vamp2: test declarations*⟩+≡

```
    public :: mci_vamp2_3
```

⟨*MCI vamp2: tests*⟩+≡

```
    subroutine mci_vamp2_3 (u)
      integer, intent(in) :: u
      type(mci_vamp2_config_t) :: config
      class(mci_t), allocatable, target :: mci
      class(mci_instance_t), pointer :: mci_instance => null ()
      class(mci_sampler_t), allocatable :: sampler
      class(rng_t), allocatable :: rng
      type(string_t) :: filename

      write (u, "(A)")  "* Test output: mci_vamp2_3"
      write (u, "(A)")  "*   Purpose: integrate function in one dimension &
           &(single channel)"
      write (u, "(A)")  "*            and adapt grid"

      write (u, "(A)")
      write (u, "(A)")  "* Initialize integrator, sampler, instance"
      write (u, "(A)")
```

```
allocate (mci_vamp2_t :: mci)
call mci%set_dimensions (1, 1)
filename = "mci_vamp2_3"
select type (mci)
type is (mci_vamp2_t)
   call mci%set_integrator_filename (filename)
   call mci%set_config (config)
end select

allocate (rng_stream_t :: rng)
call rng%init ()
call mci%import_rng (rng)

call mci%allocate_instance (mci_instance)
call mci_instance%init (mci)

allocate (test_sampler_1_t :: sampler)
select type (sampler)
type is (test_sampler_1_t)
   sampler%mode = 2
end select
call sampler%write (u)

write (u, "(A)")
write (u, "(A)")  "* Integrate with n_it = 3 and n_calls = 100"
write (u, "(A)")

select type (mci)
type is (mci_vamp2_t)
   call mci%add_pass (adapt_grids = .true.)
end select
call mci%integrate (mci_instance, sampler, 3, 100)
call mci%write (u)

write (u, "(A)")
write (u, "(A)")  "* Contents of mci_instance:"
write (u, "(A)")

call mci_instance%write (u)

write (u, "(A)")
write (u, "(A)") "* Dump channel weights and grids"
write (u, "(A)")

select type (mci)
type is (mci_vamp2_t)
   call mci%write_grids ()
end select

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call mci_instance%final ()
```

```
      call mci%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: mci_vamp2_3"

    end subroutine mci_vamp2_3
```

20.7 Dispatch

⟨dispatch_mci.f90⟩≡

```
  ⟨File header⟩

  module dispatch_mci

  ⟨Use strings⟩
    use diagnostics
    use variables

    use mci_base
    use mci_midpoint
    use mci_vamp
    use mci_vamp2

  ⟨Standard module head⟩

  ⟨Dispatch mci: public⟩

  contains

  ⟨Dispatch mci: procedures⟩

  end module dispatch_mci
```

Allocate an integrator according to the variable `$integration_method`.

⟨*Dispatch mci: public*⟩≡

```
    public :: dispatch_mci_s
```

⟨*Dispatch mci: procedures*⟩≡

```
    subroutine dispatch_mci_s (mci, var_list, process_id, is_nlo)
      class(mci_t), allocatable, intent(inout) :: mci
      logical, intent(in), optional :: is_nlo
      type(var_list_t), intent(in) :: var_list
      type(string_t), intent(in) :: process_id
      type(string_t) :: run_id
      type(string_t) :: integration_method
      type(grid_parameters_t) :: grid_par
      type(history_parameters_t) :: history_par
      type(mci_vamp2_config_t) :: mci_vamp2_config
      logical :: rebuild_grids, check_grid_file, negative_weights, verbose
      logical :: dispatch_nlo = .false.
      type(string_t) :: grid_path
      if (present (is_nlo)) dispatch_nlo = is_nlo
      integration_method = &
```

```
        var_list%get_sval (var_str ("$integration_method"))
    select case (char (integration_method))
    case ("midpoint")
       allocate (mci_midpoint_t :: mci)
    case ("vamp", "default")
       call unpack_options_vamp ()
       allocate (mci_vamp_t :: mci)
       select type (mci)
       type is (mci_vamp_t)
          call mci%set_grid_parameters (grid_par)
          if (run_id /= "") then
             call mci%set_grid_filename (process_id, run_id)
          else
             call mci%set_grid_filename (process_id)
          end if
          grid_path = var_list%get_sval (var_str ("$grid_path"))
          if (grid_path /= "") call mci%prepend_grid_path (grid_path)
          call mci%set_history_parameters (history_par)
          call mci%set_rebuild_flag (rebuild_grids, check_grid_file)
          mci%negative_weights = negative_weights
          mci%verbose = verbose
       end select
    case ("vamp2")
       call unpack_options_vamp2 ()
       allocate (mci_vamp2_t :: mci)
       select type (mci)
       type is (mci_vamp2_t)
          call mci%set_config (mci_vamp2_config)
          if (run_id /= "") then
             call mci%set_integrator_filename (process_id, run_id)
          else
             call mci%set_integrator_filename (process_id)
          end if
          grid_path = var_list%get_sval (var_str ("$grid_path"))
          if (grid_path /= "") call mci%prepend_integrator_path (grid_path)
          call mci%set_rebuild_flag (rebuild_grids, check_grid_file)
          mci%verbose = verbose
       end select
    case default
       call msg_fatal ("Integrator '" &
            // char (integration_method) // "' not implemented")
    end select
  contains
    ⟨Dispatch mci s: procedures⟩
  end subroutine dispatch_mci_s
```

⟨*Dispatch mci s: procedures*⟩≡

```
  subroutine unpack_options_vamp ()
    grid_par%threshold_calls = &
         var_list%get_ival (var_str ("threshold_calls"))
    grid_par%min_calls_per_channel = &
         var_list%get_ival (var_str ("min_calls_per_channel"))
    grid_par%min_calls_per_bin = &
         var_list%get_ival (var_str ("min_calls_per_bin"))
```

```
  grid_par%min_bins = &
       var_list%get_ival (var_str ("min_bins"))
  grid_par%max_bins = &
       var_list%get_ival (var_str ("max_bins"))
  grid_par%stratified = &
       var_list%get_lval (var_str ("?stratified"))
  if (.not. dispatch_nlo) then
     grid_par%use_vamp_equivalences = &
          var_list%get_lval (var_str ("?use_vamp_equivalences"))
  else
     grid_par%use_vamp_equivalences = .false.
  end if
  grid_par%channel_weights_power = &
       var_list%get_rval (var_str ("channel_weights_power"))
  grid_par%accuracy_goal = &
       var_list%get_rval (var_str ("accuracy_goal"))
  grid_par%error_goal = &
       var_list%get_rval (var_str ("error_goal"))
  grid_par%rel_error_goal = &
       var_list%get_rval (var_str ("relative_error_goal"))
  history_par%global = &
       var_list%get_lval (var_str ("?vamp_history_global"))
  history_par%global_verbose = &
       var_list%get_lval (var_str ("?vamp_history_global_verbose"))
  history_par%channel = &
       var_list%get_lval (var_str ("?vamp_history_channels"))
  history_par%channel_verbose = &
       var_list%get_lval (var_str ("?vamp_history_channels_verbose"))
  verbose = &
       var_list%get_lval (var_str ("?vamp_verbose"))
  check_grid_file = &
       var_list%get_lval (var_str ("?check_grid_file"))
  run_id = &
       var_list%get_sval (var_str ("$run_id"))
  rebuild_grids = &
       var_list%get_lval (var_str ("?rebuild_grids"))
  negative_weights = &
       var_list%get_lval (var_str ("?negative_weights")) .or. dispatch_nlo
end subroutine unpack_options_vamp

subroutine unpack_options_vamp2 ()
  mci_vamp2_config%n_bins_max = &
       var_list%get_ival (var_str ("max_bins"))
  mci_vamp2_config%n_calls_min_per_channel = &
       var_list%get_ival (var_str ("min_calls_per_channel"))
  mci_vamp2_config%n_calls_threshold = &
       var_list%get_ival (var_str ("threshold_calls"))
  mci_vamp2_config%beta = &
       var_list%get_rval (var_str ("channel_weights_power"))
  mci_vamp2_config%stratified = &
       var_list%get_lval (var_str ("?stratified"))
  mci_vamp2_config%accuracy_goal = &
       var_list%get_rval (var_str ("accuracy_goal"))
  mci_vamp2_config%error_goal = &
```

```
          var_list%get_rval (var_str ("error_goal"))
      mci_vamp2_config%rel_error_goal = &
          var_list%get_rval (var_str ("relative_error_goal"))
      verbose = &
          var_list%get_lval (var_str ("?vamp_verbose"))
      check_grid_file = &
          var_list%get_lval (var_str ("?check_grid_file"))
      run_id = &
          var_list%get_sval (var_str ("$run_id"))
      rebuild_grids = &
          var_list%get_lval (var_str ("?rebuild_grids"))
    end subroutine unpack_options_vamp2
```

20.7.1 Unit tests

Test module, followed by the corresponding implementation module.

⟨dispatch_mci_ut.f90⟩≡

```
  ⟨File header⟩

  module dispatch_mci_ut
    use unit_tests
    use dispatch_mci_uti

  ⟨Standard module head⟩

  ⟨Dispatch mci: public test⟩

  contains

  ⟨Dispatch mci: test driver⟩

  end module dispatch_mci_ut
```

⟨dispatch_mci_uti.f90⟩≡

```
  ⟨File header⟩

  module dispatch_mci_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use variables
    use mci_base
    use mci_midpoint
    use mci_vamp
    use dispatch_mci

  ⟨Standard module head⟩

  ⟨Dispatch mci: test declarations⟩

  contains

  ⟨Dispatch mci: tests⟩
```

```
  end module dispatch_mci_uti
```

API: driver for the unit tests below.

⟨*Dispatch mci: public test*⟩≡

```
    public ::dispatch_mci_test
```

⟨*Dispatch mci: test driver*⟩≡

```
    subroutine dispatch_mci_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Dispatch mci: execute tests⟩
    end subroutine dispatch_mci_test
```

Select type: integrator core

⟨*Dispatch mci: execute tests*⟩≡

```
    call test (dispatch_mci_1, "dispatch_mci_1", &
         "integration method", &
         u, results)
```

⟨*Dispatch mci: test declarations*⟩≡

```
    public :: dispatch_mci_1
```

⟨*Dispatch mci: tests*⟩≡

```
    subroutine dispatch_mci_1 (u)
      integer, intent(in) :: u
      type(var_list_t) :: var_list
      class(mci_t), allocatable :: mci
      type(string_t) :: process_id

      write (u, "(A)")  "* Test output: dispatch_mci_1"
      write (u, "(A)")  "*   Purpose: select integration method"
      write (u, "(A)")

      call var_list%init_defaults (0)

      process_id = "dispatch_mci_1"

      write (u, "(A)")  "* Allocate MCI as midpoint_t"
      write (u, "(A)")

      call var_list%set_string (&
           var_str ("$integration_method"), &
           var_str ("midpoint"), is_known = .true.)
      call dispatch_mci_s (mci, var_list, process_id)
      select type (mci)
      type is (mci_midpoint_t)
         call mci%write (u)
      end select

      call mci%final ()
      deallocate (mci)
```

```
write (u, "(A)")
write (u, "(A)")  "* Allocate MCI as vamp_t"
write (u, "(A)")

call var_list%set_string (&
     var_str ("$integration_method"), &
     var_str ("vamp"), is_known = .true.)
call var_list%set_int (var_str ("threshold_calls"), &
     1, is_known = .true.)
call var_list%set_int (var_str ("min_calls_per_channel"), &
     2, is_known = .true.)
call var_list%set_int (var_str ("min_calls_per_bin"), &
     3, is_known = .true.)
call var_list%set_int (var_str ("min_bins"), &
     4, is_known = .true.)
call var_list%set_int (var_str ("max_bins"), &
     5, is_known = .true.)
call var_list%set_log (var_str ("?stratified"), &
     .false., is_known = .true.)
call var_list%set_log (var_str ("?use_vamp_equivalences"),&
     .false., is_known = .true.)
call var_list%set_real (var_str ("channel_weights_power"),&
     4._default, is_known = .true.)
call var_list%set_log (&
     var_str ("?vamp_history_global_verbose"), &
     .true., is_known = .true.)
call var_list%set_log (&
     var_str ("?vamp_history_channels"), &
     .true., is_known = .true.)
call var_list%set_log (&
     var_str ("?vamp_history_channels_verbose"), &
     .true., is_known = .true.)
call var_list%set_log (var_str ("?stratified"), &
     .false., is_known = .true.)

call dispatch_mci_s (mci, var_list, process_id)
select type (mci)
type is (mci_vamp_t)
   call mci%write (u)
   call mci%write_history_parameters (u)
end select

call mci%final ()
deallocate (mci)

write (u, "(A)")
write (u, "(A)")  "* Allocate MCI as vamp_t, allow for negative weights"
write (u, "(A)")

call var_list%set_string (&
     var_str ("$integration_method"), &
     var_str ("vamp"), is_known = .true.)
call var_list%set_log (var_str ("?negative_weights"), &
     .true., is_known = .true.)
```

```
  call dispatch_mci_s (mci, var_list, process_id)
  select type (mci)
  type is (mci_vamp_t)
     call mci%write (u)
     call mci%write_history_parameters (u)
  end select

  call mci%final ()
  deallocate (mci)

  call var_list%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: dispatch_mci_1"

end subroutine dispatch_mci_1
```

Chapter 21

Parton shower and interface to PYTHIA6

This is the code for the WHIZARD QCD parton shower for final state radiation (FSR) and initial state radiation (ISR) as well as the interface to the PYTHIA module for showering and hadronization.

21.1 Basics of the shower

⟨shower_base.f90⟩≡
  ⟨*File header*⟩

```
module shower_base
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩

```
  use io_units
  use constants
  use diagnostics
  use format_utils, only: write_separator
  use lorentz
  use particles
  use os_interface
  use rng_base
  use physics_defs
  use sm_physics, only: running_as_lam
  use particles
  use variables
  use model_data
  use pdf
  use tauola_interface
```

  ⟨*Standard module head*⟩

  ⟨*Shower base: public*⟩

  ⟨*Shower base: parameters*⟩

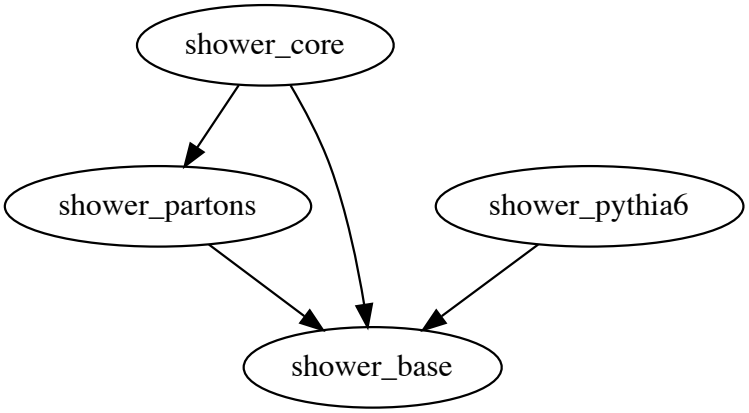

Figure 21.1: Module dependencies in `src/shower`.

```
⟨Shower base: types⟩

⟨Shower base: interfaces⟩

contains

⟨Shower base: procedures⟩

end module shower_base
```

21.1.1 Shower implementations

⟨*Shower base: public*⟩≡
```
public :: PS_WHIZARD, PS_PYTHIA6, PS_PYTHIA8, PS_UNDEFINED
```

⟨*Shower base: parameters*⟩≡
```
integer, parameter :: PS_UNDEFINED = 0
integer, parameter :: PS_WHIZARD = 1
integer, parameter :: PS_PYTHIA6 = 2
integer, parameter :: PS_PYTHIA8 = 3
```

A dictionary

⟨*Shower base: public*⟩+≡
```
public :: shower_method_of_string
```

⟨*Shower base: procedures*⟩≡
```
elemental function shower_method_of_string (string) result (i)
  integer :: i
  type(string_t), intent(in) :: string
  select case (char(string))
```

```
    case ("WHIZARD")
       i = PS_WHIZARD
    case ("PYTHIA6")
       i = PS_PYTHIA6
    case ("PYTHIA8")
       i = PS_PYTHIA8
    case default
       i = PS_UNDEFINED
    end select
  end function shower_method_of_string
```

⟨*Shower base: public*⟩+≡

```
  public :: shower_method_to_string
```

⟨*Shower base: procedures*⟩+≡

```
  elemental function shower_method_to_string (i) result (string)
    type(string_t) :: string
    integer, intent(in) :: i
    select case (i)
    case (PS_WHIZARD)
       string = "WHIZARD"
    case (PS_PYTHIA6)
       string = "PYTHIA6"
    case (PS_PYTHIA8)
       string = "PYTHIA8"
    case default
       string = "UNDEFINED"
    end select
  end function shower_method_to_string
```

21.1.2 Shower settings

These the general shower settings, the settings and parameters for the matching are defined in the corresponding matching modules. The width and the cutoff of the Gaussian primordial k_t distribution, `PARP(91)` and `PARP(93)`, in GeV, are called `isr_primordial_kt_width` and `isr_primordial_kt_cutoff` in `WHIZARD`. The parameter `MSTJ(45)` gives the maximum number of flavors in gluon decay to quarks, and is here called `max_n_flavors`.

The two parameters `isr_alpha_s_running] and [[fsr_alpha_s_running` decide whether to use constant or running $alpha_s$ in the form of the function $D_alpha_s(t)$ for the FSR and ISR (`MSTJ(44)`, `MSTP(64)`), respectively. The next parameter, `fixed_alpha_s` is the parameter `PARU(111)`, which sets the value for constant α_s, and the flag whether to use P_t-ordered ISR is `isr_pt_ordered`. From the entry `min_voirtuality` on, parameters have meanings both for the `PYTHIA` and `WHIZARD` parton shower(s), where `PYTHIA` values are denoted at the end of the line.

⟨*Shower base: public*⟩+≡

```
  public :: shower_settings_t
```

⟨*Shower base: types*⟩≡

```
  type :: shower_settings_t
     logical :: active = .false.
```

```
    logical :: isr_active = .false.
    logical :: fsr_active = .false.
    logical :: muli_active = .false.
    logical :: hadronization_active = .false.
    logical :: tau_dec = .false.
    logical :: verbose = .false.
    integer :: method = PS_UNDEFINED
    logical :: hadron_collision = .false.
    logical :: mlm_matching = .false.
    logical :: ckkw_matching = .false.
    logical :: powheg_matching = .false.
    type(string_t) :: pythia6_pygive
    real(default) :: min_virtuality = 1._default   ! PARJ(82)^2
    real(default) :: fsr_lambda = 0.29_default     ! PARP(72)
    real(default) :: isr_lambda = 0.29_default     ! PARP(61)
    integer :: max_n_flavors = 5                   ! MSTJ(45)
    logical :: isr_alpha_s_running = .true.        ! MSTP(64)
    logical :: fsr_alpha_s_running = .true.        ! MSTJ(44)
    real(default) :: fixed_alpha_s = 0.2_default   ! PARU(111)
    logical :: alpha_s_fudged = .true.
    logical :: isr_pt_ordered = .false.
    logical :: isr_angular_ordered = .true.        ! MSTP(62)
    real(default) :: isr_primordial_kt_width = 1.5_default  ! PARP(91)
    real(default) :: isr_primordial_kt_cutoff = 5._default  ! PARP(93)
    real(default) :: isr_z_cutoff = 0.999_default  ! 1-PARP(66)
    real(default) :: isr_minenergy = 2._default    ! PARP(65)
    real(default) :: isr_tscalefactor = 1._default
    logical :: isr_only_onshell_emitted_partons = .true.   ! MSTP(63)
  contains
  ⟨Shower base: shower settings: TBP⟩
end type shower_settings_t
```

Read in the shower settings (and flags whether matching and hadronization are switched on).

⟨*Shower base: shower settings: TBP*⟩≡

```
procedure :: init => shower_settings_init
```

⟨*Shower base: procedures*⟩+≡

```
subroutine shower_settings_init (settings, var_list)
  class(shower_settings_t), intent(out) :: settings
  type(var_list_t), intent(in) :: var_list

  settings%fsr_active = &
       var_list%get_lval (var_str ("?ps_fsr_active"))
  settings%isr_active = &
       var_list%get_lval (var_str ("?ps_isr_active"))
  settings%tau_dec = &
       var_list%get_lval (var_str ("?ps_taudec_active"))
  settings%muli_active = &
       var_list%get_lval (var_str ("?muli_active"))
  settings%hadronization_active = &
       var_list%get_lval (var_str ("?hadronization_active"))
  settings%mlm_matching = &
       var_list%get_lval (var_str ("?mlm_matching"))
```

```
    settings%ckkw_matching = &
         var_list%get_lval (var_str ("?ckkw_matching"))
    settings%powheg_matching = &
         var_list%get_lval (var_str ("?powheg_matching"))
    settings%method = shower_method_of_string ( &
         var_list%get_sval (var_str ("$shower_method")))
    settings%active = settings%isr_active .or. &
         settings%fsr_active .or. &
         settings%powheg_matching .or. &
         settings%muli_active .or. &
         settings%hadronization_active
    if (.not. settings%active)  return
    settings%verbose = &
         var_list%get_lval (var_str ("?shower_verbose"))
    settings%pythia6_pygive = &
         var_list%get_sval (var_str ("$ps_PYTHIA_PYGIVE"))
    settings%min_virtuality = &
         (var_list%get_rval (var_str ("ps_mass_cutoff"))**2)
    settings%fsr_lambda = &
         var_list%get_rval (var_str ("ps_fsr_lambda"))
    settings%isr_lambda = &
         var_list%get_rval (var_str ("ps_isr_lambda"))
    settings%max_n_flavors = &
         var_list%get_ival (var_str ("ps_max_n_flavors"))
    settings%isr_alpha_s_running = &
         var_list%get_lval (var_str ("?ps_isr_alpha_s_running"))
    settings%fsr_alpha_s_running = &
         var_list%get_lval (var_str ("?ps_fsr_alpha_s_running"))
    settings%fixed_alpha_s = &
         var_list%get_rval (var_str ("ps_fixed_alpha_s"))
    settings%isr_pt_ordered = &
         var_list%get_lval (var_str ("?ps_isr_pt_ordered"))
    settings%isr_angular_ordered = &
         var_list%get_lval (var_str ("?ps_isr_angular_ordered"))
    settings%isr_primordial_kt_width = &
         var_list%get_rval (var_str ("ps_isr_primordial_kt_width"))
    settings%isr_primordial_kt_cutoff = &
         var_list%get_rval (var_str ("ps_isr_primordial_kt_cutoff"))
    settings%isr_z_cutoff = &
         var_list%get_rval (var_str ("ps_isr_z_cutoff"))
    settings%isr_minenergy = &
         var_list%get_rval (var_str ("ps_isr_minenergy"))
    settings%isr_tscalefactor = &
         var_list%get_rval (var_str ("ps_isr_tscalefactor"))
    settings%isr_only_onshell_emitted_partons = &
         var_list%get_lval (&
         var_str ("?ps_isr_only_onshell_emitted_partons"))
  end subroutine shower_settings_init
```

⟨*Shower base: shower settings: TBP*⟩+≡

```
  procedure :: write => shower_settings_write
```

⟨*Shower base: procedures*⟩+≡

```
  subroutine shower_settings_write (settings, unit)
```

```
class(shower_settings_t), intent(in) :: settings
integer, intent(in), optional :: unit
integer :: u
u = given_output_unit (unit);  if (u < 0)  return
write (u, "(1x,A)")  "Shower settings:"
call write_separator (u)
write (u, "(1x,A)")  "Master switches:"
write (u, "(3x,A,1x,L1)") &
     "ps_isr_active                = ", settings%isr_active
write (u, "(3x,A,1x,L1)") &
     "ps_fsr_active                = ", settings%fsr_active
write (u, "(3x,A,1x,L1)") &
     "ps_tau_dec                   = ", settings%tau_dec
write (u, "(3x,A,1x,L1)") &
     "muli_active                  = ", settings%muli_active
write (u, "(3x,A,1x,L1)") &
     "hadronization_active         = ", settings%hadronization_active
write (u, "(1x,A)")  "General settings:"
if (settings%isr_active .or. settings%fsr_active) then
   write (u, "(3x,A)") &
        "method                       =  " // &
        char (shower_method_to_string (settings%method))
   write (u, "(3x,A,1x,L1)") &
        "shower_verbose               = ", settings%verbose
   write (u, "(3x,A,ES19.12)") &
        "ps_mass_cutoff               = ", &
        sqrt (abs (settings%min_virtuality))
   write (u, "(3x,A,1x,I1)") &
        "ps_max_n_flavors             = ", settings%max_n_flavors
else
   write (u, "(3x,A)") " [ISR and FSR off]"
end if
if (settings%isr_active) then
   write (u, "(1x,A)")  "ISR settings:"
   write (u, "(3x,A,1x,L1)") &
        "ps_isr_pt_ordered            = ", settings%isr_pt_ordered
   write (u, "(3x,A,ES19.12)") &
        "ps_isr_lambda                = ", settings%isr_lambda
   write (u, "(3x,A,1x,L1)") &
        "ps_isr_alpha_s_running       = ", settings%isr_alpha_s_running
   write (u, "(3x,A,ES19.12)") &
        "ps_isr_primordial_kt_width   = ", settings%isr_primordial_kt_width
   write (u, "(3x,A,ES19.12)") &
        "ps_isr_primordial_kt_cutoff  = ", &
        settings%isr_primordial_kt_cutoff
   write (u, "(3x,A,ES19.12)") &
        "ps_isr_z_cutoff              = ", settings%isr_z_cutoff
   write (u, "(3x,A,ES19.12)") &
        "ps_isr_minenergy             = ", settings%isr_minenergy
   write (u, "(3x,A,ES19.12)") &
        "ps_isr_tscalefactor          = ", settings%isr_tscalefactor
else if (settings%fsr_active) then
   write (u, "(3x,A)") " [ISR off]"
end if
```

```
  if (settings%fsr_active) then
     write (u, "(1x,A)")  "FSR settings:"
     write (u, "(3x,A,ES19.12)") &
          "ps_fsr_lambda                = ", settings%fsr_lambda
     write (u, "(3x,A,1x,L1)") &
          "ps_fsr_alpha_s_running       = ", settings%fsr_alpha_s_running
  else if (settings%isr_active) then
     write (u, "(3x,A)") " [FSR off]"
  end if
  write (u, "(1x,A)")  "Matching Settings:"
  write (u, "(3x,A,1x,L1)") &
       "mlm_matching                 = ", settings%mlm_matching
  write (u, "(3x,A,1x,L1)") &
       "ckkw_matching                = ", settings%ckkw_matching
  write (u, "(1x,A)")  "PYTHIA6 specific settings:"
  write (u, "(3x,A,A,A)") &
       "ps_PYTHIA_PYGIVE             = '", &
       char(settings%pythia6_pygive), "'"
end subroutine shower_settings_write
```

21.1.3 Abstract Shower Type

Any parton shower implementation will use random numbers to generate emissions.

⟨*Shower base: public*⟩+≡

```
public :: shower_base_t
```

⟨*Shower base: types*⟩+≡

```
type, abstract :: shower_base_t
   class(rng_t), allocatable :: rng
   type(string_t) :: name
   type(pdf_data_t) :: pdf_data
   type(shower_settings_t) :: settings
   type(taudec_settings_t) :: taudec_settings
 contains
 ⟨Shower base: shower base: TBP⟩
end type shower_base_t
```

⟨*Shower base: shower base: TBP*⟩≡

```
procedure :: write_msg => shower_base_write_msg
```

⟨*Shower base: procedures*⟩+≡

```
subroutine shower_base_write_msg (shower)
  class(shower_base_t), intent(inout) :: shower
  call msg_message ("Shower: Using " // char(shower%name) // " shower")
end subroutine shower_base_write_msg
```

⟨*Shower base: shower base: TBP*⟩+≡

```
procedure :: import_rng => shower_base_import_rng
```

⟨*Shower base: procedures*⟩+≡

```
pure subroutine shower_base_import_rng (shower, rng)
  class(shower_base_t), intent(inout) :: shower
  class(rng_t), intent(inout), allocatable :: rng
  call move_alloc (from = rng, to = shower%rng)
end subroutine shower_base_import_rng
```

Shower implementations need to know the overall settings as well as `pdf_data_t` if ISR needs to be simulated.

⟨*Shower base: shower base: TBP*⟩+≡

```
procedure (shower_base_init), deferred :: init
```

⟨*Shower base: interfaces*⟩≡

```
abstract interface
  subroutine shower_base_init (shower, settings, taudec_settings, pdf_data)
    import
    class(shower_base_t), intent(out) :: shower
    type(shower_settings_t), intent(in) :: settings
    type(taudec_settings_t), intent(in) :: taudec_settings
    type(pdf_data_t), intent(in) :: pdf_data
  end subroutine shower_base_init
 end interface
```

⟨*Shower base: shower base: TBP*⟩+≡

```
procedure (shower_base_prepare_new_event), deferred :: prepare_new_event
```

⟨*Shower base: interfaces*⟩+≡

```
abstract interface
   subroutine shower_base_prepare_new_event &
          (shower)
     import
     class(shower_base_t), intent(inout) :: shower
   end subroutine shower_base_prepare_new_event
end interface
```

⟨*Shower base: shower base: TBP*⟩+≡

```
procedure (shower_base_import_particle_set), deferred :: import_particle_set
```

⟨*Shower base: interfaces*⟩+≡

```
abstract interface
   subroutine shower_base_import_particle_set &
          (shower, particle_set, os_data, scale)
     import
     class(shower_base_t), target, intent(inout) :: shower
     type(particle_set_t), intent(in) :: particle_set
     type(os_data_t), intent(in) :: os_data
     real(default), intent(in) :: scale
   end subroutine shower_base_import_particle_set
end interface
```

⟨*Shower base: shower base: TBP*⟩+≡

```
procedure (shower_base_generate_emissions), deferred :: generate_emissions
```

⟨*Shower base: interfaces*⟩+≡

```
abstract interface
   subroutine shower_base_generate_emissions &
          (shower, valid, number_of_emissions)
    import
    class(shower_base_t), intent(inout), target :: shower
     logical, intent(out) :: valid
    integer, optional, intent(in) :: number_of_emissions
  end subroutine shower_base_generate_emissions
 end interface
```

⟨*Shower base: shower base: TBP*⟩+≡

```
procedure (shower_base_make_particle_set), deferred :: make_particle_set
```

⟨*Shower base: interfaces*⟩+≡

```
abstract interface
   subroutine shower_base_make_particle_set &
       (shower, particle_set, model, model_hadrons)
     import
     class(shower_base_t), intent(in) :: shower
     type(particle_set_t), intent(inout) :: particle_set
     class(model_data_t), intent(in), target :: model
     class(model_data_t), intent(in), target :: model_hadrons
   end subroutine shower_base_make_particle_set
end interface
```

⟨*Shower base: shower base: TBP*⟩+≡

```
procedure (shower_base_get_final_colored_ME_momenta), deferred :: &
     get_final_colored_ME_momenta
```

⟨*Shower base: interfaces*⟩+≡

```
abstract interface
   subroutine shower_base_get_final_colored_ME_momenta &
          (shower, momenta)
     import
     class(shower_base_t), intent(in) :: shower
     type(vector4_t), dimension(:), allocatable, intent(out) :: momenta
   end subroutine shower_base_get_final_colored_ME_momenta
end interface
```

21.1.4 Additional parameters

These parameters are the cut-off scale t_{cut}, given in GeV^2 (`PARJ(82)`), the cut-off scale for the P_t^2-ordered shower in GeV^2, and the two shower parameters `PARP(72)` and `PARP(61)`, respectively.

⟨*Shower base: parameters*⟩+≡

```
real(default), public :: D_min_scale = 0.5_default
```

Treating either u and d, or all quarks except t as massless:

⟨*Shower base: parameters*⟩+≡

```
logical, public :: treat_light_quarks_massless = .true.
logical, public :: treat_duscb_quarks_massless = .false.
```

Temporary parameters for the P_t-ordered shower:

⟨*Shower base: parameters*⟩+≡
```
real(default), public :: scalefactor1 = 0.02_default
real(default), public :: scalefactor2 = 0.02_default
```

⟨*Shower base: public*⟩+≡
```
public :: D_alpha_s_isr
public :: D_alpha_s_fsr
```

⟨*Shower base: procedures*⟩+≡
```
function D_alpha_s_isr (tin, settings) result (alpha_s)
  real(default), intent(in) :: tin
  type(shower_settings_t), intent(in) :: settings
  real(default) :: min_virtuality, d_constalpha_s, d_lambda_isr
  integer :: d_nf
  real(default) :: t
  real(default) :: alpha_s
  min_virtuality = settings%min_virtuality
  d_lambda_isr = settings%isr_lambda
  d_constalpha_s = settings%fixed_alpha_s
  d_nf = settings%max_n_flavors
  if (settings%alpha_s_fudged) then
     t = max (max (0.1_default * min_virtuality, &
                   1.1_default * d_lambda_isr**2), abs(tin))
  else
     t = abs(tin)
  end if
  if (settings%isr_alpha_s_running) then
     alpha_s = running_as_lam (number_of_flavors(t, d_nf, min_virtuality), &
          sqrt(t), d_lambda_isr, 0)
  else
     alpha_s = d_constalpha_s
  end if
end function D_alpha_s_isr

function D_alpha_s_fsr (tin, settings) result (alpha_s)
  real(default), intent(in) :: tin
  type(shower_settings_t), intent(in) :: settings
  real(default) :: min_virtuality, d_lambda_fsr, d_constalpha_s
  integer :: d_nf
  real(default) :: t
  real(default) :: alpha_s
  min_virtuality = settings%min_virtuality
  d_lambda_fsr = settings%fsr_lambda
  d_constalpha_s = settings%fixed_alpha_s
  d_nf = settings%max_n_flavors
  if (settings%alpha_s_fudged) then
     t = max (max (0.1_default * min_virtuality, &
                   1.1_default * d_lambda_fsr**2), abs(tin))
  else
     t = abs(tin)
  end if
  if (settings%fsr_alpha_s_running) then
     alpha_s = running_as_lam (number_of_flavors (t, d_nf, min_virtuality), &
          sqrt(t), d_lambda_fsr, 0)
```

```
    else
       alpha_s = d_constalpha_s
    end if
  end function D_alpha_s_fsr
```

Mass and mass squared selection functions. All masses are in GeV. Light quarks are assumed to be ordered, $m_1 < m_2 < m_3 \ldots$, and they get current masses, not elementary ones. Mesons and baryons other than proton and neutron are needed as beam-remnants. Particles with PDG number zero are taken massless, as well as proper beam remnants and any other particles.

⟨*Shower base: public*⟩+≡

```
  public :: mass_type
  public :: mass_squared_type
```

⟨*Shower base: procedures*⟩+≡

```
  elemental function mass_type (type, m2_default) result (mass)
    integer, intent(in) :: type
    real(default), intent(in) :: m2_default
    real(default) :: mass
    mass = sqrt (mass_squared_type (type, m2_default))
  end function mass_type

  elemental function mass_squared_type (type, m2_default) result (mass2)
    integer, intent(in) :: type
    real(default), intent(in) :: m2_default
    real(default) :: mass2
    select case (abs (type))
    !!!    case (1,2)
    !!!       if (treat_light_quarks_massless .or. &
    !!!            treat_duscb_quarks_massless) then
    !!!          mass2 = zero
    !!!       else
    !!!          mass2 = 0.330_default**2
    !!!       end if
    !!!    case (3)
    !!!       if (treat_duscb_quarks_massless) then
    !!!          mass2 = zero
    !!!       else
    !!!          mass2 = 0.500_default**2
    !!!       end if
    !!!    case (4)
    !!!       if (treat_duscb_quarks_massless) then
    !!!          mass2 = zero
    !!!       else
    !!!          mass2 = 1.500_default**2
    !!!       end if
    !!!    case (5)
    !!!       if (treat_duscb_quarks_massless) then
    !!!          mass2 = zero
    !!!       else
    !!!          mass2 = 4.800_default**2
    !!!       end if
    !!!    case (GLUON)
    !!!       mass2 = zero
```

```
    case (NEUTRON)
       mass2 = 0.939565_default**2
    case (PROTON)
       mass2 = 0.93827_default**2
    case (DPLUS)
       mass2 = 1.86960_default**2
    case (D0)
       mass2 = 1.86483_default**2
    case (B0)
       mass2 = 5.27950_default**2
    case (BPLUS)
       mass2 = 5.27917_default**2
    case (DELTAPLUSPLUS)
       mass2 = 1.232_default**2
    case (SIGMA0)
       mass2 = 1.192642_default**2
    case (SIGMAPLUS)
       mass2 = 1.18937_default**2
    case (SIGMACPLUS)
       mass2 = 2.4529_default**2
    case (SIGMACPLUSPLUS)
       mass2 = 2.45402_default**2
    case (SIGMAB0)
       mass2 = 5.8152_default**2
    case (SIGMABPLUS)
       mass2 = 5.8078_default**2
    case (BEAM_REMNANT)
       mass2 = zero !!! don't know how to handle the beamremnant
    case default
       mass2 = m2_default
    end select
  end function mass_squared_type
```

The number of flavors active at a certain scale (virtuality) t.

⟨*Shower base: public*⟩+≡

```
  public :: number_of_flavors
```

⟨*Shower base: procedures*⟩+≡

```
  elemental function number_of_flavors (t, d_nf, min_virtuality) result (nr)
    real(default), intent(in) :: t, min_virtuality
    integer, intent(in) :: d_nf
    real(default) :: nr
    integer :: i
    nr = 0
    if (t < min_virtuality) return   ! arbitrary cut off
    ! TODO: do i = 1, min (max (3, d_nf), 6)
    do i = 1, min (3, d_nf)
    !!! to do: take heavier quarks(-> cuts on allowed costheta in g->qq)
    !!!        into account
       if ((four * mass_squared_type (i, zero) + min_virtuality) < t ) then
          nr = i
       else
          exit
       end if
```

```
    end do
  end function number_of_flavors
```

21.1.5 Unit tests

Test module, followed by the corresponding implementation module.

⟨shower_base_ut.f90⟩≡

```
⟨File header⟩

module shower_base_ut
  use unit_tests
  use shower_base_uti

⟨Standard module head⟩

⟨Shower base: public test⟩

contains

⟨Shower base: test driver⟩

end module shower_base_ut
```

⟨shower_base_uti.f90⟩≡

```
⟨File header⟩

module shower_base_uti

⟨Use kinds⟩
⟨Use strings⟩
  use format_utils, only: write_separator
  use variables
  use shower_base

⟨Standard module head⟩

⟨Shower base: test declarations⟩

contains

⟨Shower base: tests⟩

end module shower_base_uti
```

API: driver for the unit tests below.

⟨*Shower base: public test*⟩≡

```
  public :: shower_base_test
```

⟨*Shower base: test driver*⟩≡

```
  subroutine shower_base_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨Shower base: execute tests⟩
  end subroutine shower_base_test
```

Shower settings

This test dispatches an `shower_settings` object, which is used to steer the initial and final state showers.

⟨*Shower base: execute tests*⟩≡
```
call test (shower_base_1, "shower_base_1", &
     "Shower settings", &
     u, results)
```

⟨*Shower base: test declarations*⟩≡
```
public :: shower_base_1
```

⟨*Shower base: tests*⟩≡
```
subroutine shower_base_1 (u)
  integer, intent(in) :: u
  type(var_list_t) :: var_list
  type(shower_settings_t) :: shower_settings

  write (u, "(A)")  "* Test output: shower_base_1"
  write (u, "(A)")  "*   Purpose: setting ISR/FSR shower"
  write (u, "(A)")

  write (u, "(A)")  "* Default settings"
  write (u, "(A)")

  call var_list%init_defaults (0)
  call var_list%set_log (var_str ("?alpha_s_is_fixed"), &
       .true., is_known = .true.)
  call shower_settings%init (var_list)
  call write_separator (u)
  call shower_settings%write (u)
  call write_separator (u)

  write (u, "(A)")
  write (u, "(A)")  "* Switch on ISR/FSR showers, hadronization"
  write (u, "(A)")  "      and MLM matching"
  write (u, "(A)")

  call var_list%set_string (var_str ("$shower_method"), &
       var_str ("PYTHIA6"), is_known = .true.)
  call var_list%set_log (var_str ("?ps_fsr_active"), &
       .true., is_known = .true.)
  call var_list%set_log (var_str ("?ps_isr_active"), &
       .true., is_known = .true.)
  call var_list%set_log (var_str ("?hadronization_active"), &
       .true., is_known = .true.)
  call var_list%set_log (var_str ("?mlm_matching"), &
       .true., is_known = .true.)
  call var_list%set_int &
       (var_str ("ps_max_n_flavors"), 4, is_known = .true.)
  call var_list%set_real &
```

```
          (var_str ("ps_isr_z_cutoff"), 0.1234_default, &
           is_known=.true.)
     call var_list%set_real (&
          var_str ("mlm_etamax"), 3.456_default, is_known=.true.)
     call var_list%set_string (&
          var_str ("$ps_PYTHIA_PYGIVE"), var_str ("abcdefgh"), is_known=.true.)
     call shower_settings%init (var_list)
     call write_separator (u)
     call shower_settings%write (u)
     call write_separator (u)

     call var_list%final ()

     write (u, "(A)")
     write (u, "(A)")  "* Test output end: shower_base_1"

   end subroutine shower_base_1
```

21.2 Parton module for the shower

⟨shower_partons.f90⟩≡

```
  ⟨File header⟩

  module shower_partons

  ⟨Use kinds with double⟩
    use io_units
    use constants
    use system_defs, only: TAB
    use diagnostics
    use physics_defs
    use lorentz
    use sm_physics
    use particles
    use flavors
    use colors
    use subevents
    use model_data
    use shower_base
    use rng_base

  ⟨Standard module head⟩

  ⟨Shower partons: public⟩

  ⟨Shower partons: types⟩

  contains

  ⟨Shower partons: procedures⟩

  end module shower_partons
```

21.2.1 The basic type defintions

The type parton_t defines a parton for the shower. The x value of the parton is only needed for spacelike showers. The pointer initial is only needed for partons in initial showers, it points to the hadron the parton is coming from. An auxiliary value for the P_t-ordered ISR is aux_pt. Then, there are two auxiliary entries for the clustering of CKKW pseudo weights and CKKW matching, ckkwlabel and ckkwscale. In order to make shower settings available to all operations on the shower partons, we endow the parton_t type with a pointer to shower_settings_t.

⟨*Shower partons: public*⟩≡
```
public :: parton_t
```

⟨*Shower partons: types*⟩≡
```
type :: parton_t
   integer :: nr = 0
   integer :: type = 0
   type(shower_settings_t), pointer :: settings => null()
   type(vector4_t) :: momentum = vector4_null
   real(default) :: t  = zero
   real(default) :: mass2  = zero
   real(default) :: scale = zero
   real(default) :: z = zero
   real(default) :: costheta = zero
   real(default) :: x = zero
   logical :: simulated = .false.
   logical :: belongstoFSR = .true.
   logical :: belongstointeraction = .false.
   type(parton_t), pointer :: parent => null ()
   type(parton_t), pointer :: child1 => null ()
   type(parton_t), pointer :: child2 => null ()
   type(parton_t), pointer :: initial => null ()
   integer :: c1 = 0, c2 = 0
   integer :: aux_pt = 0
   integer :: ckkwlabel = 0
   real(default) :: ckkwscale = zero
   integer :: ckkwtype = -1
   integer :: interactionnr = 0
 contains
 ⟨Shower partons: parton: TBP⟩
end type parton_t
```

⟨*Shower partons: public*⟩+≡
```
public :: parton_pointer_t
```

⟨*Shower partons: types*⟩+≡
```
type :: parton_pointer_t
   type(parton_t), pointer :: p => null ()
end type parton_pointer_t
```

21.2.2 Routines

⟨*Shower partons: parton: TBP*⟩≡

```
    procedure :: to_particle => parton_to_particle
```

⟨*Shower partons: procedures*⟩≡

```
    function parton_to_particle (parton, model, from_hard_int) result (particle)
      type(particle_t) :: particle
      class(parton_t), intent(in) :: parton
      class(model_data_t), pointer, intent(in) :: model
      logical, intent(in), optional :: from_hard_int
      integer :: col, anti_col
      call parton%to_color (col, anti_col, from_hard_int)
      call particle%init (parton%to_status (from_hard_int), parton%type, &
           model, col, anti_col, parton%momentum)
    end function parton_to_particle
```

⟨*Shower partons: public*⟩+≡

```
    public :: parton_of_particle
```

⟨*Shower partons: procedures*⟩+≡

```
   ! pure
    function parton_of_particle (particle, nr) result (parton)
      type(parton_t) :: parton
      type(particle_t), intent(in) :: particle
      integer, intent(in) :: nr
      integer, dimension(2) :: col_array
      parton%nr = nr
      parton%momentum = particle%p
      parton%t = particle%p2
      parton%type = particle%flv%get_pdg ()
      col_array = particle%get_color ()
      parton%c1 = col_array (1)
      parton%c2 = col_array (2)
      parton%interactionnr = 1
      parton%mass2 = particle%flv%get_mass () ** 2
    end function parton_of_particle
```

⟨*Shower partons: parton: TBP*⟩+≡

```
    procedure :: to_status => parton_to_status
```

⟨*Shower partons: procedures*⟩+≡

```
    pure function parton_to_status (parton, from_hard_int) result (status)
      integer :: status
      class(parton_t), intent(in) :: parton
      logical, intent(in), optional :: from_hard_int
      logical :: fhi
      fhi = .false.; if (present (from_hard_int))  fhi = from_hard_int
      if (fhi .or. parton%is_colored ()) then
         if (associated (parton%initial) .and. .not. parton%belongstoFSR) then
            status = PRT_INCOMING
         else
            status = PRT_OUTGOING
         end if
      else
         status = PRT_BEAM_REMNANT
      end if
    end function parton_to_status
```

⟨*Shower partons: parton: TBP*⟩+≡

```
procedure :: to_color => parton_to_color
```

⟨*Shower partons: procedures*⟩+≡

```
pure subroutine parton_to_color (parton, c1, c2, from_hard_int)
  class(parton_t), intent(in) :: parton
  integer, intent(out) :: c1, c2
  logical, intent(in), optional :: from_hard_int
  logical :: fhi
  fhi = .false.; if (present (from_hard_int))  fhi = from_hard_int
  c1 = 0
  c2 = 0
  if (parton%is_colored ()) then
     if (fhi) then
        if (parton%c1 /= 0) c1 = parton%c1
        if (parton%c2 /= 0) c2 = parton%c2
     else
        if (parton%c1 /= 0) c1 = 500 + parton%c1
        if (parton%c2 /= 0) c2 = 500 + parton%c2
     end if
  end if
end subroutine parton_to_color
```

⟨*Shower partons: public*⟩+≡

```
public :: parton_copy
```

⟨*Shower partons: procedures*⟩+≡

```
subroutine parton_copy (prt1, prt2)
  type(parton_t), intent(in) :: prt1
  type(parton_t), intent(out) :: prt2
  if (associated (prt1%settings))  prt2%settings => prt1%settings
  prt2%nr = prt1%nr
  prt2%type = prt1%type
  prt2%momentum = prt1%momentum
  prt2%t = prt1%t
  prt2%mass2 = prt1%mass2
  prt2%scale = prt1%scale
  prt2%z = prt1%z
  prt2%costheta = prt1%costheta
  prt2%x = prt1%x
  prt2%simulated = prt1%simulated
  prt2%belongstoFSR = prt1%belongstoFSR
  prt2%belongstointeraction = prt1%belongstointeraction
  prt2%interactionnr = prt1%interactionnr
  if (associated (prt1%parent))  prt2%parent  => prt1%parent
  if (associated (prt1%child1))  prt2%child1  => prt1%child1
  if (associated (prt1%child2))  prt2%child2  => prt1%child2
  if (associated (prt1%initial)) prt2%initial => prt1%initial
  prt2%c1 = prt1%c1
  prt2%c2 = prt1%c2
  prt2%aux_pt = prt1%aux_pt
end subroutine parton_copy
```

This returns the angle between the daughters assuming them to be massless.

⟨*Shower partons: parton: TBP*⟩+≡
```
procedure :: get_costheta => parton_get_costheta
```

⟨*Shower partons: procedures*⟩+≡
```
elemental function parton_get_costheta (prt) result (costheta)
  class(parton_t), intent(in) :: prt
  real(default) :: costheta
  real(default) :: denom
  denom = two * prt%z * (one - prt%z) * prt%momentum%p(0)**2
  if (denom > eps0) then
     costheta = one - prt%t / denom
  else
     costheta = - one
  end if
end function parton_get_costheta
```

The same for massive daughters.

⟨*Shower partons: parton: TBP*⟩+≡
```
procedure :: get_costheta_mass => parton_get_costheta_mass
```

⟨*Shower partons: procedures*⟩+≡
```
elemental function parton_get_costheta_mass (prt) result (costheta)
  class(parton_t), intent(in) :: prt
  real(default) :: costheta, sqrt12
  if (prt%is_branched ()) then
     if (prt%child1%simulated .and. &
         prt%child2%simulated) then
        sqrt12 = sqrt (max (zero, (prt%z)**2 * prt%momentum%p(0)**2 &
                                - prt%child1%t)) * &
                 sqrt (max (zero, (one - prt%z)**2 * prt%momentum%p(0)**2 &
                                - prt%child2%t))
        if (sqrt12 > eps0) then
           costheta = (prt%t - prt%child1%t - prt%child2%t - &
                two * prt%z * (one - prt%z) * prt%momentum%p(0)**2) / &
                (- two * sqrt12)
           return
        end if
     end if
  end if
  costheta = prt%get_costheta ()
end function parton_get_costheta_mass
```

This function returns the angle between the momentum vectors of the parton and first daughter. This is only used for debugging.

⟨*Shower partons: parton: TBP*⟩+≡
```
procedure :: get_costheta_motherfirst => parton_get_costheta_motherfirst
```

⟨*Shower partons: procedures*⟩+≡
```
elemental function parton_get_costheta_motherfirst (prt) result (costheta)
  class(parton_t), intent(in) :: prt
  real(default) :: costheta
  if (prt%is_branched ()) then
     if ((prt%child1%simulated .or. &
```

```
            prt%child1%is_final () .or. &
            prt%child1%is_branched ()) .and. &
            (prt%child2%simulated .or. &
             prt%child2%is_final () .or. &
             prt%child2%is_branched ())) then
          costheta = enclosed_angle_ct (prt%momentum, prt%child1%momentum)
          return
       end if
    end if
    costheta = - two
  end function parton_get_costheta_motherfirst
```

Get the parton velocities.

⟨*Shower partons: parton: TBP*⟩+≡

```
  procedure :: get_beta => parton_get_beta
```

⟨*Shower partons: procedures*⟩+≡

```
  pure function get_beta (t,E) result (beta)
    real(default), intent(in) :: t,E
    real(default) :: beta
    beta = sqrt (max (tiny_07, one - t /(E**2)))
  end function get_beta

  elemental function parton_get_beta (prt) result (beta)
    class(parton_t), intent(in) :: prt
    real(default) :: beta
    beta = sqrt (max (tiny_07, one - prt%t / prt%momentum%p(0)**2))
  end function parton_get_beta
```

Write routine.

⟨*Shower partons: parton: TBP*⟩+≡

```
  procedure :: write => parton_write
```

⟨*Shower partons: procedures*⟩+≡

```
  subroutine parton_write (prt, unit)
    class(parton_t), intent(in) :: prt
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit); if (u < 0) return

    write (u, "(1x,7A)") "Shower parton <nr>", TAB, "<type>", TAB // TAB, &
         "<parent>", TAB, "<mom(0:3)>"
    write (u, "(2x,I5,3A)", advance = "no")  prt%nr, TAB, TAB, TAB
    if (prt%is_final ()) then
       write (u, "(1x,I5,1x,A)", advance = "no") prt%type, TAB // TAB
    else
       write (u, "('[',I5,']',A)", advance = "no") prt%type, TAB // TAB
    end if
    if (associated (prt%parent)) then
       write (u, "(I5,A)", advance = "no") prt%parent%nr, TAB // TAB
    else
       write (u, "(5x,2A)", advance = "no") TAB, TAB
    end if
    write (u, "(4(ES12.5,A))") prt%momentum%p(0), TAB, &
```

```
                          prt%momentum%p(1), TAB, &
                          prt%momentum%p(2), TAB, &
                          prt%momentum%p(3)
write (u, "(1x,9A)") "<p4square>", TAB // TAB, "<t>", TAB // TAB, &
     "<scale>", TAB // TAB, "<c1>", TAB, "<c2>", TAB, "<mass2>"
write (u, "(1x,3(ES12.5,A))", advance = "no") &
     prt%momentum ** 2, TAB // TAB, prt%t, TAB, prt%scale, TAB, prt%mass2
write (u, "(2(I4,A))") prt%c1, TAB, prt%c2, TAB
if (prt%is_branched ()) then
   if (prt%belongstoFSR) then
      write (u, "(1x,9A)") "costheta(prt)", TAB, &
           "costheta_correct(prt)", TAB, &
           "prt%costheta", TAB, "prt%z", TAB, &
           "costheta_motherfirst(prt)"
      write (u, "(1X,5(ES12.5,A))") &
           prt%get_costheta (), TAB, &
           prt%get_costheta_mass (), TAB // TAB, &
           prt%costheta, TAB, prt%z, TAB, &
           prt%get_costheta_motherfirst (), TAB
   else
      write (u, "(1x,9A)") "prt%z", TAB, "prt%x", TAB, &
           "costheta_correct(prt)", TAB, &
           "prt%costheta", TAB, &
           "costheta_motherfirst(prt)"
      write (u, "(1X,5(ES12.5,A))") &
           prt%z, TAB, prt%x, TAB, &
           prt%get_costheta_mass (), TAB, &
           prt%costheta, TAB, &
           prt%get_costheta_motherfirst (), TAB
   end if
else
   if (prt%belongstoFSR) then
      write (u, "(1X,A)") "not branched."
   else
      write (u, "(1X,A,ES12.5)") "not branched. x = ",  prt%x
   end if
end if
write (u, "(A)", advance = "no") " Parton"
if (prt%belongstoFSR) then
   write (u, "(A)", advance = "no")  " is FSR,"
else
   if (associated (prt%initial)) then
      write (u, "(A,I1)", advance = "no")  " from hadron,", prt%initial%nr
   else
      write (u, "(A)", advance = "no")  ""
   end if
end if
if (prt%is_final ()) then
   write (u, "(A)", advance = "no")  " is final,"
else
   write (u, "(A)", advance = "no")  ""
end if
if (prt%simulated) then
   write (u, "(A)", advance = "no")  " is simulated,"
```

```
      else
         write (u, "(A)", advance = "no")  ""
      end if
      if (associated (prt%child1) .and. associated (prt%child2)) then
         write (u, "(A,2(I5),A)", advance = "no") &
              " has children: ", prt%child1%nr, prt%child2%nr, ","
      else if (associated (prt%child1)) then
         write (u, "(A,1(I5),A)", advance = "no") &
              " has one child: ", prt%child1%nr, ", "
      end if
      if (prt%belongstointeraction) then
         write (u, "(A,I2)") " belongs to interaction ", &
              prt%interactionnr
      else
         write (u, "(A,I2)") " does not belong to interaction ", &
              prt%interactionnr
      end if
      write (u,"(A)")  TAB
    end subroutine parton_write
```

⟨*Shower partons: parton: TBP*⟩+≡

```
    procedure :: is_final => parton_is_final
```

⟨*Shower partons: procedures*⟩+≡

```
    elemental function parton_is_final (prt) result (is_final)
      class(parton_t), intent(in) :: prt
      logical :: is_final
      is_final = .false.
      if (prt%belongstoFSR) then
         is_final = .not. associated (prt%child1) .and. &
              (.not. prt%belongstointeraction .or. &
              (prt%belongstointeraction .and. prt%simulated))
      end if
    end function parton_is_final
```

⟨*Shower partons: parton: TBP*⟩+≡

```
    procedure :: is_branched => parton_is_branched
```

⟨*Shower partons: procedures*⟩+≡

```
    elemental function parton_is_branched (prt) result (is_branched)
      class(parton_t), intent(in) :: prt
      logical :: is_branched
      is_branched = associated (prt%child1) .and. associated (prt%child2)
    end function parton_is_branched
```

⟨*Shower partons: parton: TBP*⟩+≡

```
    procedure :: set_simulated => parton_set_simulated
```

⟨*Shower partons: procedures*⟩+≡

```
    pure subroutine parton_set_simulated (prt, sim)
      class(parton_t), intent(inout) :: prt
      logical, intent(in), optional :: sim
      if (present (sim)) then
         prt%simulated = sim
```

```
    else
       prt%simulated = .true.
    end if
  end subroutine parton_set_simulated
```

⟨*Shower partons: public*⟩+≡

```
  public :: parton_set_parent
```

⟨*Shower partons: procedures*⟩+≡

```
  subroutine parton_set_parent (prt, parent)
    type(parton_t), intent(inout) :: prt
    type(parton_t), intent(in) , target :: parent
    prt%parent => parent
  end subroutine parton_set_parent
```

⟨*Shower partons: public*⟩+≡

```
  public :: parton_get_parent
```

⟨*Shower partons: procedures*⟩+≡

```
  function parton_get_parent (prt) result (parent)
    type(parton_t), intent(in) :: prt
    type(parton_t), pointer :: parent
    parent => prt%parent
  end function parton_get_parent
```

⟨*Shower partons: public*⟩+≡

```
  public :: parton_set_initial
```

⟨*Shower partons: procedures*⟩+≡

```
  subroutine parton_set_initial (prt, initial)
    type(parton_t), intent(inout) :: prt
    type(parton_t), intent(in) , target :: initial
    prt%initial => initial
  end subroutine parton_set_initial
```

⟨*Shower partons: public*⟩+≡

```
  public :: parton_get_initial
```

⟨*Shower partons: procedures*⟩+≡

```
  function parton_get_initial (prt) result (initial)
    type(parton_t), intent(in) :: prt
    type(parton_t), pointer :: initial
    initial => prt%initial
  end function parton_get_initial
```

⟨*Shower partons: public*⟩+≡

```
  public :: parton_set_child
```

⟨*Shower partons: procedures*⟩+≡

```
  subroutine parton_set_child (prt, child, i)
    type(parton_t), intent(inout) :: prt
    type(parton_t), intent(in), target :: child
    integer, intent(in) ::  i
```

```
    if (i == 1) then
       prt%child1 => child
    else
       prt%child2 => child
    end if
  end subroutine parton_set_child
```

⟨*Shower partons: public*⟩+≡

```
  public :: parton_get_child
```

⟨*Shower partons: procedures*⟩+≡

```
  function parton_get_child (prt, i) result (child)
    type(parton_t), pointer :: child
    type(parton_t), intent(in) :: prt
    integer, intent(in) :: i
    child => null ()
    if (i == 1) then
       child => prt%child1
    else
       child => prt%child2
    end if
  end function parton_get_child
```

⟨*Shower partons: parton: TBP*⟩+≡

```
  procedure :: is_quark => parton_is_quark
```

⟨*Shower partons: procedures*⟩+≡

```
  elemental function parton_is_quark (prt) result (is_quark)
    class(parton_t), intent(in) ::prt
    logical :: is_quark
    is_quark = abs (prt%type) <= 6 .and. prt%type /= 0
  end function parton_is_quark
```

⟨*Shower partons: parton: TBP*⟩+≡

```
  procedure :: is_squark => parton_is_squark
```

⟨*Shower partons: procedures*⟩+≡

```
  elemental function parton_is_squark (prt) result (is_squark)
    class(parton_t), intent(in) ::prt
    logical :: is_squark
    is_squark = ((abs(prt%type) >= 1000001) .and. (abs(prt%type) <= 1000006)) &
             .or. ((abs(prt%type) >= 2000001) .and. (abs(prt%type) <= 2000006))
  end function parton_is_squark
```

9 can be used for gluons in codes for glueballs

⟨*Shower partons: parton: TBP*⟩+≡

```
  procedure :: is_gluon => parton_is_gluon
```

⟨*Shower partons: procedures*⟩+≡

```
  elemental function parton_is_gluon (prt) result (is_gluon)
    class(parton_t), intent(in) :: prt
    logical :: is_gluon
    is_gluon = prt%type == GLUON .or. prt%type == 9
  end function parton_is_gluon
```

⟨*Shower partons: parton: TBP*⟩+≡
```
procedure :: is_gluino => parton_is_gluino
```
⟨*Shower partons: procedures*⟩+≡
```
elemental function parton_is_gluino (prt) result (is_gluino)
  class(parton_t), intent(in) :: prt
  logical :: is_gluino
  is_gluino = prt%type == 1000021
end function parton_is_gluino
```

⟨*Shower partons: parton: TBP*⟩+≡
```
procedure :: is_proton => parton_is_proton
```
⟨*Shower partons: procedures*⟩+≡
```
elemental function parton_is_proton (prt) result (is_hadron)
  class(parton_t), intent(in) :: prt
  logical :: is_hadron
  is_hadron = abs (prt%type) == PROTON
end function parton_is_proton
```

TODO: SUSY partons.

⟨*Shower partons: parton: TBP*⟩+≡
```
procedure :: is_colored => parton_is_colored
```
⟨*Shower partons: procedures*⟩+≡
```
pure function parton_is_colored (parton) result (is_colored)
  logical :: is_colored
  class(parton_t), intent(in) :: parton
  is_colored = parton_is_quark (parton) .or. parton_is_gluon (parton)
end function parton_is_colored
```

⟨*Shower partons: parton: TBP*⟩+≡
```
procedure :: mass => parton_mass
```
⟨*Shower partons: procedures*⟩+≡
```
elemental function parton_mass (prt) result (mass)
  class(parton_t), intent(in) :: prt
  real(default) :: mass
  mass = mass_type (prt%type, prt%mass2)
end function parton_mass
```

⟨*Shower partons: parton: TBP*⟩+≡
```
procedure :: mass_squared => parton_mass_squared
```
⟨*Shower partons: procedures*⟩+≡
```
elemental function parton_mass_squared (prt) result (mass_squared)
  class(parton_t), intent(in) :: prt
  real(default) :: mass_squared
  mass_squared = mass_squared_type (prt%type, prt%mass2)
end function parton_mass_squared
```

⟨*Shower partons: parton: TBP*⟩+≡
```
procedure :: momentum_to_pythia6 => parton_momentum_to_pythia6
```

⟨*Shower partons: procedures*⟩+≡

```
pure function parton_momentum_to_pythia6 (prt) result (p)
  real(double), dimension(1:5) :: p
  class(parton_t), intent(in) :: prt
  real(default) :: mass
  !!! gfortran 5.1 complains about 'ELEMENTAL procedure pointer
  !!! component mass is not allowed as an actual argument'
  !!! p = prt%momentum%to_pythia6 (prt%mass ())
  mass = prt%mass ()
  p = prt%momentum%to_pythia6 (mass)
end function parton_momentum_to_pythia6
```

⟨*Shower partons: public*⟩+≡

```
public :: P_prt_to_child1
```

⟨*Shower partons: procedures*⟩+≡

```
function P_prt_to_child1 (prt) result (retvalue)
  type(parton_t), intent(in) :: prt
  real(default) :: retvalue
  retvalue = zero
  if (prt%is_gluon ()) then
     if (prt%child1%is_quark ()) then
        retvalue = P_gqq (prt%z)
     else if (prt%child1%is_gluon ()) then
        retvalue = P_ggg (prt%z) + P_ggg (one - prt%z)
     end if
  else if (prt%is_quark ()) then
     if (prt%child1%is_quark ()) then
        retvalue = P_qqg (prt%z)
     else if (prt%child1%is_gluon ()) then
        retvalue = P_qqg (one - prt%z)
     end if
  end if
end function P_prt_to_child1
```

This function returns whether the kinematics of the branching of parton `prt` into its daughters are allowed or not.

⟨*Shower partons: public*⟩+≡

```
public :: thetabar
```

⟨*Shower partons: procedures*⟩+≡

```
function thetabar (prt, recoiler, isr_ang, E3out) result (retvalue)
  type(parton_t), intent(inout) :: prt
  type(parton_t), intent(in) :: recoiler
  real(default), intent(out), optional :: E3out
  logical, intent(in) :: isr_ang
  logical :: retvalue
  real(default) :: ctheta, cthetachild1
  real(default) p1, p4, p3, E3, shat

  shat = (prt%child1%momentum + recoiler%momentum)**2
  E3 = 0.5_default * (shat / prt%z -recoiler%t + prt%child1%t - &
       prt%child2%mass_squared ()) / sqrt(shat)
  if (present (E3out)) then
```

```
      E3out = E3
   end if
   !!! absolute values of momenta in a 3 -> 1 + 4 branching
   p3 = sqrt (E3**2 - prt%t)
   p1 = sqrt (prt%child1%momentum%p(0)**2 - prt%child1%t)
   p4 = sqrt (max (zero, (E3 - prt%child1%momentum%p(0))**2 &
                         - prt%child2%t))
   if (p3 > zero) then
      retvalue = ((p1 + p4 >= p3) .and. (p3 >= abs(p1 - p4)) )
      if (retvalue .and. isr_ang) then
         !!! check angular ordering
         if (associated (prt%child1)) then
            if (associated (prt%child1%child2)) then
               ctheta = (E3**2 - p1**2 - p4**2 + prt%t) / (two * p1 * p4)
               cthetachild1 = (prt%child1%momentum%p(0)**2 - &
                    space_part (prt%child1%child1%momentum)**2 &
                    - space_part (prt%child1%child2%momentum)**2 + prt%child1%t) &
                    / (two * space_part (prt%child1%child1%momentum)**1 * &
                             space_part (prt%child1%child2%momentum)**1)
               retvalue = (ctheta > cthetachild1)
            end if
         end if
      end if
   else
      retvalue = .false.
   end if
end function thetabar
```

⟨*Shower partons: public*⟩+≡

```
public :: parton_apply_costheta
```

⟨*Shower partons: procedures*⟩+≡

```
recursive subroutine parton_apply_costheta (prt, rng)
   type(parton_t), intent(inout) :: prt
   class(rng_t), intent(inout), allocatable :: rng
   if (debug2_active (D_SHOWER)) then
      print *, "D: parton_apply_costheta for parton " , prt%nr
      print *, 'prt%momentum%p =    ', prt%momentum%p
      call msg_debug2 (D_SHOWER, "prt%type", prt%type)
   end if
   prt%z = 0.5_default * (one + prt%get_beta () * prt%costheta)
   if (associated (prt%child1) .and. associated (prt%child2)) then
      if (prt%child1%simulated .and. prt%child2%simulated) then
         prt%z = 0.5_default * (one + (prt%child1%t - prt%child2%t) / &
              prt%t + prt%get_beta () * prt%costheta * &
               sqrt((prt%t - prt%child1%t - prt%child2%t)**2 - &
               4 * prt%child1%t * prt%child2%t) / prt%t)
         if (prt%type /= INTERNAL) then
            prt%child1%momentum%p(0) = prt%z * prt%momentum%p(0)
            prt%child2%momentum%p(0) = (one - prt%z) * prt%momentum%p(0)
         end if
         call prt%generate_ps (rng)
         call parton_apply_costheta (prt%child1, rng)
         call parton_apply_costheta (prt%child2, rng)
```

```
      end if
    end if
  end subroutine parton_apply_costheta
```

⟨*Shower partons: public*⟩+≡

```
  public :: parton_apply_lorentztrafo
```

⟨*Shower partons: procedures*⟩+≡

```
  subroutine parton_apply_lorentztrafo (prt, L)
    type(parton_t), intent(inout) :: prt
    type(lorentz_transformation_t), intent(in) :: L
    prt%momentum = L * prt%momentum
  end subroutine parton_apply_lorentztrafo
```

⟨*Shower partons: public*⟩+≡

```
  public :: parton_apply_lorentztrafo_recursive
```

⟨*Shower partons: procedures*⟩+≡

```
  recursive subroutine parton_apply_lorentztrafo_recursive (prt, L)
    type(parton_t), intent(inout) :: prt
    type(lorentz_transformation_t) ,intent(in) :: L
    if (prt%type /= PROTON .and. prt%type /= BEAM_REMNANT) then
       !!! don't boost hadrons and beam-remnants
       call parton_apply_lorentztrafo (prt, L)
    end if
    if (associated (prt%child1) .and. associated (prt%child2)) then
       if ((space_part_norm (prt%child1%momentum) < eps0) .and. &
           (space_part_norm (prt%child2%momentum) < eps0) .and. &
           (.not. prt%child1%belongstointeraction) .and. &
           (.not. prt%child2%belongstointeraction)) then
          !!! don't boost unevolved timelike partons
       else
          call parton_apply_lorentztrafo_recursive (prt%child1, L)
          call parton_apply_lorentztrafo_recursive (prt%child2, L)
       end if
    else
       if (associated (prt%child1)) then
          call parton_apply_lorentztrafo_recursive (prt%child1, L)
       end if
       if (associated (prt%child2)) then
          call parton_apply_lorentztrafo_recursive (prt%child2, L)
       end if
    end if
  end subroutine parton_apply_lorentztrafo_recursive
```

This takes the three-momentum of a parton and generates three-momenta of its children given their energy and virtuality

⟨*Shower partons: parton: TBP*⟩+≡

```
  procedure :: generate_ps => parton_generate_ps
```

⟨*Shower partons: procedures*⟩+≡

```
  subroutine parton_generate_ps (prt, rng)
    class(parton_t), intent(inout) :: prt
    class(rng_t), intent(inout), allocatable :: rng
```

```
real(default), dimension(1:3, 1:3) :: directions
integer i,j
real(default) :: scproduct, pabs, p1abs, p2abs, x, ptabs, phi
real(default), dimension(1:3) :: momentum
type(vector3_t) :: pchild1_direction
type(lorentz_transformation_t) :: L, rotation
if (debug2_active (D_SHOWER)) print *, "D: parton_generate_ps for parton " , prt%nr
if (debug_active (D_SHOWER)) then
   if (.not. (associated (prt%child1) .and. associated (prt%child2))) then
      call msg_fatal ("no children for generate_ps")
   end if
end if
!!! test if parton is a virtual parton from the imagined parton shower history
if (prt%type == INTERNAL) then
   L = inverse (boost (prt%momentum, sqrt(prt%t)))
   !!! boost to restframe of mother
   call parton_apply_lorentztrafo (prt, L)
   call parton_apply_lorentztrafo (prt%child1, L)
   call parton_apply_lorentztrafo (prt%child2, L)
   !!! Store child1's momenta
   pchild1_direction = direction (space_part (prt%child1%momentum))
   !!! Redistribute energy
   prt%child1%momentum%p(0) = (prt%momentum%p(0)**2 - &
        prt%child2%t + prt%child1%t) / (two * prt%momentum%p(0))
   prt%child2%momentum%p(0) = prt%momentum%p(0) - &
        prt%child1%momentum%p(0)

   ! rescale momenta and set momenta to be along z-axis
   prt%child1%momentum = vector4_moving (prt%child1%momentum%p(0), &
        vector3_canonical(3) * &
        sqrt(prt%child1%momentum%p(0)**2 - prt%child1%t))
   prt%child2%momentum = vector4_moving (prt%child2%momentum%p(0), &
        - vector3_canonical(3) * &
        sqrt(prt%child2%momentum%p(0)**2 - prt%child2%t))

   !!! rotate so that total momentum is along former total momentum
   rotation = rotation_to_2nd (space_part (prt%child1%momentum), &
        pchild1_direction)
   call parton_apply_lorentztrafo (prt%child1, rotation)
   call parton_apply_lorentztrafo (prt%child2, rotation)

   L = inverse (L)             !!! inverse of the boost to restframe of mother
   call parton_apply_lorentztrafo (prt, L)
   call parton_apply_lorentztrafo (prt%child1, L)
   call parton_apply_lorentztrafo (prt%child2, L)
else
   !!! directions(1,:) -> direction of the parent parton
   if (space_part_norm (prt%momentum) < eps0) return
   directions(1,1:3) = prt%momentum%p(1:3) / space_part_norm (prt%momentum)
   !!! directions(2,:) and directions(3,:) -> two random directions
   !!!   perpendicular to the direction of the parent parton
    do j = 2, 3
       call rng%generate (directions(j,:))
    end do
```

```
do i = 2, 3
   scproduct = zero
   do j = 1, i - 1
      scproduct = directions(i,1) * directions(j,1) + &
           directions(i,2) * directions(j,2) + &
           directions(i,3) * directions(j,3)
      directions(i,1) = directions(i,1) - directions(j,1) * scproduct
      directions(i,2) = directions(i,2) - directions(j,2) * scproduct
      directions(i,3) = directions(i,3) - directions(j,3) * scproduct
   end do
   scproduct = directions(i,1)**2 + directions(i,2)**2 + &
        directions(i,3)**2
   do j = 1, 3
      directions(i,j) = directions(i,j) / sqrt(scproduct)
   end do
end do
```

⟨*Enforce right-handed system for* `directions`⟩

```
pabs = space_part_norm (prt%momentum)
if ((prt%child1%momentum%p(0)**2 - prt%child1%t < 0) .or. &
    (prt%child2%momentum%p(0)**2 - prt%child2%t < 0)) then
   call msg_debug(D_SHOWER, "generate_ps error at E^2 < t")
   return
end if
p1abs = sqrt (prt%child1%momentum%p(0)**2 - prt%child1%t)
p2abs = sqrt (prt%child2%momentum%p(0)**2 - prt%child2%t)
x = (pabs**2 + p1abs**2 - p2abs**2) / (two * pabs)
if (pabs > p1abs + p2abs .or. &
     pabs < abs(p1abs - p2abs)) then
   if (debug_active (D_SHOWER)) then
      print *, "D: parton_generate_ps Dreiecksungleichung error &
           &for parton ", prt%nr, " ", &
           space_part_norm (prt%momentum), " ", p1abs, " ", p2abs
      call prt%write ()
      call prt%child1%write ()
      call prt%child2%write ()
   end if
   return
end if
!!! Due to numerical problems transverse momentum could be imaginary ->
!!!     set transverse momentum to zero
ptabs = sqrt (max (p1abs * p1abs - x * x, zero))
call rng%generate (phi)
phi = twopi * phi
do i = 1, 3
   momentum(i) = x * directions(1,i) + ptabs * &
         (cos(phi) * directions(2,i) + sin(phi) * directions(3,i))
end do
prt%child1%momentum%p(1:3) = momentum(1:3)
do i = 1, 3
   momentum(i) = (space_part_norm (prt%momentum) - x) * directions(1,i) - &
        ptabs * (cos(phi) * directions(2,i) + sin(phi) * directions(3,i))
end do
prt%child2%momentum%p(1:3) = momentum(1:3)
```

```
      end if
    end subroutine parton_generate_ps
```

⟨*Enforce right-handed system for* directions⟩≡

```
  if ((directions(1,1) * (directions(2,2) * directions(3,3) - &
       directions(2,3) * directions(3,2)) + &
       directions(1,2) * (directions(2,3) * directions(3,1) - &
       directions(2,1) * directions(3,3)) + &
       directions(1,3) * (directions(2,1) * directions(3,2) - &
       directions(2,2) * directions(3,1))) < 0) then
     directions(3,:) = - directions(3,:)
  end if
```

This routine is similar to parton_generate_ps, but now for the ISR. It takes the three-momentum of a parton's first child as fixed and generates the two remaining three-momenta.

⟨*Shower partons: parton: TBP*⟩+≡

```
    procedure :: generate_ps_ini => parton_generate_ps_ini
```

⟨*Shower partons: procedures*⟩+≡

```
    subroutine parton_generate_ps_ini (prt, rng)
      class(parton_t), intent(inout) :: prt
      class(rng_t), intent(inout), allocatable :: rng
      real(default), dimension(1:3, 1:3) :: directions
      integer :: i,j
      real(default) :: scproduct, pabs, p1abs, p2abs, x, ptabs, phi
      real(default), dimension(1:3) :: momentum
      if (debug_active (D_SHOWER)) print *, "D: parton_generate_ps_ini: for parton " , prt%nr
      if (debug_active (D_SHOWER)) then
         if (.not. (associated (prt%child1) .and. associated (prt%child2))) then
            call msg_fatal ("no children for generate_ps")
         end if
      end if

      if (.not. prt%is_proton()) then
         !!! generate ps for normal partons
         do i = 1, 3
            directions(1,i) = prt%child1%momentum%p(i) / &
                 space_part_norm(prt%child1%momentum)
         end do
         do j = 2, 3
            call rng%generate (directions(j,:))
         end do
         do i = 2, 3
            scproduct = zero
            do j = 1, i - 1
               scproduct = directions(i,1) * directions(j,1) + &
                    directions(i,2) * directions(j,2) + &
                    directions(i,3) * directions(j,3)
               directions(i,1) = directions(i,1) - directions(j,1) * scproduct
               directions(i,2) = directions(i,2) - directions(j,2) * scproduct
               directions(i,3) = directions(i,3) - directions(j,3) * scproduct
            end do
            scproduct = directions(i,1)**2 + directions(i,2)**2 + &
```

```
            directions(i,3)**2
         do j = 1, 3
            directions(i,j) = directions(i,j) / sqrt(scproduct)
         end do
      end do
      ⟨Enforce right-handed system for directions⟩

      pabs = space_part_norm (prt%child1%momentum)
      p1abs = sqrt (prt%momentum%p(0)**2 - prt%t)
      p2abs = sqrt (max(zero, prt%child2%momentum%p(0)**2 - &
           prt%child2%t))

      x = (pabs**2 + p1abs**2 - p2abs**2) / (two * pabs)
      if (debug_active (D_SHOWER)) then
         if (pabs > p1abs + p2abs .or. pabs < abs(p1abs - p2abs)) then
            print *, "error at generate_ps, Dreiecksungleichung for parton ", &
                 prt%nr, " ", pabs," ",p1abs," ",p2abs
            call prt%write ()
            call prt%child1%write ()
            call prt%child2%write ()
            call msg_fatal ("parton_generate_ps_ini: Dreiecksungleichung")
         end if
      end if
      if (debug_active (D_SHOWER)) print *, "D: parton_generate_ps_ini: x = ", x
      ptabs = sqrt (p1abs * p1abs - x**2)
      call rng%generate (phi)
      phi = twopi * phi
      do i = 1,3
         momentum(i) = x * directions(1,i) + ptabs * (cos(phi) * &
              directions(2,i) + sin(phi) * directions(3,i))
      end do
      prt%momentum%p(1:3) = momentum
      do i = 1, 3
         momentum(i) = (x - pabs) * directions(1,i) + ptabs * (cos(phi) * &
              directions(2,i) + sin(phi) * directions(3,i))
      end do
      prt%child2%momentum%p(1:3) = momentum(1:3)
   else
      !!! for first partons just set beam remnants momentum
      prt%child2%momentum = prt%momentum - prt%child1%momentum
   end if
end subroutine parton_generate_ps_ini
```

21.2.3 The analytic FSR

⟨*Shower partons: parton: TBP*⟩+≡

```
   procedure :: next_t_ana => parton_next_t_ana
```

⟨*Shower partons: procedures*⟩+≡

```
   subroutine parton_next_t_ana (prt, rng)
     class(parton_t), intent(inout) :: prt
     class(rng_t), intent(inout), allocatable :: rng
     integer :: gtoqq
```

```
    real(default) :: integral, random
    if (signal_is_pending ()) return
    call msg_debug (D_SHOWER, "next_t_ana")
    ! check if branchings are possible at all
    if (min (prt%t, prt%momentum%p(0)**2) < &
         prt%mass_squared () + prt%settings%min_virtuality) then
       prt%t = prt%mass_squared ()
       call prt%set_simulated ()
       return
    end if
    integral = zero
    call rng%generate (random)
    do
       call parton_simulate_stept (prt, rng, integral, random, gtoqq, .false.)
       if (prt%simulated) then
          if (prt%is_gluon ()) then
             !!! Abusing the x-variable to store the information to which
             !!! quark flavor the gluon branches (if any)
             prt%x = one * gtoqq + 0.1_default
             !!! x = gtoqq + 0.1 -> int(x) will be the quark flavor or
             !!! zero for g -> gg
          end if
          exit
       end if
    end do
  end subroutine parton_next_t_ana
```

The shower is actually sensitive to how close we go to the one here.

⟨*Shower partons: procedures*⟩+≡

```
  function cmax (prt, tt) result (cmaxx)
    type(parton_t), intent(in) :: prt
    real(default), intent(in), optional :: tt
    real(default) :: t, cost, cmaxx, radicand
    t = prt%t;  if (present (tt))  t = tt
    if (associated (prt%parent)) then
       cost = prt%parent%get_costheta ()
       radicand = max(zero, one - &
            t / (prt%get_beta () * prt%momentum%p(0))**2 * &
            (one + cost) / (one - cost))
       call msg_debug2 (D_SHOWER, "cmax: sqrt (radicand)", sqrt (radicand))
       cmaxx = min (0.99999_default, sqrt (radicand))
    else
       cmaxx = 0.99999_default
    end if
  end function cmax
```

Simulation routine. The variable `lookatsister` takes constraints from the sister parton into account, if not given it is assumed `.true.`. `a` and `x` are three-dimensional arrays for values used for the integration.

⟨*Shower partons: public*⟩+≡

```
  public :: parton_simulate_stept
```

⟨*Shower partons: procedures*⟩+≡

```
subroutine parton_simulate_stept &
     (prt, rng, integral, random, gtoqq, lookatsister)
  type(parton_t), intent(inout) :: prt
  class(rng_t), intent(inout), allocatable :: rng
  real(default), intent(inout) :: integral
  real(default), intent(inout) :: random
  integer, intent(out) :: gtoqq
  logical, intent(in), optional :: lookatsister

  type(parton_t), pointer :: sister
  real(default) :: tstep, tmin, oldt
  real(default) :: c, cstep
  real(default), dimension(3) :: z, P
  real(default) :: to_integral
  real(default) :: a11,a12,a13,a21,a22,a23
  real(default) :: cmax_t
  real(default) :: temprand
  real(default), dimension(3) :: a, x

  ! higher values -> faster but coarser
  real(default), parameter :: tstepfactor = 0.02_default
  real(default), parameter :: tstepmin = 0.5_default
  real(default), parameter :: cstepfactor = 0.8_default
  real(default), parameter :: cstepmin = 0.03_default

  if (signal_is_pending ()) return
  call msg_debug (D_SHOWER, "parton_simulate_stept")
  gtoqq = 111 ! illegal value
  call prt%set_simulated (.false.)

  ⟨Set sister if lookatsister is true or not given⟩

  tmin = prt%settings%min_virtuality + prt%mass_squared ()
  if (prt%is_quark ()) then
     to_integral = three *pi * log(one / random)
  else if (prt%is_gluon ()) then
     to_integral = four *pi * log(one / random)
  else
     prt%t = prt%mass_squared ()
     call prt%set_simulated ()
     return
  end if

  if (associated (sister)) then
     if (sqrt(prt%t) > sqrt(prt%parent%t) - &
          sqrt(sister%mass_squared ())) then
        prt%t = (sqrt (prt%parent%t) - sqrt (sister%mass_squared ()))**2
     end if
  end if
  if (prt%t > prt%momentum%p(0)**2) then
      prt%t = prt%momentum%p(0)**2
  end if

  if (prt%t <= tmin) then
```

```
   prt%t = prt%mass_squared ()
   call prt%set_simulated ()
   return
end if

! simulate the branchings between prt%t and prt%t - tstep
tstep = max(tstepfactor * (prt%t - 0.9_default * tmin), tstepmin)
cmax_t = cmax(prt)
c = - cmax_t ! take highest t -> minimal constraint
cstep = max(cstepfactor * (one - abs(c)), cstepmin)
! get values at border of "previous" bin -> to be used in first bin
z(3) = 0.5_default + 0.5_default * get_beta (prt%t - &
     0.5_default * tstep, prt%momentum%p(0)) * c
if (prt%is_gluon ()) then
   P(3) = P_ggg (z(3)) + P_gqq (z(3)) * number_of_flavors &
        (prt%t, prt%settings%max_n_flavors, prt%settings%min_virtuality)
else
   P(3) = P_qqg (z(3))
end if
a(3) = D_alpha_s_fsr (z(3) * (one - z(3)) * prt%t, &
     prt%settings) * P(3) / (prt%t - 0.5_default * tstep)

do while (c < cmax_t .and. (integral < to_integral))
   if (signal_is_pending ()) return
   cmax_t = cmax (prt)
   cstep = max (cstepfactor * (one - abs(c)**2), cstepmin)
   if (c + cstep > cmax_t) then
      cstep = cmax_t - c
   end if
   if (cstep < 1E-9_default) then
      !!! reject too small bins
      exit
   end if
   z(1) = z(3)
   z(2) = 0.5_default + 0.5_default * get_beta &
        (prt%t - 0.5_default * tstep, prt%momentum%p(0)) * &
        (c + 0.5_default * cstep)
   z(3) = 0.5_default + 0.5_default * get_beta &
        (prt%t - 0.5_default * tstep, prt%momentum%p(0)) * (c + cstep)
   P(1) = P(3)
   if (prt%is_gluon ()) then
      P(2) = P_ggg(z(2)) + P_gqq(z(2)) * number_of_flavors &
           (prt%t, prt%settings%max_n_flavors, prt%settings%min_virtuality)
      P(3) = P_ggg(z(3)) + P_gqq(z(3)) * number_of_flavors &
           (prt%t, prt%settings%max_n_flavors, prt%settings%min_virtuality)
   else
      P(2) = P_qqg(z(2))
      P(3) = P_qqg(z(3))
   end if
   ! get values at borders of the intgral and in the middle
   a(1) = a(3)
   a(2) = D_alpha_s_fsr (z(2) * (one - z(2)) * prt%t, &
        prt%settings) * P(2) / &
        (prt%t - 0.5_default * tstep)
```

```
a(3) = D_alpha_s_fsr (z(3) * (one - z(3)) * prt%t, &
     prt%settings) * P(3) / &
     (prt%t - 0.5_default * tstep)

!!! a little tricky:
!!! fit x(1) + x(2)/(1 + c) + x(3)/(1 - c) to these values
a11 = (one+c+0.5_default*cstep) * (one-c-0.5_default*cstep) - &
      (one-c) * (one+c+0.5_default*cstep)
a12 = (one-c-0.5_default*cstep) - (one+c+0.5_default*cstep) * &
      (one-c) / (one+c)
a13 = a(2) * (one+c+0.5_default*cstep) * (one-c-0.5_default*cstep) - &
      a(1) * (one-c) * (one+c+0.5_default*cstep)
a21 = (one+c+cstep) * (one-c-cstep) - (one+c+cstep) * (one-c)
a22 = (one-c-cstep) - (one+c+cstep) * (one-c) / (one+c)
a23 = a(3) * (one+c+cstep) * (one-c-cstep) - &
     a(1) * (one-c) * (one+c+cstep)

x(2) = (a23 - a21 * a13 / a11) / (a22 - a12 * a21 / a11)
x(1) = (a13 - a12 * x(2)) / a11
x(3) = a(1) * (one - c) - x(1) * (one - c) - x(2) * (one - c) / (one + c)

integral = integral + tstep * (x(1) * cstep + x(2) * &
     log((one + c + cstep) / (one + c)) - x(3) * &
     log((one - c - cstep) / (one - c)))

if (integral > to_integral) then
   oldt = prt%t
   call rng%generate (temprand)
   prt%t = prt%t - temprand * tstep
   call rng%generate (temprand)
   prt%costheta = c + (0.5_default - temprand) * cstep
   call prt%set_simulated ()

   if (prt%t < prt%settings%min_virtuality + prt%mass_squared ()) then
      prt%t = prt%mass_squared ()
   end if
   if (abs(prt%costheta) > cmax_t) then
      ! reject branching due to violation of costheta-limits
      call rng%generate (random)
      if (prt%is_quark ()) then
         to_integral = three * pi * log(one / random)
      else if (prt%is_gluon ()) then
         to_integral = four * pi * log(one / random)
      end if
      integral = zero
      prt%t = oldt
      call prt%set_simulated (.false.)
   end if
   if (prt%is_gluon ()) then
      ! decide between g->gg and g->qqbar splitting
      z(1) = 0.5_default + 0.5_default * prt%costheta
      call rng%generate (temprand)
      if (P_ggg(z(1)) > temprand * (P_ggg (z(1)) + P_gqq (z(1)) * &
           number_of_flavors(prt%t, prt%settings%max_n_flavors, &
```

```
                 prt%settings%min_virtuality))) then
              gtoqq = 0
           else
              call rng%generate (temprand)
              gtoqq = 1 + int (temprand * number_of_flavors &
                   (prt%t, prt%settings%max_n_flavors, &
                    prt%settings%min_virtuality))
           end if
        end if
     else
        c = c + cstep
     end if
     cmax_t = cmax (prt)
  end do
  if (integral <= to_integral) then
     prt%t = prt%t - tstep
     if (prt%t < prt%settings%min_virtuality + prt%mass_squared ()) then
        prt%t = prt%mass_squared ()
        call prt%set_simulated ()
     end if
  end if
end subroutine parton_simulate_stept
```

⟨*Set* `sister` *if* `lookatsister` *is true or not given*⟩≡

```
sister => null()
SET_SISTER: do
   if (present (lookatsister)) then
      if (.not. lookatsister) then
         exit SET_SISTER
      end if
   end if
   if (prt%nr == prt%parent%child1%nr) then
      sister => prt%parent%child2
   else
      sister => prt%parent%child1
   end if
   exit SET_SISTER
end do SET_SISTER
```

From the whole ISR algorithm all functionality has been moved to `shower_core.f90`. Only `maxzz` remains here, because more than one module needs to access it.

⟨*Shower partons: public*⟩+≡

```
public :: maxzz
```

⟨*Shower partons: procedures*⟩+≡

```
function maxzz (shat, s, maxz_isr, minenergy_timelike) result (maxz)
  real(default), intent(in) :: shat, s, minenergy_timelike, maxz_isr
  real(default) :: maxz
  maxz = min (maxz_isr, one - (two * minenergy_timelike * sqrt(shat)) / s)
end function maxzz
```

21.3 Main shower module

⟨shower_core.f90⟩≡
⟨*File header*⟩

```
module shower_core
```

⟨*Use kinds with double*⟩
⟨*Use strings*⟩

```
  use io_units
  use constants
  use format_utils, only: write_separator
  use numeric_utils
  use diagnostics
  use physics_defs
  use os_interface
  use lorentz
  use sm_physics
  use particles
  use model_data
  use flavors
  use colors
  use subevents
  use pdf
  use rng_base
  use shower_base
  use shower_partons
  use muli, only: muli_t
  use hep_common
  use tauola_interface
```

⟨*Standard module head*⟩

⟨*Shower core: public*⟩

⟨*Shower core: parameters*⟩

⟨*Shower core: types*⟩

⟨*Shower core: interfaces*⟩

```
contains
```

⟨*Shower core: procedures*⟩

```
end module shower_core
```

⟨*Shower core: public*⟩≡

```
  public :: shower_interaction_t
```

⟨*Shower core: types*⟩≡

```
  type :: shower_interaction_t
     type(parton_pointer_t), dimension(:), allocatable :: partons
  end type shower_interaction_t
```

```
type :: shower_interaction_pointer_t
   type(shower_interaction_t), pointer :: i => null ()
end type shower_interaction_pointer_t
```

The WHIZARD internal shower. Flags distinguish between analytic and k_T-ordered showers.

⟨*Shower core: public*⟩+≡

```
public :: shower_t
```

⟨*Shower core: types*⟩+≡

```
type, extends (shower_base_t) :: shower_t
   type(shower_interaction_pointer_t), dimension(:), allocatable :: &
        interactions
   type(parton_pointer_t), dimension(:), allocatable :: partons
   type(muli_t) :: mi
   integer :: next_free_nr
   integer :: next_color_nr
   logical :: valid
 contains
 ⟨Shower core: shower: TBP⟩
end type shower_t
```

⟨*Shower core: shower: TBP*⟩≡

```
procedure :: init => shower_init
```

⟨*Shower core: procedures*⟩≡

```
subroutine shower_init (shower, settings, taudec_settings, pdf_data)
  class(shower_t), intent(out) :: shower
  type(shower_settings_t), intent(in) :: settings
  type(taudec_settings_t), intent(in) :: taudec_settings
  type(pdf_data_t), intent(in) :: pdf_data
  call msg_debug (D_SHOWER, "shower_init")
  shower%settings = settings
  shower%taudec_settings = taudec_settings
  call shower%pdf_data%init (pdf_data)
  shower%name = "WHIZARD internal"
  call shower%write_msg ()
end subroutine shower_init
```

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: prepare_new_event => shower_prepare_new_event
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_prepare_new_event (shower)
  class(shower_t), intent(inout) :: shower
  call shower%cleanup ()
  shower%next_free_nr = 1
  shower%next_color_nr = 1
  if (debug_active (D_SHOWER)) then
     if (allocated (shower%interactions)) then
        call msg_bug ("Shower: creating new shower while old one " // &
             "is still associated (interactions)")
     end if
     if (allocated (shower%partons)) then
```

```
            call msg_bug ("Shower: creating new shower while old one " // &
                 "is still associated (partons)")
         end if
      end if
      treat_light_quarks_massless = .true.
      treat_duscb_quarks_massless = .false.
      shower%valid = .true.
   end subroutine shower_prepare_new_event
```

It would be better to have the muli type outside of the shower.

⟨*Shower core: shower: TBP*⟩+≡

```
   procedure :: activate_multiple_interactions => shower_activate_multiple_interactions
```

⟨*Shower core: procedures*⟩+≡

```
   subroutine shower_activate_multiple_interactions (shower, os_data)
      class(shower_t), intent(inout) :: shower
      type(os_data_t), intent(in) :: os_data
      if (shower%mi%is_initialized ()) then
         call shower%mi%restart ()
      else
         call shower%mi%initialize (&
              GeV2_scale_cutoff=shower%settings%min_virtuality, &
              GeV2_s=shower_interaction_get_s &
              (shower%interactions(1)%i), &
              muli_dir=char(os_data%whizard_mulipath))
      end if
      call shower%mi%apply_initial_interaction ( &
           GeV2_s=shower_interaction_get_s(shower%interactions(1)%i), &
           x1=shower%interactions(1)%i%partons(1)%p%parent%x, &
           x2=shower%interactions(1)%i%partons(2)%p%parent%x, &
           pdg_f1=shower%interactions(1)%i%partons(1)%p%parent%type, &
           pdg_f2=shower%interactions(1)%i%partons(2)%p%parent%type, &
           n1=shower%interactions(1)%i%partons(1)%p%parent%nr, &
           n2=shower%interactions(1)%i%partons(2)%p%parent%nr)
   end subroutine shower_activate_multiple_interactions
```

⟨*Shower core: shower: TBP*⟩+≡

```
   procedure :: import_particle_set => shower_import_particle_set
```

⟨*Shower core: procedures*⟩+≡

```
   subroutine shower_import_particle_set (shower, particle_set, os_data, scale)
      class(shower_t), target, intent(inout) :: shower
      type(particle_set_t), intent(in) :: particle_set
      type(os_data_t), intent(in) :: os_data
      real(default), intent(in) :: scale
      !integer, dimension(:), allocatable :: connections
      type(parton_t), dimension(:), allocatable, target, save :: partons, hadrons
      type(parton_pointer_t), dimension(:), allocatable :: &
           parton_pointers
      integer :: n_beam, n_in, n_out, n_tot
      integer :: i, j, nr, max_color_nr
      call msg_debug (D_SHOWER, 'shower_import_particle_set')
      call count_and_allocate ()
      call setup_hadrons_from_particle_set ()
```

```
    call setup_partons_from_particle_set ()
    call shower%update_max_color_nr (1 + max_color_nr)
    call shower%add_interaction_2ton (parton_pointers)
    if (shower%settings%muli_active) then
       call shower%activate_multiple_interactions (os_data)
    end if
    call msg_debug2 (D_SHOWER, 'shower%write() after shower_import_particle_set')
    if (debug2_active (D_SHOWER)) then
       call shower%write ()
    end if
  contains
  ⟨Shower core: shower import particle set: procedures⟩
  end subroutine shower_import_particle_set
```

⟨*Shower core: shower import particle set: procedures*⟩≡

```
  subroutine count_and_allocate ()
    max_color_nr = 0
    n_beam = particle_set%get_n_beam ()
    n_in = particle_set%get_n_in ()
    n_out = particle_set%get_n_out ()
    n_tot = particle_set%get_n_tot ()
    if (allocated (partons))  deallocate (partons)
    allocate (partons (n_in + n_out))
    allocate (parton_pointers (n_in+n_out))
  end subroutine count_and_allocate
```

⟨*Shower core: shower import particle set: procedures*⟩+≡

```
  subroutine setup_hadrons_from_particle_set ()
    j = 0
    !!! !!! !!! Workaround for Portland 16.1 compiler bug
    !!! if (n_beam > 0 .and. all (particle_set%prt(1:2)%flv%get_pdg_abs () > TAU)) then
    if (n_beam > 0 .and. particle_set%prt(1)%flv%get_pdg_abs () > TAU .and. &
         particle_set%prt(2)%flv%get_pdg_abs () > TAU) then
       call msg_debug (D_SHOWER, 'Copy hadrons from particle_set to hadrons')
       if (.not. allocated (hadrons))  allocate (hadrons (1:2))
       do i = 1, n_tot
          if (particle_set%prt(i)%status == PRT_BEAM) then
             j = j + 1
             nr = shower%get_next_free_nr ()
             hadrons(j) = parton_of_particle (particle_set%prt(i), nr)
             hadrons(j)%settings => shower%settings
             max_color_nr = max (max_color_nr, abs(hadrons(j)%c1), &
                                               abs(hadrons(j)%c2))
       end if
       end do
    end if
  end subroutine setup_hadrons_from_particle_set
```

⟨*Shower core: shower import particle set: procedures*⟩+≡

```
  subroutine setup_partons_from_particle_set ()
    integer, dimension(1) :: parent
    j = 0
    call msg_debug (D_SHOWER, "Copy partons from particle_set to partons")
```

```
    do i = 1, n_tot
       if (particle_set%prt(i)%get_status () == PRT_INCOMING .or. &
           particle_set%prt(i)%get_status () == PRT_OUTGOING) then
          j = j + 1
          nr = shower%get_next_free_nr ()
          partons(j) = parton_of_particle (particle_set%prt(i), nr)
          partons(j)%settings => shower%settings
          parton_pointers(j)%p => partons(j)
          max_color_nr = max (max_color_nr, abs (partons(j)%c1), &
                                            abs (partons(j)%c2))
          if (particle_set%prt(i)%get_status () == PRT_INCOMING .and. &
              particle_set%prt(i)%get_n_parents () == 1 .and. &
              allocated (hadrons)) then
             parent = particle_set%prt(i)%get_parents ()
             partons(j)%initial => hadrons (parent(1))
             partons(j)%x = space_part_norm (partons(j)%momentum) / &
                            space_part_norm (partons(j)%initial%momentum)
                end if
       end if
    end do
  end subroutine setup_partons_from_particle_set
```

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: generate_emissions => shower_generate_emissions
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine shower_generate_emissions &
         (shower, valid, number_of_emissions)
    class(shower_t), intent(inout), target :: shower
    logical, intent(out) :: valid
    integer, optional, intent(in) :: number_of_emissions

    type(parton_t), dimension(:), allocatable, target :: partons
    type(parton_pointer_t), dimension(:), allocatable :: &
         parton_pointers
    real(default) :: mi_scale, ps_scale, shat, phi
    type(parton_pointer_t) :: temppp
    integer :: i, j, k
    integer :: n_int, max_color_nr
    integer, dimension(2,4) :: color_corr
    call msg_debug (D_SHOWER, "shower_generate_emissions")
    if (shower%settings%isr_active) then
       call msg_debug (D_SHOWER, "Generate ISR with FSR")
       i = 0
       BRANCHINGS: do
          i = i + 1
          if (signal_is_pending ()) return
          if (shower%settings%muli_active) then
          call shower%mi%generate_gev2_pt2 &
                  (shower%get_ISR_scale (), mi_scale)
          else
             mi_scale = 0.0
          end if
```

```
!!! Shower: debugging
!!! shower%generate_next_isr_branching returns a pointer to
!!! the parton with the next ISR-branching, this parton's
!!! scale is the scale of the next branching
! temppp=shower%generate_next_isr_branching_veto ()
temppp = shower%generate_next_isr_branching ()

if (.not. associated (temppp%p) .and. &
  mi_scale < shower%settings%min_virtuality) then
   exit BRANCHINGS
end if
!!! check if branching or interaction occurs next
if (associated (temppp%p)) then
   ps_scale = abs(temppp%p%t)
else
   ps_scale = 0._default
end if
if (mi_scale > ps_scale) then
   !!! discard branching evolution lower than mi_scale
   call shower%set_max_ISR_scale (mi_scale)
   if (associated (temppp%p)) &
        call temppp%p%set_simulated (.false.)

   !!! execute new interaction
   deallocate (partons)
   deallocate (parton_pointers)
   allocate (partons(1:4))
   allocate (parton_pointers(1:4))
   do j = 1, 4
      partons(j)%nr = shower%get_next_free_nr ()
      partons(j)%belongstointeraction = .true.
      parton_pointers(j)%p => partons(j)
   end do
call shower%mi%generate_partons (partons(1)%nr, partons(2)%nr, &
        partons(1)%x, partons(2)%x, &
        partons(1)%type, partons(2)%type, &
        partons(3)%type, partons(4)%type)
   !!! calculate momenta
   shat = partons(1)%x *partons(2)%x * &
        shower_interaction_get_s (shower%interactions(1)%i)
   partons(1)%momentum = [0.5_default * sqrt(shat), &
        zero, zero, 0.5_default*sqrt(shat)]
   partons(2)%momentum = [0.5_default * sqrt(shat), &
        zero, zero, -0.5_default*sqrt(shat)]
   call parton_set_initial (partons(1), &
        shower%interactions(1)%i%partons(1)%p%initial)
   call parton_set_initial (partons(2), &
        shower%interactions(1)%i%partons(2)%p%initial)
   partons(1)%belongstoFSR = .false.
   partons(2)%belongstoFSR = .false.
   !!! calculate color connection
call shower%mi%get_color_correlations &
        (shower%get_next_color_nr (), &
        max_color_nr,color_corr)
```

```
      call shower%update_max_color_nr (max_color_nr)

         partons(1)%c1 = color_corr(1,1)
         partons(1)%c2 = color_corr(2,1)
         partons(2)%c1 = color_corr(1,2)
         partons(2)%c2 = color_corr(2,2)
         partons(3)%c1 = color_corr(1,3)
         partons(3)%c2 = color_corr(2,3)
         partons(4)%c1 = color_corr(1,4)
         partons(4)%c2 = color_corr(2,4)

         call shower%rng%generate (phi)
         phi = 2 * pi * phi
         partons(3)%momentum = [0.5_default*sqrt(shat), &
              sqrt(mi_scale)*cos(phi), &
              sqrt(mi_scale)*sin(phi), &
              sqrt(0.25_default*shat - mi_scale)]
         partons(4)%momentum = [ 0.5_default*sqrt(shat), &
              -sqrt(mi_scale)*cos(phi), &
              -sqrt(mi_scale)*sin(phi), &
              -sqrt(0.25_default*shat - mi_scale)]
         partons(3)%belongstoFSR = .true.
         partons(4)%belongstoFSR = .true.

         call shower%add_interaction_2ton (parton_pointers)
         n_int = size (shower%interactions)
         do k = 1, 2
         call shower%mi%replace_parton &
              (shower%interactions(n_int)%i%partons(k)%p%initial%nr, &
               shower%interactions(n_int)%i%partons(k)%p%nr, &
               shower%interactions(n_int)%i%partons(k)%p%parent%nr, &
               shower%interactions(n_int)%i%partons(k)%p%type, &
               shower%interactions(n_int)%i%partons(k)%p%x, &
               mi_scale)
         end do
      else
      !!! execute the next branching 'found' in the previous step
         call shower%execute_next_isr_branching (temppp)
         if (shower%settings%muli_active) then
         call shower%mi%replace_parton (temppp%p%initial%nr, &
                 temppp%p%child1%nr, temppp%p%nr, &
                 temppp%p%type, temppp%p%x, ps_scale)
         end if

      end if
   end do BRANCHINGS

   call shower%generate_fsr_for_isr_partons ()
else
   if (signal_is_pending ()) return
   call msg_debug (D_SHOWER, "Generate FSR without ISR")
   call shower%simulate_no_isr_shower ()
end if
```

```
    !!! some bookkeeping, needed after the shower is done
    call shower%boost_to_labframe ()
    call shower%generate_primordial_kt ()
    call shower%update_beamremnants ()

    if (shower%settings%fsr_active) then
       do i = 1, size (shower%interactions)
          if (signal_is_pending ()) return
          call shower%interaction_generate_fsr_2ton &
               (shower%interactions(i)%i)
       end do
    else
       call shower%simulate_no_fsr_shower ()
    end if
    call msg_debug (D_SHOWER, "Shower finished:")
    if (debug_active (D_SHOWER))  call shower%write ()

    valid = shower%valid
    !!! clean-up muli: we should finalize the muli pdf sets
    !!!      when _all_ runs are done. Not after every event if possible
    ! call shower%mi%finalize()
  end subroutine shower_generate_emissions
```

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: make_particle_set => shower_make_particle_set
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine shower_make_particle_set &
         (shower, particle_set, model, model_hadrons)
    class(shower_t), intent(in) :: shower
    type(particle_set_t), intent(inout) :: particle_set
    class(model_data_t), intent(in), target :: model
    class(model_data_t), intent(in), target :: model_hadrons
    call shower%combine_with_particle_set (particle_set, model, &
         model_hadrons)
    if (shower%settings%hadronization_active) then
       call shower%converttopythia ()
    end if
  end subroutine shower_make_particle_set
```

The parameters of the shower module:

⟨*Shower core: parameters*⟩≡

```
  real(default), save :: alphasxpdfmax = 12._default
```

In this routine, `y` and `ymin` are the jet measures, `w` and `wmax` are weights, `s` is the kinematic energy squared of the interaction. The flag `isr_is_possible_and_allowed` checks whether the initial parton is set, lepton-hadron collisions are not implemented (yet).

As a workaround: as WHIZARD can treat partons as massless, there might be partons with $E < m$: if such a parton is found, quarks will be treated massless.

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: add_interaction_2ton => shower_add_interaction_2ton
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_add_interaction_2ton (shower, partons)
  class(shower_t), intent(inout) :: shower
  type(parton_pointer_t), intent(in), dimension(:), allocatable :: partons
  !type(ckkw_pseudo_shower_weights_t), intent(in) :: ckkw_pseudo_weights

  integer :: n_partons, n_out
  integer :: i, j, imin, jmin
  real(default) :: y, ymin
  !real(default) :: w, wmax
  !real(default) :: random, sum
  type(parton_pointer_t), dimension(:), allocatable :: new_partons
  type(parton_t), pointer :: prt
  integer :: n_int
  type(shower_interaction_pointer_t), dimension(:), allocatable :: temp
  type(vector4_t) :: prtmomentum, childmomentum
  logical :: isr_is_possible_and_allowed
  type(lorentz_transformation_t) :: L

  if (signal_is_pending ()) return
  call msg_debug (D_SHOWER, "Add interaction2toN")
  n_partons = size (partons)
  n_out = n_partons - 2
  if (n_out < 2) then
     call msg_bug &
          ("Shower core: trying to add a 2-> (something<2) interaction")
  end if

  isr_is_possible_and_allowed = (associated (partons(1)%p%initial) &
       .and. associated (partons(2)%p%initial)) .and. &
       shower%settings%isr_active
  call msg_debug (D_SHOWER, "isr_is_possible_and_allowed", &
       isr_is_possible_and_allowed)

  if (associated (partons(1)%p%initial) .and. &
       partons(1)%p%is_quark ()) then
     if (partons(1)%p%momentum%p(0) < &
          two * partons(1)%p%mass()) then
        if (abs(partons(1)%p%type) < 2) then
           treat_light_quarks_massless = .true.
        else
           treat_duscb_quarks_massless = .true.
        end if
     end if
  end if
  if (associated (partons(2)%p%initial) .and. &
       partons(2)%p%is_quark ()) then
     if (partons(2)%p%momentum%p(0) < &
          two * partons(2)%p%mass()) then
        if (abs(partons(2)%p%type) < 2) then
           treat_light_quarks_massless = .true.
        else
           treat_duscb_quarks_massless = .true.
        end if
```

```
   end if
end if
```

⟨*Add a new interaction to* shower%interactions⟩

```
if (associated (shower%interactions(n_int)%i%partons(1)%p%initial))  &
     call shower%interactions(n_int)%i%partons(1)%p%initial%set_simulated ()
if (associated (shower%interactions(n_int)%i%partons(2)%p%initial))  &
     call shower%interactions(n_int)%i%partons(2)%p%initial%set_simulated ()
if (isr_is_possible_and_allowed) then
   !!! boost to the CMFrame of the incoming partons
   L = boost (-(shower%interactions(n_int)%i%partons(1)%p%momentum + &
        shower%interactions(n_int)%i%partons(2)%p%momentum), &
        (shower%interactions(n_int)%i%partons(1)%p%momentum + &
        shower%interactions(n_int)%i%partons(2)%p%momentum)**1 )
   do i = 1, n_partons
      call parton_apply_lorentztrafo &
           (shower%interactions(n_int)%i%partons(i)%p, L)
   end do
end if
do i = 1, size (partons)
   if (signal_is_pending ()) return
   !!! partons are marked as belonging to the hard interaction
   shower%interactions(n_int)%i%partons(i)%p%belongstointeraction &
        = .true.
   shower%interactions(n_int)%i%partons(i)%p%belongstoFSR = i > 2
   shower%interactions(n_int)%i%partons(i)%p%interactionnr = n_int
   !!! include a 2^(i - 1) number as a label for the ckkw clustering
   shower%interactions(n_int)%i%partons(i)%p%ckkwlabel = 2**(i - 1)
end do
```

⟨*Add partons from* shower%interactions *to* shower%partons⟩

```
if (isr_is_possible_and_allowed) then
   if (shower%settings%isr_pt_ordered) then
      call shower_prepare_for_simulate_isr_pt &
           (shower, shower%interactions(size (shower%interactions))%i)
   else
      call shower_prepare_for_simulate_isr_ana_test &
           (shower, shower%interactions(n_int)%i%partons(1)%p, &
           shower%interactions(n_int)%i%partons(2)%p)
   end if
end if

!!! generate pseudo parton shower history and add all partons to
!!!          shower%partons-array
!!! TODO initial -> initial + final branchings ??
allocate (new_partons(1:(n_partons - 2)))
do i = 1, size (new_partons)
   nullify (new_partons(i)%p)
end do
do i = 1, size (new_partons)
   new_partons(i)%p => shower%interactions(n_int)%i%partons(i + 2)%p
end do
```

```
      imin = 0
      jmin = 0

      ! TODO: (bcn 2015-04-24) make this a clustering step of the matching
!     if (allocated (ckkw_pseudo_weights%weights)) then
!        !<Perform clustering using the CKKW weights>>
!     else
         ⟨Perform clustering in the usual way⟩
!     end if

      !!! set the FSR starting scale for all partons
      do i = 1, size (new_partons)
         !!! the imaginary mother is the only parton remaining in new_partons
         if (.not. associated (new_partons(i)%p)) cycle
         call set_starting_scale (new_partons(i)%p, &
              get_starting_scale (new_partons(i)%p))
         exit
      end do

    contains

      ⟨Procedures of shower_add_interaction_2ton⟩
    end subroutine shower_add_interaction_2ton
```

⟨*Add a new interaction to* shower%interactions⟩≡

```
  if (allocated (shower%interactions)) then
     n_int = size (shower%interactions) + 1
  else
     n_int = 1
  end if
  allocate (temp (1:n_int))
  do i = 1, n_int - 1
     allocate (temp(i)%i)
     temp(i)%i = shower%interactions(i)%i
  end do
  allocate (temp(n_int)%i)
  allocate (temp(n_int)%i%partons(1:n_partons))
  do i = 1, n_partons
     allocate (temp(n_int)%i%partons(i)%p)
     call parton_copy (partons(i)%p, temp(n_int)%i%partons(i)%p)
  end do
  if (allocated (shower%interactions)) deallocate(shower%interactions)
  allocate (shower%interactions(1:n_int))
  do i = 1, n_int
     shower%interactions(i)%i => temp(i)%i
  end do
  deallocate (temp)
```

⟨*Add partons from* shower%interactions *to* shower%partons⟩≡

```
  if (allocated (shower%partons)) then
     allocate (new_partons(1:size(shower%partons) + &
          size(shower%interactions(n_int)%i%partons)))
     do i = 1, size (shower%partons)
        new_partons(i)%p => shower%partons(i)%p
```

```
      end do
      do i = 1, size (shower%interactions(n_int)%i%partons)
         new_partons(size(shower%partons) + i)%p => &
              shower%interactions(n_int)%i%partons(i)%p
      end do
      deallocate (shower%partons)
   else
      allocate (new_partons(1:size(shower%interactions(n_int)%i%partons)))
      do i = 1, size (partons)
         new_partons(i)%p => shower%interactions(n_int)%i%partons(i)%p
      end do
   end if
   allocate (shower%partons(1:size (new_partons)))
   do i = 1, size (new_partons)
      shower%partons(i)%p => new_partons(i)%p
   end do
   deallocate (new_partons)
```

⟨*Perform clustering using the CKKW weights*⟩≡

```
   CKKW_CLUSTERING: do
      !!! search for the combination with the highest weight
      wmax = zero
      CKKW_OUTER: do i = 1, size (new_partons)
         CKKW_INNER: do j = i + 1, size (new_partons)
            if (.not. associated (new_partons(i)%p)) cycle
            if (.not. associated (new_partons(j)%p)) cycle
            w = ckkw_pseudo_weights%weights(new_partons(i)%p%ckkwlabel + &
                 new_partons(j)%p%ckkwlabel)
            if (w > wmax .or. vanishes(wmax)) then
               wmax = w
               imin = i
               jmin = j
            end if
         end do CKKW_INNER
      end do CKKW_OUTER
      if (wmax > zero) then
         call shower%add_parent (new_partons(imin)%p)
         call parton_set_child (new_partons(imin)%p%parent, &
              new_partons(jmin)%p, 2)
         call parton_set_parent (new_partons(jmin)%p, &
              new_partons(imin)%p%parent)
         prt => new_partons(imin)%p%parent
         prt%nr = shower_get_next_free_nr (shower)
         prt%type = INTERNAL

         prt%momentum = new_partons(imin)%p%momentum + &
              new_partons(jmin)%p%momentum
         prt%t = prt%momentum**2

         !!! auxilliary values for the ckkw matching
         !!! for now, randomly choose the type of the intermediate
         prt%ckkwlabel = new_partons(imin)%p%ckkwlabel + &
              new_partons(jmin)%p%ckkwlabel
         sum = zero
         call shower%rng%generate (random)
```

```
      CKKW_TYPE: do i = 0, 4
         if (sum + &
              ckkw_pseudo_weights%weights_by_type(prt%ckkwlabel, i) > &
              random * ckkw_pseudo_weights%weights(prt%ckkwlabel) ) then
            prt%ckkwtype = i
            exit ckkw_type
         end if
         sum = sum + &
              ckkw_pseudo_weights%weights_by_type(prt%ckkwlabel, i)
      end do CKKW_TYPE

      !!! TODO -> calculate costheta and store it for
      !!!        later use in generate_ps

      if (space_part_norm(prt%momentum) > tiny_10) then
         prtmomentum = prt%momentum
         childmomentum = prt%child1%momentum
         prtmomentum = boost (- prt%get_beta() / &
              sqrt (one - &
              (prt%get_beta ())**2), space_part (prt%momentum) / &
              space_part_norm(prt%momentum)) * prtmomentum
         childmomentum = boost (- prt%get_beta () / &
              sqrt(one - &
              (prt%get_beta ())**2), space_part (prt%momentum) / &
              space_part_norm(prt%momentum)) * childmomentum
         prt%costheta = enclosed_angle_ct(prtmomentum, childmomentum)
      else
         prt%costheta = - one
      end if

      prt%belongstointeraction = .true.
      prt%belongstoFSR = &
           new_partons(imin)%p%belongstoFSR .and. &
           new_partons(jmin)%p%belongstoFSR

      nullify (new_partons(imin)%p)
      nullify (new_partons(jmin)%p)
      new_partons(imin)%p => prt
   else
      exit CKKW_CLUSTERING
   end if
end do CKKW_CLUSTERING
```

⟨*Perform clustering in the usual way*⟩≡

```
CLUSTERING: do
   !!! search for the partons to be clustered together
   ymin = zero
   OUTER: do i = 1, size (new_partons)
      INNER: do j = i + 1, size (new_partons)
         !!! calculate the jet measure
         if (.not.associated (new_partons(i)%p)) cycle INNER
         if (.not.associated (new_partons(j)%p)) cycle INNER
         !if (.not. shower_clustering_allowed &
              !(shower, new_partons, i,j)) &
              !cycle inner
```

```
         !!! Durham jet-measure ! don't care about constants
         y = min (new_partons(i)%p%momentum%p(0), &
              new_partons(j)%p%momentum%p(0)) * &
              (one - enclosed_angle_ct &
              (new_partons(i)%p%momentum, &
              new_partons(j)%p%momentum))
         if (y < ymin .or. vanishes(ymin)) then
            ymin = y
            imin = i
            jmin = j
         end if
      end do INNER
   end do OUTER
   if (ymin > zero) then
      call shower%add_parent (new_partons(imin)%p)
      call parton_set_child &
           (new_partons(imin)%p%parent, new_partons(jmin)%p, 2)
      call parton_set_parent &
           (new_partons(jmin)%p, new_partons(imin)%p%parent)
      prt => new_partons(imin)%p%parent
      prt%nr = shower_get_next_free_nr (shower)
      prt%type = INTERNAL

      prt%momentum = new_partons(imin)%p%momentum + &
           new_partons(jmin)%p%momentum
      prt%t = prt%momentum**2
      !!! TODO -> calculate costheta and store it for
      !!!       later use in generate_ps

      if (space_part_norm(prt%momentum) > tiny_10) then
         prtmomentum = prt%momentum
         childmomentum = prt%child1%momentum
         prtmomentum = boost (- prt%get_beta () / sqrt(one - &
              (prt%get_beta ())**2), space_part(prt%momentum) / &
              space_part_norm(prt%momentum)) * prtmomentum
         childmomentum = boost (- prt%get_beta() / &
              sqrt(one - &
              (prt%get_beta ())**2), space_part(prt%momentum) / &
              space_part_norm(prt%momentum)) * childmomentum
         prt%costheta = enclosed_angle_ct (prtmomentum, childmomentum)
      else
         prt%costheta = - one
      end if

      prt%belongstointeraction = .true.
      nullify (new_partons(imin)%p)
      nullify (new_partons(jmin)%p)
      new_partons(imin)%p => prt
   else
      exit CLUSTERING
   end if
end do CLUSTERING
```

⟨*Procedures of* shower_add_interaction_2ton⟩≡

```
recursive subroutine transfer_pointers (destiny, start, prt)
```

```
    type(parton_pointer_t), dimension(:), allocatable :: destiny
    integer, intent(inout) :: start
    type(parton_t), pointer :: prt
    destiny(start)%p => prt
    start = start + 1
    if (associated (prt%child1)) then
       call transfer_pointers (destiny, start, prt%child1)
    end if
    if (associated (prt%child2)) then
       call transfer_pointers (destiny, start, prt%child2)
    end if
  end subroutine transfer_pointers
```

⟨*Procedures of* shower_add_interaction_2ton⟩+≡

```
  recursive function get_starting_scale (prt) result (scale)
    type(parton_t), pointer :: prt
    real(default) :: scale
    scale = huge (scale)
    if (associated (prt%child1) .and. associated (prt%child2)) then
       scale = min(scale, prt%t)
    end if
    if (associated (prt%child1)) then
       scale = min (scale, get_starting_scale (prt%child1))
    end if
    if (associated (prt%child2)) then
       scale = min (scale, get_starting_scale (prt%child2))
    end if
  end function get_starting_scale
```

⟨*Procedures of* shower_add_interaction_2ton⟩+≡

```
  recursive subroutine set_starting_scale (prt, scale)
    type(parton_t), pointer :: prt
    real(default) :: scale
    if (prt%type /= INTERNAL) then
       if (scale > prt%settings%min_virtuality + prt%mass_squared ()) then
          prt%t = scale
       else
          prt%t = prt%mass_squared ()
          call prt%set_simulated ()
       end if
    end if
    if (associated (prt%child1)) then
       call set_starting_scale (prt%child1, scale)
    end if
    if (associated (prt%child2)) then
       call set_starting_scale (prt%child2, scale)
    end if
  end subroutine set_starting_scale
```

⟨*Shower core: shower: TBP*⟩+≡

```
    procedure :: simulate_no_isr_shower => shower_simulate_no_isr_shower
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_simulate_no_isr_shower (shower)
  class(shower_t), intent(inout) :: shower
  integer :: i, j
  type(parton_t), pointer :: prt
  call msg_debug (D_SHOWER, "shower_simulate_no_isr_shower")
  do i = 1, size (shower%interactions)
     do j = 1, 2
        prt => shower%interactions(i)%i%partons(j)%p
        if (associated (prt%initial)) then
           !!! for virtuality ordered: remove unneeded partons
           if (associated (prt%parent)) then
              if (.not. prt%parent%is_proton ()) then
                 if (associated (prt%parent%parent)) then
                    if (.not. prt%parent%is_proton ()) then
                       call shower_remove_parton_from_partons &
                            (shower, prt%parent%parent)
                    end if
                 end if
                 call shower_remove_parton_from_partons &
                      (shower, prt%parent)
              end if
           end if
           call parton_set_parent (prt, prt%initial)
           call parton_set_child (prt%initial, prt, 1)
           if (associated (prt%initial%child2)) then
              call shower_remove_parton_from_partons &
                   (shower,prt%initial%child2)
              deallocate (prt%initial%child2)
           end if
           call shower%add_child (prt%initial, 2)
        end if
     end do
  end do
end subroutine shower_simulate_no_isr_shower
```

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: simulate_no_fsr_shower => shower_simulate_no_fsr_shower
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_simulate_no_fsr_shower (shower)
  class(shower_t), intent(inout) :: shower
  integer :: i, j
  type(parton_t), pointer :: prt
  do i = 1, size (shower%interactions)
     do j = 3, size (shower%interactions(i)%i%partons)
        prt => shower%interactions(i)%i%partons(j)%p
        call prt%set_simulated ()
        prt%scale = zero
        prt%t = prt%mass_squared ()
     end do
  end do
end subroutine shower_simulate_no_fsr_shower
```

⟨*Shower core: procedures*⟩+≡

```
subroutine swap_pointers (prtp1, prtp2)
  type(parton_pointer_t), intent(inout) :: prtp1, prtp2
  type(parton_pointer_t) :: prtptemp
  prtptemp%p => prtp1%p
  prtp1%p => prtp2%p
  prtp2%p => prtptemp%p
end subroutine swap_pointers
```

This removes emitted timelike partons.

⟨*Shower core: procedures*⟩+≡

```
recursive subroutine shower_remove_parton_from_partons (shower, prt)
  type(shower_t), intent(inout) :: shower
  type(parton_t), pointer :: prt
  integer :: i
  if (.not. prt%belongstoFSR .and. associated (prt%child2)) then
     call shower_remove_parton_from_partons_recursive (shower, prt%child2)
  end if
  do i = 1, size (shower%partons)
     if (associated (shower%partons(i)%p, prt)) then
        shower%partons(i)%p => null()
        ! TODO: (bcn 2015-05-05) memory leak here? no deallocation?
        exit
     end if
     if (debug_active (D_SHOWER)) then
        if (i == size (shower%partons)) then
           call msg_bug ("shower_remove_parton_from_partons: parton&
                &to be removed not found")
        end if
     end if
  end do
end subroutine shower_remove_parton_from_partons
```

This removes the parton `prt` and all its children.

⟨*Shower core: procedures*⟩+≡

```
recursive subroutine shower_remove_parton_from_partons_recursive (shower, prt)
  type(shower_t), intent(inout) :: shower
  type(parton_t), pointer :: prt
  if (associated (prt%child1)) then
     call shower_remove_parton_from_partons_recursive (shower, prt%child1)
     deallocate (prt%child1)
  end if
  if (associated (prt%child2)) then
     call shower_remove_parton_from_partons_recursive (shower, prt%child2)
     deallocate (prt%child2)
  end if
  call shower_remove_parton_from_partons (shower, prt)
end subroutine shower_remove_parton_from_partons_recursive
```

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: sort_partons => shower_sort_partons
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_sort_partons (shower)
```

```
class(shower_t), intent(inout) :: shower
integer :: i, j, maxsort, size_partons
logical :: changed
call msg_debug2 (D_SHOWER, "shower_sort_partons")
if (.not. allocated (shower%partons)) return
size_partons = size (shower%partons)
maxsort = 0
do i = 1, size_partons
   if (associated (shower%partons(i)%p))  maxsort = i
end do
if (signal_is_pending ()) return
size_partons = size (shower%partons)
if (size_partons <= 1) return
do i = 1, maxsort
   if (.not. associated (shower%partons(i)%p)) cycle
   if (.not. shower%settings%isr_pt_ordered) then
      !!! set unsimulated ISR partons to be "typeless" to prevent
      !!!     influences from "wrong" masses
      if (.not. shower%partons(i)%p%belongstoFSR .and. &
          .not. shower%partons(i)%p%simulated .and. &
          .not. shower%partons(i)%p%belongstointeraction) then
         shower%partons(i)%p%type = 0
      end if
   end if
end do
if (signal_is_pending ()) return
!!! Just a Bubblesort
!!! Different algorithms needed for t-ordered and pt^2-ordered shower
!!! Pt-ordered:
if (shower%settings%isr_pt_ordered) then
   OUTERDO_PT: do i = 1, maxsort - 1
      changed = .false.
      INNERDO_PT: do j = 1, maxsort - i
         if (.not. associated (shower%partons(j + 1)%p)) cycle
         if (.not. associated (shower%partons(j)%p)) then
            !!! change if j + 1 ist assoaciated and j is not
            call swap_pointers (shower%partons(j), shower%partons(j + 1))
            changed = .true.
         else if (shower%partons(j)%p%scale < &
              shower%partons(j + 1)%p%scale) then
            call swap_pointers (shower%partons(j), shower%partons(j + 1))
            changed = .true.
         else if (nearly_equal(shower%partons(j)%p%scale, &
              shower%partons(j + 1)%p%scale)) then
            if (shower%partons(j)%p%nr > shower%partons(j + 1)%p%nr) then
               call swap_pointers (shower%partons(j), shower%partons(j + 1))
               changed = .true.
            end if
         end if
      end do INNERDO_PT
      if (.not. changed) exit OUTERDO_PT
   end do outerdo_pt
!!! |t|-ordered
else
```

```
      OUTERDO_T: do i = 1, maxsort - 1
         changed = .false.
         INNERDO_T: do j = 1, maxsort - i
            if (.not. associated (shower%partons(j + 1)%p)) cycle
            if (.not. associated (shower%partons(j)%p)) then
               !!! change if j+1 is associated and j isn't
               call swap_pointers (shower%partons(j), shower%partons(j + 1))
               changed = .true.
            else if (.not. shower%partons(j)%p%belongstointeraction .and. &
                 shower%partons(j + 1)%p%belongstointeraction) then
               !!! move partons belonging to the interaction to the front
               call swap_pointers (shower%partons(j), shower%partons(j + 1))
               changed = .true.
            else if (.not. shower%partons(j)%p%belongstointeraction .and. &
                 .not. shower%partons(j + 1)%p%belongstointeraction ) then
               if (abs (shower%partons(j)%p%t) - &
                    shower%partons(j)%p%mass_squared () < &
                    abs(shower%partons(j + 1)%p%t) - &
                    shower%partons(j + 1)%p%mass_squared ()) then
                  call swap_pointers (shower%partons(j), shower%partons(j + 1))
                  changed = .true.
               else
                  if (nearly_equal(abs (shower%partons(j)%p%t) - &
                       shower%partons(j)%p%mass_squared (), &
                       abs(shower%partons(j + 1)%p%t) - &
                       shower%partons(j + 1)%p%mass_squared ())) then
                     if (shower%partons(j)%p%nr > &
                          shower%partons(j + 1)%p%nr) then
                        call swap_pointers (shower%partons(j), &
                             shower%partons(j + 1))
                        changed = .true.
                     end if
                  end if
               end if
            end if
         end do INNERDO_T
         if (.not. changed) exit OUTERDO_T
      end do OUTERDO_T
   end if
end subroutine shower_sort_partons
```

Deallocate the interaction pointers.

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: cleanup => shower_cleanup
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_cleanup (shower)
  class(shower_t), intent(inout) :: shower
  integer :: i
  if (allocated (shower%interactions)) then
  do i = 1, size (shower%interactions)
     if (allocated (shower%interactions(i)%i%partons)) &
          deallocate (shower%interactions(i)%i%partons)
     deallocate (shower%interactions(i)%i)
```

```
    end do
    deallocate (shower%interactions)
    end if
    if (allocated (shower%partons))  deallocate (shower%partons)
  end subroutine shower_cleanup
```

Bookkeeping functions.

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: get_next_free_nr => shower_get_next_free_nr
```

⟨*Shower core: procedures*⟩+≡

```
  function shower_get_next_free_nr (shower) result (next_number)
    class(shower_t), intent(inout) :: shower
    integer :: next_number
    next_number = shower%next_free_nr
    shower%next_free_nr = shower%next_free_nr + 1
  end function shower_get_next_free_nr
```

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: update_max_color_nr => shower_update_max_color_nr
```

⟨*Shower core: procedures*⟩+≡

```
  pure subroutine shower_update_max_color_nr (shower, index)
    class(shower_t), intent(inout) :: shower
    integer, intent(in) :: index
    if (index > shower%next_color_nr) then
       shower%next_color_nr = index
    end if
  end subroutine shower_update_max_color_nr
```

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: get_next_color_nr => shower_get_next_color_nr
```

⟨*Shower core: procedures*⟩+≡

```
  function shower_get_next_color_nr (shower) result (next_color)
    class(shower_t), intent(inout) :: shower
    integer :: next_color
    next_color = shower%next_color_nr
    shower%next_color_nr = shower%next_color_nr + 1
  end function shower_get_next_color_nr
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine shower_enlarge_partons_array (shower, custom_length)
    type(shower_t), intent(inout) :: shower
    integer, intent(in), optional :: custom_length
    integer :: i, length, oldlength
    type(parton_pointer_t), dimension(:), allocatable :: tmp_partons
    call msg_debug (D_SHOWER, "shower_enlarge_partons_array")
    if (present(custom_length)) then
       length = custom_length
    else
       length = 10
    end if
```

```
    if (debug_active (D_SHOWER)) then
       if (length < 1) then
          call msg_bug ("Shower: no parton_pointers added in shower%partons")
       end if
    end if
    if (allocated (shower%partons)) then
       oldlength = size (shower%partons)
       allocate (tmp_partons(1:oldlength))
       do i = 1, oldlength
          tmp_partons(i)%p => shower%partons(i)%p
       end do
       deallocate (shower%partons)
    else
       oldlength = 0
    end if
    allocate (shower%partons(1:oldlength + length))
    do i = 1, oldlength
       shower%partons(i)%p => tmp_partons(i)%p
    end do
    do i = oldlength + 1, oldlength + length
       shower%partons(i)%p => null()
    end do
  end subroutine shower_enlarge_partons_array
```

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: add_child => shower_add_child
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine shower_add_child (shower, prt, child)
    class(shower_t), intent(inout) :: shower
    type(parton_t), pointer :: prt
    integer, intent(in) :: child
    integer :: i, lastfree
    type(parton_pointer_t) :: newprt
    if (child /= 1 .and. child /= 2) then
       call msg_bug ("Shower: Adding child in nonexisting place")
    end if
    allocate (newprt%p)
    newprt%p%nr = shower%get_next_free_nr ()
    !!! add new parton as child
    if (child == 1) then
       prt%child1 => newprt%p
    else
       prt%child2 => newprt%p
    end if
    newprt%p%parent => prt
    if (associated (prt%settings)) then
       newprt%p%settings => prt%settings
    end if
    newprt%p%interactionnr = prt%interactionnr
    !!! add new parton to shower%partons list
    if (associated (shower%partons (size(shower%partons))%p)) then
       call shower_enlarge_partons_array (shower)
    end if
```

```
      !!! find last free pointer and let it point to the new parton
      lastfree = 0
      do i = size (shower%partons), 1, -1
         if (.not. associated (shower%partons(i)%p)) then
            lastfree = i
         end if
      end do
      if (lastfree == 0) then
         call msg_bug ("Shower: no free pointers found")
      end if
      shower%partons(lastfree)%p => newprt%p
    end subroutine shower_add_child
```

⟨*Shower core: shower: TBP*⟩+≡

```
    procedure :: add_parent => shower_add_parent
```

⟨*Shower core: procedures*⟩+≡

```
    subroutine shower_add_parent (shower, prt)
      class(shower_t), intent(inout) :: shower
      type(parton_t), intent(inout), target :: prt
      integer :: i, lastfree
      type(parton_pointer_t) :: newprt
      call msg_debug2 (D_SHOWER, "shower_add_parent: for parton nr", prt%nr)
      allocate (newprt%p)
      newprt%p%nr = shower%get_next_free_nr ()
      !!! add new parton as parent
      newprt%p%child1 => prt
      prt%parent => newprt%p
      if (associated (prt%settings)) then
         newprt%p%settings => prt%settings
      end if
      newprt%p%interactionnr = prt%interactionnr
      !!! add new parton to shower%partons list
      if (.not. allocated (shower%partons) .or. &
           associated (shower%partons(size(shower%partons))%p)) then
         call shower_enlarge_partons_array (shower)
      end if
      !!! find last free pointer and let it point to the new parton
      lastfree = 0
      do i = size(shower%partons), 1, -1
         if (.not. associated (shower%partons(i)%p)) then
            lastfree = i
         end if
      end do
      if (debug_active (D_SHOWER)) then
         if (lastfree == 0) then
            call msg_bug ("Shower: no free pointers found")
         end if
      end if
      shower%partons(lastfree)%p => newprt%p
    end subroutine shower_add_parent
```

For debugging:

⟨*Shower core: procedures*⟩+≡

```
pure function shower_get_total_momentum (shower) result (mom)
  type(shower_t), intent(in) :: shower
  type(vector4_t) :: mom
  integer :: i
  if (.not. allocated (shower%partons)) return
  mom = vector4_null
  do i = 1, size (shower%partons)
     if (.not. associated (shower%partons(i)%p)) cycle
     if (shower%partons(i)%p%is_final ()) then
        mom = mom + shower%partons(i)%p%momentum
     end if
  end do
end function shower_get_total_momentum
```

Count the number of partons by going through `shower%partons` whereby you can require a minimum energy `mine` and specify whether to `include_remnants`, which is done if not given.

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: get_nr_of_partons => shower_get_nr_of_partons
```

⟨*Shower core: procedures*⟩+≡

```
function shower_get_nr_of_partons (shower, mine, &
       include_remnants, no_hard_prts, only_colored) result (nr)
  class(shower_t), intent(in) :: shower
  real(default), intent(in), optional :: mine
  logical, intent(in), optional :: include_remnants, no_hard_prts, &
                                   only_colored
  logical :: no_hard, only_col, include_rem
  integer :: nr, i
  nr = 0
  no_hard = .false.; if (present (no_hard_prts)) &
       no_hard = no_hard_prts
  only_col = .false.; if (present (only_colored)) &
       only_col = only_colored
  include_rem = .true.; if (present (include_remnants)) &
       include_rem = include_remnants
  do i = 1, size (shower%partons)
     if (.not. associated (shower%partons(i)%p)) cycle
     associate (prt => shower%partons(i)%p)
        if (.not. prt%is_final ()) cycle
        if (present (only_colored)) then
           if (only_col) then
              if (.not. prt%is_colored ()) cycle
           else
              if (prt%is_colored ()) cycle
           end if
        end if
        if (no_hard) then
           if (shower%partons(i)%p%belongstointeraction) cycle
        end if
        if (.not. include_rem) then
           if (prt%type == BEAM_REMNANT) cycle
        end if
        if (present(mine)) then
```

```
            if (prt%momentum%p(0) < mine) cycle
         end if
         nr = nr + 1
      end associate
   end do
end function shower_get_nr_of_partons
```

⟨*Shower core: procedures*⟩+≡

```
function shower_get_nr_of_final_colored_ME_partons (shower) result (nr)
   type(shower_t), intent(in) :: shower
   integer :: nr
   integer :: i, j
   type(parton_t), pointer :: prt
   nr = 0
   do i = 1, size (shower%interactions)
      do j = 1, size (shower%interactions(i)%i%partons)
         prt => shower%interactions(i)%i%partons(j)%p
         if (.not. associated (prt)) cycle
         if (.not. prt%is_colored ()) cycle
         if (prt%belongstointeraction .and. prt%belongstoFSR .and. &
              (prt%type /= INTERNAL)) then
            nr = nr +1
         end if
      end do
   end do
end function shower_get_nr_of_final_colored_ME_partons
```

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: get_final_colored_ME_momenta => &
     shower_get_final_colored_ME_momenta
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_get_final_colored_ME_momenta (shower, momenta)
   class(shower_t), intent(in) :: shower
   type(vector4_t), dimension(:), allocatable, intent(out) :: momenta
   type(parton_pointer_t), dimension(:), allocatable :: partons
   integer :: i, j, index, s
   type(parton_t), pointer :: prt
   s = shower_get_nr_of_final_colored_ME_partons (shower)
   if (s == 0) return
   allocate (partons(1:s))
   allocate (momenta(1:s))
   index = 0
   do i = 1, size (shower%interactions)
      do j = 1, size (shower%interactions(i)%i%partons)
         prt => shower%interactions(i)%i%partons(j)%p
         if (.not. associated (prt)) cycle
         if (.not. prt%is_colored ()) cycle
         if (prt%belongstointeraction .and. prt%belongstoFSR .and. &
              (prt%type /= INTERNAL)) then
            index = index + 1
            partons(index)%p => prt
         end if
      end do
```

```
    end do
    do i = 1, s   ! pointers forbid array notation
       momenta(i) = partons(i)%p%momentum
    end do
  end subroutine shower_get_final_colored_ME_momenta
```

⟨*Shower core: procedures*⟩+≡

```
  recursive function interaction_fsr_is_finished_for_parton &
       (prt) result (finished)
    type(parton_t), intent(in) :: prt
    logical :: finished
    if (prt%belongstoFSR) then
       !!! FSR partons
       if (associated (prt%child1)) then
          finished = interaction_fsr_is_finished_for_parton (prt%child1) &
               .and. interaction_fsr_is_finished_for_parton (prt%child2)
       else
          finished = prt%t <= prt%mass_squared ()
       end if
    else
       !!! search for emitted timelike partons in ISR shower
       if (.not. associated (prt%initial)) then
          !!! no inital -> no ISR
          finished = .true.
       else if (.not. associated (prt%parent)) then
          finished = .false.
       else
          if (.not. prt%parent%is_proton ()) then
             if (associated (prt%child2)) then
                finished = interaction_fsr_is_finished_for_parton (prt%parent) .and. &
                     interaction_fsr_is_finished_for_parton (prt%child2)
             else
                finished = interaction_fsr_is_finished_for_parton (prt%parent)
             end if
          else
             if (associated (prt%child2)) then
                finished = interaction_fsr_is_finished_for_parton (prt%child2)
             else
                !!! only second partons can come here -> if that happens FSR
                !!!     evolution is not existing
                finished = .true.
             end if
          end if
       end if
    end if
  end function interaction_fsr_is_finished_for_parton
```

⟨*Shower core: procedures*⟩+≡

```
  function interaction_fsr_is_finished (interaction) result (finished)
    type(shower_interaction_t), intent(in) :: interaction
    logical :: finished
    integer :: i
    finished = .true.
```

```
    if (.not. allocated (interaction%partons)) return
    do i = 1, size (interaction%partons)
       if (.not. interaction_fsr_is_finished_for_parton &
            (interaction%partons(i)%p)) then
          finished = .false.
          exit
       end if
    end do
  end function interaction_fsr_is_finished
```

⟨*Shower core: public*⟩+≡

```
  public :: shower_interaction_get_s
```

⟨*Shower core: procedures*⟩+≡

```
  function shower_interaction_get_s (interaction) result (s)
    type(shower_interaction_t), intent(in) :: interaction
    real(default) :: s
    s = (interaction%partons(1)%p%initial%momentum + &
         interaction%partons(2)%p%initial%momentum)**2
  end function shower_interaction_get_s
```

⟨*Shower core: procedures*⟩+≡

```
  function shower_fsr_is_finished (shower) result (finished)
    type(shower_t), intent(in) :: shower
    logical :: finished
    integer :: i
    finished = .true.
    if (.not. allocated (shower%interactions)) return
    do i = 1, size(shower%interactions)
       if (.not. interaction_fsr_is_finished (shower%interactions(i)%i)) then
          finished = .false.
          exit
       end if
    end do
  end function shower_fsr_is_finished
```

⟨*Shower core: procedures*⟩+≡

```
  function shower_isr_is_finished (shower) result (finished)
    type(shower_t), intent(in) :: shower
    logical :: finished
    integer :: i
    type(parton_t), pointer :: prt
    finished = .true.
    if (.not.allocated (shower%partons)) return
    do i = 1, size (shower%partons)
       if (.not. associated (shower%partons(i)%p)) cycle
       prt => shower%partons(i)%p
       if (shower%settings%isr_pt_ordered) then
          if (.not. prt%belongstoFSR .and. .not. prt%simulated &
              .and. prt%scale > zero) then
             finished = .false.
             exit
          end if
```

```
      else
         if (.not. prt%belongstoFSR .and. .not. prt%simulated &
             .and. prt%t < zero) then
            finished = .false.
            exit
         end if
      end if
   end do
end function shower_isr_is_finished
```

⟨*Shower core: procedures*⟩+≡

```
subroutine interaction_find_partons_nearest_to_hadron &
     (interaction, prt1, prt2, isr_pt_ordered)
  type(shower_interaction_t), intent(in) :: interaction
  type(parton_t), pointer :: prt1, prt2
  logical, intent(in) :: isr_pt_ordered
  prt1 => null ()
  prt2 => null ()
  prt1 => interaction%partons(1)%p
  do
     if (associated (prt1%parent)) then
        if (prt1%parent%is_proton ()) then
           exit
        else if ((.not. isr_pt_ordered .and. .not. prt1%parent%simulated) &
              .or. (isr_pt_ordered .and. .not. prt1%simulated)) then
           exit
        else
           prt1 => prt1%parent
        end if
     else
        exit
     end if
  end do
  prt2 => interaction%partons(2)%p
  do
     if (associated (prt2%parent)) then
        if (prt2%parent%is_proton ()) then
           exit
        else if ((.not. isr_pt_ordered .and. .not. prt2%parent%simulated) &
              .or. (isr_pt_ordered .and. .not. prt2%simulated)) then
           exit
        else
           prt2 => prt2%parent
        end if
     else
        exit
     end if
  end do
end subroutine interaction_find_partons_nearest_to_hadron
```

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: update_beamremnants => shower_update_beamremnants
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_update_beamremnants (shower)
  class(shower_t), intent(inout) :: shower
  type(parton_t), pointer :: hadron, remnant
  integer :: i
  real(default) :: random
  !!! only proton in first interaction !!?
  !!! currently only first beam-remnant will be updated
  do i = 1,2
     if (associated (shower%interactions(1)%i%partons(i)%p%initial)) then
        hadron => shower%interactions(1)%i%partons(i)%p%initial
     else
        cycle
     end if
     remnant => hadron%child2
     if (associated (remnant)) then
        remnant%momentum = hadron%momentum - hadron%child1%momentum
     end if
     !!! generate flavor of the beam-remnant if beam was proton
     if (abs (hadron%type) == PROTON .and. associated (hadron%child1)) then
        if (hadron%child1%is_quark ()) then
           !!! decide if valence (u,d) or sea quark (s,c,b)
           if ((abs (hadron%child1%type) <= 2) .and. &
                (hadron%type * hadron%child1%type > zero)) then
              !!! valence quark
              if (abs (hadron%child1%type) == 1) then
                 !!! if d then remaining diquark is uu_1
                 remnant%type = sign (UU1, hadron%type)
              else
                 call shower%rng%generate (random)
                 !!! if u then remaining diquark is ud_0 or ud_1
                 if (random < 0.75_default) then
                    remnant%type = sign (UD0, hadron%type)
                 else
                    remnant%type = sign (UD1, hadron%type)
                 end if
              end if
              remnant%c1 = hadron%child1%c2
              remnant%c2 = hadron%child1%c1
           else if ((hadron%type * hadron%child1%type) < zero) then
              !!! antiquark
              if (.not. associated (remnant%child1)) then
                 call shower%add_child (remnant, 1)
              end if
              if (.not. associated (remnant%child2)) then
                 call shower%add_child (remnant, 2)
              end if
              call shower%rng%generate (random)
              if (random < 0.6666_default) then
                 !!! 2/3 into udq + u
                 if (abs (hadron%child1%type) == 1) then
                    remnant%child1%type = sign (NEUTRON, hadron%type)
                 else if (abs (hadron%child1%type) == 2) then
                    remnant%child1%type = sign (PROTON, hadron%type)
                 else if (abs (hadron%child1%type) == 3) then
```

```
            remnant%child1%type = sign (SIGMA0, hadron%type)
         else if (abs (hadron%child1%type) == 4) then
            remnant%child1%type = sign (SIGMACPLUS, hadron%type)
         else if (abs (hadron%child1%type) == 5) then
            remnant%child1%type = sign (SIGMAB0, hadron%type)
         end if
         remnant%child2%type = sign (2, hadron%type)
      else
         !!! 1/3 into uuq + d
         if (abs (hadron%child1%type) == 1) then
            remnant%child1%type = sign (PROTON, hadron%type)
         else if (abs (hadron%child1%type) == 2) then
            remnant%child1%type = sign (DELTAPLUSPLUS, hadron%type)
         else if (abs (hadron%child1%type) == 3) then
            remnant%child1%type = sign (SIGMAPLUS, hadron%type)
         else if (abs (hadron%child1%type) == 4) then
            remnant%child1%type = sign (SIGMACPLUSPLUS, hadron%type)
         else if (abs (hadron%child1%type) == 5) then
            remnant%child1%type = sign (SIGMABPLUS, hadron%type)
         end if
         remnant%child2%type = sign (1, hadron%type)
      end if
      remnant%c1 = hadron%child1%c2
      remnant%c2 = hadron%child1%c1
      remnant%child1%c1 = 0
      remnant%child1%c2 = 0
      remnant%child2%c1 = remnant%c1
      remnant%child2%c2 = remnant%c2
   else
      !!! sea quark
      if (.not. associated (remnant%child1)) then
         call shower%add_child (remnant, 1)
      end if
      if (.not. associated (remnant%child2)) then
         call shower%add_child (remnant, 2)
      end if
      call shower%rng%generate (random)
      if (random < 0.5_default) then
         !!! 1/2 into usbar + ud_0
         if (abs (hadron%child1%type) == 3) then
            remnant%child1%type = sign (KPLUS, hadron%type)
         else if (abs (hadron%child1%type) == 4) then
            remnant%child1%type = sign (D0, hadron%type)
         else if (abs (hadron%child1%type) == 5) then
            remnant%child1%type = sign (BPLUS, hadron%type)
         end if
         remnant%child2%type = sign (UD0, hadron%type)
      else if (random < 0.6666_default) then
         !!! 1/6 into usbar + ud_1
         if (abs (hadron%child1%type) == 3) then
            remnant%child1%type = sign (KPLUS, hadron%type)
         else if (abs (hadron%child1%type) == 4) then
            remnant%child1%type = sign (D0, hadron%type)
         else if (abs (hadron%child1%type) == 5) then
```

```
            remnant%child1%type = sign (BPLUS, hadron%type)
         end if
         remnant%child2%type = sign (UD1, hadron%type)
      else
         !!! 1/3 into dsbar + uu_1
         if (abs (hadron%child1%type) == 3) then
            remnant%child1%type = sign (K0, hadron%type)
         else if (abs (hadron%child1%type) == 4) then
            remnant%child1%type = sign (DPLUS, hadron%type)
         else if (abs (hadron%child1%type) == 5) then
            remnant%child1%type = sign (B0, hadron%type)
         end if
         remnant%child2%type = sign (UU1, hadron%type)
      end if
      remnant%c1 = hadron%child1%c2
      remnant%c2 = hadron%child1%c1
      remnant%child1%c1 = 0
      remnant%child1%c2 = 0
      remnant%child2%c1 = remnant%c1
      remnant%child2%c2 = remnant%c2
   end if
else if (hadron%child1%is_gluon ()) then
   if (.not.associated (remnant%child1)) then
      call shower%add_child (remnant, 1)
   end if
   if (.not.associated (remnant%child2)) then
      call shower%add_child (remnant, 2)
   end if
   call shower%rng%generate (random)
   if (random < 0.5_default) then
      !!! 1/2 into u + ud_0
      remnant%child1%type = sign (2, hadron%type)
      remnant%child2%type = sign (UD0, hadron%type)
   else if (random < 0.6666_default) then
      !!! 1/6 into u + ud_1
      remnant%child1%type = sign (2, hadron%type)
      remnant%child2%type = sign (UD1, hadron%type)
   else
      !!! 1/3 into d + uu_1
      remnant%child1%type = sign (1, hadron%type)
      remnant%child2%type = sign (UU1, hadron%type)
   end if
   remnant%c1 = hadron%child1%c2
   remnant%c2 = hadron%child1%c1
   if (hadron%type > 0) then
      remnant%child1%c1 = remnant%c1
      remnant%child2%c2 = remnant%c2
   else
      remnant%child1%c2 = remnant%c2
      remnant%child2%c1 = remnant%c1
   end if
end if
remnant%initial => hadron
if (associated (remnant%child1)) then
```

```
            remnant%child1%initial => hadron
            remnant%child2%initial => hadron
            !!! don't care about on-shellness for now
            remnant%child1%momentum = 0.5_default * remnant%momentum
            remnant%child2%momentum = 0.5_default * remnant%momentum
            !!! but care about on-shellness for baryons
            if (mod (remnant%child1%type, 100) >= 10) then
               !!! check if the third quark is set -> meson or baryon
               remnant%child1%t = remnant%child1%mass_squared ()
               remnant%child1%momentum = [remnant%child1%momentum%p(0), &
                    (remnant%child1%momentum%p(1:3) / &
                     remnant%child1%momentum%p(1:3)**1) * &
                    sqrt (remnant%child1%momentum%p(0)**2 - remnant%child1%t)]
               remnant%child2%momentum = remnant%momentum &
                    - remnant%child1%momentum
            end if
         end if
      end if
   end do
end subroutine shower_update_beamremnants
```

⟨*Shower core: procedures*⟩+≡

```
subroutine interaction_apply_lorentztrafo (interaction, L)
  type(shower_interaction_t), intent(inout) :: interaction
  type(lorentz_transformation_t), intent(in) :: L
  type(parton_t), pointer :: prt
  integer :: i
  !!! ISR part
  do i = 1,2
     prt => interaction%partons(i)%p
     !!! loop over ancestors
     MOTHERS: do
        !!! boost parton
        call parton_apply_lorentztrafo (prt, L)
        if (associated (prt%child2)) then
           !!! boost emitted timelike parton (and daughters)
           call parton_apply_lorentztrafo_recursive (prt%child2, L)
        end if
        if (associated (prt%parent)) then
           if (.not. prt%parent%is_proton ()) then
              prt => prt%parent
           else
              exit
           end if
        else
           exit
        end if
     end do MOTHERS
  end do
  !!! FSR part
  if (associated (interaction%partons(3)%p%parent)) then
     !!! pseudo Parton-Shower historia has been generated -> find
     !!!     mother and go on from there recursively
     prt => interaction%partons(3)%p
```

```
      do while (associated (prt%parent))
         prt => prt%parent
      end do
      call parton_apply_lorentztrafo_recursive (prt, L)
   else
      do i = 3, size (interaction%partons)
         call parton_apply_lorentztrafo (interaction%partons(i)%p, L)
      end do
   end if
end subroutine interaction_apply_lorentztrafo
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_apply_lorentztrafo (shower, L)
   type(shower_t), intent(inout) :: shower
   type(lorentz_transformation_t), intent(in) :: L
   integer :: i
   do i = 1, size (shower%interactions)
      call interaction_apply_lorentztrafo (shower%interactions(i)%i, L)
   end do
end subroutine shower_apply_lorentztrafo
```

This boosts partons belonging to the interaction to the center-of-mass frame of its partons nearest to the hadron.

⟨*Shower core: procedures*⟩+≡

```
subroutine interaction_boost_to_CMframe (interaction, isr_pt_ordered)
   type(shower_interaction_t), intent(inout) :: interaction
   logical, intent(in) :: isr_pt_ordered
   type(vector4_t) :: beta
   type(parton_t), pointer :: prt1, prt2
   call interaction_find_partons_nearest_to_hadron &
        (interaction, prt1, prt2, isr_pt_ordered)
   beta = prt1%momentum + prt2%momentum
   beta = beta / beta%p(0)
   if (debug_active (D_SHOWER)) then
      if (beta**2 > one) then
         call msg_error ("Shower: boost to CM frame: beta > 1")
         return
      end if
   end if
   if (space_part(beta)**2 > tiny_13) then
      call interaction_apply_lorentztrafo (interaction, &
           boost(space_part(beta)**1 / &
             sqrt (one - space_part(beta)**2), -direction(beta)))
   end if
end subroutine interaction_boost_to_CMframe
```

This boosts every interaction to the center-of-mass-frame of its partons nearest to the hadron.

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: boost_to_CMframe => shower_boost_to_CMframe
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_boost_to_CMframe (shower)
```

```
    class(shower_t), intent(inout) :: shower
    integer :: i
    do i = 1, size (shower%interactions)
       call interaction_boost_to_CMframe &
            (shower%interactions(i)%i, shower%settings%isr_pt_ordered)
    end do
    ! TODO: (bcn 2015-03-23) this shouldnt be here !
    call shower%update_beamremnants ()
  end subroutine shower_boost_to_CMframe
```

This boost all partons so that initial partons have their assigned x-value.

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: boost_to_labframe => shower_boost_to_labframe
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine shower_boost_to_labframe (shower)
    class(shower_t), intent(inout) :: shower
    integer :: i
    do i = 1, size (shower%interactions)
       call interaction_boost_to_labframe &
            (shower%interactions(i)%i, shower%settings%isr_pt_ordered)
    end do
  end subroutine shower_boost_to_labframe
```

This boosts all partons so that initial partons have their assigned x-value.

⟨*Shower core: procedures*⟩+≡

```
  subroutine interaction_boost_to_labframe (interaction, isr_pt_ordered)
    type(shower_interaction_t), intent(inout) :: interaction
    logical, intent(in) :: isr_pt_ordered
    type(parton_t), pointer :: prt1, prt2
    type(vector3_t) :: beta
    call interaction_find_partons_nearest_to_hadron &
         (interaction, prt1, prt2, isr_pt_ordered)
    if (.not. associated (prt1%initial) .or. .not. &
         associated (prt2%initial)) then
       return
    end if
    !!! transform partons to overall labframe.
    beta = vector3_canonical(3) * &
         ((prt1%x * prt2%momentum%p(0) - &
           prt2%x * prt1%momentum%p(0)) / &
          (prt1%x * prt2%momentum%p(3) - &
           prt2%x * prt1%momentum%p(3)))
    if (beta**1 > tiny_10) &
         call interaction_apply_lorentztrafo (interaction, &
             boost (beta**1 / sqrt(one - beta**2), -direction(beta)))
  end subroutine interaction_boost_to_labframe
```

Only rotate to z if inital hadrons are given (and they are assumed to be aligned along the z-axis).

⟨*Shower core: procedures*⟩+≡

```
  subroutine interaction_rotate_to_z (interaction, isr_pt_ordered)
```

```
    type(shower_interaction_t), intent(inout) :: interaction
    logical, intent(in) :: isr_pt_ordered
    type(parton_t), pointer :: prt1, prt2
    call interaction_find_partons_nearest_to_hadron &
         (interaction, prt1, prt2, isr_pt_ordered)
    if (associated (prt1%initial)) then
       call interaction_apply_lorentztrafo (interaction, &
            rotation_to_2nd (space_part (prt1%momentum), &
            vector3_canonical(3) * sign (one, &
            prt1%initial%momentum%p(3))))
    end if
  end subroutine interaction_rotate_to_z
```

Rotate initial partons to lie along $\pm z$ axis.

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: rotate_to_z => shower_rotate_to_z
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine shower_rotate_to_z (shower)
    class(shower_t), intent(inout) :: shower
    integer :: i
    do i = 1, size (shower%interactions)
       call interaction_rotate_to_z &
            (shower%interactions(i)%i, shower%settings%isr_pt_ordered)
    end do
    ! TODO: (bcn 2015-03-23) this shouldnt be here !
    call shower%update_beamremnants ()
  end subroutine shower_rotate_to_z
```

Return if there are no initials, electron-hadron collision not implemented.

⟨*Shower core: procedures*⟩+≡

```
  subroutine interaction_generate_primordial_kt &
       (interaction, primordial_kt_width, primordial_kt_cutoff, rng)
    type(shower_interaction_t), intent(inout) :: interaction
    real(default), intent(in) :: primordial_kt_width, primordial_kt_cutoff
    class(rng_t), intent(inout), allocatable :: rng
    type(parton_t), pointer :: had1, had2
    type(vector4_t) :: momenta(2)
    type(vector3_t) :: beta
    real(default) :: pt (2), phi(2)
    real(default) :: shat
    real(default) :: btheta, bphi
    integer :: i
    if (vanishes (primordial_kt_width))  return
    if (.not. associated (interaction%partons(1)%p%initial) .or. &
        .not. associated (interaction%partons(2)%p%initial)) then
       return
    end if
    had1 => interaction%partons(1)%p%initial
    had2 => interaction%partons(2)%p%initial
    !!! copy momenta and energy
    momenta(1) = had1%child1%momentum
    momenta(2) = had2%child1%momentum
```

```
    GENERATE_PT_PHI: do i = 1, 2
       !!! generate transverse momentum and phi
       GENERATE_PT: do
          call rng%generate (pt (i))
          pt(i) = primordial_kt_width * sqrt(-log(pt(i)))
          if (pt(i) < primordial_kt_cutoff) exit
       end do GENERATE_PT
       call rng%generate (phi (i))
       phi(i) = twopi * phi(i)
    end do GENERATE_PT_PHI
    !!! adjust momenta
    shat = (momenta(1) + momenta(2))**2
    momenta(1) = [momenta(1)%p(0), &
         pt(1) * cos(phi(1)), &
         pt(1) * sin(phi(1)), &
         momenta(1)%p(3)]
    momenta(2) = [momenta(2)%p(0), &
         pt(2) * cos(phi(2)), &
         pt(2) * sin(phi(2)), &
         momenta(2)%p(3)]
    beta = [momenta(1)%p(1) + momenta(2)%p(1), &
         momenta(1)%p(2) + momenta(2)%p(2), zero] / sqrt(shat)
    momenta(1) = boost (beta**1 / sqrt(one - beta**2), -direction(beta)) &
         * momenta(1)
    bphi = azimuthal_angle (momenta(1))
    btheta = polar_angle (momenta(1))
    call interaction_apply_lorentztrafo (interaction, &
         rotation (cos(bphi), sin(bphi), 3) * rotation(cos(btheta), &
         sin(btheta), 2) * rotation(cos(-bphi), sin(-bphi), 3))
    call interaction_apply_lorentztrafo (interaction, &
         boost (beta**1 / sqrt(one - beta**2), -direction(beta)))
  end subroutine interaction_generate_primordial_kt
```

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: generate_primordial_kt => shower_generate_primordial_kt
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine shower_generate_primordial_kt (shower)
    class(shower_t), intent(inout) :: shower
    integer :: i
    do i = 1, size (shower%interactions)
       call interaction_generate_primordial_kt (shower%interactions(i)%i, &
            shower%settings%isr_primordial_kt_width, &
            shower%settings%isr_primordial_kt_cutoff, shower%rng)
    end do
    ! TODO: (bcn 2015-03-23) this shouldnt be here !
    call shower%update_beamremnants ()
  end subroutine shower_generate_primordial_kt
```

Output.

⟨*Shower core: procedures*⟩+≡

```
  subroutine interaction_write (interaction, unit)
    type(shower_interaction_t), intent(in) :: interaction
```

```
    integer, intent(in), optional :: unit
    integer :: i, u
    u = given_output_unit (unit); if (u < 0) return
    if (associated (interaction%partons(1)%p)) then
       if (associated (interaction%partons(1)%p%initial)) &
            call interaction%partons(1)%p%initial%write (u)
    end if
    if (associated (interaction%partons(2)%p)) then
       if (associated (interaction%partons(2)%p%initial)) &
            call interaction%partons(2)%p%initial%write (u)
    end if
    if (allocated (interaction%partons)) then
       do i = 1, size (interaction%partons)
          call interaction%partons(i)%p%write (u)
       end do
    end if
    write (u, "(A)")
  end subroutine interaction_write
```

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: write => shower_write
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine shower_write (shower, unit)
    class(shower_t), intent(in) :: shower
    integer, intent(in), optional :: unit
    integer :: i, u
    u = given_output_unit (unit); if (u < 0) return
    write (u, "(1x,A)") "------------------------------"
    write (u, "(1x,A)") "WHIZARD internal parton shower"
    write (u, "(1x,A)") "------------------------------"
    call shower%pdf_data%write (u)
    if (size (shower%interactions) > 0) then
       write (u, "(3x,A)") "Interactions: "
       do i = 1, size (shower%interactions)
          write (u, "(4x,A,I0)") "Interaction number ", i
          if (.not. associated (shower%interactions(i)%i)) then
             call msg_fatal ("Shower: missing interaction in shower")
          end if
          call interaction_write (shower%interactions(i)%i, u)
       end do
    else
       write (u, "(3x,A)") "[no interactions in shower]"
    end if
    write (u, "(A)")
    if (allocated (shower%partons)) then
       write (u, "(5x,A)")  "Partons:"
       do i = 1, size (shower%partons)
          if (associated (shower%partons(i)%p)) then
             call shower%partons(i)%p%write (u)
             if (i < size (shower%partons)) then
                if (associated (shower%partons(i + 1)%p)) then
                   if (shower%partons(i)%p%belongstointeraction .and. &
                        .not. shower%partons(i + 1)%p%belongstointeraction) then
```

```
                    call write_separator (u)
                 end if
              end if
           end if
        end if
     end do
  else
     write (u, "(5x,A)")  "[no partons in shower]"
  end if
  write (u, "(4x,A)")  "Total Momentum: "
  call vector4_write (shower_get_total_momentum (shower))
  write (u, "(1x,A,L1)") "ISR finished: ", shower_isr_is_finished (shower)
  write (u, "(1x,A,L1)") "FSR finished: ", shower_fsr_is_finished (shower)
end subroutine shower_write
```

We combine the `particle_set` from the hard interaction with the partons of the shower. For simplicity, we do not maintain the mother-daughter-relations of the shower. Hadronic `beam_remnants` of the old `particle_set` are removed as they are provided, including proper flavor information, by the ISR shower.

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: combine_with_particle_set => shower_combine_with_particle_set
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_combine_with_particle_set (shower, particle_set, &
      model_in, model_hadrons)
  class(shower_t), intent(in) :: shower
  type(particle_set_t), intent(inout) :: particle_set
  class(model_data_t), intent(in), target :: model_in
  class(model_data_t), intent(in), target :: model_hadrons
  type(particle_t), dimension(:), allocatable :: particles
  integer, dimension(:), allocatable :: hard_colored_ids, &
       shower_partons_ids, incoming_ids, outgoing_ids
  class(model_data_t), pointer :: model
  logical, dimension(:), allocatable :: hard_colored_mask
  integer :: n_shower_partons, n_remnants, i, j
  integer :: n_in, n_out, n_beam, n_tot_old
  if (signal_is_pending ()) return
  call msg_debug (D_SHOWER, "shower_combine_with_particle_set")
  call msg_debug (D_SHOWER, "Particle set before replacing")
  if (debug_active (D_SHOWER))  &
       call particle_set%write (summary=.true., compressed=.true.)

  n_shower_partons = shower%get_nr_of_partons (only_colored = &
                                     .true., no_hard_prts = .true.)
  n_remnants = shower%get_nr_of_partons (only_colored = .false., &
                                          no_hard_prts = .true.)
  if (n_shower_partons + n_remnants > 0) then
     call particle_set%without_hadronic_remnants &
          (particles, n_tot_old, n_shower_partons + n_remnants)
     call count_and_allocate ()
     call replace_outgoings ()
     call set_hard_colored_as_resonant_parents_for_shower ()
     call add_to_pset (n_tot_old, .true.)
     call add_to_pset (n_tot_old + n_remnants, .false.)
```

```
      call particle_set%replace (particles)
    end if

    call msg_debug (D_SHOWER, 'Particle set after replacing')
    if (debug_active (D_SHOWER))  &
         call particle_set%write (summary=.true., compressed=.true.)

  contains

  ⟨Shower core: shower combine with particle set: procedures⟩

  end subroutine shower_combine_with_particle_set
```

⟨*Shower core: shower combine with particle set: procedures*⟩≡

```
  subroutine count_and_allocate ()
    n_beam = particle_set%get_n_beam ()
    n_in = particle_set%get_n_in ()
    n_out = particle_set%get_n_out ()
    allocate (hard_colored_mask (size (particles)))
    !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
    do i = 1, size (particles)
       hard_colored_mask(i) = (particles(i)%get_status () == PRT_INCOMING .or. &
            particles(i)%get_status () == PRT_OUTGOING) .and. &
            particles(i)%is_colored ()
    end do
    !!! hard_colored_mask = (particles%get_status () == PRT_INCOMING .or. &
    !!!                      particles%get_status () == PRT_OUTGOING) .and. &
    !!!                     particles%is_colored ()
    allocate (hard_colored_ids (count (hard_colored_mask)))
    hard_colored_ids = pack ([(i, i=1, size (particles))], hard_colored_mask)
    allocate (shower_partons_ids (n_shower_partons))
    shower_partons_ids = [(n_tot_old + n_remnants + i, i=1, n_shower_partons)]
    allocate (incoming_ids(n_in))
    incoming_ids = [(n_beam +  i, i=1, n_in)]
    allocate (outgoing_ids(n_out))
    outgoing_ids = [(n_tot_old - n_out  + i, i=1, n_out )]
    if (debug_active (D_SHOWER)) then
        print *, 'n_remnants =    ', n_remnants
        print *, 'n_shower_partons =    ', n_shower_partons
        print *, 'n_tot_old =    ', n_tot_old
        print *, 'n_beam =    ', n_beam
        print *, 'n_in, n_out =    ', n_in, n_out
    end if
  end subroutine count_and_allocate
```

⟨*Shower core: shower combine with particle set: procedures*⟩+≡

```
  subroutine replace_outgoings ()
    do i = 1, size (shower%interactions)
       if (i > 1) then
          call msg_bug ('shower_combine_with_particle_set assumes 1 interaction')
       end if
       associate (interaction => shower%interactions(i)%i)
          do j = 3, size (interaction%partons)
```

```
            if (associated (interaction%partons(j)%p)) then
               call replace_parton_in_particles (j, interaction%partons(j)%p)
            end if
         end do
      end associate
   end do
end subroutine replace_outgoings
```

⟨*Shower core: shower combine with particle set: procedures*⟩+≡

```
subroutine replace_parton_in_particles (j, prt)
   integer, intent(in) :: j
   type(parton_t), intent(in) :: prt
   integer :: idx
   if (j <= 2) then
      idx = n_beam + j
   else
      idx = n_tot_old - n_out - n_in + j
   end if
   call particles(idx)%set_momentum (prt%momentum)
end subroutine replace_parton_in_particles
```

⟨*Shower core: shower combine with particle set: procedures*⟩+≡

```
subroutine set_hard_colored_as_resonant_parents_for_shower ()
   do i = 1, n_tot_old
      if (hard_colored_mask (i)) then
         if (has_splitted (i)) then
           call particles(i)%add_children (shower_partons_ids)
           if (particles(i)%get_status () == PRT_OUTGOING) then
              call particles(i)%set_status (PRT_RESONANT)
           end if
         end if
      end if
   end do
end subroutine set_hard_colored_as_resonant_parents_for_shower
```

⟨*Shower core: shower combine with particle set: procedures*⟩+≡

```
function has_splitted (i) result (splitted)
   logical :: splitted
   integer, intent(in) :: i
   splitted = .false.
   do j = 1, size (shower%partons)
      if (.not. associated (shower%partons(j)%p)) cycle
      if (particles(i)%flv%get_pdg () == shower%partons(j)%p%type) then
         if (all (nearly_equal (particles(i)%p%p, &
                                shower%partons(j)%p%momentum%p))) then
            splitted = shower%partons(j)%p%is_branched ()
         end if
      end if
   end do
end function has_splitted
```

⟨*Shower core: shower combine with particle set: procedures*⟩+≡

```
subroutine add_to_pset (offset, remnants)
```

```
    integer, intent(in) :: offset
    logical, intent(in) :: remnants
    integer :: i, j
    j = offset
    do i = 1, size (shower%partons)
       if (.not. associated (shower%partons(i)%p)) cycle
       associate (prt => shower%partons(i)%p)
          if (.not. prt%is_final () .or. &
              prt%belongstointeraction) cycle
          if (remnants) then
             if (prt%is_colored ()) cycle
          else
             if (.not. (prt%is_colored ())) cycle
          end if
          j = j + 1
          call find_model (model, prt%type, model_in, model_hadrons)
          particles (j) = prt%to_particle (model)
          if (remnants) then
             call particles(j)%set_parents ([prt%initial%nr])
             call particles(prt%initial%nr)%add_child (j)
          else
             call particles(j)%set_parents (hard_colored_ids)
          end if
       end associate
    end do
  end subroutine add_to_pset
```

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: write_lhef => shower_write_lhef
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine shower_write_lhef (shower, unit)
    class(shower_t), intent(in) :: shower
    integer, intent(in), optional :: unit
    integer :: u
    integer :: i
    integer :: c1, c2
    u = given_output_unit (unit);  if (u < 0)  return
    write(u,'(A)') '<LesHouchesEvents version="1.0">'
    write(u,'(A)') '<-- not a complete lhe file - just one event -->'
    write(u,'(A)') '<event>'
    write(u, *) 2 + shower%get_nr_of_partons (), 1, 1.0, 1.0, 1.0, 1.0
    !!! write incoming partons
    do i = 1, 2
       if (abs (shower%partons(i)%p%type) < 1000) then
          c1 = 0
          c2 = 0
          if (shower%partons(i)%p%is_colored ()) then
             if (shower%partons(i)%p%c1 /= 0) c1 = 500 + shower%partons(i)%p%c1
             if (shower%partons(i)%p%c2 /= 0) c2 = 500 + shower%partons(i)%p%c2
          end if
          write (u,*) shower%partons(i)%p%type, -1, 0, 0, c1, c2, &
               shower%partons(i)%p%momentum%p(1), &
               shower%partons(i)%p%momentum%p(2), &
```

```
            shower%partons(i)%p%momentum%p(3), &
            shower%partons(i)%p%momentum%p(0), &
            shower%partons(i)%p%momentum**2, zero, 9.0
      else
         write (u,*) shower%partons(i)%p%type, -9, 0, 0, 0, 0, &
            shower%partons(i)%p%momentum%p(1), &
            shower%partons(i)%p%momentum%p(2), &
            shower%partons(i)%p%momentum%p(3), &
            shower%partons(i)%p%momentum%p(0), &
            shower%partons(i)%p%momentum**2, zero, 9.0
      end if
   end do
   !!! write outgoing partons
   do i = 3, size (shower%partons)
      if (.not. associated (shower%partons(i)%p)) cycle
      if (.not. shower%partons(i)%p%is_final ()) cycle
      c1 = 0
      c2 = 0
      if (shower%partons(i)%p%is_colored ()) then
         if (shower%partons(i)%p%c1 /= 0) c1 = 500 + shower%partons(i)%p%c1
         if (shower%partons(i)%p%c2 /= 0) c2 = 500 + shower%partons(i)%p%c2
      end if
      write (u,*) shower%partons(i)%p%type, 1, 1, 2, c1, c2, &
           shower%partons(i)%p%momentum%p(1), &
           shower%partons(i)%p%momentum%p(2), &
           shower%partons(i)%p%momentum%p(3), &
           shower%partons(i)%p%momentum%p(0), &
           shower%partons(i)%p%momentum**2, zero, 9.0
   end do
   write(u,'(A)') '</event>'
   write(u,'(A)') '</LesHouchesEvents>'
end subroutine shower_write_lhef
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_replace_parent_by_hadron (shower, prt)
   type(shower_t), intent(inout) :: shower
   type(parton_t), intent(inout), target :: prt
   type(parton_t), pointer :: remnant => null()
   if (associated (prt%parent)) then
      call shower_remove_parton_from_partons (shower, prt%parent)
      deallocate (prt%parent)
   end if
   if (.not. associated (prt%initial%child2)) then
      call shower%add_child (prt%initial, 2)
   end if
   prt%parent => prt%initial
   prt%parent%child1 => prt
   ! make other child to be a beam-remnant
   remnant => prt%initial%child2
   remnant%type = BEAM_REMNANT
   remnant%momentum = prt%parent%momentum - prt%momentum
   remnant%x = one - prt%x
   remnant%parent => prt%initial
   remnant%t = zero
```

```
end subroutine shower_replace_parent_by_hadron
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_get_first_ISR_scale_for_parton (shower, prt, tmax)
  type(shower_t), intent(inout), target :: shower
  type(parton_t), intent(inout), target :: prt
  real(default), intent(in), optional :: tmax
  real(default) :: t, tstep, random, integral, temp1
  real(default) :: temprand
  if (present(tmax)) then
     t = max (max (-shower%settings%isr_tscalefactor * prt%momentum%p(0)**2, &
          -abs(tmax)), prt%t)
  else
     t = max (-shower%settings%isr_tscalefactor * prt%momentum%p(0)**2, prt%t)
  end if
  call shower%rng%generate (random)
  random = -twopi * log(random)
  !!! compare Integral and log(random) instead of random and exp(-Integral)
  integral = zero
  call prt%set_simulated (.false.)
  do
     call shower%rng%generate (temprand)
     tstep = max (abs (0.01_default * t) * temprand, 0.1_default * &
          shower%settings%min_virtuality)
     if (t + 0.5_default * tstep > - shower%settings%min_virtuality) then
        prt%t = prt%mass_squared ()
        call prt%set_simulated ()
        exit
     end if
     prt%t = t + 0.5_default * tstep
     temp1 = integral_over_z_simple (prt, (random - integral) / tstep)
     integral = integral + tstep * temp1
     if (integral > random) then
        prt%t = t + 0.5_default * tstep
        exit
     end if
     t = t + tstep
  end do
  if (prt%t > - shower%settings%min_virtuality) then
     call shower_replace_parent_by_hadron (shower, prt)
  end if

contains

  function integral_over_z_simple (prt, final) result (integral)
    type(parton_t), intent(inout) :: prt
    real(default), intent(in) :: final
    real(default), volatile :: integral

    real(default), parameter :: zstepfactor = one
    real(default), parameter :: zstepmin = 0.0001_default
    real(default) :: z, zstep, minz, maxz
    real(default) :: pdfsum
    integer :: quark, d_nf
```

```
integral = zero
if (debug2_active (D_SHOWER)) then
   print *, "D: integral_over_z_simple: t = ", prt%t
end if
minz = prt%x
! maxz = maxzz(shat, s, shower%settings%isr_z_cutoff, shower%settings%isr_minenergy)
maxz = shower%settings%isr_z_cutoff
z = minz
d_nf = shower%settings%max_n_flavors
!!! TODO -> Adapt zstep to structure of divergencies
if (prt%child1%is_gluon ()) then
   !!! gluon coming from g->gg
   do
      call shower%rng%generate (temprand)
      zstep = max(zstepmin, temprand * zstepfactor * z * (one - z))
      zstep = min(zstep, maxz - z)
      integral = integral + zstep * (D_alpha_s_isr ((one - &
           (z + 0.5_default * zstep)) * abs(prt%t), &
           shower%settings) / (abs(prt%t))) * &
           P_ggg (z + 0.5_default * zstep) * &
           shower%get_pdf (prt%initial%type, &
           prt%x / (z + 0.5_default * zstep), abs(prt%t), GLUON)
      if (integral > final) then
         exit
      end if
      z = z + zstep
      if (z >= maxz) then
         exit
      end if
   end do
   !!! gluon coming from q->qg  ! correctly implemented yet?
   if (integral < final) then
      z = minz
      do
         call shower%rng%generate (temprand)
         zstep = max(zstepmin, temprand * zstepfactor * z * (one - z))
         zstep = min(zstep, maxz - z)
         pdfsum = zero
         do quark = -d_nf, d_nf
            if (quark == 0) cycle
            pdfsum = pdfsum + shower%get_pdf (prt%initial%type, &
                 prt%x / (z + 0.5_default * zstep), abs(prt%t), quark)
         end do
         integral = integral + zstep * (D_alpha_s_isr &
              ((z + 0.5_default * zstep) * abs(prt%t), &
              shower%settings) / (abs(prt%t))) * &
              P_qqg (one - (z + 0.5_default * zstep)) * pdfsum
         if (integral > final) then
            exit
         end if
         z = z + zstep
         if (z >= maxz) then
            exit
```

```
            end if
         end do
      end if
   else if (prt%child1%is_quark ()) then
      !!! quark coming from q->qg
      do
         call shower%rng%generate(temprand)
         zstep = max(zstepmin, temprand * zstepfactor * z * (one - z))
         zstep = min(zstep, maxz - z)
         integral = integral + zstep * (D_alpha_s_isr ((one - &
              (z + 0.5_default * zstep)) * abs(prt%t), &
              shower%settings) / (abs(prt%t))) * &
              P_qqg (z + 0.5_default * zstep) * &
              shower%get_pdf (prt%initial%type, &
              prt%x / (z + 0.5_default * zstep), abs(prt%t), prt%type)
         if (integral > final) then
            exit
         end if
         z = z + zstep
         if (z >= maxz) then
            exit
         end if
      end do
      !!! quark coming from g->qqbar
      if (integral < final) then
         z = minz
         do
            call shower%rng%generate (temprand)
            zstep = max(zstepmin, temprand * zstepfactor * z*(one - z))
            zstep = min(zstep, maxz - z)
            integral = integral + zstep * (D_alpha_s_isr &
                 ((one - (z + 0.5_default * zstep)) * abs(prt%t), &
                 shower%settings) / (abs(prt%t))) * &
                 P_gqq (z + 0.5_default * zstep) * &
                 shower%get_pdf (prt%initial%type, &
                 prt%x / (z + 0.5_default * zstep), abs(prt%t), GLUON)
            if (integral > final) then
               exit
            end if
            z = z + zstep
            if (z >= maxz) then
               exit
            end if
         end do
      end if

   end if
   integral = integral / shower%get_pdf (prt%initial%type, prt%x, &
        abs(prt%t), prt%type)
 end function integral_over_z_simple

end subroutine shower_get_first_ISR_scale_for_parton
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_prepare_for_simulate_isr_pt (shower, interaction)
  type(shower_t), intent(inout) :: shower
  type(shower_interaction_t), intent(inout) :: interaction
  real(default) :: s
  s = (interaction%partons(1)%p%momentum + &
       interaction%partons(2)%p%momentum)**2
  interaction%partons(1)%p%scale = shower%settings%isr_tscalefactor * 0.25_default * s
  interaction%partons(2)%p%scale = shower%settings%isr_tscalefactor * 0.25_default * s
end subroutine shower_prepare_for_simulate_isr_pt
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_prepare_for_simulate_isr_ana_test (shower, prt1, prt2)
  type(shower_t), intent(inout) :: shower
  type(parton_t), intent(inout), target :: prt1, prt2
  type(parton_t), pointer :: prt, prta, prtb
  real(default) ::  scale, factor, E
  integer :: i
  if (.not. associated (prt1%initial) .or. .not. associated (prt2%initial)) then
     return
  end if
  scale = - (prt1%momentum + prt2%momentum) ** 2
  call prt1%set_simulated ()
  call prt2%set_simulated ()
  call shower%add_parent (prt1)
  call shower%add_parent (prt2)
  factor = sqrt (energy (prt1%momentum)**2 - scale) / &
       space_part_norm(prt1%momentum)
  prt1%parent%type = prt1%type
  prt1%parent%z = one
  prt1%parent%momentum = prt1%momentum
  prt1%parent%t = scale
  prt1%parent%x = prt1%x
  prt1%parent%initial => prt1%initial
  prt1%parent%belongstoFSR = .false.
  prt1%parent%c1 = prt1%c1
  prt1%parent%c2 = prt1%c2

  prt2%parent%type= prt2%type
  prt2%parent%z = one
  prt2%parent%momentum = prt2%momentum
  prt2%parent%t = scale
  prt2%parent%x = prt2%x
  prt2%parent%initial => prt2%initial
  prt2%parent%belongstoFSR = .false.
  prt2%parent%c1 = prt2%c1
  prt2%parent%c2 = prt2%c2

  call shower_get_first_ISR_scale_for_parton (shower, prt1%parent)
  call shower_get_first_ISR_scale_for_parton (shower, prt2%parent)

  !!! redistribute energy among first partons
  prta => prt1%parent
  prtb => prt2%parent
```

```
    E = energy (prt1%momentum + prt2%momentum)
    prta%momentum%p(0) = (E**2 - prtb%t + prta%t) / (two * E)
    prtb%momentum%p(0) = E - prta%momentum%p(0)

    call prt1%parent%set_simulated ()
    call prt2%parent%set_simulated ()
    !!! rescale momenta
    do i = 1, 2
       if (i == 1) then
          prt => prt1%parent
       else
          prt => prt2%parent
       end if
       factor = sqrt (energy (prt%momentum)**2 - prt%t) &
            / space_part_norm (prt%momentum)
       prt%momentum = vector4_moving (energy (prt%momentum), &
            factor * space_part (prt%momentum))
    end do

    if (prt1%parent%t < zero) then
       call shower%add_parent (prt1%parent)
       prt1%parent%parent%momentum = prt1%parent%momentum
       prt1%parent%parent%t = prt1%parent%t
       prt1%parent%parent%x = prt1%parent%x
       prt1%parent%parent%initial => prt1%parent%initial
       prt1%parent%parent%belongstoFSR = .false.
       call shower%add_child (prt1%parent%parent, 2)
    end if

    if (prt2%parent%t < zero) then
       call shower%add_parent (prt2%parent)
       prt2%parent%parent%momentum = prt2%parent%momentum
       prt2%parent%parent%t = prt2%parent%t
       prt2%parent%parent%x = prt2%parent%x
       prt2%parent%parent%initial => prt2%parent%initial
       prt2%parent%parent%belongstoFSR = .false.
       call shower%add_child (prt2%parent%parent, 2)
    end if

  end subroutine shower_prepare_for_simulate_isr_ana_test
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine shower_add_children_of_emitted_timelike_parton (shower, prt)
    type(shower_t), intent(inout) :: shower
    type(parton_t), pointer :: prt

    if (prt%t > prt%mass_squared () + shower%settings%min_virtuality) then
       if (prt%is_quark ()) then
          !!! q -> qg
          call shower%add_child (prt, 1)
          prt%child1%type = prt%type
          prt%child1%momentum%p(0) = prt%z * prt%momentum%p(0)
          prt%child1%t = prt%t
          call shower%add_child (prt, 2)
```

```
            prt%child2%type = GLUON
            prt%child2%momentum%p(0) = (one - prt%z) * prt%momentum%p(0)
            prt%child2%t = prt%t
         else
            if (int (prt%x) > 0) then
               call shower%add_child (prt, 1)
               prt%child1%type = int (prt%x)
               prt%child1%momentum%p(0) = prt%z * prt%momentum%p(0)
               prt%child1%t = prt%t
               call shower%add_child (prt, 2)
               prt%child2%type = -int (prt%x)
               prt%child2%momentum%p(0) = (one - prt%z) * prt%momentum%p(0)
               prt%child2%t= prt%t
            else
               call shower%add_child (prt, 1)
               prt%child1%type = GLUON
               prt%child1%momentum%p(0) = prt%z * prt%momentum%p(0)
               prt%child1%t = prt%t
               call shower%add_child (prt, 2)
               prt%child2%type = GLUON
               prt%child2%momentum%p(0) = (one - prt%z) * prt%momentum%p(0)
               prt%child2%t = prt%t
            end if
         end if
      end if
   end subroutine shower_add_children_of_emitted_timelike_parton
```

⟨*Shower core: procedures*⟩+≡

```
   subroutine shower_simulate_children_ana (shower,prt)
      type(shower_t), intent(inout), target :: shower
      type(parton_t), intent(inout) :: prt
      real(default), dimension(1:2) :: random, integral
      integer, dimension(1:2) :: gtoqq
      integer :: daughter
      type(parton_t), pointer :: daughterprt
      integer :: n_loop

      if (signal_is_pending ()) return
      if (debug2_active (D_SHOWER))  &
           print *, "D: shower_simulate_children_ana: for parton " , prt%nr
      gtoqq = 0
      if (.not. associated (prt%child1) .or. .not. associated (prt%child2)) then
         call msg_error ("Shower: error in simulate_children_ana: no children.")
         return
      end if

      ⟨Set beam-remnants and internal partons as simulated⟩

      integral = zero

      !!! impose constraints by angular ordering -> cf. (26) of Gaining analytic control
      !!! check if no branchings are possible
      if (.not. prt%child1%simulated) then
         prt%child1%t = min (prt%child1%t, &
```

```
        0.5_default * prt%child1%momentum%p(0)**2 * (one - &
        prt%get_costheta ()))
   if (.not. associated (prt%child1%settings)) &
        prt%child1%settings => shower%settings
   if (min (prt%child1%t, prt%child1%momentum%p(0)**2) < &
        prt%child1%mass_squared () + &
        prt%child1%settings%min_virtuality) then
      prt%child1%t = prt%child1%mass_squared ()
      call prt%child1%set_simulated ()
   end if
end if
if (.not. prt%child2%simulated) then
   prt%child2%t = min (prt%child2%t, &
        0.5_default * prt%child2%momentum%p(0)**2 * (one - &
        prt%get_costheta ()))
   if (.not. associated (prt%child2%settings)) &
        prt%child2%settings => shower%settings
   if (min (prt%child2%t, prt%child2%momentum%p(0)**2) < &
        prt%child2%mass_squared () + &
        prt%child2%settings%min_virtuality) then
      prt%child2%t = prt%child2%mass_squared ()
      call prt%child2%set_simulated ()
   end if
end if

call shower%rng%generate (random)

n_loop = 0
do
   if (signal_is_pending ()) return
   n_loop = n_loop + 1
   if (n_loop > 900) then
      !!! try with massless quarks
      treat_duscb_quarks_massless = .true.
   end if
   if (n_loop > 1000) then
      call msg_message ("simulate_children_ana failed for parton ", prt%nr)
      call msg_warning ("too many loops in simulate_children_ana")
      call shower%write ()
      shower%valid = .false.
      return
   end if

   !!! check if a branching in the range t(i) to t(i) - tstep(i) occurs
   if (.not. prt%child1%simulated) then
      call parton_simulate_stept &
           (prt%child1, shower%rng, integral(1), random(1), gtoqq(1))
   end if
   if (.not. prt%child2%simulated) then
      call parton_simulate_stept &
           (prt%child2, shower%rng, integral(2), random(2), gtoqq(2))
   end if

   if (prt%child1%simulated .and. prt%child2%simulated) then
```

```
            if (sqrt (prt%t) <= sqrt (prt%child1%t) + sqrt (prt%child2%t)) then
              ⟨Repeat the simulation for the parton with the lower virtuality⟩
            else
               exit
            end if
         end if
      end do

      call parton_apply_costheta (prt, shower%rng)

      ⟨Add children⟩
      call shower_parton_update_color_connections (shower, prt)
    end subroutine shower_simulate_children_ana
```

⟨*Set beam-remnants and internal partons as simulated*⟩≡

```
  if (HADRON_REMNANT <= abs (prt%type) .and. abs (prt%type) <= HADRON_REMNANT_OCTET) then
     !!! prt is beam-remnant
     call prt%set_simulated ()
     return
  end if

  !!! check if partons are "internal" -> fixed scale
  if (prt%child1%type == INTERNAL) then
     call prt%child1%set_simulated ()
  end if
  if (prt%child2%type == INTERNAL) then
     call prt%child2%set_simulated ()
  end if
```

⟨*Repeat the simulation for the parton with the lower virtuality*⟩≡

```
  !!! virtuality : t - m**2 (assuming it's not fixed)
  if (prt%child1%type == INTERNAL .and. prt%child2%type == INTERNAL) then
     call msg_fatal &
          ("Shower: both partons fixed, but momentum not conserved")
  else if (prt%child1%type == INTERNAL) then
     !!! reset child2
     call prt%child2%set_simulated (.false.)
     prt%child2%t = min (prt%child1%t, (sqrt (prt%t) - &
          sqrt (prt%child1%t))**2)
     integral(2) = zero
     call shower%rng%generate (random(2))
  else if (prt%child2%type == INTERNAL) then
     ! reset child1
     call prt%child1%set_simulated (.false.)
     prt%child1%t = min (prt%child2%t, (sqrt (prt%t) - &
          sqrt (prt%child2%t))**2)
     integral(1) = zero
     call shower%rng%generate (random(1))
  else if (prt%child1%t - prt%child1%mass_squared () > &
       prt%child2%t - prt%child2%mass_squared ()) then
     !!! reset child2
     call prt%child2%set_simulated (.false.)
     prt%child2%t = min (prt%child1%t, (sqrt (prt%t) - &
          sqrt (prt%child1%t))**2)
```

```
      integral(2) = zero
      call shower%rng%generate (random(2))
   else
      !!! reset child1 ! TODO choose child according to their t
      call prt%child1%set_simulated (.false.)
      prt%child1%t = min (prt%child2%t, (sqrt (prt%t) - &
           sqrt (prt%child2%t))**2)
      integral(1) = zero
      call shower%rng%generate (random(1))
   end if
```

⟨*Add children*⟩≡

```
   if (.not. associated (prt%child1%settings)) &
      prt%child1%settings => shower%settings
   if (.not. associated (prt%child2%settings)) &
      prt%child2%settings => shower%settings
   do daughter = 1, 2
      if (signal_is_pending ()) return
      if (daughter == 1) then
         daughterprt => prt%child1
      else
         daughterprt => prt%child2
      end if
      if (daughterprt%t < daughterprt%mass_squared () + &
           daughterprt%settings%min_virtuality) then
         cycle
      end if
      if (.not. (daughterprt%is_quark () .or. daughterprt%is_gluon ())) then
         cycle
      end if
      if (daughterprt%is_quark ()) then
         !!! q -> qg
         call shower%add_child (daughterprt, 1)
         daughterprt%child1%type = daughterprt%type
         daughterprt%child1%momentum%p(0) = daughterprt%z * &
              daughterprt%momentum%p(0)
         daughterprt%child1%t = daughterprt%t
         call shower%add_child (daughterprt, 2)
         daughterprt%child2%type = GLUON
         daughterprt%child2%momentum%p(0) = (one - daughterprt%z) * &
              daughterprt%momentum%p(0)
         daughterprt%child2%t = daughterprt%t
      else if (daughterprt%is_gluon ()) then
         if (gtoqq(daughter) > 0) then
            call shower%add_child (daughterprt, 1)
            daughterprt%child1%type = gtoqq (daughter)
            daughterprt%child1%momentum%p(0) = &
                 daughterprt%z * daughterprt%momentum%p(0)
            daughterprt%child1%t = daughterprt%t
            call shower%add_child (daughterprt, 2)
            daughterprt%child2%type = - gtoqq (daughter)
            daughterprt%child2%momentum%p(0) = (one - &
                 daughterprt%z) * daughterprt%momentum%p(0)
            daughterprt%child2%t = daughterprt%t
         else
```

```
          call shower%add_child (daughterprt, 1)
          daughterprt%child1%type = GLUON
          daughterprt%child1%momentum%p(0) = &
               daughterprt%z * daughterprt%momentum%p(0)
          daughterprt%child1%t = daughterprt%t
          call shower%add_child (daughterprt, 2)
          daughterprt%child2%type = GLUON
          daughterprt%child2%momentum%p(0) = (one - &
               daughterprt%z) * daughterprt%momentum%p(0)
          daughterprt%child2%t = daughterprt%t
        end if
     end if
  end do
```

The recoiler is `otherprt`. Instead of the random number and the exponential of the integral, we compare the logarithm of the random number and the integral.

⟨*Shower core: procedures*⟩+≡

```
    subroutine shower_isr_step_pt (shower, prt)
      type(shower_t), intent(inout) :: shower
      type(parton_t), target, intent(inout) :: prt
      type(parton_t), pointer :: otherprt

      real(default) :: scale, scalestep
      real(default), volatile :: integral
      real(default) :: random, factor
      real(default) :: temprand1, temprand2

      otherprt => shower%find_recoiler (prt)

      scale = prt%scale
      call shower%rng%generate (temprand1)
      call shower%rng%generate (temprand2)
      scalestep = max (abs (scalefactor1 * scale) * temprand1, &
           scalefactor2 * temprand2 * D_Min_scale)
      call shower%rng%generate (random)
      random = - twopi * log(random)
      integral = zero

      if (scale - 0.5_default * scalestep < D_Min_scale) then
         !!! close enough to cut-off scale -> ignore
         prt%scale = zero
         prt%t = prt%mass_squared ()
         call prt%set_simulated ()
      else
         prt%scale = scale - 0.5_default * scalestep
         factor = scalestep * (D_alpha_s_isr (prt%scale, &
              shower%settings) / (prt%scale * &
              shower%get_pdf (prt%initial%type, prt%x, prt%scale, prt%type)))
         integral = integral + factor * integral_over_z_isr_pt &
              (prt, otherprt, (random - integral) / factor)
         if (integral > random) then
            !!! prt%scale set above and prt%z set in integral_over_z_isr_pt
            call prt%set_simulated ()
            prt%t = - prt%scale / (one - prt%z)
```

```
      else
         prt%scale = scale - scalestep
      end if
   end if

contains

   function integral_over_z_isr_pt (prt, otherprt, final) &
        result (integral)
     type(parton_t), intent(inout) :: prt, otherprt
     real(default), intent(in) :: final
     real(default), volatile :: integral
     real(default) :: mbr, r
     real(default) :: zmin, zmax, z, zstep
     integer :: n_bin
     integer, parameter :: n_total_bins = 100
     real(default) :: quarkpdfsum
     real(default) :: temprand
     integer :: quark, d_nf

     quarkpdfsum = zero
     d_nf = shower%settings%max_n_flavors
     if (debug2_active (D_SHOWER)) then
        print *, "D: integral_over_z_isr_pt: for scale = ", prt%scale
     end if

     integral = zero
     mbr = (prt%momentum + otherprt%momentum)**1
     zmin = prt%x
     zmax = min (one - (sqrt (prt%scale) / mbr) * &
          (sqrt(one + 0.25_default * prt%scale / mbr**2) - &
          0.25_default * sqrt(prt%scale) / mbr), shower%settings%isr_z_cutoff)
     zstep = (zmax - zmin) / n_total_bins

     if (debug_active (D_SHOWER)) then
        if (zmin > zmax) then
           call msg_bug(" error in integral_over_z_isr_pt: zmin > zmax ")
           integral = zero
        end if
     end if

     !!! divide the range [zmin:zmax] in n_total_bins
     BINS: do n_bin = 1, n_total_bins
        z = zmin + zstep * (n_bin - 0.5_default)
        !!! z-value in the middle of the bin

        if (prt%is_gluon ()) then
           QUARKS: do quark = -d_nf, d_nf
              if (quark == 0) cycle quarks
              quarkpdfsum = quarkpdfsum + shower%get_pdf &
                   (prt%initial%type, prt%x / z, prt%scale, quark)
           end do QUARKS
           !!! g -> gg or q -> gq
           integral = integral + (zstep / z) * ((P_ggg (z) + &
```

```
        P_ggg (one - z)) * shower%get_pdf (prt%initial%type, &
        prt%x / z, prt%scale, GLUON) + P_qqg (one - z) * quarkpdfsum)
else if (prt%is_quark ()) then
   !!! q -> qg or g -> qq
   integral = integral + (zstep / z) * ( P_qqg (z) * &
        shower%get_pdf (prt%initial%type, prt%x / z, prt%scale, &
        prt%type) + &
        P_gqq(z) * shower%get_pdf (prt%initial%type, prt%x / z, &
        prt%scale, GLUON))
else
   ! call msg_fatal ("Bug neither quark nor gluon in" &
   !           // " integral_over_z_isr_pt")
end if
if (integral > final) then
   prt%z = z
   call shower%rng%generate (temprand)
   !!! decide type of father partons
     if (prt%is_gluon ()) then
        if (temprand > (P_qqg (one - z) * quarkpdfsum) / &
             ((P_ggg (z) + P_ggg (one - z)) * shower%get_pdf &
             (prt%initial%type, prt%x / z, prt%scale, GLUON) &
             + P_qqg (one - z) * quarkpdfsum)) then
           !!! gluon => gluon + gluon
           prt%aux_pt = GLUON
        else
           !!! quark => quark + gluon
           !!! decide which quark flavor the parent is
           r = temprand * quarkpdfsum
           WHICH_QUARK: do quark = -d_nf, d_nf
              if (quark == 0) cycle WHICH_QUARK
              if (r > quarkpdfsum - shower%get_pdf (prt%initial%type, &
                   prt%x / z, prt%scale, quark)) then
                 prt%aux_pt = quark
                 exit WHICH_QUARK
              else
                 quarkpdfsum = quarkpdfsum - shower%get_pdf &
                      (prt%initial%type, prt%x / z, prt%scale, quark)
              end if
           end do WHICH_QUARK
        end if

     else if (prt%is_quark ()) then
        if (temprand > (P_qqg (z) * shower%get_pdf (prt%initial%type, &
             prt%x / z, prt%scale, prt%type)) / &
             (P_qqg (z) * shower%get_pdf (prt%initial%type, prt%x / z, &
             prt%scale, prt%type) + &
             P_gqq (z) * shower%get_pdf (prt%initial%type, prt%x / z, &
             prt%scale, GLUON))) then
           !!! gluon => quark + antiquark
           prt%aux_pt = GLUON
        else
           !!! quark => quark + gluon
           prt%aux_pt = prt%type
        end if
```

```
            end if
            exit BINS
         end if
      end do BINS
    end function integral_over_z_isr_pt
  end subroutine shower_isr_step_pt
```

This function returns a pointer to the parton with the next ISR branching, while FSR branchings are ignored.

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: generate_next_isr_branching_veto => &
       shower_generate_next_isr_branching_veto
```

⟨*Shower core: procedures*⟩+≡

```
  function shower_generate_next_isr_branching_veto &
       (shower) result (next_brancher)
    class(shower_t), intent(inout) :: shower
    type(parton_pointer_t) :: next_brancher
    integer :: i
    type(parton_t),  pointer :: prt
    real(default) :: random
    !!! pointers to branchable partons
    type(parton_pointer_t), dimension(:), allocatable :: partons
    integer :: n_partons
    real(default) :: weight
    real(default) :: temp1, temp2, temp3, E3

    if (signal_is_pending ()) return

    if (shower%settings%isr_pt_ordered) then
       next_brancher = shower%generate_next_isr_branching ()
       return
    end if
    next_brancher%p => null()
    !!! branchable partons
    n_partons = 0
    do i = 1,size (shower%partons)
       prt => shower%partons(i)%p
       if (.not. associated (prt)) cycle
       if (prt%belongstoFSR) cycle
       if (prt%is_final ()) cycle
       if (.not. prt%belongstoFSR .and. prt%simulated) cycle
       n_partons = n_partons + 1
    end do
    if (n_partons == 0) then
       return
    end if
    allocate (partons(1:n_partons))
    n_partons = 1
    do i = 1, size (shower%partons)
       prt => shower%partons(i)%p
       if (.not. associated (prt)) cycle
       if (prt%belongstoFSR) cycle
       if (prt%is_final ()) cycle
```

```
   if (.not. prt%belongstoFSR .and. prt%simulated) cycle
   partons(n_partons)%p => shower%partons(i)%p
   n_partons = n_partons + 1
end do
!!! generate initial trial scales
do i = 1, size (partons)
   if (signal_is_pending ()) return
   call generate_next_trial_scale (partons(i)%p)
end do

do
   !!! search for parton with the highest trial scale
   prt => partons(1)%p
   do i = 1, size (partons)
      if (prt%t >= zero) cycle
      if (abs (partons(i)%p%t) > abs (prt%t)) then
         prt => partons(i)%p
      end if
   end do

   if (prt%t >= zero) then
      next_brancher%p => null()
      exit
   end if
   !!! generate trial z and type of mother prt
   call generate_trial_z_and_typ (prt)

   !!! weight with pdf and alpha_s
   temp1 = (D_alpha_s_isr ((one - prt%z) * abs(prt%t), &
        shower%settings) / sqrt (alphasxpdfmax))
   temp2 = shower%get_xpdf (prt%initial%type, prt%x, prt%t, &
        prt%type) / sqrt (alphasxpdfmax)
   temp3 = shower%get_xpdf (prt%initial%type, prt%child1%x, prt%child1%t, &
        prt%child1%type) / &
        shower%get_xpdf (prt%initial%type, prt%child1%x, prt%t, &
        prt%child1%type)
   ! TODO: (bcn 2015-02-19) ???
   if (temp1 * temp2 * temp3 > one) then
      print *, "weights:", temp1, temp2, temp3
   end if
   weight = (D_alpha_s_isr ((one - prt%z) * abs(prt%t), &
        shower%settings)) * &
        shower%get_xpdf (prt%initial%type, prt%x, prt%t, prt%type) * &
        shower%get_xpdf (prt%initial%type, prt%child1%x, prt%child1%t, &
        prt%child1%type) / &
        shower%get_xpdf &
        (prt%initial%type, prt%child1%x, prt%t, prt%child1%type)
   if (weight > alphasxpdfmax) then
      print *, "Setting alphasxpdfmax from ", alphasxpdfmax, " to ", weight
      alphasxpdfmax = weight
   end if
   weight = weight / alphasxpdfmax
   call shower%rng%generate (random)
   if (weight < random) then
```

```
         !!! discard branching
         call generate_next_trial_scale (prt)
         cycle
      end if
      !!! branching accepted so far
      !!! generate emitted parton
      prt%child2%t = abs(prt%t)
      prt%child2%momentum%p(0) = sqrt (abs(prt%t))
      if (shower%settings%isr_only_onshell_emitted_partons) then
         prt%child2%t = prt%child2%mass_squared ()
      else
         call prt%child2%next_t_ana (shower%rng)
      end if

      if (thetabar (prt, shower%find_recoiler (prt), &
           shower%settings%isr_angular_ordered, E3)) then
         prt%momentum%p(0) = E3
         prt%child2%momentum%p(0) = E3 - prt%child1%momentum%p(0)

         !!! found branching
         call prt%generate_ps_ini (shower%rng)
         next_brancher%p => prt
         call prt%set_simulated ()
         exit
      else
         call generate_next_trial_scale (prt)
         cycle
      end if
   end do
   if (.not. associated (next_brancher%p)) then
      !!! no further branching found -> all partons emitted by hadron
      print *, "--all partons emitted by hadrons---"
      do i = 1, size(partons)
         call shower_replace_parent_by_hadron (shower, partons(i)%p%child1)
      end do
   end if
   !!! some bookkeeping
   call shower%sort_partons ()
   ! call shower%boost_to_CMframe ()        ! really necessary?
   ! call shower%rotate_to_z ()             ! really necessary?
contains

   subroutine generate_next_trial_scale (prt)
     type(parton_t), pointer, intent(inout) :: prt
     real(default) :: random, F
     real(default) :: zmax = 0.99_default !! ??
     call shower%rng%generate (random)
     F = one   !!! TODO
     F = alphasxpdfmax / (two * pi)
     if (prt%child1%is_quark ()) then
        F = F * (integral_over_P_gqq (prt%child1%x, zmax) + &
                 integral_over_P_qqg (prt%child1%x, zmax))
     else if (prt%child1%is_gluon ()) then
        F = F * (integral_over_P_ggg (prt%child1%x, zmax) + &
```

```
          two * shower%settings%max_n_flavors * &
          integral_over_P_qqg (one - zmax, one - prt%child1%x))
   else
      call msg_bug("neither quark nor gluon in generate_next_trial_scale")
   end if
   F = F / shower%get_xpdf (prt%child1%initial%type, prt%child1%x, &
        prt%child1%t, prt%child1%type)
   prt%t = prt%t * random**(one / F)
   if (abs (prt%t) - prt%mass_squared () < &
        prt%settings%min_virtuality) then
      prt%t = prt%mass_squared ()
   end if
end subroutine generate_next_trial_scale

subroutine generate_trial_z_and_typ (prt)
   type(parton_t), pointer, intent(inout) :: prt
   real(default) :: random
   real(default) :: z, zstep, zmin, integral
   real(default) :: zmax = 0.99_default !! ??
   call msg_debug (D_SHOWER, "generate_trial_z_and_typ")
   call shower%rng%generate (random)
   integral = zero
   !!! decide which branching a->bc occurs
   if (prt%child1%is_quark ()) then
      if (random < integral_over_P_qqg (prt%child1%x, zmax) / &
           (integral_over_P_qqg (prt%child1%x, zmax) + &
            integral_over_P_gqq (prt%child1%x, zmax))) then
         prt%type = prt%child1%type
         prt%child2%type = GLUON
         integral = integral_over_P_qqg (prt%child1%x, zmax)
      else
         prt%type = GLUON
         prt%child2%type = - prt%child1%type
         integral = integral_over_P_gqq (prt%child1%x, zmax)
      end if
   else if (prt%child1%is_gluon ()) then
      if (random < integral_over_P_ggg (prt%child1%x, zmax) / &
           (integral_over_P_ggg (prt%child1%x, zmax) + two * &
           shower%settings%max_n_flavors * &
           integral_over_P_qqg (one - zmax, &
           one - prt%child1%x))) then
         prt%type = GLUON
         prt%child2%type = GLUON
         integral = integral_over_P_ggg (prt%child1%x, zmax)
      else
         call shower%rng%generate (random)
         prt%type = 1 + floor(random * shower%settings%max_n_flavors)
         call shower%rng%generate (random)
         if (random > 0.5_default) prt%type = - prt%type
         prt%child2%type = prt%type
         integral = integral_over_P_qqg (one - zmax, &
              one - prt%child1%x)
      end if
   else
```

```
   call msg_bug("neither quark nor gluon in generate_next_trial_scale")
end if
!!! generate the z-value
!!! z between prt%child1%x and zmax
! prt%z = one - random * (one - prt%child1%x)      ! TODO

call shower%rng%generate (random)
zmin = prt%child1%x
zstep = max(0.1_default, 0.5_default * (zmax - zmin))
z = zmin
if (zmin > zmax) then
   print *, " zmin = ", zmin, " zmax = ", zmax
   call msg_fatal ("Shower: zmin greater than zmax")
end if
!!! procedure pointers would be helpful here
if (prt%is_quark () .and. prt%child1%is_quark ()) then
   do
      zstep = min(zstep, 0.5_default * (zmax - z))
      if (abs(zstep) < 0.00001) exit
      if (integral_over_P_qqg (zmin, z) < random * integral) then
         if (integral_over_P_qqg (zmin, min(z + zstep, zmax)) &
              < random * integral) then
            z = min (z + zstep, zmax)
            cycle
         else
            zstep = zstep * 0.5_default
            cycle
         end if
      end if
   end do
else if (prt%is_quark () .and. prt%child1%is_gluon ()) then
   do
      zstep = min(zstep, 0.5_default * (zmax - z))
      if (abs(zstep) < 0.00001) exit
      if (integral_over_P_qqg (zmin, z) < random * integral) then
         if (integral_over_P_qqg (zmin, min(z + zstep, zmax)) &
              < random * integral) then
            z = min(z + zstep, zmax)
            cycle
         else
            zstep = zstep * 0.5_default
            cycle
         end if
      end if
   end do
else if (prt%is_gluon () .and. prt%child1%is_quark ()) then
   do
      zstep = min(zstep, 0.5_default * (zmax - z))
      if (abs (zstep) < 0.00001) exit
      if (integral_over_P_gqq (zmin, z) < random * integral) then
         if (integral_over_P_gqq (zmin, min(z + zstep, zmax)) &
              < random * integral) then
            z = min (z + zstep, zmax)
            cycle
```

```
            else
               zstep = zstep * 0.5_default
               cycle
            end if
         end if
      end do
   else if (prt%is_gluon () .and. prt%child1%is_gluon ()) then
      do
         zstep = min(zstep, 0.5_default * (zmax - z))
         if (abs (zstep) < 0.00001) exit
         if (integral_over_P_ggg (zmin, z) < random * integral) then
            if (integral_over_P_ggg (zmin, min(z + zstep, zmax)) &
                 < random * integral) then
               z = min(z + zstep, zmax)
               cycle
            else
               zstep = zstep * 0.5_default
               cycle
            end if
         end if
      end do
   else
   end if
   prt%z = z
   prt%x = prt%child1%x / prt%z
 end subroutine generate_trial_z_and_typ
end function shower_generate_next_isr_branching_veto
```

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: find_recoiler => shower_find_recoiler
```

⟨*Shower core: procedures*⟩+≡

```
function shower_find_recoiler (shower, prt) result(recoiler)
  class(shower_t), intent(inout) :: shower
  type(parton_t), intent(inout), target :: prt
  type(parton_t), pointer :: recoiler
  type(parton_t), pointer :: otherprt1, otherprt2
  integer :: n_int
  otherprt1 => null()
  otherprt2 => null()
  DO_INTERACTIONS: do n_int = 1, size(shower%interactions)
     otherprt1 => shower%interactions(n_int)%i%partons(1)%p
     otherprt2 => shower%interactions(n_int)%i%partons(2)%p
     PARTON1: do
        if (associated (otherprt1%parent)) then
           if (.not. otherprt1%parent%is_proton () .and. &
                otherprt1%parent%simulated) then
              otherprt1 => otherprt1%parent
              if (associated (otherprt1, prt)) then
                 exit PARTON1
              end if
           else
              exit PARTON1
           end if
```

```
            else
               exit PARTON1
            end if
         end do PARTON1
         PARTON2: do
            if (associated (otherprt2%parent)) then
               if (.not. otherprt2%parent%is_proton () .and. &
                    otherprt2%parent%simulated) then
                  otherprt2 => otherprt2%parent
                  if (associated (otherprt2, prt)) then
                     exit PARTON2
                  end if
               else
                  exit PARTON2
               end if
            else
               exit PARTON2
            end if
         end do PARTON2

         if (associated (otherprt1, prt) .or. associated (otherprt2, prt)) then
            exit DO_INTERACTIONS
         end if
         if (associated (otherprt1%parent, prt) .or. &
              associated (otherprt2%parent, prt)) then
            exit DO_INTERACTIONS
         end if
      end do DO_INTERACTIONS

      recoiler => null()
      if (associated (otherprt1%parent, prt)) then
         recoiler => otherprt2
      else if (associated (otherprt2%parent, prt)) then
         recoiler => otherprt1
      else if (associated (otherprt1, prt)) then
         recoiler => otherprt2
      else if (associated (otherprt2, prt)) then
         recoiler => otherprt1
      else
         call shower%write ()
         call prt%write ()
         call msg_error ("shower_find_recoiler: no otherparton found")
      end if
   end function shower_find_recoiler
```

⟨*Shower core: procedures*⟩+≡

```
   subroutine shower_isr_step (shower, prt)
      type(shower_t), intent(inout) :: shower
      type(parton_t), target, intent(inout) :: prt
      type(parton_t), pointer :: otherprt => null()
      real(default) :: t, tstep
      real(default), volatile :: integral
      real(default) :: random
      real(default) :: temprand1, temprand2
```

```
    otherprt => shower%find_recoiler (prt)
    ! if (.not. otherprt%child1%belongstointeraction) then
    !    otherprt => otherprt%child1
    ! end if

    if (signal_is_pending ()) return
    t = max(prt%t, prt%child1%t)
    call shower%rng%generate (random)
    ! compare Integral and log(random) instead of random and exp(-Integral)
    random = - twopi * log(random)
    integral = zero
    call shower%rng%generate (temprand1)
    call shower%rng%generate (temprand2)
    tstep = max (abs (0.02_default * t) * temprand1, &
         0.02_default * temprand2 * shower%settings%min_virtuality)
    if (t + 0.5_default * tstep > - shower%settings%min_virtuality) then
       prt%t = prt%mass_squared ()
       call prt%set_simulated ()
    else
       prt%t = t + 0.5_default * tstep
       integral = integral + tstep * &
            integral_over_z_isr (shower, prt, otherprt,(random - integral) / tstep)
       if (integral > random) then
          prt%t = t + 0.5_default * tstep
          prt%x = prt%child1%x / prt%z
          call prt%set_simulated ()
       else
          prt%t = t + tstep
       end if
    end if
  end subroutine shower_isr_step
```

⟨*Shower core: procedures*⟩+≡

```
  function integral_over_z_isr (shower, prt, otherprt, final) result (integral)
    type(shower_t), intent(inout) :: shower
    type(parton_t), intent(inout) :: prt, otherprt
    real(default), intent(in) :: final
    !!! !!! !!! volatile argument: gfortran 7 aggressive optimization (#809)
    real(default), volatile :: integral
    real(default) :: minz, maxz, shat,s
    integer :: quark

    !!! calculate shat -> s of parton-parton system
    shat = (otherprt%momentum + prt%child1%momentum)**2
    !!! calculate s -> s of hadron-hadron system
    s = (otherprt%initial%momentum + prt%initial%momentum)**2
    integral = zero
    minz = prt%child1%x
    maxz = maxzz (shat, s, shower%settings%isr_z_cutoff, &
         shower%settings%isr_minenergy)

    !!! for gluon
    if (prt%child1%is_gluon ()) then
       !!! 1: g->gg
```

```
      prt%type = GLUON
      prt%child2%type = GLUON
        prt%child2%t = abs(prt%t)
        call integral_over_z_part_isr &
             (shower, prt, otherprt, shat, minz, maxz, integral, final)
        if (integral > final) then
           return
        else
           !!! 2: q->gq
           do quark = - shower%settings%max_n_flavors, &
                   shower%settings%max_n_flavors
              if (quark == 0) cycle
              prt%type = quark
              prt%child2%type = quark
              prt%child2%t = abs(prt%t)
              call integral_over_z_part_isr &
                   (shower, prt, otherprt, shat, minz, maxz, integral, final)
              if (integral > final) then
                 return
              end if
           end do
        end if
     else if (prt%child1%is_quark ()) then
        !!! 1: q->qg
        prt%type = prt%child1%type
        prt%child2%type = GLUON
        prt%child2%t = abs(prt%t)
        call integral_over_z_part_isr &
             (shower, prt,otherprt, shat, minz, maxz, integral, final)
        if (integral > final) then
           return
        else
           !!! 2: g->qqbar
           prt%type = GLUON
           prt%child2%type = -prt%child1%type
           prt%child2%t = abs(prt%t)
           call integral_over_z_part_isr &
                (shower, prt,otherprt, shat, minz, maxz, integral, final)
        end if
     end if
  end function integral_over_z_isr
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine integral_over_z_part_isr &
       (shower, prt, otherprt, shat ,minz, maxz, retvalue, final)
    type(shower_t), intent(inout) :: shower
    type(parton_t), intent(inout) :: prt, otherprt
    real(default), intent(in) :: shat, minz, maxz, final
    real(default), intent(inout) :: retvalue
    real(default) :: z, zstep
    real(default) :: r1,r3,s1,s3
    real(default) :: pdf_divisor
    real(default) :: temprand
    real(default), parameter :: zstepfactor = 0.1_default
```

```
real(default), parameter :: zstepmin = 0.0001_default
call msg_debug2 (D_SHOWER, "integral_over_z_part_isr")
if (signal_is_pending ()) return
pdf_divisor = shower%get_pdf &
     (prt%initial%type, prt%child1%x, prt%t, prt%child1%type)
z = minz
s1 = shat + abs(otherprt%t) + abs(prt%child1%t)
r1 = sqrt (s1**2 - four * abs(otherprt%t * prt%child1%t))
ZLOOP: do
   if (signal_is_pending ()) return
   if (z >= maxz) then
      exit
   end if
   call shower%rng%generate (temprand)
   if (prt%child1%is_gluon ()) then
      if (prt%is_gluon ()) then
         !!! g-> gg -> divergencies at z->0 and z->1
         zstep = max(zstepmin, temprand * zstepfactor * z * (one - z))
      else
         !!! q-> gq -> divergencies at z->0
         zstep = max(zstepmin, temprand * zstepfactor * (one - z))
      end if
   else
      if (prt%is_gluon ()) then
         !!! g-> qqbar -> no divergencies
         zstep = max(zstepmin, temprand * zstepfactor)
      else
         !!! q-> qg -> divergencies at z->1
         zstep = max(zstepmin, temprand * zstepfactor * (one - z))
      end if
   end if
   zstep = min(zstep, maxz - z)
   prt%z = z + 0.5_default * zstep
   s3 = shat / prt%z + abs(otherprt%t) + abs(prt%t)
   r3 = sqrt (s3**2 - four * abs(otherprt%t * prt%t))
   !!! TODO: WHY is this if needed?
   if (abs(otherprt%t) > eps0) then
      prt%child2%t = min ((s1 * s3 - r1 * r3) / &
           (two * abs(otherprt%t)) - abs(prt%child1%t) - &
           abs(prt%t), abs(prt%child1%t))
   else
      prt%child2%t = abs(prt%child1%t)
   end if
   do
      prt%child2%momentum%p(0) = sqrt (abs(prt%child2%t))
      if (shower%settings%isr_only_onshell_emitted_partons) then
         prt%child2%t = prt%child2%mass_squared ()
      else
         call prt%child2%next_t_ana (shower%rng)
      end if
      !!! take limits by recoiler into account
      prt%momentum%p(0) = (shat / prt%z + &
           abs(otherprt%t) - abs(prt%child1%t) - &
           prt%child2%t) / (two * sqrt(shat))
```

```
            prt%child2%momentum%p(0) = &
                 prt%momentum%p(0) - prt%child1%momentum%p(0)
            !!! check if E and t of prt%child2 are consistent
            if (prt%child2%momentum%p(0)**2 < prt%child2%t &
                 .and. prt%child2%t > prt%child2%mass_squared ()) then
               !!! E is too small to have p_T^2 = E^2 - t > 0
               !!!      -> cycle to find another solution
               cycle
            else
               !!! E is big enough -> exit
               exit
            end if
         end do
         if (thetabar (prt, otherprt, shower%settings%isr_angular_ordered) &
              .and. pdf_divisor > zero &
              .and. prt%child2%momentum%p(0) > zero) then
            retvalue = retvalue + (zstep / prt%z) * &
                 (D_alpha_s_isr ((one - prt%z) * prt%t, &
                 shower%settings) * &
                 P_prt_to_child1 (prt) * &
                 shower%get_pdf (prt%initial%type, prt%child1%x / prt%z, &
                 prt%t, prt%type)) / (abs(prt%t) * pdf_divisor)
         end if
         if (retvalue > final) then
            exit
         else
            z = z + zstep
         end if
      end do ZLOOP
   end subroutine integral_over_z_part_isr
```

This returns a pointer to the parton with the next ISR branching, again FSR branchings are ignored.

⟨*Shower core: shower: TBP*⟩+≡

```
   procedure :: generate_next_isr_branching => &
        shower_generate_next_isr_branching
```

⟨*Shower core: procedures*⟩+≡

```
   function shower_generate_next_isr_branching &
        (shower) result (next_brancher)
      class(shower_t), intent(inout) :: shower
      type(parton_pointer_t) :: next_brancher
      integer i, index
      type(parton_t),  pointer :: prt
      next_brancher%p => null()
      do
         if (signal_is_pending ()) return
         if (shower_isr_is_finished (shower)) exit
         !!! find mother with highest |t| or pt to be simulated
         index = 0
         call shower%sort_partons ()
         do i = 1,size (shower%partons)
            prt => shower%partons(i)%p
            if (.not. associated (prt)) cycle
```

```
         if (.not. shower%settings%isr_pt_ordered) then
            if (prt%belongstointeraction) cycle
         end if
         if (prt%belongstoFSR) cycle
         if (prt%is_final ()) cycle
         if (.not. prt%belongstoFSR .and. prt%simulated) cycle
         index = i
         exit
      end do
      if (debug_active (D_SHOWER)) then
         if (index == 0) then
            call msg_fatal(" no branchable partons found")
         end if
      end if

      prt => shower%partons(index)%p

      !!! ISR simulation
      if (shower%settings%isr_pt_ordered) then
         call shower_isr_step_pt (shower, prt)
      else
         call shower_isr_step (shower, prt)
      end if
      if (prt%simulated) then
         if (prt%t < zero) then
            next_brancher%p => prt
            if (.not. shower%settings%isr_pt_ordered) &
                 call prt%generate_ps_ini (shower%rng)
            exit
         else
            if (.not. shower%settings%isr_pt_ordered) then
               call shower_replace_parent_by_hadron (shower, prt%child1)
            else
               call shower_replace_parent_by_hadron (shower, prt)
            end if
         end if
      end if
   end do

   !!! some bookkeeping
   call shower%sort_partons ()
   call shower%boost_to_CMframe ()        !!! really necessary?
   call shower%rotate_to_z ()             !!! really necessary?
end function shower_generate_next_isr_branching
```

This is a loop which searches for all emitted and branched partons.

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: generate_fsr_for_isr_partons => &
     shower_generate_fsr_for_partons_emitted_in_ISR
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_generate_fsr_for_partons_emitted_in_ISR (shower)
  class(shower_t), intent(inout) :: shower
  integer :: n_int, i
```

```
    type(parton_t), pointer :: prt
    if (shower%settings%isr_only_onshell_emitted_partons) return
    call msg_debug (D_SHOWER, "shower_generate_fsr_for_partons_emitted_in_ISR")
    INTERACTIONS_LOOP: do n_int = 1, size (shower%interactions)
       INCOMING_PARTONS_LOOP: do i = 1, 2
          if (signal_is_pending ()) return
          prt => shower%interactions(n_int)%i%partons(i)%p
          PARENT_PARTONS_LOOP: do
             if (associated (prt%parent)) then
                if (.not. prt%parent%is_proton ()) then
                   prt => prt%parent
                else
                   exit
                end if
             else
                exit
             end if
             if (associated (prt%child2)) then
                if (prt%child2%is_branched ()) then
                   call shower_parton_generate_fsr (shower, prt%child2)
                end if
             else
                ! call msg_fatal ("Shower: no child2 associated?")
             end if
          end do PARENT_PARTONS_LOOP
       end do INCOMING_PARTONS_LOOP
    end do INTERACTIONS_LOOP
  end subroutine shower_generate_fsr_for_partons_emitted_in_ISR
```

This executes the branching generated by `shower_generate_next_isr_branching`, that means it generates the flavors, momenta, etc.

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: execute_next_isr_branching => shower_execute_next_isr_branching
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine shower_execute_next_isr_branching (shower, prtp)
    class(shower_t), intent(inout) :: shower
    type(parton_pointer_t), intent(inout) :: prtp
    type(parton_t), pointer :: prt, otherprt
    type(parton_t), pointer :: prta, prtb, prtc, prtr
    real(default) :: mbr
    real(default) :: phirand
    call msg_debug (D_SHOWER, "shower_execute_next_isr_branching")
    if (.not. associated (prtp%p)) then
       call msg_fatal ("Shower: prtp not associated")
    end if

    prt => prtp%p

    if ((.not. shower%settings%isr_pt_ordered .and. &
         prt%t > - shower%settings%min_virtuality) .or. &
         (shower%settings%isr_pt_ordered .and. prt%scale < D_Min_scale)) then
       call msg_error ("Shower: no branching to be executed.")
    end if
```

```
otherprt => shower%find_recoiler (prt)
if (shower%settings%isr_pt_ordered) then
   !!! get the recoiler
   otherprt => shower%find_recoiler (prt)
   if (associated (otherprt%parent)) then
      !!! Why only for pt ordered
      if (.not. otherprt%parent%is_proton () .and. &
           shower%settings%isr_pt_ordered) otherprt => otherprt%parent
   end if
   if (.not. associated (prt%parent)) then
      call shower%add_parent (prt)
   end if
   prt%parent%belongstoFSR = .false.
   if (.not. associated (prt%parent%child2)) then
      call shower%add_child (prt%parent, 2)
   end if

   prta => prt%parent          !!! new parton a with branching a->bc
   prtb => prt                 !!! former parton
   prtc => prt%parent%child2   !!! emitted parton
   prtr => otherprt            !!! recoiler

   mbr = (prtb%momentum + prtr%momentum)**1

   !!! 1. assume you are in the restframe
   !!! 2. rotate by random phi
   call shower%rng%generate (phirand)
   phirand = twopi * phirand
   call shower_apply_lorentztrafo (shower, &
        rotation(cos(phirand), sin(phirand),vector3_canonical(3)))
   !!! 3. Put the b off-shell
   !!! and
   !!! 4. construct the massless a
   !!! and the parton (eventually emitted by a)

   !!! generate the flavor of the parent (prta)
   if (prtb%aux_pt /= 0) prta%type = prtb%aux_pt
   if (prtb%is_quark ()) then
      if (prta%type == prtb%type) then
         !!! (anti)-quark -> (anti-)quark + gluon
         prta%type = prtb%type   ! quarks have same flavor
         prtc%type = GLUON        ! emitted gluon
      else
         !!! gluon -> quark + antiquark
         prta%type = GLUON
         prtc%type = - prtb%type
      end if
   else if (prtb%is_gluon ()) then
      prta%type = GLUON
      prtc%type = GLUON
   else
      ! STOP "Bug in shower_execute_next_branching: neither quark nor gluon"
   end if
```

```
   prta%initial => prtb%initial
   prta%belongstoFSR = .false.
   prta%scale = prtb%scale
   prta%x = prtb%x / prtb%z

   prtb%momentum = vector4_moving ((mbr**2 + prtb%t) / (two * mbr), &
        vector3_canonical(3) * &
        sign ((mbr**2 - prtb%t) / (two * mbr), &
        prtb%momentum%p(3)))
   prtr%momentum = vector4_moving ((mbr**2 - prtb%t) / (two * mbr), &
        vector3_canonical(3) * &
        sign( (mbr**2 - prtb%t) / (two * mbr), &
        prtr%momentum%p(3)))

   prta%momentum = vector4_moving ((0.5_default / mbr) * &
        ((mbr**2 / prtb%z) + prtb%t - prtc%mass_squared ()), &
        vector3_null)
   prta%momentum = vector4_moving (prta%momentum%p(0), &
        vector3_canonical(3) * &
        (0.5_default / prtb%momentum%p(3)) * &
        ((mbr**2 / prtb%z) - two &
        * prtr%momentum%p(0) * prta%momentum%p(0) ) )
   if (prta%momentum%p(0)**2 - prta%momentum%p(3)**2 - &
        prtc%mass_squared () > zero) then
      !!! This SHOULD be always fulfilled???
      prta%momentum = vector4_moving (prta%momentum%p(0), &
           vector3_moving([sqrt (prta%momentum%p(0)**2 - &
           prta%momentum%p(3)**2 - &
           prtc%mass_squared ()), zero, &
           prta%momentum%p(3)]))
   end if
   prtc%momentum = prta%momentum - prtb%momentum

   !!! 5. rotate to have a along z-axis
   call shower%boost_to_CMframe ()
   call shower%rotate_to_z ()
   !!! 6. rotate back in phi
   call shower_apply_lorentztrafo (shower, rotation &
        (cos(-phirand), sin(-phirand), vector3_canonical(3)))
else
   if (prt%child2%t > prt%child2%mass_squared ()) then
      call shower_add_children_of_emitted_timelike_parton &
           (shower, prt%child2)
      call prt%child2%set_simulated ()
   end if

   call shower%add_parent (prt)
   call shower%add_child (prt%parent, 2)

   prt%parent%momentum = prt%momentum
   prt%parent%t = prt%t
   prt%parent%x = prt%x
   prt%parent%initial => prt%initial
```

```
   prt%parent%belongstoFSR = .false.

   prta => prt
   prtb => prt%child1
   prtc => prt%child2
end if
if (signal_is_pending ()) return
if (shower%settings%isr_pt_ordered) then
   call prt%parent%generate_ps_ini (shower%rng)
else
   call prt%generate_ps_ini (shower%rng)
end if

!!! add color connections
if (prtb%is_quark ()) then

   if (prta%type == prtb%type) then
      if (prtb%type > 0) then
         !!! quark -> quark + gluon
         prtc%c2 = prtb%c1
         prtc%c1 = shower%get_next_color_nr ()
         prta%c1 = prtc%c1
      else
         !!! antiquark -> antiquark + gluon
         prtc%c1 = prtb%c2
         prtc%c2 = shower%get_next_color_nr ()
         prta%c2 = prtc%c2
      end if
   else
      !!! gluon -> quark + antiquark
      if (prtb%type > 0) then
         !!! gluon -> quark + antiquark
         prta%c1 = prtb%c1
         prtc%c1 = 0
         prtc%c2 = shower%get_next_color_nr ()
         prta%c2 = prtc%c2
      else
         !!! gluon -> antiquark + quark
         prta%c2 = prtb%c2
         prtc%c1 = shower%get_next_color_nr ()
         prtc%c2 = 0
         prta%c1 = prtc%c1
        end if
     end if
  else if (prtb%is_gluon ()) then
     if (prta%is_gluon ()) then
        !!! g -> gg
        prtc%c2 = prtb%c1
        prtc%c1 = shower%get_next_color_nr ()
        prta%c1 = prtc%c1
        prta%c2 = prtb%c2
     else if (prta%is_quark ()) then
        if (prta%type > 0) then
           prta%c1 = prtb%c1
```

```
            prta%c2 = 0
            prtc%c1 = prtb%c2
            prtc%c2 = 0
         else
            prta%c1 = 0
            prta%c2 = prtb%c2
            prtc%c1 = 0
            prtc%c2 = prtb%c1
         end if
      end if
   end if

   call shower%sort_partons ()
   call shower%boost_to_CMframe ()
   call shower%rotate_to_z ()

end subroutine shower_execute_next_isr_branching
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_remove_parents_and_stuff (shower, prt)
  type(shower_t), intent(inout) :: shower
  type(parton_t), intent(inout), target :: prt
  type(parton_t), pointer :: actprt, nextprt
  nextprt => prt%parent
  actprt => null()
  !!! remove children of emitted timelike parton
  if (associated (prt%child2)) then
     if (associated (prt%child2%child1)) then
        call shower_remove_parton_from_partons_recursive &
             (shower, prt%child2%child1)
     end if
     prt%child2%child1 => null()
     if (associated (prt%child2%child2)) then
        call shower_remove_parton_from_partons_recursive &
             (shower, prt%child2%child2)
     end if
     prt%child2%child2 => null()
  end if
  do
     actprt => nextprt
     if (.not. associated (actprt)) then
        exit
     else if (actprt%is_proton ()) then
        !!! remove beam-remnant
        call shower_remove_parton_from_partons (shower, actprt%child2)
        exit
     end if
     if (associated (actprt%parent)) then
        nextprt => actprt%parent
     else
        nextprt => null()
     end if
     call shower_remove_parton_from_partons_recursive &
          (shower, actprt%child2)
```

```
      call shower_remove_parton_from_partons (shower, actprt)

    end do
    prt%parent=>null()

  end subroutine shower_remove_parents_and_stuff
```

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: get_ISR_scale => shower_get_ISR_scale
```

⟨*Shower core: procedures*⟩+≡

```
  function shower_get_ISR_scale (shower) result (scale)
    class(shower_t), intent(in) :: shower
    real(default) :: scale
    type(parton_t), pointer :: prt1, prt2
    integer :: i
    scale = zero
    do i = 1, size (shower%interactions)
       call interaction_find_partons_nearest_to_hadron &
            (shower%interactions(i)%i, prt1, prt2, &
            shower%settings%isr_pt_ordered)
       if (.not. prt1%simulated .and. abs(prt1%scale) > scale) &
            scale = abs(prt1%scale)
       if (.not. prt1%simulated .and. abs(prt2%scale) > scale) &
            scale = abs(prt2%scale)
    end do
  end function shower_get_ISR_scale
```

⟨*Shower core: shower: TBP*⟩+≡

```
  procedure :: set_max_isr_scale => shower_set_max_isr_scale
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine shower_set_max_isr_scale (shower, newscale)
    class(shower_t), intent(inout) :: shower
    real(default), intent(in) :: newscale
    real(default) :: scale
    type(parton_t), pointer :: prt
    integer :: i,j
    call msg_debug (D_SHOWER, "shower_set_max_isr_scale: newscale", &
         newscale)
    if (shower%settings%isr_pt_ordered) then
       scale = newscale
    else
       scale = - abs (newscale)
    end if

    INTERACTIONS: do i = 1, size (shower%interactions)
       PARTONS: do j = 1, 2
          prt => shower%interactions(i)%i%partons(j)%p
          do
             if (.not. shower%settings%isr_pt_ordered) then
                if (prt%belongstointeraction) prt => prt%parent
             end if
               if (prt%t < scale) then
```

```
            if (associated (prt%parent)) then
               prt => prt%parent
            else
               exit   !!! unresolved prt found
            end if
         else
            exit   !!! prt with scale above newscale found
         end if
      end do
      if (.not. shower%settings%isr_pt_ordered) then
         if (prt%child1%belongstointeraction .or. &
              prt%is_proton ()) then
            !!! don't reset scales of "first" spacelike partons
            !!!    in virtuality ordered shower or hadrons
            cycle
         end if
      else
         if (prt%is_proton ()) then
            !!! don't reset scales of hadrons
            cycle
         end if
      end if
      if (shower%settings%isr_pt_ordered) then
         prt%scale = scale
      else
         prt%t = scale
      end if
      call prt%set_simulated (.false.)
      call shower_remove_parents_and_stuff (shower, prt)
   end do PARTONS
 end do INTERACTIONS
end subroutine shower_set_max_isr_scale
```

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: interaction_generate_fsr_2ton => &
     shower_interaction_generate_fsr_2ton
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_interaction_generate_fsr_2ton (shower, interaction)
  class(shower_t), intent(inout) :: shower
  type(shower_interaction_t), intent(inout) :: interaction
  type(parton_t), pointer :: prt
  prt => interaction%partons(3)%p
  do
     if (.not. associated (prt%parent)) exit
     prt => prt%parent
  end do
  call shower_parton_generate_fsr (shower, prt)
  call shower_parton_update_color_connections (shower, prt)
end subroutine shower_interaction_generate_fsr_2ton
```

Perform the FSR for one parton, it is assumed, that the parton already branched. Hence, its children are to be simulated. This procedure is intended for branched

FSR-partons emitted in the ISR.

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_parton_generate_fsr (shower, prt)
  type(shower_t), intent(inout) :: shower
  type(parton_t), intent(inout), target :: prt
  type(parton_pointer_t), dimension(:), allocatable :: partons
  logical :: single_emission = .false.
  call msg_debug (D_SHOWER, "shower_parton_generate_fsr")
  if (signal_is_pending ()) return
  if (debug_active (D_SHOWER)) then
     if (.not. prt%is_branched ()) then
        call msg_error ("shower_parton_generate_fsr: parton not branched")
        return
     end if
     if (prt%child1%simulated .or. &
         prt%child2%simulated) then
        print *, "children already simulated for parton ", prt%nr
        return
     end if
  end if
  allocate (partons(1))
  partons(1)%p => prt
  if (single_emission) then
     call shower%parton_pointer_array_generate_fsr (partons, partons)
  else
     call shower%parton_pointer_array_generate_fsr_recursive (partons)
  end if
end subroutine shower_parton_generate_fsr
```

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: parton_pointer_array_generate_fsr_recursive => &
     shower_parton_pointer_array_generate_fsr_recursive
```

⟨*Shower core: procedures*⟩+≡

```
  recursive subroutine shower_parton_pointer_array_generate_fsr_recursive &
       (shower, partons)
    class(shower_t), intent(inout) :: shower
    type(parton_pointer_t), dimension(:), allocatable, intent(inout) :: &
         partons
    type(parton_pointer_t), dimension(:), allocatable :: partons_new
    call msg_debug (D_SHOWER, "shower_parton_pointer_array_generate_fsr_recursive")
    if (signal_is_pending ()) return
    if (size (partons) == 0) return
    call shower%parton_pointer_array_generate_fsr (partons, partons_new)
    call shower%parton_pointer_array_generate_fsr_recursive (partons_new)
  end subroutine shower_parton_pointer_array_generate_fsr_recursive
```

⟨*Shower core: shower: TBP*⟩+≡

```
procedure :: parton_pointer_array_generate_fsr => &
     shower_parton_pointer_array_generate_fsr
```

⟨*Shower core: procedures*⟩+≡

```
subroutine shower_parton_pointer_array_generate_fsr &
     (shower, partons, partons_new)
  class(shower_t), intent(inout) :: shower
```

```
    type(parton_pointer_t), dimension(:), allocatable, intent(inout) :: &
         partons
    type(parton_pointer_t), dimension(:), allocatable, intent(out) :: &
         partons_new
    integer :: i, size_partons, size_partons_new
    call msg_debug (D_SHOWER, "shower_parton_pointer_array_generate_fsr")
    !!! Simulate highest/first parton
    call shower_simulate_children_ana (shower, partons(1)%p)
    !!! check for new daughters to be included in new_partons
    size_partons = size (partons)
    size_partons_new = size_partons - 1   !!! partons(1) not needed anymore
    if (partons(1)%p%child1%is_branched ()) &
         size_partons_new = size_partons_new + 1
    if (partons(1)%p%child2%is_branched ()) &
         size_partons_new = size_partons_new + 1

    allocate (partons_new (1:size_partons_new))

    if (size_partons > 1) then
       do i = 2, size_partons
          partons_new (i - 1)%p => partons(i)%p
       end do
    end if
    if (partons(1)%p%child1%is_branched ()) &
         partons_new (size_partons)%p => partons(1)%p%child1
    if (partons(1)%p%child2%is_branched ()) then
    !!! check if child1 is already included
       if (size_partons_new == size_partons) then
          partons_new (size_partons)%p => partons(1)%p%child2
       else if (size_partons_new == size_partons + 1) then
          partons_new (size_partons + 1)%p => partons(1)%p%child2
       else
          call msg_fatal ("Shower: wrong sizes in" &
               // "shower_parton_pointer_array_generate_fsr")
       end if
    end if
    deallocate (partons)

  end subroutine shower_parton_pointer_array_generate_fsr
```

⟨*Shower core: procedures*⟩+≡

```
  recursive subroutine shower_parton_update_color_connections &
       (shower, prt)
    type(shower_t), intent(inout) :: shower
    type(parton_t), intent(inout) :: prt
    real(default) :: temprand
    if (.not. associated (prt%child1) .or. &
        .not. associated (prt%child2)) return

    if (signal_is_pending ()) return
    if (prt%is_gluon ()) then
       if (prt%child1%is_quark ()) then
          !!! give the quark the colorpartner and the antiquark
          !!!     the anticolorpartner
```

```
      if (prt%child1%type > 0) then
         !!! child1 is quark, child2 is antiquark
         prt%child1%c1 = prt%c1
         prt%child2%c2 = prt%c2
      else
         !!! child1 is antiquark, child2 is quark
         prt%child1%c2 = prt%c2
         prt%child2%c1 = prt%c1
      end if
   else
      !!! g -> gg splitting -> random choosing of partners
      call shower%rng%generate (temprand)
      if (temprand > 0.5_default) then
         prt%child1%c1 = prt%c1
         prt%child1%c2 = shower%get_next_color_nr ()
         prt%child2%c1 = prt%child1%c2
         prt%child2%c2 = prt%c2
      else
         prt%child1%c2 = prt%c2
         prt%child2%c1 = prt%c1
         prt%child2%c2 = shower%get_next_color_nr ()
         prt%child1%c1 = prt%child2%c2
      end if
   end if
else if (prt%is_quark ()) then
   if (prt%child1%is_quark ()) then
      if (prt%child1%type > 0) then
         !!! q -> q + g
         prt%child2%c1 = prt%c1
         prt%child2%c2 = shower%get_next_color_nr ()
         prt%child1%c1 = prt%child2%c2
      else
         !!! qbar -> qbar + g
         prt%child2%c2 = prt%c2
         prt%child2%c1 = shower%get_next_color_nr ()
         prt%child1%c2 = prt%child2%c1
      end if
   else
      if (prt%child2%type > 0) then
         !!! q -> g + q
         prt%child1%c1 = prt%c1
         prt%child1%c2 = shower%get_next_color_nr ()
         prt%child2%c1 = prt%child1%c2
      else
         !!! qbar -> g + qbar
         prt%child1%c2 = prt%c2
         prt%child1%c1 = shower%get_next_color_nr ()
         prt%child2%c2 = prt%child1%c1
      end if
   end if
end if

call shower_parton_update_color_connections (shower, prt%child1)
call shower_parton_update_color_connections (shower, prt%child2)
```

```
    end subroutine shower_parton_update_color_connections
```

The next two routines are for PDFs. Wrapper function to return parton densities.

⟨*Shower core: shower: TBP*⟩+≡

```
    procedure :: get_pdf => shower_get_pdf
```

⟨*Shower core: procedures*⟩+≡

```
    function shower_get_pdf (shower, mother, x, Q2, daughter) result (pdf)
      ⟨get pdf⟩
      if (x > eps0) then
         pdf = pdf / x
      end if
    end function shower_get_pdf
```

⟨*Shower core: shower: TBP*⟩+≡

```
    procedure :: get_xpdf => shower_get_xpdf
```

⟨*Shower core: procedures*⟩+≡

```
    function shower_get_xpdf (shower, mother, x, Q2, daughter) result (pdf)
      ⟨get pdf⟩
    end function shower_get_xpdf
```

⟨*get pdf*⟩≡

```
  class(shower_t), intent(inout), target :: shower
  integer, intent(in) :: mother, daughter
  real(default), intent(in) :: x, Q2
  real(default) :: pdf
  real(double), save :: f(-6:6) = 0._double
  real(double), save :: lastx, lastQ2 = 0._double
  pdf = zero
  if (debug_active (D_SHOWER)) then
     if (abs (mother) /= PROTON) then
        call msg_debug (D_SHOWER, "mother", mother)
        call msg_fatal ("Shower: pdf only implemented for (anti-)proton")
     end if
     if (.not. (abs (daughter) >= 1 .and. abs (daughter) <= 6 .or. &
               daughter == GLUON)) then
        call msg_debug (D_SHOWER, "daughter", daughter)
        call msg_fatal ("Shower: error in pdf, unknown daughter")
     end if
  end if
  if (x > zero .and. x < one) then
     if ((dble(Q2) - lastQ2) > eps0 .or. (dble(x) - lastx) > eps0) then
        call shower%pdf_data%evolve &
             (dble(x), sqrt (abs (dble(Q2))), f)
     end if
     if (abs (daughter) >= 1 .and. abs (daughter) <= 6) then
        pdf = max (f(daughter * sign (1,mother)), tiny_10)
     else
        pdf = max (f(0), tiny_10)
     end if
  end if
  lastQ2 = dble(Q2)
```

```
  lastx  = dble(x)
```

Convert Whizard shower to Pythia6. Currently only works for one interaction.

⟨*Shower core: shower: TBP*⟩+≡

```
    procedure :: converttopythia => shower_converttopythia
```

⟨*Shower core: procedures*⟩+≡

```
    subroutine shower_converttopythia (shower)
      class(shower_t), intent(in) :: shower
    ⟨PYJETS COMMON BLOCK⟩
      type(parton_t), pointer :: pp, ppparent
      integer :: i
      K = 0
      do i = 1, 2
         !!! get history of the event
         pp => shower%interactions(1)%i%partons(i)%p
         !!! add these partons to the event record
         if (associated (pp%initial)) then
            !!! add hadrons
            K(i,1) = 21
            K(i,2) = pp%initial%type
            K(i,3) = 0
            P(i,1:5) = pp%initial%momentum_to_pythia6 ()
            !!! add partons emitted by the hadron
            ppparent => pp
            do while (associated (ppparent%parent))
               if (ppparent%parent%is_proton ()) then
                  exit
               else
                  ppparent => ppparent%parent
               end if
            end do
            K(i+2,1) = 21
            K(i+2,2) = ppparent%type
            K(i+2,3) = i
            P(i+2,1:5) = ppparent%momentum_to_pythia6 ()
            !!! add partons in the initial state of the ME
            K(i+4,1) = 21
            K(i+4,2) = pp%type
            K(i+4,3) = i
            P(i+4,1:5) = pp%momentum_to_pythia6 ()
         else
            !!! for e+e- without ISR all entries are the same
            K(i,1) = 21
            K(i,2) = pp%type
            K(i,3) = 0
            P(i,1:5) = pp%momentum_to_pythia6 ()
            P(i+2,:) = P(1,:)
            K(i+2,:) = K(1,:)
            K(i+2,3) = i
            P(i+4,:) = P(1,:)
            K(i+4,:) = K(1,:)
            K(i+4,3) = i
            P(i+4,5) = 0.
         end if
```

```
    end do
    N = 6
    !!! create intermediate (fake) Z-Boson
    !K(7,1) = 21
    !K(7,2) = 23
    !K(7,3) = 0
    !P(7,1:4) = P(5,1:4) + P(6,1:4)
    !P(7,5) = P(7,4)**2 - P(7,3)**2 - P(7,2)**2 - P(7,1)**2
    !N = 7
    !!! include partons in the final state of the hard matrix element
    do i = 1, size (shower%interactions(1)%i%partons) - 2
       !!! get partons that are in the final state of the hard matrix element
       pp => shower%interactions(1)%i%partons(2+i)%p
       !!! add these partons to the event record
       K(7+I,1) = 21
       K(7+I,2) = pp%type
       K(7+I,3) = 7
       P(7+I,1:5) = pp%momentum_to_pythia6 ()
       !N = 7 + I
       N = 6 + I
    end do
    !!! include "Z" (again)
    !N = N + 1
    !K(N,1) = 11
    !K(N,2) = 23
    !K(N,3) = 7
    !P(N,1:5) = P(7,1:5)
    !nz = N
    !!! include partons from the final state of the parton shower
    call shower_transfer_final_partons_to_pythia (shower, 8)
    !!! set "children" of "Z"
    !K(nz,4) = 11
    !K(nz,5) = N

    !!! be sure to remove the next partons (=first obsolete partons)
    !!! otherwise they might be interpreted as thrust information
    K(N+1:N+3,1:3) = 0
  end subroutine shower_converttopythia
```

⟨*Shower core: procedures*⟩+≡

```
  subroutine shower_transfer_final_partons_to_pythia (shower, first)
  ⟨PYJETS COMMON BLOCK⟩
    type(shower_t), intent(in) :: shower
    integer, intent(in) :: first
    type(parton_t), pointer :: prt
    integer :: i, j, n_finals
    type(parton_t), dimension(:), allocatable :: final_partons
    type(parton_t) :: temp_parton
    integer :: minindex, maxindex

    prt => null()

    !!! get total number of final partons
    n_finals = 0
```

```
do i = 1, size (shower%partons)
   if (.not. associated (shower%partons(i)%p)) cycle
   prt => shower%partons(i)%p
   if (.not. prt%belongstoFSR) cycle
   if (associated (prt%child1)) cycle
   n_finals = n_finals + 1
end do

allocate (final_partons(1:n_finals))
j = 1
do i = 1, size (shower%partons)
   if (.not. associated (shower%partons(i)%p)) cycle
   prt => shower%partons(i)%p
   if (.not. prt%belongstoFSR) cycle
   if (associated (prt%child1)) cycle
   final_partons(j) = shower%partons(i)%p
   j = j + 1
end do

!!! move quark to front as beginning of color string
minindex = 1
maxindex = size (final_partons)
FIND_Q: do i = minindex, maxindex
   if (final_partons(i)%type >= 1 .and. final_partons(i)%type <= 6) then
      temp_parton = final_partons(minindex)
      final_partons(minindex) = final_partons(i)
      final_partons(i) = temp_parton
      exit FIND_Q
   end if
end do FIND_Q

!!! sort so that connected partons are next to each other, don't care about zeros
do i = 1, size (final_partons)
   !!! ensure that final_partnons begins with a color (not an anticolor)
   if (final_partons(i)%c1 > 0 .and. final_partons(i)%c2 == 0) then
      if (i == 1) then
         exit
      else
         temp_parton = final_partons(1)
         final_partons(1) = final_partons(i)
         final_partons(i) = temp_parton
         exit
      end if
   end if
end do

do i = 1, size (final_partons) - 1
   !!! search for color partner and move it to i + 1
   PARTNERS: do j = i + 1, size (final_partons)
      if (final_partons(j)%c2 == final_partons(i)%c1) exit PARTNERS
   end do PARTNERS
   if (j > size (final_partons)) then
      print *, "no color connected parton found" !WRONG???
      print *, "particle: ", final_partons(i)%nr, " index: ", &
```

```
            final_partons(i)%c1
         exit
      end if
      temp_parton = final_partons(i + 1)
      final_partons(i + 1) = final_partons(j)
      final_partons(j) = temp_parton
   end do

   !!! transfering partons
   do i = 1, size (final_partons)
      prt = final_partons(i)
      N = N + 1
      K(N,1) = 2
      if (prt%c1 == 0) K(N,1) = 1       !!! end of color string
      K(N,2) = prt%type
      !K(N,3) = first
      K(N,3) = 0
      K(N,4) = 0
      K(N,5) = 0
      P(N,1:5) = prt%momentum_to_pythia6()
   end do
   deallocate (final_partons)
end subroutine shower_transfer_final_partons_to_pythia
```

21.4 Interface to PYTHIA

⟨shower_pythia6.f90⟩≡

```
⟨File header⟩

module shower_pythia6

⟨Use kinds with double⟩
⟨Use strings⟩
  use constants
  use numeric_utils, only: vanishes
  use io_units
  use physics_defs
  use diagnostics
  use os_interface
  use lorentz
  use subevents
  use shower_base
  use particles
  use model_data
  use hep_common
  use pdf
  use helicities
  use tauola_interface

⟨Standard module head⟩

⟨Shower pythia6: public⟩
```

```
  ⟨Shower pythia6: variables⟩
  ⟨Shower pythia6: types⟩

  contains

  ⟨Shower pythia6: procedures⟩

    end module shower_pythia6
```

⟨*PYJETS COMMON BLOCK*⟩≡

```
    integer :: N, NPAD, K
    real(double) :: P, V
    COMMON/PYJETS/N,NPAD,K(4000,5),P(4000,5),V(4000,5)
    SAVE /PYJETS/
```

⟨*Shower pythia6: variables*⟩≡

```
    integer :: N_old
```

The PYTHIA6 shower type.

⟨*Shower pythia6: public*⟩≡

```
    public :: shower_pythia6_t
```

⟨*Shower pythia6: types*⟩≡

```
    type, extends (shower_base_t) :: shower_pythia6_t
       integer :: initialized_for_NPRUP = 0
       logical :: warning_given = .false.
     contains
       ⟨Shower pythia6: shower pythia6: TBP⟩
    end type shower_pythia6_t
```

Initialize the PYTHIA6 shower.

⟨*Shower pythia6: shower pythia6: TBP*⟩≡

```
    procedure :: init => shower_pythia6_init
```

⟨*Shower pythia6: procedures*⟩≡

```
    subroutine shower_pythia6_init (shower, settings, taudec_settings, pdf_data)
      class(shower_pythia6_t), intent(out) :: shower
      type(shower_settings_t), intent(in) :: settings
      type(taudec_settings_t), intent(in) :: taudec_settings
      type(pdf_data_t), intent(in) :: pdf_data
      call msg_debug (D_SHOWER, "shower_pythia6_init")
      shower%settings = settings
      shower%taudec_settings = taudec_settings
      call pythia6_set_verbose (settings%verbose)
      call shower%pdf_data%init (pdf_data)
      shower%name = "PYTHIA6"
      call shower%write_msg ()
    end subroutine shower_pythia6_init
```

⟨*Shower pythia6: shower pythia6: TBP*⟩+≡

```
    procedure :: prepare_new_event => shower_pythia6_prepare_new_event
```

⟨*Shower pythia6: procedures*⟩+≡

```
subroutine shower_pythia6_prepare_new_event (shower)
  class(shower_pythia6_t), intent(inout) :: shower
end subroutine shower_pythia6_prepare_new_event
```

⟨*Shower pythia6: shower pythia6: TBP*⟩+≡

```
procedure :: import_particle_set => shower_pythia6_import_particle_set
```

⟨*Shower pythia6: procedures*⟩+≡

```
subroutine shower_pythia6_import_particle_set &
       (shower, particle_set, os_data, scale)
  class(shower_pythia6_t), target, intent(inout) :: shower
  type(particle_set_t), intent(in) :: particle_set
  type(os_data_t), intent(in) :: os_data
  real(default), intent(in) :: scale
  type(particle_set_t) :: pset_reduced
  call msg_debug (D_SHOWER, "shower_pythia6_import_particle_set")
  if (debug_active (D_SHOWER)) then
     print *, 'IDBMUP(1:2) =    ', IDBMUP(1:2)
     print *, 'EBMUP, PDFGUP =    ', EBMUP, PDFGUP
     print *, 'PDFSUP, IDWTUP =    ', PDFSUP, IDWTUP
     print *, "NPRUP = ", NPRUP
     call particle_set%write (summary=.true., compressed=.true.)
  end if
  call particle_set%reduce (pset_reduced)
  if (debug2_active (D_SHOWER)) then
     print *, 'After particle_set%reduce: pset_reduced'
     call pset_reduced%write (summary=.true., compressed=.true.)
  end if
  call hepeup_from_particle_set (pset_reduced, tauola_convention=.true.)
  call hepeup_set_event_parameters (proc_id = 1)
  call hepeup_set_event_parameters (scale = scale)
end subroutine shower_pythia6_import_particle_set
```

⟨*Shower pythia6: shower pythia6: TBP*⟩+≡

```
procedure :: generate_emissions => shower_pythia6_generate_emissions
```

⟨*Shower pythia6: procedures*⟩+≡

```
subroutine shower_pythia6_generate_emissions &
       (shower, valid, number_of_emissions)
  IMPLICIT DOUBLE PRECISION(A-H, O-Z)
  IMPLICIT INTEGER(I-N)
  COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
  SAVE/PYDAT1/
  class(shower_pythia6_t), intent(inout), target :: shower
  logical, intent(out) :: valid
  integer, optional, intent(in) :: number_of_emissions
  integer :: N, NPAD, K
  real(double) :: P, V
  common /PYJETS/ N, NPAD, K(4000,5), P(4000,5), V(4000,5)
  COMMON/PYDAT2/KCHG(500,4),PMAS(500,4),PARF(2000),VCKM(4,4)
  COMMON/PYINT4/MWID(500),WIDS(500,5)
  save /PYJETS/,/PYDAT2/,/PYINT4/
  integer :: u_W2P
```

```
integer :: i
real(double) :: beta_z, pz_in, E_in
integer, parameter :: lower = 5
real(double), parameter :: beta_x = 0.0_double
real(double), parameter :: beta_y = 0.0_double
real(double), parameter :: theta = 0.0_double
real(double), parameter :: phi = 0.0_double
if (signal_is_pending ()) return
call pythia6_setup_lhe_io_units (u_W2P)
call w2p_write_lhef_event (u_W2P)
rewind (u_W2P)
call pythia6_set_last_treated_line(6)
call shower%transfer_settings ()
if (debug_active (D_SHOWER)) then
   print *, ' Before pyevnt, before boosting :'
   call pylist(2)
end if
call msg_debug (D_SHOWER, "calling pyevnt")
! TODO: (bcn 2015-04-24) doesnt change anything I think
! P(1,1:5) = pset_reduced%prt(1)%momentum_to_pythia6 ()
! P(2,1:5) = pset_reduced%prt(2)%momentum_to_pythia6 ()
call pyevnt ()
call pyedit(12)
do i = 1, n
  if (K(i,1) == 14 .and. abs(K(i,2)) >= 11 .and. abs(K(i,2)) <= 16) then
    if (K(i,4) > 0 .and. K(i,5) > 0 .and. K(i,4) < N .and. K(i,5) < N) then
      K(i,1) = 11
      K(i,4) = K(K(i,4),3)
      K(i,5) = K(K(i,5),3)
    end if
  end if
end do
if (.not. shower%settings%hadron_collision) then
   pz_in = pup(3,1) + pup(3,2)
   E_in = pup(4,1) + pup(4,2)
   beta_z = pz_in / E_in
   call pyrobo (lower, N, theta, phi, beta_x, beta_y, beta_z)
end if
if (debug_active (D_SHOWER)) then
   print *, ' After pyevnt, after boosting :'
   call pylist(2)
   if (debug2_active (D_SHOWER)) then
      call pystat (5)
      do i = 1, 200
         print *, 'MSTJ (', i, ') = ', MSTJ(i)
         print *, 'MSTU (', i, ') = ', MSTU(i)
         print *, 'PMAS (', i, ') = ', PMAS(i,1), PMAS(i,2)
         print *, 'MWID (', i, ') = ', MWID(i)
         print *, 'PARJ (', i, ') = ', PARJ(i)
      end do
   end if
end if
close (u_W2P)
valid = pythia6_handle_errors ()
```

```
    end subroutine shower_pythia6_generate_emissions
```

⟨*Shower pythia6: shower pythia6: TBP*⟩+≡

```
    procedure :: make_particle_set => shower_pythia6_make_particle_set
```

⟨*Shower pythia6: procedures*⟩+≡

```
    subroutine shower_pythia6_make_particle_set &
           (shower, particle_set, model, model_hadrons)
      class(shower_pythia6_t), intent(in) :: shower
      type(particle_set_t), intent(inout) :: particle_set
      class(model_data_t), intent(in), target :: model
      class(model_data_t), intent(in), target :: model_hadrons
      call shower%combine_with_particle_set (particle_set, model, model_hadrons)
    end subroutine shower_pythia6_make_particle_set
```

⟨*Shower pythia6: shower pythia6: TBP*⟩+≡

```
    procedure :: transfer_settings => shower_pythia6_transfer_settings
```

⟨*Shower pythia6: procedures*⟩+≡

```
    subroutine shower_pythia6_transfer_settings (shower)
      class(shower_pythia6_t), intent(inout) :: shower
      character(len=10) :: buffer
      real(default) :: rand
      logical, save :: tauola_initialized = .false.
      call msg_debug (D_SHOWER, "shower_pythia6_transfer_settings")
      !!! We repeat these as they are overwritten by the hadronization
      call pygive ("MSTP(111)=1")     !!! Allow hadronization and decays
      call pygive ("MSTJ(1)=0")       !!! No jet fragmentation
      call pygive ("MSTJ(21)=1")      !!! Allow decays but no jet fragmentation

      if (shower%initialized_for_NPRUP >= NPRUP) then
        call msg_debug (D_SHOWER, "calling upinit")
        call upinit ()
      else
         if (shower%settings%isr_active) then
            call pygive ("MSTP(61)=1")
         else
            call pygive ("MSTP(61)=0")  !!! switch off ISR
         end if
         if (shower%settings%fsr_active) then
            call pygive ("MSTP(71)=1")
         else
            call pygive ("MSTP(71)=0")   !!! switch off FSR
         end if
         call pygive ("MSTP(11)=0")      !!! Disable Pythias QED-ISR per default
         call pygive ("MSTP(171)=1")     !!! Allow variable energies

         write (buffer, "(F10.5)") sqrt (abs (shower%settings%min_virtuality))
         call pygive ("PARJ(82)=" // buffer)
         write (buffer, "(F10.5)") shower%settings%isr_tscalefactor
         call pygive ("PARP(71)=" // buffer)
         write (buffer, "(F10.5)") shower%settings%fsr_lambda
         call pygive ("PARP(72)=" // buffer)
         write(buffer, "(F10.5)") shower%settings%isr_lambda
```

```
      call pygive ("PARP(61)=" // buffer)
      write (buffer, "(I10)") shower%settings%max_n_flavors
      call pygive ("MSTJ(45)=" // buffer)
      if (shower%settings%isr_alpha_s_running) then
         call pygive ("MSTP(64)=2")
      else
         call pygive ("MSTP(64)=0")
      end if
      if (shower%settings%fsr_alpha_s_running) then
         call pygive ("MSTJ(44)=2")
      else
         call pygive ("MSTJ(44)=0")
      end if
      write (buffer, "(F10.5)") shower%settings%fixed_alpha_s
      call pygive ("PARU(111)=" // buffer)
      write (buffer, "(F10.5)") shower%settings%isr_primordial_kt_width
      call pygive ("PARP(91)=" // buffer)
      write (buffer, "(F10.5)") shower%settings%isr_primordial_kt_cutoff
      call pygive ("PARP(93)=" // buffer)
      write (buffer, "(F10.5)") 1._double - shower%settings%isr_z_cutoff
      call pygive ("PARP(66)=" // buffer)
      write (buffer, "(F10.5)") shower%settings%isr_minenergy
      call pygive ("PARP(65)=" // buffer)
      if (shower%settings%isr_only_onshell_emitted_partons) then
         call pygive ("MSTP(63)=0")
      else
         call pygive ("MSTP(63)=2")
      end if
      if (shower%settings%mlm_matching) then
         call pygive ("MSTP(62)=2")
         call pygive ("MSTP(67)=0")
      end if
      call pythia6_set_config (shower%settings%pythia6_pygive)
      call msg_debug (D_SHOWER, "calling pyinit")
      call PYINIT ("USER", "", "", 0D0)
      call shower%rng%generate (rand)
      write (buffer, "(I10)") floor (rand*900000000)
      call pygive ("MRPY(1)=" // buffer)
      call pygive ("MRPY(2)=0")
      call pythia6_set_config (shower%settings%pythia6_pygive)
      shower%initialized_for_NPRUP = NPRUP
   end if
   if (shower%settings%tau_dec) then
      call pygive ("MSTJ(28)=2")
   end if
   if (pythia6_tauola_active() .and. .not. tauola_initialized) then
      call wo_tauola_init_call (shower%taudec_settings)
      tauola_initialized = .true.
   end if
end subroutine shower_pythia6_transfer_settings
```

⟨*Shower pythia6: shower pythia6: TBP*⟩+≡

```
procedure :: combine_with_particle_set => &
     shower_pythia6_combine_with_particle_set
```

⟨*Shower pythia6: procedures*⟩+≡

```
subroutine shower_pythia6_combine_with_particle_set &
      (shower, particle_set, model_in, model_hadrons)
  class(shower_pythia6_t), intent(in) :: shower
  type(particle_set_t), intent(inout) :: particle_set
  class(model_data_t), intent(in), target :: model_in
  class(model_data_t), intent(in), target :: model_hadrons
  call pythia6_combine_with_particle_set &
      (particle_set, model_in, model_hadrons, shower%settings)
end subroutine shower_pythia6_combine_with_particle_set
```

K(I,1)	pythia status code 1 = undecayed particle or unfragmented parton (single or last of parton system) 2 = unfragmented parton (followed by more partons in the same color singlet 3 = unfragmented parton (color info in K(I,4), K(I,5)) 11 = decayed particle or fragmented parton 12 = fragmented parton 13 = fragmented parton that has been removed 14 = branched parton with color info like 3 21 = documentation lines
K(I,2)	PDG code
K(I,3)	Parent where known else 0. Unphysical to assign particles partons as parents
K(I,4)	Normally first daughter
K(I,5)	Normally last daughter

The first two particles are always the beams, in Pythia and Whizard. We remove all beam remnants (including the ISR photons) since those are added back in by Pythia.

⟨*Shower pythia6: public*⟩+≡

```
public :: pythia6_combine_with_particle_set
```

⟨*Shower pythia6: procedures*⟩+≡

```
subroutine pythia6_combine_with_particle_set (particle_set, model_in, &
    model_hadrons, settings)
  type(particle_set_t), intent(inout) :: particle_set
  class(model_data_t), intent(in), target :: model_in
  class(model_data_t), intent(in), target :: model_hadrons
  type(shower_settings_t), intent(in) :: settings
  class(model_data_t), pointer :: model
  type(vector4_t) :: momentum
  type(particle_t), dimension(:), allocatable :: particles, beams
  integer :: dangling_col, dangling_anti_col, color, anti_color
  integer :: i, j, py_entries, next_color, n_tot_old, parent, real_parent
  integer :: pdg, status, child, hadro_start, i_py, i_whz
  integer, allocatable, dimension(:) :: py_index, whz_index
  logical, allocatable, dimension(:) :: valid
  real(default), parameter :: py_tiny = 1E-10_default
  integer :: N, NPAD, K
  real(double) :: P, V
  common /PYJETS/ N, NPAD, K(4000,5), P(4000,5), V(4000,5)
  save /PYJETS/
  integer, parameter :: KSUSY1 = 1000000, KSUSY2 = 2000000
```

```
  if (signal_is_pending ()) return
  if (debug_active (D_SHOWER)) then
     call msg_debug (D_SHOWER, 'Combine PYTHIA6 with particle set')
     call msg_debug (D_SHOWER, 'Particle set before replacing')
     call particle_set%write (summary=.true., compressed=.true.)
     call pylist (2)
     call msg_debug (D_SHOWER, string = "settings%hadron_collision", &
        value = settings%hadron_collision)
  end if
  if (settings%method == PS_PYTHIA6 .and. settings%hadron_collision) then
     call pythia6_set_last_treated_line(2)
     allocate (beams(2))
     beams = particle_set%prt(1:2)
     call particle_set%replace (beams)
     if (debug_active (D_SHOWER)) then
       call msg_debug (D_SHOWER, 'Resetting particle set to')
       call particle_set%write (summary=.true., compressed=.true.)
     end if
  end if
  call fill_hepevt_block_from_pythia ()
  call count_valid_entries_in_pythia_record ()
  call particle_set%without_hadronic_remnants &
       (particles, n_tot_old, py_entries)
  if (debug_active (D_SHOWER)) then
     print *, 'n_tot_old =    ', n_tot_old
     print *, 'py_entries =    ', py_entries
  end if
  call add_particles_of_pythia ()
  call particle_set%replace (particles)
  if (settings%hadron_collision) then
     call set_parent_child_relations_from_K ()
     call set_parent_child_relations_of_color_strings_to_hadrons ()
     !!! call particle_set%remove_duplicates (py_tiny * 100.0_default)
  else
     call set_parent_child_relations_from_hepevt ()
  end if
  !call fix_nonemitting_outgoings ()
  if (settings%method == PS_WHIZARD) then
     call fudge_whizard_partons_in_hadro ()
  end if
  where ((particle_set%prt%status == PRT_OUTGOING .or. &
          particle_set%prt%status == PRT_VIRTUAL .or. &
          particle_set%prt%status == PRT_BEAM_REMNANT) .and. &
          particle_set%prt%has_children ()) &
          particle_set%prt%status = PRT_RESONANT
  if (debug_active (D_SHOWER)) then
     print *, 'Particle set after replacing'
     call particle_set%write (summary=.true., compressed=.true.)
     print *, ' pythia6_set_last_treated_line will set to: ', N
  end if
  call pythia6_set_last_treated_line(N)

contains
```

⟨*Shower pythia6: combine with particle set: procedures*⟩

```
end subroutine pythia6_combine_with_particle_set
```

⟨*Shower pythia6: combine with particle set: procedures*⟩≡

```
subroutine count_valid_entries_in_pythia_record ()
⟨HEPEVT BLOCK⟩
  integer :: pset_idx
  logical :: comes_from_cmshower, emitted_zero_momentum_photon, &
       direct_decendent
  integer, parameter :: cmshower = 94
  hadro_start = 0
  allocate (valid(N))
  valid = .false.
  FIND: do i_py = 5, N
     !if (K(i_py,2) >= 91 .and. K(i_py,2) <= 94) then
     if (K(i_py,2) >= 91 .and. K(i_py,2) <= 93) then
        hadro_start = i_py
        exit FIND
     end if
  end do FIND
  do i_py = N, N_old+1, -1
     status = K(i_py,1)
     if (any (P(i_py,1:4) > 1E-8_default * P(1,4)) .and. &
          (status >= 1 .and. status <= 21)) then
        pset_idx = find_pythia_particle (i_py, more_fuzzy=.false.)
        direct_decendent = IDHEP(JMOHEP(1,i_py)) == cmshower .and. &
             JMOHEP(2,i_py) == 0
        emitted_zero_momentum_photon = find_pythia_particle &
             (JMOHEP(1,i_py), more_fuzzy=.false.) == pset_idx
        comes_from_cmshower = status == 1 .and. &
             (direct_decendent .or. emitted_zero_momentum_photon)
        valid(i_py) = pset_idx == 0 .or. comes_from_cmshower
     end if
  end do
  py_entries = count (valid)
  allocate (py_index (py_entries))
  allocate (whz_index (N))
  whz_index = 0
end subroutine count_valid_entries_in_pythia_record
```

⟨*Shower pythia6: combine with particle set: procedures*⟩+≡

```
subroutine add_particles_of_pythia ()
  integer :: whizard_status
  integer :: pset_idx, start_in_py
  integer :: ihelicity
  type(helicity_t) :: hel
  real(default) :: lifetime
  type(vector4_t) :: vertex
  dangling_col = 0
  dangling_anti_col = 0
  next_color = 500
```

```
    i_whz = 1
    if (settings%method == PS_PYTHIA6 .and. settings%hadron_collision) then
       start_in_py = 3
    else
       start_in_py = 7
    end if
    do i_py = start_in_py, N
       status = K(i_py,1)
       if (valid(i_py)) then
          call assign_colors (color, anti_color)
          momentum = real ([P(i_py,4), P(i_py,1:3)], kind=default)
          pdg = K(i_py,2)
          parent = K(i_py,3)
          call find_model (model, pdg, model_in, model_hadrons)
          if (i_py <= 4) then
             whizard_status = PRT_INCOMING
          else
             if (status <= 10) then
                whizard_status = PRT_OUTGOING
             else
                whizard_status = PRT_VIRTUAL
             end if
          end if
          call particles(n_tot_old+i_whz)%init &
               (whizard_status, pdg, model, color, anti_color, momentum)
          lifetime = V(i_py,5)
          vertex = [real (V(i_py,4), kind=default), &
                    real (V(i_py,1), kind=default), &
                    real (V(i_py,2), kind=default), &
                    real (V(i_py,3), kind=default)]
          if (.not. vanishes(lifetime)) &
               call particles(n_tot_old+i_whz)%set_lifetime (lifetime)
          if (any (.not. vanishes(real(V(i_py,1:4), kind = default)))) &
               call particles(n_tot_old+i_whz)%set_vertex (vertex)
          !!! Set tau helicity set by TAUOLA
          if (abs (pdg) == 15) then
            call wo_tauola_get_helicity (i_py, ihelicity)
            call hel%init (ihelicity)
            call particles(n_tot_old+i_whz)%set_helicity(hel)
            call particles(n_tot_old+i_whz)%set_polarization(PRT_DEFINITE_HELICITY)
          end if
          py_index(i_whz) = i_py
          whz_index(i_py) = n_tot_old + i_whz
          i_whz = i_whz + 1
       else
          pset_idx = find_pythia_particle (i_py, more_fuzzy=.true.)
          whz_index(i_py) = pset_idx
       end if
    end do
  end subroutine add_particles_of_pythia
```

⟨*Shower pythia6: combine with particle set: procedures*⟩+≡

```
  subroutine assign_colors (color, anti_color)
    integer, intent(out) :: color, anti_color
```

```
  if ((K(i_py,2) == 21) .or. (abs (K(i_py,2)) <= 8) .or. &
       (abs (K(i_py,2)) >= KSUSY1+1 .and. abs (K(i_py,2)) <= KSUSY1+8) .or. &
       (abs (K(i_py,2)) >= KSUSY2+1 .and. abs (K(i_py,2)) <= KSUSY2+8) .or. &
       (abs (K(i_py,2)) >= 1000 .and. abs (K(i_py,2)) <= 9999) .and. &
       hadro_start == 0) then
     if (dangling_col == 0 .and. dangling_anti_col == 0) then
        ! new color string
        ! Gluon and gluino only color octets implemented so far
        if (K(i_py,2) == 21 .or. K(i_py,2) == 1000021) then
           color = next_color
           dangling_col = color
           next_color = next_color + 1
           anti_color = next_color
           dangling_anti_col = anti_color
           next_color = next_color + 1
        else if (K(i_py,2) > 0) then  ! particles have color
           color = next_color
           dangling_col = color
           anti_color = 0
           next_color = next_color + 1
        else if (K(i_py,2) < 0) then  ! antiparticles have anticolor
           anti_color = next_color
           dangling_anti_col = anti_color
           color = 0
           next_color = next_color + 1
        end if
     else if(status == 1) then
        ! end of string
        color = dangling_anti_col
        anti_color = dangling_col
        dangling_col = 0
        dangling_anti_col = 0
     else
        ! inside the string
        if(dangling_col /= 0) then
           anti_color = dangling_col
           color = next_color
           dangling_col = next_color
           next_color = next_color +1
        else if(dangling_anti_col /= 0) then
           color = dangling_anti_col
           anti_color = next_color
           dangling_anti_col = next_color
           next_color = next_color +1
        else
           call msg_bug ("Couldn't assign colors")
        end if
     end if
  else
     color = 0
     anti_color = 0
  end if
end subroutine assign_colors
```

⟨*Shower pythia6: combine with particle set: procedures*⟩+≡

```
subroutine fill_hepevt_block_from_pythia ()
  integer :: first_daughter, second_mother_of_first_daughter, i_hep
  logical :: inconsistent_mother, more_than_one_points_to_first_daugther
⟨HEPEVT BLOCK⟩
  call pyhepc(1)
  do i_hep = 1, NHEP
     first_daughter = JDAHEP(1,i_hep)
     if (first_daughter > 0) then
        more_than_one_points_to_first_daugther = &
             count (JDAHEP(1,i_hep:NHEP) == first_daughter) > 1
        if (more_than_one_points_to_first_daugther) then
           second_mother_of_first_daughter = JMOHEP(2,first_daughter)
           ! Only entries with codes 91-94 should have a second mother
           if (second_mother_of_first_daughter == 0) then
              inconsistent_mother = JMOHEP(1,first_daughter) /= i_hep
              if (inconsistent_mother) then
                 JMOHEP(1,first_daughter) = i_hep
                 do j = i_hep + 1, NHEP
                   if (JDAHEP(1,j) == first_daughter) then
                      JMOHEP(2,first_daughter) = j
                   end if
                 end do
              end if
           end if
        end if
     end if
  end do
end subroutine fill_hepevt_block_from_pythia
```

⟨*HEPEVT BLOCK*⟩≡

```
integer, parameter :: NMXHEP = 4000
integer :: NEVHEP
integer :: NHEP
integer, dimension(NMXHEP) :: ISTHEP
integer, dimension(NMXHEP) :: IDHEP
integer, dimension(2, NMXHEP) :: JMOHEP
integer, dimension(2, NMXHEP) :: JDAHEP
double precision, dimension(5, NMXHEP) :: PHEP
double precision, dimension(4, NMXHEP) :: VHEP
common /HEPEVT/ &
 NEVHEP, NHEP, ISTHEP, IDHEP, &
 JMOHEP, JDAHEP, PHEP, VHEP
save /HEPEVT/
```

Use HEPEVT for parent-child informations

⟨*Shower pythia6: combine with particle set: procedures*⟩+≡

```
subroutine set_parent_child_relations_from_hepevt ()
  integer, allocatable, dimension(:) :: parents
⟨HEPEVT BLOCK⟩
  integer:: parent2, parent1, npar
  call msg_debug (D_SHOWER, &
       "set_parent_child_relations_from_hepevt")
  if (debug_active (D_SHOWER)) then
```

```
      print *, 'NHEP, n, py_entries:' , NHEP, n, py_entries
    end if
    do i_whz = 1, py_entries
      parent1 = JMOHEP(1,py_index(i_whz))
      parent2 = parent1
      if (JMOHEP(2,py_index(i_whz)) > 0) then
        parent2 = JMOHEP(2,py_index(i_whz))
      end if
      allocate (parents(parent2-parent1+1))
      parents = 0
      child = n_tot_old + i_whz
      npar = 0
      do parent = parent1, parent2
        if (parent > 0) then
          if (parent <= 2) then
            call particle_set%parent_add_child (parent, child)
          else
            if (whz_index(parent) > 0) then
              npar = npar + 1
              parents(npar) = whz_index(parent)
              call particle_set%prt(whz_index(parent))%add_child (child)
            end if
          end if
        end if
      end do
      parents = pack (parents, parents > 0)
      if (npar > 0) call particle_set%prt(child)%set_parents (parents)
      if (allocated (parents))  deallocate (parents)
    end do
    NHEP = 0
  end subroutine set_parent_child_relations_from_hepevt
```

⟨*Shower pythia6: combine with particle set: procedures*⟩+≡

```
  subroutine fix_nonemitting_outgoings ()
    integer, dimension(1) :: child
    integer, parameter :: cmshower = 94
    do i = 1, size (particle_set%prt)
      associate (p => particle_set%prt(i))
        if (p%get_n_children () == 1) then
          child = p%get_children ()
          if (particle_set%prt(child(1))%get_pdg () == cmshower) then
            j = particle_set%reverse_find_particle (p%get_pdg (), p%p)
            if (j == i) then
              deallocate (p%child)
              p%status = PRT_OUTGOING
            end if
          end if
        end if
      end associate
    end do
  end subroutine fix_nonemitting_outgoings
```

⟨*Shower pythia6: combine with particle set: procedures*⟩+≡

```
subroutine set_parent_child_relations_from_K ()
  do j = 1, py_entries
     parent = K(py_index(j),3)
     child = n_tot_old + j
     if (parent > 0) then
        if (parent >= 1 .and. parent <= 2) then
           call particle_set%parent_add_child (parent, child)
        else
           real_parent = whz_index (parent)
           if (real_parent > 0 .and. real_parent /= child) then
              call particle_set%parent_add_child (real_parent, child)
           end if
        end if
     end if
  end do
end subroutine set_parent_child_relations_from_K
```

⟨*Shower pythia6: combine with particle set: procedures*⟩+≡

```
subroutine set_parent_child_relations_of_color_strings_to_hadrons ()
  integer :: begin_string, end_string, old_start, next_start, real_child
  integer, allocatable, dimension(:) :: parents
  call msg_debug (D_SHOWER, "set_parent_child_relations_of_color_strings_to_hadrons")
  call msg_debug (D_SHOWER, "hadro_start", hadro_start)
  if (hadro_start > 0) then
     old_start = hadro_start
     do
        next_start = 0
        FIND: do i = old_start + 1, N
           if (K(i,2) >= 91 .and. K(i,2) <= 94) then
              next_start = i
              exit FIND
           end if
        end do FIND
        begin_string = K(old_start,3)
        end_string = N
        do i = begin_string, N
           if (K(i,1) == 11) then
              end_string = i
              exit
           end if
        end do
        allocate (parents (end_string - begin_string + 1))
        parents = 0
        real_child = whz_index (old_start)
        do i = begin_string, end_string
           real_parent = whz_index (i)
           if (real_parent > 0) then
              call particle_set%prt(real_parent)%add_child (real_child)
              parents (i - begin_string + 1) = real_parent
           end if
        end do
        call particle_set%prt(real_child)%set_parents (parents)
        deallocate (parents)
        if (next_start == 0) exit
```

```
          old_start = next_start
       end do
    end if
  end subroutine set_parent_child_relations_of_color_strings_to_hadrons
```

We allow to be more_fuzzy when finding particles for parent child relations than when deciding whether we add particles or not.

⟨*Shower pythia6: combine with particle set: procedures*⟩+≡

```
  function find_pythia_particle (i_py, more_fuzzy) result (j)
    integer :: j
    integer, intent(in) :: i_py
    logical, intent(in) :: more_fuzzy
    real(default) :: rel_small
    pdg = K(i_py,2)
    momentum = real([P(i_py,4), P(i_py,1:3)], kind=default)
    if (more_fuzzy) then
       rel_small = 1E-6_default
    else
       rel_small = 1E-10_default
    end if
    j = particle_set%reverse_find_particle (pdg, momentum, &
         abs_smallness = py_tiny, &
         rel_smallness = rel_small)
  end function find_pythia_particle
```

Outgoing partons after hadronization shouldn't happen and is a dirty fix to missing mother daughter relation. I suspect that it has to do with the ordering of the color string but am not sure.

⟨*Shower pythia6: combine with particle set: procedures*⟩+≡

```
  subroutine fudge_whizard_partons_in_hadro ()
    do i = 1, size (particle_set%prt)
       if (particle_set%prt(i)%status == PRT_OUTGOING .and. &
           (particle_set%prt(i)%flv%get_pdg () == GLUON .or. &
            particle_set%prt(i)%flv%get_pdg_abs () < 6)  .or. &
           particle_set%prt(i)%status == PRT_BEAM_REMNANT) then
          particle_set%prt(i)%status = PRT_VIRTUAL
       end if
    end do
  end subroutine fudge_whizard_partons_in_hadro
```

⟨*Shower pythia6: shower pythia6: TBP*⟩+≡

```
  procedure :: get_final_colored_ME_momenta => shower_pythia6_get_final_colored_ME_momenta
```

⟨*Shower pythia6: procedures*⟩+≡

```
  subroutine shower_pythia6_get_final_colored_ME_momenta &
         (shower, momenta)
    class(shower_pythia6_t), intent(in) :: shower
    type(vector4_t), dimension(:), allocatable, intent(out) :: momenta
  ⟨PYJETS COMMON BLOCK⟩
    integer :: i, j, n_jets
    if (signal_is_pending ()) return

    i = 7 !!! final ME partons start in 7th row of event record
```

```
    n_jets = 0
    do
       if (K(I,1) /= 21) exit
       if ((K(I,2) == 21) .or. (abs(K(I,2)) <= 6)) then
          n_jets = n_jets + 1
       end if
       i = i + 1
    end do
    if (n_jets == 0) return
    allocate (momenta(1:n_jets))
    i = 7
    j = 1
    do
       if (K(I,1) /= 21) exit
       if ((K(I,2) == 21) .or. (abs(K(I,2)) <= 6)) then
          momenta(j) = real ([P(i,4), P(i,1:3)], kind=default)
          j = j + 1
       end if
       i = i + 1
    end do
  end subroutine shower_pythia6_get_final_colored_ME_momenta
```

⟨*Shower pythia6: public*⟩+≡

```
  public :: pythia6_setup_lhe_io_units
```

⟨*Shower pythia6: procedures*⟩+≡

```
  subroutine pythia6_setup_lhe_io_units (u_W2P, u_P2W)
    integer, intent(out) :: u_W2P
    integer, intent(out), optional :: u_P2W
    character(len=10) :: buffer
    u_W2P = free_unit ()
    if (debug_active (D_SHOWER)) then
       open (unit=u_W2P, status="replace", file="whizardout.lhe", &
            action="readwrite")
    else
       open (unit=u_W2P, status="scratch", action="readwrite")
    end if
    write (buffer, "(I10)")  u_W2P
    call pygive ("MSTP(161)=" // buffer)  !!! Unit for PYUPIN (LHA)
    call pygive ("MSTP(162)=" // buffer)  !!! Unit for PYUPEV (LHA)
    if (present (u_P2W)) then
       u_P2W = free_unit ()
       write (buffer, "(I10)")  u_P2W
       call pygive ("MSTP(163)=" // buffer)
       if (debug_active (D_SHOWER)) then
          open (unit=u_P2W, file="pythiaout2.lhe", status="replace", &
               action="readwrite")
       else
          open (unit=u_P2W, status="scratch", action="readwrite")
       end if
    end if
  end subroutine pythia6_setup_lhe_io_units
```

⟨*Shower pythia6: public*⟩+≡

```
public :: pythia6_set_config
```

⟨*Shower pythia6: procedures*⟩+≡

```
subroutine pythia6_set_config (pygive_all)
  type(string_t), intent(in) :: pygive_all
  type(string_t) :: pygive_remaining, pygive_partial
  if (len (pygive_all) > 0) then
     pygive_remaining = pygive_all
     do while (len (pygive_remaining) > 0)
        call split (pygive_remaining, pygive_partial, ";")
        call pygive (char (pygive_partial))
     end do
     if (pythia6_get_error() /= 0) then
        call msg_fatal &
             (" PYTHIA6 did not recognize ps_PYTHIA_PYGIVE setting.")
     end if
  end if
end subroutine pythia6_set_config
```

Exchanging error messages with PYTHIA6.

⟨*Shower pythia6: public*⟩+≡

```
public :: pythia6_set_error
```

⟨*Shower pythia6: procedures*⟩+≡

```
subroutine pythia6_set_error (mstu23)
  IMPLICIT DOUBLE PRECISION(A-H, O-Z)
  IMPLICIT INTEGER(I-N)
  COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
  SAVE/PYDAT1/
  integer, intent(in) :: mstu23
  MSTU(23) = mstu23
end subroutine pythia6_set_error
```

⟨*Shower pythia6: public*⟩+≡

```
public :: pythia6_get_error
```

⟨*Shower pythia6: procedures*⟩+≡

```
function pythia6_get_error () result (mstu23)
  IMPLICIT DOUBLE PRECISION(A-H, O-Z)
  IMPLICIT INTEGER(I-N)
  COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
  SAVE/PYDAT1/
  integer :: mstu23
  mstu23 = MSTU(23)
end function pythia6_get_error
```

⟨*Shower pythia6: public*⟩+≡

```
public :: pythia6_tauola_active
```

⟨*Shower pythia6: procedures*⟩+≡

```
function pythia6_tauola_active () result (active)
  IMPLICIT DOUBLE PRECISION(A-H, O-Z)
  IMPLICIT INTEGER(I-N)
  COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
```

```
      SAVE/PYDAT1/
      logical :: active
      active = MSTJ(28) == 2
    end function pythia6_tauola_active
```

⟨*Shower pythia6: public*⟩+≡

```
    public :: pythia6_handle_errors
```

⟨*Shower pythia6: procedures*⟩+≡

```
    function pythia6_handle_errors () result (valid)
      logical :: valid
      valid = pythia6_get_error () == 0
      if (.not. valid) then
         call pythia6_set_error (0)
      end if
    end function pythia6_handle_errors
```

⟨*Shower pythia6: public*⟩+≡

```
    public :: pythia6_set_verbose
```

⟨*Shower pythia6: procedures*⟩+≡

```
    subroutine pythia6_set_verbose (verbose)
      logical, intent(in) :: verbose
      if (verbose) then
         call pygive ('MSTU(13)=1')
      else
         call pygive ('MSTU(12)=12345') !!! No title page is written
         call pygive ('MSTU(13)=0')     !!! No information is written
      end if
    end subroutine pythia6_set_verbose
```

⟨*Shower pythia6: public*⟩+≡

```
    public :: pythia6_set_last_treated_line
```

⟨*Shower pythia6: procedures*⟩+≡

```
    subroutine pythia6_set_last_treated_line (last_line)
      integer,intent(in) :: last_line
      N_old = last_line
    end subroutine pythia6_set_last_treated_line
```

⟨pythia6_up.f⟩≡

```
  C...UPINIT
  C...Is supposed to fill the HEPRUP commonblock with info
  C...on incoming beams and allowed processes.

        SUBROUTINE UPINIT

  C...Double precision and integer declarations.
        IMPLICIT DOUBLE PRECISION(A-H, O-Z)
        IMPLICIT INTEGER(I-N)

  C...PYTHIA commonblock: only used to provide read unit MSTP(161).
        COMMON/PYPARS/MSTP(200),PARP(200),MSTI(200),PARI(200)
```

```
      SAVE /PYPARS/

C...User process initialization commonblock.
      INTEGER MAXPUP
      PARAMETER (MAXPUP=100)
      INTEGER IDBMUP,PDFGUP,PDFSUP,IDWTUP,NPRUP,LPRUP
      DOUBLE PRECISION EBMUP,XSECUP,XERRUP,XMAXUP
      COMMON/HEPRUP/IDBMUP(2),EBMUP(2),PDFGUP(2),PDFSUP(2),
     &IDWTUP,NPRUP,XSECUP(MAXPUP),XERRUP(MAXPUP),XMAXUP(MAXPUP),
     &LPRUP(MAXPUP)
      SAVE /HEPRUP/

C...Lines to read in assumed never longer than 200 characters.
      PARAMETER (MAXLEN=200)
      CHARACTER*(MAXLEN) STRING

C...Format for reading lines.
      CHARACTER(len=6) STRFMT
      STRFMT='(A000)'
      WRITE(STRFMT(3:5),'(I3)') MAXLEN

C...Loop until finds line beginning with "<init>" or "<init ".
  100 READ(MSTP(161),STRFMT,END=130,ERR=130) STRING
      IBEG=0
  110 IBEG=IBEG+1
C...Allow indentation.
      IF(STRING(IBEG:IBEG).EQ.' '.AND.IBEG.LT.MAXLEN-5) GOTO 110
      IF(STRING(IBEG:IBEG+5).NE.'<init>'.AND.
     &STRING(IBEG:IBEG+5).NE.'<init ') GOTO 100

C...Read first line of initialization info.
      READ(MSTP(161),*,END=130,ERR=130) IDBMUP(1),IDBMUP(2),EBMUP(1),
     &EBMUP(2),PDFGUP(1),PDFGUP(2),PDFSUP(1),PDFSUP(2),IDWTUP,NPRUP

C...Read NPRUP subsequent lines with information on each process.
      DO 120 IPR=1,NPRUP
        READ(MSTP(161),*,END=130,ERR=130) XSECUP(IPR),XERRUP(IPR),
     &  XMAXUP(IPR),LPRUP(IPR)
  120 CONTINUE
      RETURN

C...Error exit: give up if initalization does not work.
  130 WRITE(*,*) ' Failed to read LHEF initialization information.'
      WRITE(*,*) ' Event generation will be stopped.'
      CALL PYSTOP(12)

      RETURN
      END
```

⟨pythia6_up.f⟩+≡

```
C...UPEVNT
C...Dummy routine, to be replaced by a user implementing external
C...processes. Depending on cross section model chosen, it either has
C...to generate a process of the type IDPRUP requested, or pick a type
```

```
C...itself and generate this event. The event is to be stored in the
C...HEPEUP commonblock, including (often) an event weight.

C...New example: handles a standard Les Houches Events File.

      SUBROUTINE UPEVNT

C...Double precision and integer declarations.
      IMPLICIT DOUBLE PRECISION(A-H, O-Z)
      IMPLICIT INTEGER(I-N)

C...PYTHIA commonblock: only used to provide read unit MSTP(162).
      COMMON/PYPARS/MSTP(200),PARP(200),MSTI(200),PARI(200)
      SAVE /PYPARS/

C...Added by WHIZARD
      COMMON/PYDAT1/MSTU(200),PARU(200),MSTJ(200),PARJ(200)
      SAVE/PYDAT1/

C...User process event common block.
      INTEGER MAXNUP
      PARAMETER (MAXNUP=500)
      INTEGER NUP,IDPRUP,IDUP,ISTUP,MOTHUP,ICOLUP
      DOUBLE PRECISION XWGTUP,SCALUP,AQEDUP,AQCDUP,PUP,VTIMUP,SPINUP
      COMMON/HEPEUP/NUP,IDPRUP,XWGTUP,SCALUP,AQEDUP,AQCDUP,IDUP(MAXNUP),
     &ISTUP(MAXNUP),MOTHUP(2,MAXNUP),ICOLUP(2,MAXNUP),PUP(5,MAXNUP),
     &VTIMUP(MAXNUP),SPINUP(MAXNUP)
      SAVE /HEPEUP/

C...Lines to read in assumed never longer than 200 characters.
      PARAMETER (MAXLEN=200)
      CHARACTER*(MAXLEN) STRING

C...Format for reading lines.
      CHARACTER(len=6) STRFMT
      STRFMT='(A000)'
      WRITE(STRFMT(3:5),'(I3)') MAXLEN

C...Loop until finds line beginning with "<event>" or "<event ".
  100 READ(MSTP(162),STRFMT,END=130,ERR=130) STRING
      IBEG=0
  110 IBEG=IBEG+1
C...Allow indentation.
      IF(STRING(IBEG:IBEG).EQ.' '.AND.IBEG.LT.MAXLEN-6) GOTO 110
      IF(STRING(IBEG:IBEG+6).NE.'<event>'.AND.
     &STRING(IBEG:IBEG+6).NE.'<event ') GOTO 100

C...Read first line of event info.
      READ(MSTP(162),*,END=130,ERR=130) NUP,IDPRUP,XWGTUP,SCALUP,
     &AQEDUP,AQCDUP

C...Read NUP subsequent lines with information on each particle.
      DO 120 I=1,NUP
        READ(MSTP(162),*,END=130,ERR=130) IDUP(I),ISTUP(I),
```

```
     &  MOTHUP(1,I),MOTHUP(2,I),ICOLUP(1,I),ICOLUP(2,I),
     &  (PUP(J,I),J=1,5),VTIMUP(I),SPINUP(I)
  120 CONTINUE
      RETURN

C...Error exit, typically when no more events.
  130 CONTINUE
C      WRITE(*,*) ' Failed to read LHEF event information.'
C      WRITE(*,*) ' Will assume end of file has been reached.'
      NUP=0
      MSTI(51)=1
C...Added by WHIZARD, mark these failed events
      MSTU(23)=1

      RETURN
      END
```

⟨pythia6_up.f⟩+≡

```
C...UPVETO
C...Dummy routine, to be replaced by user, to veto event generation
C...on the parton level, after parton showers but before multiple
C...interactions, beam remnants and hadronization is added.
C...If resonances like W, Z, top, Higgs and SUSY particles are handed
C...undecayed from UPEVNT, or are generated by PYTHIA, they will also
C...be undecayed at this stage; if decayed their decay products will
C...have been allowed to shower.

C...All partons at the end of the shower phase are stored in the
C...HEPEVT commonblock. The interesting information is
C...NHEP = the number of such partons, in entries 1 <= i <= NHEP,
C...IDHEP(I) = the particle ID code according to PDG conventions,
C...PHEP(J,I) = the (p_x, p_y, p_z, E, m) of the particle.
C...All ISTHEP entries are 1, while the rest is zeroed.

C...The user decision is to be conveyed by the IVETO value.
C...IVETO = 0 : retain current event and generate in full;
C...      = 1 : abort generation of current event and move to next.

      SUBROUTINE UPVETO(IVETO)

C...HEPEVT commonblock.
      PARAMETER (NMXHEP=4000)
      COMMON/HEPEVT/NEVHEP,NHEP,ISTHEP(NMXHEP),IDHEP(NMXHEP),
     &JMOHEP(2,NMXHEP),JDAHEP(2,NMXHEP),PHEP(5,NMXHEP),VHEP(4,NMXHEP)
      DOUBLE PRECISION PHEP,VHEP
      SAVE /HEPEVT/

C...Next few lines allow you to see what info PYVETO extracted from
C...the full event record for the first two events.
C...Delete if you don't want it.
      DATA NLIST/0/
      SAVE NLIST
      IF(NLIST.LE.2) THEN
        WRITE(*,*) ' Full event record at time of UPVETO call:'
```

```
          CALL PYLIST(1)
          WRITE(*,*) ' Part of event record made available to UPVETO:'
          CALL PYLIST(5)
          NLIST=NLIST+1
        ENDIF

  C...Make decision here.
        IVETO = 0

        RETURN
        END
```

⟨ktclus.f90⟩≡

```
  ⟨File header⟩

  module ktclus

  ⟨Use kinds⟩

  ⟨KTCLUS: public⟩

  contains

  ⟨KTCLUS: procedures⟩

  end module ktclus
```

⟨*KTCLUS: procedures*⟩≡

```
  !C-----------------------------------------------------------------------
  !C-----------------------------------------------------------------------
  !C-----------------------------------------------------------------------
  !C     KTCLUS: written by Mike Seymour, July 1992.
  !C     Last modified November 2000.
  !C     Please send comments or suggestions to Mike.Seymour@rl.ac.uk
  !C
  !C     This is a general-purpose kt clustering package.
  !C     It can handle ee, ep and pp collisions.
  !C     It is loosely based on the program of Siggi Bethke.
  !C
  !C     The time taken (on a 10MIP machine) is (0.2microsec)*N**3
  !C     where N is the number of particles.
  !C     Over 90 percent of this time is used in subroutine KTPMIN, which
  !C     simply finds the minimum member of a one-dimensional array.
  !C     It is well worth thinking about optimization: on the SPARCstation
  !C     a factor of two increase was obtained simply by increasing the
  !C     optimization level from its default value.
  !C
  !C     The approach is to separate the different stages of analysis.
  !C     KTCLUS does all the clustering and records a merging history.
  !C     It returns a simple list of the y values at which each merging
  !C     occured. Then the following routines can be called to give extra
  !C     information on the most recently analysed event.
  !C     KTCLUR is identical but includes an R parameter, see below.
  !C     KTYCUT gives the number of jets at each given YCUT value.
```

```
!C     KTYSUB gives the number of sub-jets at each given YCUT value.
!C     KTBEAM gives same info as KTCLUS but only for merges with the beam
!C     KTJOIN gives same info as KTCLUS but for merges of sub-jets.
!C     KTRECO reconstructs the jet momenta at a given value of YCUT.
!C     It also gives information on which jets at scale YCUT belong to
!C     which macro-jets at scale YMAC, for studying sub-jet properties.
!C     KTINCL reconstructs the jet momenta according to the inclusive jet
!C     definition of Ellis and Soper.
!C     KTISUB, KTIJOI and KTIREC are like KTYSUB, KTJOIN and KTRECO,
!C     except that they only apply to one inclusive jet at a time,
!C     with the pt of that jet automatically used for ECUT.
!C     KTWICH gives a list of which particles ended up in which jets.
!C     KTWCHS gives the same thing, but only for subjets.
!C     Note that the numbering of jets used by these two routines is
!C     guaranteed to be the same as that used by KTRECO.
!C
!C     The collision type and analysis type are indicated by the first
!C     argument of KTCLUS. IMODE=<TYPE><ANGLE><MONO><RECOM> where
!C     TYPE:  1=>ee, 2=>ep with p in -z direction, 3=>pe, 4=>pp
!C     ANGLE: 1=>angular kt def., 2=>DeltaR, 3=>f(DeltaEta,DeltaPhi)
!C            where f()=2(cosh(eta)-cos(phi)) is the QCD emission metric
!C     MONO:  1=>derive relative pseudoparticle angles from jets
!C            2=>monotonic definitions of relative angles
!C     RECOM: 1=>E recombination scheme, 2=>pt scheme, 3=>pt**2 scheme
!C
!C     There are also abbreviated forms for the most common combinations:
!C     IMODE=1 => E scheme in e+e-                              (=1111)
!C           2 => E scheme in ep                                (=2111)
!C           3 => E scheme in pe                                (=3111)
!C           4 => E scheme in pp                                (=4111)
!C           5 => covariant E scheme in pp                      (=4211)
!C           6 => covariant pt-scheme in pp                     (=4212)
!C           7 => covariant monotonic pt**2-scheme in pp        (=4223)
!C
!C     KTRECO no longer needs to reconstruct the momenta according to the
!C     same recombination scheme in which they were clustered. Its first
!C     argument gives the scheme, taking the same values as RECOM above.
!C
!C     Note that unlike previous versions, all variables which hold y
!C     values have been named in a consistent way:
!C     Y()  is the output scale at which jets were merged,
!C     YCUT is the input scale at which jets should be counted, and
!C          jet-momenta reconstructed etc,
!C     YMAC is the input macro-jet scale, used in determining whether
!C          or not each jet is a sub-jet.
!C     The original scheme defined in our papers is equivalent to always
!C     setting YMAC=1.
!C     Whenever a YCUT or YMAC variable is used, it is rounded down
!C     infinitesimally, so that for example, setting YCUT=Y(2) refers
!C     to the scale where the event is 2-jet, even if rounding errors
!C     have shifted its value slightly.
!C
!C     An R parameter can be used in hadron-hadron collisions by
!C     calling KTCLUR instead of KTCLUS.  This is as suggested by
```

```
!C     Ellis and Soper, but implemented slightly differently,
!C     as in M.H. Seymour, LU TP 94/2 (submitted to Nucl. Phys. B.).
!C     R**2 multiplies the single Kt everywhere it is used.
!C     Calling KTCLUR with R=1 is identical to calling KTCLUS.
!C     R plays a similar role to the jet radius in a cone-type algorithm,
!C     but is scaled up by about 40% (ie R=0.7 in a cone algorithm is
!C     similar to this algorithm with R=1).
!C     Note that R.EQ.1 must be used for the e+e- and ep versions,
!C     and is strongly recommended for the hadron-hadron version.
!C     However, R values smaller than 1 have been found to be useful for
!C     certain applications, particularly the mass reconstruction of
!C     highly-boosted colour-singlets such as high-pt hadronic Ws,
!C     as in M.H. Seymour, LU TP 93/8 (to appear in Z. Phys. C.).
!C     Situations in which R<1 is useful are likely to also be those in
!C     which the inclusive reconstruction method is more useful.
!C
!C     Also included is a set of routines for doing Lorentz boosts:
!C     KTLBST finds the boost matrix to/from the cm frame of a 4-vector
!C     KTRROT finds the rotation matrix from one vector to another
!C     KTMMUL multiplies together two matrices
!C     KTVMUL multiplies a vector by a matrix
!C     KTINVT inverts a transformation matrix (nb NOT a general 4 by 4)
!C     KTFRAM boosts a list of vectors between two arbitrary frames
!C     KTBREI boosts a list of vectors between the lab and Breit frames
!C     KTHADR boosts a list of vectors between the lab and hadronic cmf
!C       The last two need the momenta in the +z direction of the lepton
!C       and hadron beams, and the 4-momentum of the outgoing lepton.
!C
!C     The main reference is:
!C       S. Catani, Yu.L. Dokshitzer, M.H. Seymour and B.R. Webber,
!C         Nucl.Phys.B406(1993)187.
!C     The ep version was proposed in:
!C       S. Catani, Yu.L. Dokshitzer and B.R. Webber,
!C         Phys.Lett.285B(1992)291.
!C     The inclusive reconstruction method was proposed in:
!C       S.D. Ellis and D.E. Soper,
!C         Phys.Rev.D48(1993)3160.
!C
!C-----------------------------------------------------------------------
!C-----------------------------------------------------------------------
!C-----------------------------------------------------------------------
```

⟨*KTCLUS: public*⟩≡

```
  public :: ktclur
```

⟨*KTCLUS: procedures*⟩+≡

```
  SUBROUTINE KTCLUR(IMODE,PP,NN,R,ECUT,Y,*)
    use io_units
    IMPLICIT NONE
!C---DO CLUSTER ANALYSIS OF PARTICLES IN PP
!C
!C   IMODE   = INPUT  : DESCRIBED ABOVE
!C   PP(I,J) = INPUT  : 4-MOMENTUM OF Jth PARTICLE: I=1,4 => PX,PY,PZ,E
!C   NN      = INPUT  : NUMBER OF PARTICLES
!C   R       = INPUT  : ELLIS AND SOPER'S R PARAMETER, SEE ABOVE.
```

```
!C   ECUT    = INPUT  : DENOMINATOR OF KT MEASURE. IF ZERO, ETOT IS USED
!C   Y(J)    = OUTPUT : VALUE OF Y FOR WHICH EVENT CHANGES FROM BEING
!C                       J JET TO J-1 JET
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C   COULD NOT BE PROCESSED (MOST LIKELY DUE TO TOO MANY PARTICLES)
!C
!C   NOTE THAT THE MOMENTA ARE DECLARED DOUBLE PRECISION,
!C   AND ALL OTHER FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
    INTEGER NMAX,IM,IMODE,TYPE,ANGL,MONO,RECO,N,I,J,NN, &
         IMIN,JMIN,KMIN,NUM,HIST,INJET,IABBR,NABBR
    PARAMETER (NMAX=512,NABBR=7)
    DOUBLE PRECISION PP(4,*)
    integer :: u
!CHANGE    DOUBLE PRECISION R,ECUT,Y(*),P,KT,ETOT,RSQ,KTP,KTS,KTPAIR,KTSING, &
    DOUBLE PRECISION R,ECUT,Y(*),P,KT,ETOT,RSQ,KTP,KTS, &
         KTMIN,ETSQ,KTLAST,KTMAX,KTTMP
    LOGICAL FIRST
    CHARACTER TITLE(4,4)*10
!C---KT RECORDS THE KT**2 OF EACH MERGING.
!C---KTLAST RECORDS FOR EACH MERGING, THE HIGHEST ECUT**2 FOR WHICH THE
!C   RESULT IS NOT MERGED WITH THE BEAM (COULD BE LARGER THAN THE
!C   KT**2 AT WHICH IT WAS MERGED IF THE KT VALUES ARE NOT MONOTONIC).
!C   THIS MAY SOUND POINTLESS, BUT ITS USEFUL FOR DETERMINING WHETHER
!C   SUB-JETS SURVIVED TO SCALE Y=YMAC OR NOT.
!C---HIST RECORDS MERGING HISTORY:
!C   N=>DELETED TRACK N, M*NMAX+N=>MERGED TRACKS M AND N (M<N).
    COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
         KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM
    DIMENSION INJET(NMAX),IABBR(NABBR)
    DATA FIRST,TITLE,IABBR/.TRUE., &
         'e+e-      ','ep        ','pe        ','pp        ', &
         'angle     ','DeltaR    ','f(DeltaR) ','**********', &
         'no        ','yes       ','**********','**********', &
         'E         ','Pt        ','Pt**2     ','**********', &
         1111,2111,3111,4111,4211,4212,4223/
!C---CHECK INPUT
    IM=IMODE
    IF (IM.GE.1.AND.IM.LE.NABBR) IM=IABBR(IM)
    TYPE=MOD(IM/1000,10)
    ANGL=MOD(IM/100 ,10)
    MONO=MOD(IM/10  ,10)
    RECO=MOD(IM     ,10)
    IF (NN.GT.NMAX) CALL KTWARN('KT-MAX',100,*999)
    IF (NN.LT.1) CALL KTWARN('KT-LT1',100,*999)
    IF (NN.LT.2.AND.TYPE.EQ.1) CALL KTWARN('KT-LT2',100,*999)
    IF (TYPE.LT.1.OR.TYPE.GT.4.OR.ANGL.LT.1.OR.ANGL.GT.4.OR. &
         MONO.LT.1.OR.MONO.GT.2.OR.RECO.LT.1.OR.RECO.GT.3) CALL KTWARN('KTCLUS',101,*999)
    u = given_output_unit ()
    IF (FIRST) THEN
       WRITE (u,'(/,1X,54(''*'')/A)') &
            ' KTCLUS: written by Mike Seymour, July 1992.'
       WRITE (u,'(A)') &
            ' Last modified November 2000.'
```

```
      WRITE (u,'(A)') &
           ' Please send comments or suggestions to Mike.Seymour@rl.ac.uk'
      WRITE (u,'(/A,I2,2A)') &
           '       Collision type =',TYPE,' = ',TITLE(TYPE,1)
      WRITE (u,'(A,I2,2A)') &
           '     Angular variable =',ANGL,' = ',TITLE(ANGL,2)
      WRITE (u,'(A,I2,2A)') &
           ' Monotonic definition =',MONO,' = ',TITLE(MONO,3)
      WRITE (u,'(A,I2,2A)') &
           ' Recombination scheme =',RECO,' = ',TITLE(RECO,4)
      IF (R.NE.1) THEN
         WRITE (u,'(A,F5.2)') &
              '     Radius parameter =',R
         IF (TYPE.NE.4) WRITE (u,'(A)') &
              ' R.NE.1 is strongly discouraged for this collision type!'
      ENDIF
      WRITE (u,'(1X,54(''*'')/)')
      FIRST=.FALSE.
   ENDIF
!C---COPY PP TO P
   N=NN
   NUM=NN
   CALL KTCOPY(PP,N,P,(RECO.NE.1))
   ETOT=0
   DO I=1,N
      ETOT=ETOT+P(4,I)
   END DO
   IF (ETOT.EQ.0) CALL KTWARN('KTCLUS',102,*999)
   IF (ECUT.EQ.0) THEN
      ETSQ=1/ETOT**2
   ELSE
      ETSQ=1/ECUT**2
   ENDIF
   RSQ=R**2
!C---CALCULATE ALL PAIR KT's
   DO I=1,N-1
      DO J=I+1,N
         KTP(J,I)=-1
         KTP(I,J)=KTPAIR(ANGL,P(1,I),P(1,J),KTP(J,I))
      END DO
   END DO
!C---CALCULATE ALL SINGLE KT's
   DO I=1,N
      KTS(I)=KTSING(ANGL,TYPE,P(1,I))
   END DO
   KTMAX=0
!C---MAIN LOOP
300 CONTINUE
!C---FIND MINIMUM MEMBER OF KTP
   CALL KTPMIN(KTP,NMAX,N,IMIN,JMIN)
!C---FIND MINIMUM MEMBER OF KTS
   CALL KTSMIN(KTS,NMAX,N,KMIN)
!C---STORE Y VALUE OF TRANSITION FROM N TO N-1 JETS
   KTMIN=KTP(IMIN,JMIN)
```

```
      KTTMP=RSQ*KTS(KMIN)
      IF ((TYPE.GE.2.AND.TYPE.LE.4).AND. &
           (KTTMP.LE.KTMIN.OR.N.EQ.1)) &
           KTMIN=KTTMP
      KT(N)=KTMIN
      Y(N)=KT(N)*ETSQ
!C---IF MONO.GT.1, SEQUENCE IS SUPPOSED TO BE MONOTONIC, IF NOT, WARN
      IF (KTMIN.LT.KTMAX.AND.MONO.GT.1) CALL KTWARN('KTCLUS',1,*999)
      IF (KTMIN.GE.KTMAX) KTMAX=KTMIN
!C---IF LOWEST KT IS TO A BEAM, THROW IT AWAY AND MOVE LAST ENTRY UP
      IF (KTMIN.EQ.KTTMP) THEN
         CALL KTMOVE(P,KTP,KTS,NMAX,N,KMIN,1)
!C---UPDATE HISTORY AND CROSS-REFERENCES
         HIST(N)=KMIN
         INJET(N)=KMIN
         DO I=N,NN
            IF (INJET(I).EQ.KMIN) THEN
               KTLAST(I)=KTMAX
               INJET(I)=0
            ELSEIF (INJET(I).EQ.N) THEN
               INJET(I)=KMIN
            ENDIF
         END DO
!C---OTHERWISE MERGE JETS IMIN AND JMIN AND MOVE LAST ENTRY UP
      ELSE
         CALL KTMERG(P,KTP,KTS,NMAX,IMIN,JMIN,N,TYPE,ANGL,MONO,RECO)
         CALL KTMOVE(P,KTP,KTS,NMAX,N,JMIN,1)
!C---UPDATE HISTORY AND CROSS-REFERENCES
         HIST(N)=IMIN*NMAX+JMIN
         INJET(N)=IMIN
         DO I=N,NN
            IF (INJET(I).EQ.JMIN) THEN
               INJET(I)=IMIN
            ELSEIF (INJET(I).EQ.N) THEN
               INJET(I)=JMIN
            ENDIF
         END DO
      ENDIF
!C---THATS ALL THERE IS TO IT
      N=N-1
      IF (N.GT.1 .OR. N.GT.0.AND.(TYPE.GE.2.AND.TYPE.LE.4)) GOTO 300
      IF (N.EQ.1) THEN
         KT(N)=1D20
         Y(N)=KT(N)*ETSQ
      ENDIF
      RETURN
999 RETURN 1
  END SUBROUTINE KTCLUR
!C-----------------------------------------------------------------------
```

⟨*KTCLUS: public*⟩+≡

```
  public :: ktreco
```

⟨*KTCLUS: procedures*⟩+≡

```
!C-----------------------------------------------------------------------
```

```
    SUBROUTINE KTRECO(RECO,PP,NN,ECUT,YCUT,YMAC,PJET,JET,NJET,NSUB,*)
      IMPLICIT NONE
!C---RECONSTRUCT KINEMATICS OF JET SYSTEM, WHICH HAS ALREADY BEEN
!C   ANALYSED BY KTCLUS. NOTE THAT NO CONSISTENCY CHECK IS MADE: USER
!C   IS TRUSTED TO USE THE SAME PP VALUES AS FOR KTCLUS
!C
!C   RECO     = INPUT : RECOMBINATION SCHEME (NEED NOT BE SAME AS KTCLUS)
!C   PP(I,J)  = INPUT : 4-MOMENTUM OF Jth PARTICLE: I=1,4 => PX,PY,PZ,E
!C   NN       = INPUT : NUMBER OF PARTICLES
!C   ECUT     = INPUT : DENOMINATOR OF KT MEASURE. IF ZERO, ETOT IS USED
!C   YCUT     = INPUT : Y VALUE AT WHICH TO RECONSTRUCT JET MOMENTA
!C   YMAC     = INPUT : Y VALUE USED TO DEFINE MACRO-JETS, TO DETERMINE
!C                        WHICH JETS ARE SUB-JETS
!C   PJET(I,J)=OUTPUT : 4-MOMENTUM OF Jth JET AT SCALE YCUT
!C   JET(J)   =OUTPUT : THE MACRO-JET WHICH CONTAINS THE Jth JET,
!C                        SET TO ZERO IF JET IS NOT A SUB-JET
!C   NJET     =OUTPUT : THE NUMBER OF JETS
!C   NSUB     =OUTPUT : THE NUMBER OF SUB-JETS (EQUAL TO THE NUMBER OF
!C                        NON-ZERO ENTRIES IN JET())
!C   LAST ARGUMENT IS LABEL TO JUMP TO IF FOR ANY REASON THE EVENT
!C   COULD NOT BE PROCESSED
!C
!C   NOTE THAT THE MOMENTA ARE DECLARED DOUBLE PRECISION,
!C   AND ALL OTHER FLOATING POINT VARIABLES ARE DECLARED DOUBLE PRECISION
!C
      INTEGER NMAX,RECO,NUM,N,NN,NJET,NSUB,JET(*),HIST,IMIN,JMIN,I,J
      PARAMETER (NMAX=512)
      DOUBLE PRECISION PP(4,*),PJET(4,*)
      DOUBLE PRECISION ECUT,P,KT,KTP,KTS,ETOT,RSQ,ETSQ,YCUT,YMAC,KTLAST, &
           ROUND
      PARAMETER (ROUND=0.99999D0)
      COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
           KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM
!C---CHECK INPUT
      IF (RECO.LT.1.OR.RECO.GT.3) THEN
         PRINT *,'RECO=',RECO
         CALL KTWARN('KTRECO',100,*999)
      ENDIF
!C---COPY PP TO P
      N=NN
      IF (NUM.NE.NN) CALL KTWARN('KTRECO',101,*999)
      CALL KTCOPY(PP,N,P,(RECO.NE.1))
      IF (ECUT.EQ.0) THEN
         ETSQ=1/ETOT**2
      ELSE
         ETSQ=1/ECUT**2
      ENDIF
!C---KEEP MERGING UNTIL YCUT
 100  IF (ETSQ*KT(N).LT.ROUND*YCUT) THEN
         IF (HIST(N).LE.NMAX) THEN
            CALL KTMOVE(P,KTP,KTS,NMAX,N,HIST(N),0)
         ELSE
            IMIN=HIST(N)/NMAX
            JMIN=HIST(N)-IMIN*NMAX
```

```
            CALL KTMERG(P,KTP,KTS,NMAX,IMIN,JMIN,N,0,0,0,RECO)
            CALL KTMOVE(P,KTP,KTS,NMAX,N,JMIN,0)
         ENDIF
         N=N-1
         IF (N.GT.0) GOTO 100
      ENDIF
!C---IF YCUT IS TOO LARGE THERE ARE NO JETS
      NJET=N
      NSUB=N
      IF (N.EQ.0) RETURN
!C---SET UP OUTPUT MOMENTA
      DO I=1,NJET
         IF (RECO.EQ.1) THEN
            DO J=1,4
               PJET(J,I)=P(J,I)
            END DO
         ELSE
            PJET(1,I)=P(6,I)*COS(P(8,I))
            PJET(2,I)=P(6,I)*SIN(P(8,I))
            PJET(3,I)=P(6,I)*SINH(P(7,I))
            PJET(4,I)=P(6,I)*COSH(P(7,I))
         ENDIF
         JET(I)=I
      END DO
!C---KEEP MERGING UNTIL YMAC TO FIND THE FATE OF EACH JET
300   IF (ETSQ*KT(N).LT.ROUND*YMAC) THEN
         IF (HIST(N).LE.NMAX) THEN
            IMIN=0
            JMIN=HIST(N)
            NSUB=NSUB-1
         ELSE
            IMIN=HIST(N)/NMAX
            JMIN=HIST(N)-IMIN*NMAX
            IF (ETSQ*KTLAST(N).LT.ROUND*YMAC) NSUB=NSUB-1
         ENDIF
         DO I=1,NJET
            IF (JET(I).EQ.JMIN) JET(I)=IMIN
            IF (JET(I).EQ.N) JET(I)=JMIN
         END DO
         N=N-1
         IF (N.GT.0) GOTO 300
      ENDIF
      RETURN
999   RETURN 1
    END SUBROUTINE KTRECO
!C-----------------------------------------------------------------------
```

⟨*KTCLUS: procedures*⟩+≡

```
!C-----------------------------------------------------------------------
    FUNCTION KTPAIR(ANGL,P,Q,ANGLE)
      IMPLICIT NONE
!C---CALCULATE LOCAL KT OF PAIR, USING ANGULAR SCHEME:
!C   1=>ANGULAR, 2=>DeltaR, 3=>f(DeltaEta,DeltaPhi)
!C   WHERE f(eta,phi)=2(COSH(eta)-COS(phi)) IS THE QCD EMISSION METRIC
!C---IF ANGLE<0, IT IS SET TO THE ANGULAR PART OF THE LOCAL KT ON RETURN
```

```
!C   IF ANGLE>0, IT IS USED INSTEAD OF THE ANGULAR PART OF THE LOCAL KT
      INTEGER ANGL
! CHANGE      DOUBLE PRECISION P(9),Q(9),KTPAIR,R,KTMDPI,ANGLE,ETA,PHI,ESQ
      DOUBLE PRECISION P(9),Q(9),KTPAIR,R,ANGLE,ETA,PHI,ESQ
!C---COMPONENTS OF MOMENTA ARE PX,PY,PZ,E,1/P,PT,ETA,PHI,PT**2
      R=ANGLE
      IF (ANGL.EQ.1) THEN
         IF (R.LE.0) R=2*(1-(P(1)*Q(1)+P(2)*Q(2)+P(3)*Q(3))*(P(5)*Q(5)))
         ESQ=MIN(P(4),Q(4))**2
      ELSEIF (ANGL.EQ.2.OR.ANGL.EQ.3) THEN
         IF (R.LE.0) THEN
            ETA=P(7)-Q(7)
            PHI=KTMDPI(P(8)-Q(8))
            IF (ANGL.EQ.2) THEN
               R=ETA**2+PHI**2
            ELSE
               R=2*(COSH(ETA)-COS(PHI))
            ENDIF
         ENDIF
         ESQ=MIN(P(9),Q(9))
      ELSEIF (ANGL.EQ.4) THEN
         ESQ=(1d0/(P(5)*Q(5))-P(1)*Q(1)-P(2)*Q(2)- &
              P(3)*Q(3))*2D0/(P(5)*Q(5))/(0.0001D0+1d0/P(5)+1d0/Q(5))**2
         R=1d0
      ELSE
         CALL KTWARN('KTPAIR',200,*999)
         STOP
      ENDIF
      KTPAIR=ESQ*R
      IF (ANGLE.LT.0) ANGLE=R
999 END FUNCTION KTPAIR
!C-----------------------------------------------------------------------
    FUNCTION KTSING(ANGL,TYPE,P)
      IMPLICIT NONE
!C---CALCULATE KT OF PARTICLE, USING ANGULAR SCHEME:
!C   1=>ANGULAR, 2=>DeltaR, 3=>f(DeltaEta,DeltaPhi)
!C---TYPE=1 FOR E+E-, 2 FOR EP, 3 FOR PE, 4 FOR PP
!C   FOR EP, PROTON DIRECTION IS DEFINED AS -Z
!C   FOR PE, PROTON DIRECTION IS DEFINED AS +Z
      INTEGER ANGL,TYPE
      DOUBLE PRECISION P(9),KTSING,COSTH,R,SMALL
      DATA SMALL/1D-4/
      IF (ANGL.EQ.1.OR.ANGL.EQ.4) THEN
         COSTH=P(3)*P(5)
         IF (TYPE.EQ.2) THEN
            COSTH=-COSTH
         ELSEIF (TYPE.EQ.4) THEN
            COSTH=ABS(COSTH)
         ELSEIF (TYPE.NE.1.AND.TYPE.NE.3) THEN
            CALL KTWARN('KTSING',200,*999)
            STOP
         ENDIF
         R=2*(1-COSTH)
!C---IF CLOSE TO BEAM, USE APPROX 2*(1-COS(THETA))=SIN**2(THETA)
```

```
         IF (R.LT.SMALL) R=(P(1)**2+P(2)**2)*P(5)**2
         KTSING=P(4)**2*R
      ELSEIF (ANGL.EQ.2.OR.ANGL.EQ.3) THEN
         KTSING=P(9)
      ELSE
         CALL KTWARN('KTSING',201,*999)
         STOP
      ENDIF
999 END FUNCTION KTSING
!C-----------------------------------------------------------------------
    SUBROUTINE KTPMIN(A,NMAX,N,IMIN,JMIN)
      IMPLICIT NONE
!C---FIND THE MINIMUM MEMBER OF A(NMAX,NMAX) WITH IMIN < JMIN <= N
      INTEGER NMAX,N,IMIN,JMIN,KMIN,I,J,K
!C---REMEMBER THAT A(X+(Y-1)*NMAX)=A(X,Y)
!C   THESE LOOPING VARIABLES ARE J=Y-2, I=X+(Y-1)*NMAX
      DOUBLE PRECISION A(*),AMIN
      K=1+NMAX
      KMIN=K
      AMIN=A(KMIN)
      DO J=0,N-2
         DO I=K,K+J
            IF (A(I).LT.AMIN) THEN
               KMIN=I
               AMIN=A(KMIN)
            ENDIF
         END DO
         K=K+NMAX
      END DO
      JMIN=KMIN/NMAX+1
      IMIN=KMIN-(JMIN-1)*NMAX
    END SUBROUTINE KTPMIN
!C-----------------------------------------------------------------------
    SUBROUTINE KTSMIN(A,NMAX,N,IMIN)
      IMPLICIT NONE
!C---FIND THE MINIMUM MEMBER OF A
      INTEGER N,NMAX,IMIN,I
      DOUBLE PRECISION A(NMAX)
      IMIN=1
      DO I=1,N
         IF (A(I).LT.A(IMIN)) IMIN=I
      END DO
    END SUBROUTINE KTSMIN
!C-----------------------------------------------------------------------
    SUBROUTINE KTCOPY(A,N,B,ONSHLL)
      IMPLICIT NONE
!C---COPY FROM A TO B. 5TH=1/(3-MTM), 6TH=PT, 7TH=ETA, 8TH=PHI, 9TH=PT**2
!C   IF ONSHLL IS .TRUE. PARTICLE ENTRIES ARE PUT ON-SHELL BY SETTING E=P
      INTEGER I,N
      DOUBLE PRECISION A(4,N)
      LOGICAL ONSHLL
      DOUBLE PRECISION B(9,N),ETAMAX,SINMIN,EPS
      DATA ETAMAX,SINMIN,EPS/10,0,1D-6/
!C---SINMIN GETS CALCULATED ON FIRST CALL
```

```
      IF (SINMIN.EQ.0) SINMIN=1/COSH(ETAMAX)
      DO I=1,N
         B(1,I)=A(1,I)
         B(2,I)=A(2,I)
         B(3,I)=A(3,I)
         B(4,I)=A(4,I)
         B(5,I)=SQRT(A(1,I)**2+A(2,I)**2+A(3,I)**2)
         IF (ONSHLL) B(4,I)=B(5,I)
         IF (B(5,I).EQ.0) B(5,I)=1D-10
         B(5,I)=1/B(5,I)
         B(9,I)=A(1,I)**2+A(2,I)**2
         B(6,I)=SQRT(B(9,I))
         B(7,I)=B(6,I)*B(5,I)
         IF (B(7,I).GT.SINMIN) THEN
            B(7,I)=A(4,I)**2-A(3,I)**2
            IF (B(7,I).LE.EPS*B(4,I)**2.OR.ONSHLL) B(7,I)=B(9,I)
            B(7,I)=LOG((B(4,I)+ABS(B(3,I)))**2/B(7,I))/2
         ELSE
            B(7,I)=ETAMAX+2
         ENDIF
         B(7,I)=SIGN(B(7,I),B(3,I))
         IF (A(1,I).EQ.0 .AND. A(2,I).EQ.0) THEN
            B(8,I)=0
         ELSE
            B(8,I)=ATAN2(A(2,I),A(1,I))
         ENDIF
      END DO
    END SUBROUTINE KTCOPY
!C-----------------------------------------------------------------------
    SUBROUTINE KTMERG(P,KTP,KTS,NMAX,I,J,N,TYPE,ANGL,MONO,RECO)
      IMPLICIT NONE
!C---MERGE THE Jth PARTICLE IN P INTO THE Ith PARTICLE
!C   J IS ASSUMED GREATER THAN I. P CONTAINS N PARTICLES BEFORE MERGING.
!C---ALSO RECALCULATING THE CORRESPONDING KTP AND KTS VALUES IF MONO.GT.0
!C   FROM THE RECOMBINED ANGULAR MEASURES IF MONO.GT.1
!C---NOTE THAT IF MONO.LE.0, TYPE AND ANGL ARE NOT USED
      INTEGER ANGL,RECO,TYPE,I,J,K,N,NMAX,MONO
      DOUBLE PRECISION P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX),PT,PTT, &
! CHANGE           KTMDPI,KTUP,PI,PJ,ANG,KTPAIR,KTSING,ETAMAX,EPS
           KTUP,PI,PJ,ANG,ETAMAX,EPS
      KTUP(I,J)=KTP(MAX(I,J),MIN(I,J))
      DATA ETAMAX,EPS/10,1D-6/
      IF (J.LE.I) CALL KTWARN('KTMERG',200,*999)
!C---COMBINE ANGULAR MEASURES IF NECESSARY
      IF (MONO.GT.1) THEN
         DO K=1,N
            IF (K.NE.I.AND.K.NE.J) THEN
               IF (RECO.EQ.1) THEN
                  PI=P(4,I)
                  PJ=P(4,J)
               ELSEIF (RECO.EQ.2) THEN
                  PI=P(6,I)
                  PJ=P(6,J)
               ELSEIF (RECO.EQ.3) THEN
```

```
                  PI=P(9,I)
                  PJ=P(9,J)
               ELSE
                  CALL KTWARN('KTMERG',201,*999)
                  STOP
               ENDIF
               IF (PI.EQ.0.AND.PJ.EQ.0) THEN
                  PI=1
                  PJ=1
               ENDIF
               KTP(MAX(I,K),MIN(I,K))= &
                    (PI*KTUP(I,K)+PJ*KTUP(J,K))/(PI+PJ)
            ENDIF
         END DO
      ENDIF
      IF (RECO.EQ.1) THEN
!C---VECTOR ADDITION
         P(1,I)=P(1,I)+P(1,J)
         P(2,I)=P(2,I)+P(2,J)
         P(3,I)=P(3,I)+P(3,J)
!c         P(4,I)=P(4,I)+P(4,J) ! JA
         P(5,I)=SQRT(P(1,I)**2+P(2,I)**2+P(3,I)**2)
         P(4,I)=P(5,I) ! JA (Massless scheme)
         IF (P(5,I).EQ.0) THEN
            P(5,I)=1
         ELSE
            P(5,I)=1/P(5,I)
         ENDIF
      ELSEIF (RECO.EQ.2) THEN
!C---PT WEIGHTED ETA-PHI ADDITION
         PT=P(6,I)+P(6,J)
         IF (PT.EQ.0) THEN
            PTT=1
         ELSE
            PTT=1/PT
         ENDIF
         P(7,I)=(P(6,I)*P(7,I)+P(6,J)*P(7,J))*PTT
         P(8,I)=KTMDPI(P(8,I)+P(6,J)*PTT*KTMDPI(P(8,J)-P(8,I)))
         P(6,I)=PT
         P(9,I)=PT**2
      ELSEIF (RECO.EQ.3) THEN
!C---PT**2 WEIGHTED ETA-PHI ADDITION
         PT=P(9,I)+P(9,J)
         IF (PT.EQ.0) THEN
            PTT=1
         ELSE
            PTT=1/PT
         ENDIF
         P(7,I)=(P(9,I)*P(7,I)+P(9,J)*P(7,J))*PTT
         P(8,I)=KTMDPI(P(8,I)+P(9,J)*PTT*KTMDPI(P(8,J)-P(8,I)))
         P(6,I)=P(6,I)+P(6,J)
         P(9,I)=P(6,I)**2
      ELSE
         CALL KTWARN('KTMERG',202,*999)
```

```
         STOP
      ENDIF
!C---IF MONO.GT.0 CALCULATE NEW KT MEASURES. IF MONO.GT.1 USE ANGULAR ONES.
      IF (MONO.LE.0) RETURN
!C---CONVERTING BETWEEN 4-MTM AND PT,ETA,PHI IF NECESSARY
      IF (ANGL.NE.1.AND.RECO.EQ.1) THEN
         P(9,I)=P(1,I)**2+P(2,I)**2
         P(7,I)=P(4,I)**2-P(3,I)**2
         IF (P(7,I).LE.EPS*P(4,I)**2) P(7,I)=P(9,I)
         IF (P(7,I).GT.0) THEN
            P(7,I)=LOG((P(4,I)+ABS(P(3,I)))**2/P(7,I))/2
            IF (P(7,I).GT.ETAMAX) P(7,I)=ETAMAX+2
         ELSE
            P(7,I)=ETAMAX+2
         ENDIF
         P(7,I)=SIGN(P(7,I),P(3,I))
         IF (P(1,I).NE.0.AND.P(2,I).NE.0) THEN
            P(8,I)=ATAN2(P(2,I),P(1,I))
         ELSE
            P(8,I)=0
         ENDIF
      ELSEIF (ANGL.EQ.1.AND.RECO.NE.1) THEN
         P(1,I)=P(6,I)*COS(P(8,I))
         P(2,I)=P(6,I)*SIN(P(8,I))
         P(3,I)=P(6,I)*SINH(P(7,I))
         P(4,I)=P(6,I)*COSH(P(7,I))
         IF (P(4,I).NE.0) THEN
            P(5,I)=1/P(4,I)
         ELSE
            P(5,I)=1
         ENDIF
      ENDIF
      ANG=0
      DO K=1,N
         IF (K.NE.I.AND.K.NE.J) THEN
            IF (MONO.GT.1) ANG=KTUP(I,K)
            KTP(MIN(I,K),MAX(I,K))= &
                 KTPAIR(ANGL,P(1,I),P(1,K),ANG)
         ENDIF
      END DO
      KTS(I)=KTSING(ANGL,TYPE,P(1,I))
999 END SUBROUTINE KTMERG
!C-----------------------------------------------------------------------
    SUBROUTINE KTMOVE(P,KTP,KTS,NMAX,N,J,IOPT)
      IMPLICIT NONE
!C---MOVE THE Nth PARTICLE IN P TO THE Jth POSITION
!C---ALSO MOVING KTP AND KTS IF IOPT.GT.0
      INTEGER I,J,N,NMAX,IOPT
      DOUBLE PRECISION P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX)
      DO I=1,9
         P(I,J)=P(I,N)
      END DO
      IF (IOPT.LE.0) RETURN
      DO I=1,J-1
```

```
         KTP(I,J)=KTP(I,N)
         KTP(J,I)=KTP(N,I)
      END DO
      DO I=J+1,N-1
         KTP(J,I)=KTP(I,N)
         KTP(I,J)=KTP(N,I)
      END DO
      KTS(J)=KTS(N)
    END SUBROUTINE KTMOVE
!C-----------------------------------------------------------------------
```

⟨*KTCLUS: procedures*⟩+≡

```
    FUNCTION KTMDPI(PHI)
      IMPLICIT NONE
!C---RETURNS PHI, MOVED ONTO THE RANGE [-PI,PI)
      DOUBLE PRECISION KTMDPI,PHI,PI,TWOPI,THRPI,EPS
      PARAMETER (PI=3.14159265358979324D0,TWOPI=6.28318530717958648D0, &
           THRPI=9.42477796076937972D0)
      PARAMETER (EPS=1D-15)
      KTMDPI=PHI
      IF (KTMDPI.LE.PI) THEN
        IF (KTMDPI.GT.-PI) THEN
          GOTO 100
        ELSEIF (KTMDPI.GT.-THRPI) THEN
          KTMDPI=KTMDPI+TWOPI
        ELSE
          KTMDPI=-MOD(PI-KTMDPI,TWOPI)+PI
        ENDIF
      ELSEIF (KTMDPI.LE.THRPI) THEN
        KTMDPI=KTMDPI-TWOPI
      ELSE
        KTMDPI=MOD(PI+KTMDPI,TWOPI)-PI
      ENDIF
 100  IF (ABS(KTMDPI).LT.EPS) KTMDPI=0
    END FUNCTION KTMDPI
!C-----------------------------------------------------------------------
    SUBROUTINE KTWARN(SUBRTN,ICODE,*)
!C     DEALS WITH ERRORS DURING EXECUTION
!C     SUBRTN = NAME OF CALLING SUBROUTINE
!C     ICODE  = ERROR CODE:    - 99 PRINT WARNING & CONTINUE
!C                           100-199 PRINT WARNING & JUMP
!C                           200-    PRINT WARNING & STOP DEAD
!C-----------------------------------------------------------------------
      INTEGER ICODE
      CHARACTER(len=6) SUBRTN
      WRITE (6,10) SUBRTN,ICODE
10    FORMAT(/' KTWARN CALLED FROM SUBPROGRAM ',A6,': CODE =',I4/)
      IF (ICODE.LT.100) RETURN
      IF (ICODE.LT.200) RETURN 1
      STOP
    END SUBROUTINE KTWARN
!C-----------------------------------------------------------------------
!C-----------------------------------------------------------------------
!C-----------------------------------------------------------------------
```

Chapter 22

Multiple Interactions (MPI) Code

This is the code for the `WHIZARD` module for multiple interactions (MPI) in hadron collisions. It also provides the interleaved shower with together with the shower module.

22.1 Basic types

This file contains the module `muli_basic` which is included by all other modules of the MPI code. It's main purpose is serialization and deserialization, but it also contains some bitmodel parameters and some minor definitions common to all modules and types.

Serialization is implemented in three layers:

1. **I/O layer**: streamfile access and exact, retreivable and compiler independent representation of all intrinsic types
2. **Marking layer**: storing/restoring pointer association and better human readability
3. **Derived type layer**: abstract type as parent for all serializable derived types

The I/O Layer The I/O layer is implemented by `page_ring_t` and its type bound procedures. We chose not to use the standard formatted sequential I/O for several reasons:

- Sequential I/O is line orientated, but serialization is not.
- Formatted I/O of floating point numbers is inexact. There were problems in reading numbers like `1+epsilon(1)` or `huge(1)` including arithmetic exeptions and wrong results.
- Formatted reading is slow. This does matter, when you read files of $\mathcal{O}(100)$ MB.

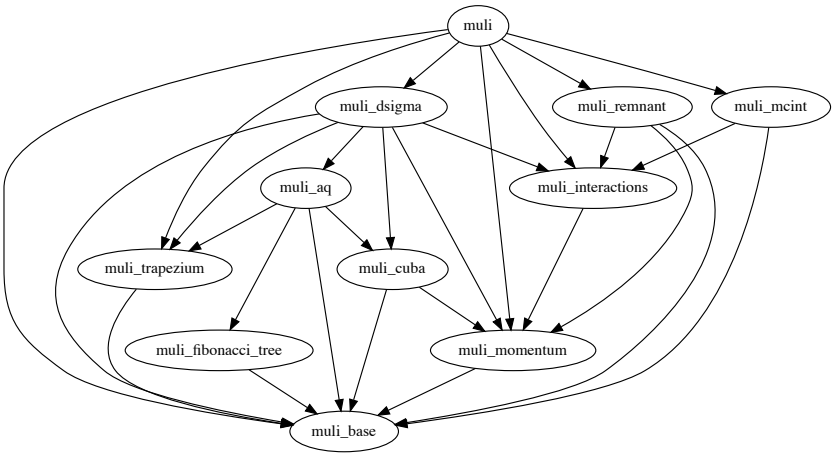

Figure 22.1: Module dependencies in `src/muli`.

`page_ring_t` is a ring buffer of default kind characters holding a region of the addressed file. All read and write procedures use the ring buffer, not the file. `page_ring` automatically reads more characters from the file when needed and writes and discards parts of the ring buffer when not any longer needed.

Marking layer Marking is done by `marking_t` and its type bound procedures. Files written by `marking_t` are regular XML files, so you can use your favorite XML editor to examine or edit serialized contents. The main purpose of this layer is to recover the type of polymorphic entities and to restore association of pointers, but it also assigns names to all contents, so it is much easier to read generated files.

Type recovery is done by a reference list. You must push a representive object to this reference list for each type that might get resolved before starting deserialization. `marker_t` will care for comparing these representives to the contents of the files.

Association restoration is done by a heap list. It is technically equal to the reference list, but holds all targets that have been processed so far. Generation of this list is handled by `marking_t`, so you dont't have to care for this aspect at all. Up to the present it is not possible to restore association, when a non-pointer target is serialized after its associated pointer is serialized. There is no trivial solution and this case does not appear here, so we will not take care of this.

Derived type layer Each instance that shall become serialized must extend `ser_class_t`. Essential for type recovery is the virtual type bound procedure `get_type`. Each non-virtual type shall override `get_type` and return the actual name of its type in lower-case letters. Each type which adds new, non-redundant components shall override `write_to_marker` and `read_from_marker`. These type-bound procedures define, what contents get serialized. While the

marker cares about tagging the type and association of the instance, every instance still has to define what to serialize. The rule is to mark the begin of its contents, then its parents procedure, then mark all non-redundant components, then mark the end of its contents. Finally, each serializable type shall override `print_to_unit`. This procedure is called for an arbitrary human-readable output. It is quite similar to `write_to_ring`, but without strict format and ignoring machine-readability.

`ser_class_t` has strictly speaking two layers. `write_to_marker` and `read_from_marker` are only for internal usage. Serialization and deserialization are triggered by the TBPs serialize and deserialize. These procedures take care of initialization and finalization of the marker. A serializable type should override these procedures to push a representive of itself and any other references to the reference list of it's marker before (de)serialization and to pop them from the list afterwards.

⟨`muli_base.f90`⟩≡

```
⟨File header⟩

module muli_base
  use, intrinsic :: iso_fortran_env
⟨Use kinds with double⟩
  use kinds, only: i64
⟨Use strings⟩
  use constants
  use io_units
  use diagnostics

⟨Standard module head⟩

⟨Muli base: variables⟩

⟨Muli base: public⟩

⟨Muli base: types⟩

⟨Muli base: interfaces⟩

contains

⟨Muli base: procedures⟩

end module muli_base
```

These are the bitmodel parameters.

⟨*Muli base: variables*⟩≡

```
  integer, public, parameter :: dik = i64
  integer(dik), public, parameter :: i_one = int(1, kind=dik)
  integer(dik), public, parameter :: i_zero = int(0, kind=dik)
```

These are the serialization parameters.

⟨*Muli base: variables*⟩+≡

```
  integer(dik), public, parameter :: serialize_page_size = 1024
  integer(dik), public, parameter :: serialize_ok = 0000
  integer(dik), public, parameter :: serialize_syntax_error = 1001
```

```
integer(dik), public, parameter :: serialize_wrong_tag = 1002
integer(dik), public, parameter :: serialize_wrong_id = 1003
integer(dik), public, parameter :: serialize_wrong_type = 1004
integer(dik), public, parameter :: serialize_wrong_name = 1005
integer(dik), public, parameter :: serialize_no_target = 1006
integer(dik), public, parameter :: serialize_no_pointer = 1007
integer(dik), public, parameter :: serialize_wrong_action = 1008
integer(dik), public, parameter :: serialize_unexpected_content = 1009
integer(dik), public, parameter :: serialize_null = 1010
integer(dik), public, parameter :: serialize_nothing = 1011
logical, public, parameter :: serialize_default_indent = .true.
logical, public, parameter :: serialize_default_line_break = .true.
logical, public, parameter :: serialize_default_asynchronous = .false.
```

And some private variables:

⟨*Muli base: variables*⟩+≡

```
integer(dik) :: last_id = 0
character(len=*), parameter :: serialize_integer_characters = &
     "-0123456789"
```

⟨*Muli base: interfaces*⟩≡

```
abstract interface
   subroutine ser_write_if (this, marker, status)
     import ser_class_t
     import marker_t
     import dik
     class(ser_class_t), intent(in) :: this
     class(marker_t), intent(inout) :: marker
     integer(dik), intent(out) :: status
   end subroutine ser_write_if
end interface

abstract interface
   subroutine ser_read_if (this, marker, status)
     import ser_class_t
     import marker_t
     import dik
     class(ser_class_t), intent(out) :: this
     class(marker_t), intent(inout) :: marker
     integer(dik), intent(out) :: status
   end subroutine ser_read_if
end interface

abstract interface
   subroutine ser_unit (this, unit, parents, components, peers)
     import ser_class_t
     import dik
     class(ser_class_t), intent(in) :: this
     integer,intent(in) :: unit
     integer(dik), intent(in) :: parents,components,peers
   end subroutine ser_unit
end interface

abstract interface
```

```
      pure subroutine ser_type (type)
        character(:), allocatable, intent(out) :: type
      end subroutine ser_type
   end interface
```

⟨*Muli base: interfaces*⟩+≡

```
   abstract interface
      elemental function measure_int (this)
        import
        class(measure_class_t), intent(in) :: this
        real(default) :: measure_int
      end function measure_int
   end interface
```

⟨*Muli base: public*⟩≡

```
   public :: operator(<)
```

⟨*Muli base: interfaces*⟩+≡

```
   interface operator(<)
      module procedure measurable_less_measurable
      module procedure measurable_less_default
   end interface operator(<)
```

⟨*Muli base: public*⟩+≡

```
   public :: operator(<=)
```

⟨*Muli base: interfaces*⟩+≡

```
   interface operator(<=)
      module procedure measurable_less_or_equal_measurable
      module procedure measurable_less_or_equal_default
   end interface operator(<=)
```

⟨*Muli base: public*⟩+≡

```
   public :: operator(==)
```

⟨*Muli base: interfaces*⟩+≡

```
   interface operator(==)
      module procedure measurable_equal_measurable
      module procedure measurable_equal_default
   end interface operator(==)
```

⟨*Muli base: public*⟩+≡

```
   public :: operator(>=)
```

⟨*Muli base: interfaces*⟩+≡

```
   interface operator(>=)
      module procedure measurable_equal_or_greater_measurable
      module procedure measurable_equal_or_greater_default
   end interface operator(>=)
```

⟨*Muli base: public*⟩+≡

```
   public :: operator(>)
```

⟨*Muli base: interfaces*⟩+≡

```
   interface operator(>)
      module procedure measurable_greater_measurable
      module procedure measurable_greater_default
   end interface operator(>)
```

⟨*Muli base: interfaces*⟩+≡

```
interface page_ring_position_is_before
   module procedure page_ring_position_is_before_int_pos
   module procedure page_ring_position_is_before_pos_pos
   module procedure page_ring_position_is_before_pos_int
end interface
```

⟨*Muli base: public*⟩+≡

```
public :: ser_class_t
```

⟨*Muli base: types*⟩≡

```
type, abstract :: ser_class_t
 contains
⟨Muli base: serial class: TBP⟩
end type ser_class_t
```

⟨*Muli base: serial class: TBP*⟩≡

```
procedure(ser_write_if), deferred :: write_to_marker
```

This is a dummy procedure. Usually, you do not need to deserialize targets, so by implementing this dummy we don't force all descendants to override this procedure. Then again this is the only way to read targets from markers.

⟨*Muli base: serial class: TBP*⟩+≡

```
procedure(ser_read_if), deferred :: read_from_marker
```

⟨*Muli base: procedures*⟩≡

```
subroutine serializable_read_target_from_marker (this, marker, status)
  class(ser_class_t), target, intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  write (output_unit, "(A)") "serializable_read_target_from_marker:"
  write (output_unit, "(A)") "This is a dummy procedure. Usually, this " &
       // "message indicates a missing overridden " &
       // "read_target_from_marker TPB for "
  call this%write_type (output_unit)
  write (output_unit, "(A)") ""
  call this%read_from_marker (marker, status)
end subroutine serializable_read_target_from_marker
```

⟨*Muli base: serial class: TBP*⟩+≡

```
procedure(ser_unit), deferred :: print_to_unit
```

⟨*Muli base: procedures*⟩+≡

```
subroutine serializable_serialize_to_unit (this, unit, name)
  class(ser_class_t), intent(in) :: this
  integer, intent(in) :: unit
  character (len=*), intent(in) :: name
  logical :: opened
  character(32) :: file
  !!!    gfortran bug
  !!!    character::stream
  character::write
  type(marker_t)::marker
  !    inquire(unit=unit,opened=opened,stream=stream,write=write)
```

```
    inquire (unit=unit,opened=opened,write=write)
    if (opened) then
       !!! if(stream=="Y")then
          if(write=="Y")then
             print *,"dummy: serializable_serialize_to_unit"
             stop
          else
             print *,"serializable_serialize_to_unit: cannot write to read-only unit."
          end if
          !!!       else
          !!!          print *,"serializable_serialize_to_unit: access kind of unit is not 'stream
          !!!       end if
    else
       call msg_error ("serializable_serialize_to_unit: file is not opened.")
    end if
  end subroutine serializable_serialize_to_unit
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure(ser_type), nopass, deferred :: get_type
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure, nopass :: verify_type => serializable_verify_type
```

⟨*Muli base: procedures*⟩+≡

```
  elemental function serializable_verify_type (type) result (match)
    character(*), intent(in) :: type
    logical :: match
    match = type == "ser_class_t"
  end function serializable_verify_type
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure :: read_target_from_marker => &
       serializable_read_target_from_marker
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure :: write_type => serializable_write_type
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine serializable_write_type (this, unit)
    class(ser_class_t), intent(in) :: this
    integer,intent(in) :: unit
    character(:), allocatable :: this_type
    call this%get_type (this_type)
    write (unit, "(A)", advance="no") this_type
  end subroutine serializable_write_type
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure :: print => serializable_print
```

⟨*Muli base: procedures*⟩+≡

```
  recursive subroutine serializable_print &
       (this, parents, components, peers, unit)
    class(ser_class_t), intent(in) :: this
    integer(dik), intent(in) :: parents, components, peers
    integer, intent(in), optional :: unit
```

```
    integer :: u
    u = given_output_unit (unit)
    write (u, "(A)")
    write (u, "(A)", advance="no")  "Instance of type: "
    call this%write_type (u)
    write (u, "(A)")
    call this%print_to_unit (u, parents, components, peers)
  end subroutine serializable_print
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure :: print_error => serializable_print_error
```

⟨*Muli base: procedures*⟩+≡

```
  recursive subroutine serializable_print_error (this)
    class(ser_class_t), intent(in) :: this
    call this%print_to_unit (error_unit, i_zero, i_zero, i_zero)
  end subroutine serializable_print_error
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure :: print_all => serializable_print_all
```

⟨*Muli base: procedures*⟩+≡

```
  recursive subroutine serializable_print_all (this, unit)
    class(ser_class_t), intent(in) :: this
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(A)")
    write (u, "(A)", advance="no")  "Instance of type: "
    call this%write_type (u)
    write (u, "(A)")
    call this%print_to_unit (u, huge(i_one), huge(i_one), huge(i_one))
  end subroutine serializable_print_all
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure :: print_little => serializable_print_little
```

⟨*Muli base: procedures*⟩+≡

```
  recursive subroutine serializable_print_little (this, unit)
    class(ser_class_t), intent(in) :: this
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (u)
    write(u, "(A)")
    write(u, "(A)", advance="no")  "Instance of type: "
    call this%write_type (u)
    write(u, "(A)")
    call this%print_to_unit (u, i_zero, i_zero, i_zero)
  end subroutine serializable_print_little
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure :: print_parents => serializable_print_parents
```

⟨*Muli base: procedures*⟩+≡

```
recursive subroutine serializable_print_parents (this)
  class(ser_class_t), intent(in) :: this
  write(output_unit, "(A)")
  write(output_unit, "(A)", advance="no")  "Instance of type: "
  call this%write_type (output_unit)
  write (output_unit, "(A)")
  call this%print_to_unit (output_unit, huge(i_one), i_zero, i_zero)
end subroutine serializable_print_parents
```

⟨*Muli base: serial class: TBP*⟩+≡

```
procedure :: print_components => serializable_print_components
```

⟨*Muli base: procedures*⟩+≡

```
recursive subroutine serializable_print_components(this)
  class(ser_class_t), intent(in) :: this
  write (output_unit, "(A)")
  write (output_unit, "(A)", advance="no")  "Instance of type: "
  call this%write_type (output_unit)
  write(output_unit, "(A)")
  call this%print_to_unit (output_unit, i_zero, huge(i_one), i_zero)
end subroutine serializable_print_components
```

⟨*Muli base: serial class: TBP*⟩+≡

```
procedure :: print_peers => serializable_print_peers
```

⟨*Muli base: procedures*⟩+≡

```
recursive subroutine serializable_print_peers (this)
  class(ser_class_t), intent(in) :: this
  write (output_unit, "(A)")
  write (output_unit, "(A)", advance="no")  "Instance of type: "
  call this%write_type (output_unit)
  write (output_unit, "(A)")
  call this%print_to_unit (output_unit, i_zero, i_zero, huge(i_one))
end subroutine serializable_print_peers
```

⟨*Muli base: serial class: TBP*⟩+≡

```
procedure :: serialize_to_file => serializable_serialize_to_file
```

⟨*Muli base: procedures*⟩+≡

```
subroutine serializable_serialize_to_file (this, name, file)
  class(ser_class_t), intent(in) :: this
  character(len=*), intent(in) :: file, name
  type(marker_t) :: marker
  call marker%open_for_write_access (file)
  write (output_unit, "(A,A)")  &
       "Serializable_serialize_to_file: writing xml preamble to ", file
  call marker%activate_next_page ()
  call marker%push ('<?xml version="1.0"?>')
  call marker%mark_begin (tag="file", name = file)
  flush(marker%unit)
  call this%serialize_to_marker (marker, name)
  call marker%mark_end ("file")
  call marker%close ()
```

```
    call marker%finalize ()
  end subroutine serializable_serialize_to_file
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure :: serialize_to_unit => serializable_serialize_to_unit
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure :: serialize_to_marker => serializable_serialize_to_marker
```

⟨*Muli base: procedures*⟩+≡

```
  recursive subroutine serializable_serialize_to_marker (this, marker, name)
    class(ser_class_t), intent(in) :: this
    class(marker_t), intent(inout) :: marker
    character(len=*), intent(in) :: name
    if (marker%action == 1) then
       call marker%mark_instance (this, name)
    else
       call msg_error ("serializable_serialize_to_marker: Marker is " &
            // "not ready for write access.")
    end if
  end subroutine serializable_serialize_to_marker
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure :: deserialize_from_file => serializable_deserialize_from_file
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine serializable_deserialize_from_file (this, name, file)
    class(ser_class_t), intent(out) :: this
    character(*), intent(in) :: name, file
    type(marker_t) :: marker
    integer(dik), dimension(2) :: p1, p2
    call marker%open_for_read_access (file, "</file>")
    marker%eof_int = huge(i_one)
    marker%eof_pos = page_ring_position (marker%eof_int)
    call marker%read_page ()
    call marker%find ('<?', skip=2, proceed=.true., pos=p1)
    call marker%find ('?>', skip=3, proceed=.false., pos=p2)
    if ((p1(2) <= 0) .or. (p2(2) <= 0)) then
       call msg_error ("serializable_deserialize_from_file: no " &
            // "version substring found.")
    end if
    call marker%set_position (p2)
    call marker%find ('<file ', skip=4, proceed=.true., pos=p1)
    call marker%find ('>', skip=1, proceed=.false., pos=p2)
    if((p1(2)>0) .and. (p2(2)>0))then
       call marker%push_position (p2)
       call marker%find ('name="', skip=4, proceed=.true., pos=p1)
       call marker%find ('"', skip=1, proceed=.false., pos=p2)
       call marker%pop_position ()
    else
       call msg_error ("serializable_deserialize_from_file: no file " &
            // "header found.")
    end if
    call this%deserialize_from_marker (name, marker)
```

```
    call marker%close ()
    call marker%finalize ()
  end subroutine serializable_deserialize_from_file
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure :: deserialize_from_unit => &
       serializable_deserialize_from_unit
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine serializable_deserialize_from_unit (this, unit, name)
    class(ser_class_t), intent(inout) :: this
    integer, intent(in) :: unit
    character(len=*), intent(in) :: name
    logical::opened
    !!!    gfortran bug
    !!!    character::stream
    character::read
    type(marker_t)::marker
    !!!    inquire(unit=unit,opened=opened,stream=stream,read=read)
    inquire(unit=unit,opened=opened,read=read)
    if(opened)then
       !!!       if(stream=="Y")then
          if(read=="Y")then
             print *,"dummy: serializable_serialize_from_unit"
             stop
          else
             print *,"serializable_serialize_from_unit: cannot write from read-only unit."
          end if
          !!!       else
          !!!          print *,"serializable_serialize_from_unit: access kind of unit is not 'stre
          !!!       end if
    else
       print *,"serializable_serialize_from_unit: file is not opened."
    end if
  end subroutine serializable_deserialize_from_unit
```

This needs to be made public, and not only be present as a TBP.

⟨*Muli base: public*⟩+≡

```
  public :: serializable_deserialize_from_marker
```

⟨*Muli base: serial class: TBP*⟩+≡

```
  procedure :: deserialize_from_marker => &
       serializable_deserialize_from_marker
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine serializable_deserialize_from_marker (this, name, marker)
    class(ser_class_t), intent(out) :: this
    character(*), intent(in) :: name
    class(marker_t), intent(inout) :: marker
    integer(dik) :: status
    if (marker%action == 2) then
       call marker%pick_instance (name, this, status)
    else
       call msg_error ("serializable_deserialize_from_marker: Marker is " &
```

```
            // "not ready for read access.")
      end if
    end subroutine serializable_deserialize_from_marker
```

⟨*Muli base: serial class: TBP*⟩+≡

```
    generic :: serialize => serialize_to_file, serialize_to_unit, &
         serialize_to_marker
```

⟨*Muli base: serial class: TBP*⟩+≡

```
    generic :: deserialize => deserialize_from_file, &
         deserialize_from_unit, deserialize_from_marker
```

⟨*Muli base: public*⟩+≡

```
    public :: serialize_print_peer_pointer
```

⟨*Muli base: procedures*⟩+≡

```
    recursive subroutine serialize_print_peer_pointer &
         (ser, unit, parents, components, peers, name)
      class(ser_class_t), pointer, intent(in) :: ser
      integer, intent(in) :: unit
      integer(dik) :: parents, components, peers
      character(len=*), intent(in) :: name
      if (associated (ser)) then
         write (unit,*) name, " is associated."
         if (peers>0) then
            write (unit,*) "Printing components of ", name
            call ser%print_to_unit (unit, parents, components, peers - i_one)
         else
            write (unit,*) "Skipping components of ", name
         end if
      else
         write (unit,*) name, " is not associated."
      end if
    end subroutine serialize_print_peer_pointer
```

⟨*Muli base: public*⟩+≡

```
    public :: serialize_print_comp_pointer
```

⟨*Muli base: procedures*⟩+≡

```
    recursive subroutine serialize_print_comp_pointer &
         (ser, unit, parents, components, peers, name)
      class(ser_class_t), pointer, intent(in) :: ser
      integer, intent(in) :: unit
      integer(dik), intent(in) :: parents, components, peers
      character(len=*), intent(in) :: name
      if (associated (ser)) then
         write (unit,*)  name," is associated."
         if (components > 0) then
            write (unit,*) "Printing components of ", name
            call ser%print_to_unit (unit, parents, components - i_one, peers)
         else
            write (unit,*) "Skipping components of ", name
         end if
      else
         write (unit,*) name," is not associated."
```

```
    end if
  end subroutine serialize_print_comp_pointer
```

⟨*Muli base: public*⟩+≡

```
  public :: serialize_print_allocatable
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine serialize_print_allocatable &
       (ser, unit, parents, components, peers, name)
    class(ser_class_t), allocatable, intent(in) :: ser
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    character(len=*), intent(in) :: name
    if (allocated (ser)) then
       write (unit,*) name, " is allocated."
       if (components > 0) then
          write (unit,*) "Printing components of ",name
          call ser%print_to_unit (unit, parents, components-1, peers)
       else
          write (unit,*) "Skipping components of ",name
       end if
    else
       write (unit,*) name," is not allocated."
    end if
  end subroutine serialize_print_allocatable
```

⟨*Muli base: public*⟩+≡

```
  public :: measure_class_t
```

⟨*Muli base: types*⟩+≡

```
  type, abstract, extends (ser_class_t) :: measure_class_t
    contains
      procedure(measure_int), public, deferred :: measure
  end type measure_class_t
```

⟨*Muli base: public*⟩+≡

```
  public :: identified_t
```

⟨*Muli base: types*⟩+≡

```
  type, extends (ser_class_t) :: identified_t
     private
     integer(dik) :: id
     type(string_t) :: name
   contains
   ⟨Muli base: identified type: TBP⟩
  end type identified_t
```

⟨*Muli base: identified type: TBP*⟩≡

```
  procedure :: base_write_to_marker => identified_write_to_marker
  procedure :: write_to_marker => identified_write_to_marker
```

⟨*Muli base: procedures*⟩+≡

```
subroutine identified_write_to_marker (this, marker, status)
  class(identified_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  integer(dik) :: id
  id = this%get_id ()
  call marker%mark_begin ("identified_t")
  call marker%mark ("name", this%get_name ())
  call marker%mark ("id", id)
  call marker%mark_end ("identified_t")
end subroutine identified_write_to_marker
```

⟨*Muli base: identified type: TBP*⟩+≡

```
procedure :: base_read_from_marker => identified_read_from_marker
procedure :: read_from_marker => identified_read_from_marker
```

⟨*Muli base: procedures*⟩+≡

```
subroutine identified_read_from_marker (this, marker, status)
  class(identified_t), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  character(:), allocatable :: name
  call marker%pick_begin ("identified_t", status=status)
  call marker%pick ("name", name, status)
  call marker%pick ("id", this%id, status)
  call marker%pick_end ("identified_t", status=status)
  this%name = name
end subroutine identified_read_from_marker
```

⟨*Muli base: identified type: TBP*⟩+≡

```
procedure :: base_print_to_unit => identified_print_to_unit
procedure :: print_to_unit => identified_print_to_unit
```

⟨*Muli base: procedures*⟩+≡

```
subroutine identified_print_to_unit (this, unit, parents, components, peers)
  class(identified_t), intent(in) :: this
  integer, intent(in) :: unit
  integer(dik), intent(in) :: parents, components, peers
  write (unit, "(A)")      "Components of identified_t:"
  write (unit, "(A,A)")    "Name:             ", this%get_name ()
  write (unit, "(A,I10)")  "ID:               ", this%get_id ()
end subroutine identified_print_to_unit
```

⟨*Muli base: identified type: TBP*⟩+≡

```
procedure, nopass :: get_type => identified_get_type
```

⟨*Muli base: procedures*⟩+≡

```
pure subroutine identified_get_type (type)
  character(:), allocatable, intent(out) :: type
  allocate (type, source="identified_t")
end subroutine identified_get_type
```

⟨*Muli base: identified type: TBP*⟩+≡
```
procedure, nopass :: verify_type => identified_verify_type
```

⟨*Muli base: procedures*⟩+≡
```
elemental logical function identified_verify_type (type)
  character(len=*), intent(in) ::type
  identified_verify_type = (type == "identified_t")
end function identified_verify_type
```

⟨*Muli base: identified type: TBP*⟩+≡
```
generic :: initialize => identified_initialize
procedure, private :: identified_initialize
```

⟨*Muli base: procedures*⟩+≡
```
subroutine identified_initialize (this, id, name)
  class(identified_t), intent(out) :: this
  integer(dik), intent(in) :: id
  character(len=*), intent(in) :: name
  this%name = name
  this%id = id
end subroutine identified_initialize
```

⟨*Muli base: identified type: TBP*⟩+≡
```
procedure :: get_id => identified_get_id
```

⟨*Muli base: procedures*⟩+≡
```
elemental function identified_get_id (this) result(id)
  class(identified_t), intent(in) :: this
  integer(dik) :: id
  id = this%id
end function identified_get_id
```

⟨*Muli base: identified type: TBP*⟩+≡
```
procedure :: get_name => identified_get_name
```

⟨*Muli base: procedures*⟩+≡
```
pure function identified_get_name (this)
  class(identified_t), intent(in) :: this
  character(len (this%name)) :: identified_get_name
  identified_get_name = char (this%name)
end function identified_get_name
```

⟨*Muli base: public*⟩+≡
```
public :: unique_t
```

⟨*Muli base: types*⟩+≡
```
type, extends (identified_t) :: unique_t
   private
   integer(dik) :: unique_id
 contains
 ⟨Muli base: unique type: TBP⟩
end type unique_t
```

⟨*Muli base: unique type: TBP*⟩≡

```
procedure, nopass :: get_type => unique_get_type
```

⟨*Muli base: procedures*⟩+≡

```
pure subroutine unique_get_type (type)
  character(:), allocatable, intent(out) :: type
  allocate (type, source="unique_t")
end subroutine unique_get_type
```

⟨*Muli base: unique type: TBP*⟩+≡

```
procedure, nopass :: verify_type => unique_verify_type
```

⟨*Muli base: procedures*⟩+≡

```
elemental logical function unique_verify_type (type)
  character(len=*), intent(in) :: type
  unique_verify_type = (type == "unique_t")
end function unique_verify_type
```

⟨*Muli base: unique type: TBP*⟩+≡

```
procedure :: write_to_marker => unique_write_to_marker
```

⟨*Muli base: procedures*⟩+≡

```
subroutine unique_write_to_marker (this, marker, status)
  class(unique_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%mark_begin ("unique_t")
  call identified_write_to_marker (this, marker, status)
  call marker%mark ("unique_id", this%get_unique_id ())
  call marker%mark_end ("unique_t")
end subroutine unique_write_to_marker
```

⟨*Muli base: unique marker: TBP*⟩≡

```
procedure :: read_from_marker => unique_read_from_marker
```

⟨*Muli base: procedures*⟩+≡

```
subroutine unique_read_from_marker (this, marker, status)
  class(unique_t), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%pick_begin ("unique_t", status=status)
  call identified_read_from_marker (this, marker, status)
  call marker%pick ("unique_id", this%unique_id, status)
  call marker%pick_end ("unique_t", status)
end subroutine unique_read_from_marker
```

⟨*Muli base: unique type: TBP*⟩+≡

```
procedure :: print_to_unit => unique_print_to_unit
```

⟨*Muli base: procedures*⟩+≡

```
subroutine unique_print_to_unit (this, unit, parents, components, peers)
  class(unique_t), intent(in) :: this
  integer,intent(in) :: unit
  integer(dik), intent(in) :: parents, components, peers
```

```
    if (parents > 0) call identified_print_to_unit &
         (this, unit, parents-1, components, peers)
    write (unit, "(A,I10)")  "Unique ID:        ", this%get_unique_id ()
  end subroutine unique_print_to_unit
```

⟨*Muli base: unique type: TBP*⟩+≡

```
  procedure :: identified_initialize => unique_initialize
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine unique_initialize(this,id,name)
    class(unique_t), intent(out) :: this
    integer(dik), intent(in) :: id
    character(len=*), intent(in) :: name
    call identified_initialize (this, id, name)
    last_id = last_id + 1
    this%unique_id = last_id
  end subroutine unique_initialize
```

⟨*Muli base: unique type: TBP*⟩+≡

```
  procedure :: get_unique_id => unique_get_unique_id
```

⟨*Muli base: procedures*⟩+≡

```
  pure function unique_get_unique_id (this)
    class(unique_t), intent(in) :: this
    integer(dik) :: unique_get_unique_id
    unique_get_unique_id = this%unique_id
  end function unique_get_unique_id
```

⟨*Muli base: types*⟩+≡

```
  type :: serializable_ref_type
     private
     integer(dik) :: id
     class(ser_class_t), pointer :: ref => null()
     class(serializable_ref_type), pointer :: next => null()
   contains
   ⟨Muli base: serial ref: TBP⟩
  end type serializable_ref_type
```

⟨*Muli base: serial ref: TBP*⟩≡

```
  procedure :: finalize => serializable_ref_finalize
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine serializable_ref_finalize (this)
    class(serializable_ref_type), intent(inout) :: this
    class(serializable_ref_type), pointer :: next
    do while (associated (this%next))
       next => this%next
       this%next => next%next
       nullify (next%ref)
       deallocate (next)
    end do
    if (associated (this%ref))  nullify (this%ref)
  end subroutine serializable_ref_finalize
```

⟨*Muli base: types*⟩+≡

```
type :: position_stack_t
   private
   integer(dik), dimension(2) :: position
   class(position_stack_t), pointer :: next => null()
 contains
 ⟨Muli base: position stack: TBP⟩
end type position_stack_t
```

⟨*Muli base: position stack: TBP*⟩≡

```
generic :: push => push_head, push_given
procedure :: push_head => position_stack_push_head
procedure :: push_given => position_stack_push_given
```

⟨*Muli base: procedures*⟩+≡

```
subroutine position_stack_push_head (this)
  class(position_stack_t) :: this
  class(position_stack_t), pointer :: new
  allocate (new)
  new%next => this%next
  new%position = this%position
  this%next => new
end subroutine position_stack_push_head
```

⟨*Muli base: procedures*⟩+≡

```
subroutine position_stack_push_given (this, position)
  class(position_stack_t) :: this
  integer(dik), dimension(2), intent(in) :: position
  class(position_stack_t), pointer:: new
  allocate (new)
  new%next => this%next
  new%position = position
  this%next => new
end subroutine position_stack_push_given
```

⟨*Muli base: position stack: TBP*⟩+≡

```
generic :: pop => position_stack_pop, position_stack_drop
procedure :: position_stack_pop
procedure :: position_stack_drop
```

⟨*Muli base: procedures*⟩+≡

```
subroutine position_stack_pop (this)
  class(position_stack_t) :: this
  class(position_stack_t), pointer :: old
  if (associated (this%next)) then
     old => this%next
     this%next => old%next
     this%position = old%position
     deallocate (old)
  end if
end subroutine position_stack_pop
```

⟨*Muli base: procedures*⟩+≡

```
subroutine position_stack_drop (this, position)
  class(position_stack_t) :: this
  integer(dik), dimension(2), intent(out) :: position
  class(position_stack_t), pointer :: old
  if (associated (this%next)) then
     old => this%next
     this%next => old%next
     position = old%position
     deallocate (old)
  else
     position= [0,0]
  end if
end subroutine position_stack_drop
```

⟨*Muli base: position stack: TBP*⟩+≡

```
procedure :: nth_position => position_stack_nth_position
```

⟨*Muli base: procedures*⟩+≡

```
function position_stack_nth_position (this, n) result (position)
  class(position_stack_t), intent(in) :: this
  integer(dik), intent(in) :: n
  integer(dik), dimension(2) :: position
  class(position_stack_t), pointer :: tmp
  integer(dik) :: pos
  tmp => this%next
  pos = n
  do while (associated (tmp) .and. pos>0)
     tmp => tmp%next
     pos = pos - 1
  end do
  if (associated(tmp)) then
     position = tmp%position
  else
     position = [0,0]
  end if
end function position_stack_nth_position
```

⟨*Muli base: position stack: TBP*⟩+≡

```
procedure :: first => position_stack_first
```

⟨*Muli base: procedures*⟩+≡

```
function position_stack_first(this) result(position)
  class(position_stack_t), intent(in) :: this
  integer(kind=dik), dimension(2) :: position, tmp_position
  class(position_stack_t), pointer :: tmp_stack
  tmp_position = this%position
  tmp_stack => this%next
  do while (associated (tmp_stack))
     if (page_ring_position_is_before (tmp_stack%position, tmp_position)) then
        tmp_position = tmp_stack%position
     end if
     tmp_stack => tmp_stack%next
  end do
```

```
end function position_stack_first
```

⟨*Muli base: position stack: TBP*⟩+≡

```
procedure :: last => position_stack_last
```

⟨*Muli base: procedures*⟩+≡

```
function position_stack_last (this) result (position)
  class(position_stack_t), intent(in) :: this
  integer(dik), dimension(2) :: position, tmp_position
  class(position_stack_t), pointer :: tmp_stack
  tmp_position = this%position
  tmp_stack => this%next
  do while (associated (tmp_stack))
     if (page_ring_position_is_before (tmp_position, tmp_stack%position)) then
        tmp_position = tmp_stack%position
     end if
     tmp_stack => tmp_stack%next
  end do
end function position_stack_last
```

⟨*Muli base: position stack: TBP*⟩+≡

```
procedure :: range => position_stack_range
```

⟨*Muli base: procedures*⟩+≡

```
pure function position_stack_range (this) result (position)
  class(position_stack_t), intent(in) :: this
  integer(dik), dimension(2) :: position
  class(position_stack_t), pointer :: tmp
end function position_stack_range
```

⟨*Muli base: public*⟩+≡

```
public :: page_ring_t
```

⟨*Muli base: types*⟩+≡

```
type :: page_ring_t
   private
   logical :: asynchronous = serialize_default_asynchronous
   logical :: eof_reached = .false.
   integer :: unit = -1
   integer(dik) :: ring_size = 2
   integer(dik) :: action = 0
   integer(dik) :: eof_int = -1
   integer(dik) :: out_unit = output_unit
   integer(dik) :: err_unit = error_unit
   integer(dik), dimension(2) :: active_pages = [0,-1]
   integer(dik), dimension(2) :: eof_pos = [-1,-1]
   type(string_t) :: eof_string
   type(position_stack_t) :: position_stack
   character(serialize_page_size), dimension(:), allocatable::ring
 contains
 ⟨Muli base: page ring: TBP⟩
end type page_ring_t
```

These are the page_ring_t procedures, here for read access only:

⟨*Muli base: page ring: TBP*⟩≡
```
procedure :: open_for_read_access => page_ring_open_for_read_access
```

⟨*Muli base: procedures*⟩+≡
```
subroutine page_ring_open_for_read_access &
     (this, file, eof_string, asynchronous)
  class(page_ring_t), intent(inout) :: this
  character(*), intent(in) :: file, eof_string
  logical, intent(in), optional :: asynchronous
  logical :: exist
  this%eof_string = eof_string
  inquire (file=file, exist=exist)
  if (exist) then
     this%action = 2
  else
     call msg_error ("page_ring_open: File " // file // " is opened " &
          // "for read access but does not exist.")
  end if
  if (present (asynchronous)) this%asynchronous = asynchronous
  if (this%unit < 0)  call generate_unit (this%unit, 100, 1000)
  if (this%unit < 0) then
     call msg_error ("page_ring_open: No free unit found.")
  end if
  this%ring_size = 2
  call this%set_position ([i_zero,i_one])
  this%active_pages = [i_zero,-i_one]
  if (allocated (this%ring)) deallocate (this%ring)
  allocate (this%ring (i_zero:this%ring_size - i_one))
  if (this%asynchronous) then
     open (this%unit, file=file, access="stream", &
          action="read", asynchronous="yes", status="old")
  else
     open (this%unit, file=file, access="stream", action="read", &
          asynchronous="no", status="old")
  end if
  call this%read_page ()
end subroutine page_ring_open_for_read_access
```

⟨*Muli base: page ring: TBP*⟩+≡
```
procedure :: read_page => page_ring_read_page
```

⟨*Muli base: procedures*⟩+≡
```
subroutine page_ring_read_page (this)
  class(page_ring_t), intent(inout) :: this
  integer(dik) :: iostat
  character(8) :: iomsg
  if (.not. this%eof_reached) then
     call this%activate_next_page ()
     read (this%unit, iostat=iostat) this%ring (this%last_index ())
     if (iostat == iostat_end) then
        this%eof_reached = .true.
        this%eof_pos(1) = this%last_page ()
        this%eof_pos(2) = index(this%ring(this%last_index()), &
```

```
            char(this%eof_string))
         this%eof_pos(2) = this%eof_pos(2) + len(this%eof_string) - 1
         this%eof_int = page_ring_ordinal(this%eof_pos)
      end if
   end if
end subroutine page_ring_read_page
```

Those are the write access only type-bound procedures of page_ring_t:

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: open_for_write_access => page_ring_open_for_write_access
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_open_for_write_access (this, file, asynchronous)
  class(page_ring_t), intent(inout) :: this
  character(*), intent(in) :: file
  logical, intent(in), optional :: asynchronous
  this%action = 1
  if (present (asynchronous)) this%asynchronous = asynchronous
  if (this%unit < 0)  call generate_unit (this%unit, 100, 1000)
  if (this%unit < 0) then
     call msg_error ("page_ring_open: No free unit found.")
  end if
  this%ring_size = 2
  call this%set_position ([i_zero,i_one])
  this%active_pages = [i_zero,-i_one]
  if (allocated (this%ring))  deallocate (this%ring)
  allocate (this%ring (i_zero:this%ring_size-i_one))
  if (this%asynchronous) then
     open (this%unit, file=file, access="stream", action="write", &
          asynchronous="yes", status="replace")
  else
     open (this%unit, file=file, access="stream", action="write", &
          asynchronous="no",status="replace")
  end if
end subroutine page_ring_open_for_write_access
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: flush => page_ring_flush
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_flush (this)
  class(page_ring_t), intent(inout) :: this
  integer(dik) :: page
  do while (this%active_pages(1) < this%actual_page ())
     if (this%asynchronous) then
        write (this%unit, asynchronous="yes") &
             this%ring(mod(this%active_pages(1), this%ring_size))
     else
        write (this%unit, asynchronous="no") &
             this%ring(mod(this%active_pages(1), this%ring_size))
     end if
     this%active_pages(1) = this%active_pages(1) + 1
  end do
end subroutine page_ring_flush
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: break => page_ring_break
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_break(this)
  class(page_ring_t), intent(inout) :: this
  if (this%actual_page () >= this%active_pages(2)) &
       call this%activate_next_page ()
  call this%turn_page ()
end subroutine page_ring_break
```

For comparisons:

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: str_equal => page_ring_str_equal
```

⟨*Muli base: procedures*⟩+≡

```
pure logical function page_ring_str_equal (this, string, pos)
  class(page_ring_t), intent(in) :: this
  character(*), intent(in) :: string
  integer(dik), dimension(2,2), intent(in) :: pos
  page_ring_str_equal = string == this%substring (pos)
end function page_ring_str_equal
```

Routines for searching:

⟨*Muli base: page ring: TBP*⟩+≡

```
generic :: find => page_ring_find, page_ring_find_default
procedure, private :: page_ring_find
procedure, private :: page_ring_find_default
```

⟨*Muli base: procedures*⟩+≡

```
recursive subroutine page_ring_find &
     (this, exp, start, limit, skip, proceed, pos)
  class(page_ring_t), intent(inout) :: this
  integer(dik), dimension(2), intent(in) :: start
  integer(dik), dimension(2), intent(in) :: limit
  character(*), intent(in) :: exp
  integer, intent(in) :: skip
  logical, intent(in) :: proceed
  integer(dik), dimension(2), intent(out) :: pos
  integer(dik) :: page, page2, ind
  page = this%ring_index (start(1))
  if (limit(1) == start(1)) then
     ind = index(this%ring(page) (start(2):limit(2)), exp)
     if (ind > 0) then
        select case (skip)
        case (1)
           pos= [start(1), start(2)+ind-2]
           if (pos(2) == 0) then
              pos(1) = pos(1) - 1
              pos(2) = serialize_page_size
           end if
        case (2)
           pos = [start(1), start(2)+ind-1]
```

```
      case (3)
         pos = [start(1), start(2)+ind+len(exp)-2]
      case (4)
         pos = [start(1),start(2)+ind+len(exp)-1]
         if (pos(1) == this%last_page())  call this%read_page ()
         if (pos(2) > serialize_page_size) then
            pos(1) = pos(1) + 1
            pos(2) = pos(2) - serialize_page_size
         end if
      end select
      if (proceed)  call this%set_position (pos)
   else
      Call msg_warning ("page_ring_find: limit reached.")
      pos = [-1, -1]
   end if
else
   ind = index (this%ring(page) (start(2):), exp)
   if (ind > 0) then
      select case (skip)
      case (1)
         pos = [start(1), start(2)+ind-2]
         if (pos(2) == 0) then
            pos(1) = pos(1) - 1
            pos(2) = serialize_page_size
         end if
      case (2)
         pos = [start(1), start(2)+ind-1]
      case (3)
         pos = [start(1), start(2)+ind+len(exp)-2]
      case (4)
         pos = [start(1), start(2)+ind+len(exp)-1]
         if (pos(1) == this%last_page ())  call this%read_page ()
         if (pos(2) > serialize_page_size) then
            pos(1) = pos(1) + 1
            pos(2) = i_one
         end if
      end select
      if(proceed)call this%set_position(pos)
   else
      if (start(1) + 1 > this%active_pages (2)) then
         call this%read_page ()
         page = this%ring_index(start(1))
      end if
      page2 = this%ring_index(start(1)+1)
      ind = index(this%ring(page) (serialize_page_size - &
           len(exp)+1:)//this%ring(page2)(:len(exp)),exp)
      if (ind > 0) then
         select case (skip)
         case (1)
            pos = [start(1), serialize_page_size-len(exp)+ind-1]
         case (2)
            pos = [start(1), serialize_page_size-len(exp)+ind]
         case (3)
            pos = [start(1)+1, ind-1]
```

```
            case (4)
               pos = [start(1)+1, ind]
            end select
            if (pos(2) > serialize_page_size) then
               pos(1) = pos(1) + 1
               pos(2) = pos(2) - serialize_page_size
            else
               if (pos(2) < 0) then
                  pos(1) = pos(1) - 1
                  pos(2) = pos(2) + serialize_page_size
               end if
            end if
            if (proceed)  call this%set_position (pos)
         else
            if (proceed)  this%active_pages(1) = this%active_pages(2)
            call this%find (exp, [start(1) + i_one, i_one], &
                 limit, skip, proceed, pos)
         end if
      end if
   end if
end subroutine page_ring_find
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_find_default (this, exp, skip, proceed, pos)
  class(page_ring_t), intent(inout) :: this
  character(*), intent(in), optional :: exp
  integer, intent(in) :: skip
  logical, intent(in) :: proceed
  integer(dik), dimension(2), intent(out) :: pos
  call this%find (exp, this%position_stack%position, this%eof_pos, &
       skip, proceed, pos)
end subroutine page_ring_find_default
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: find_pure => page_ring_find_pure
```

⟨*Muli base: procedures*⟩+≡

```
pure recursive function page_ring_find_pure &
     (this, exp, start, limit, skip) result (pos)
  class(page_ring_t),intent(in) :: this
  integer(dik), dimension(2), intent(in) :: start
  integer(dik), dimension(2), intent(in) :: limit
  character(*),intent(in) :: exp
  integer,optional,intent(in) :: skip
  integer(dik), dimension(2) :: pos
  integer(dik) :: page, page2, ind, actual_skip
  !!! Is the starting point before limit?
  if (start(1) <= limit(1)) then
     !!! Default skip is what you expect from the build-in index function
     if (present(skip)) then
        actual_skip = skip
     else
        actual_skip = 2
     end if
```

```
page = mod(start(1), this%ring_size)
!!! Does the scanning region end on the page?
if (start(1) == limit(1)) then
   ind = index (this%ring (page) (start(2):limit(2)),exp)
else
   ind = index (this%ring (page) (start(2):),exp)
end if
if (ind > 0) then
   !!! substring found on first page
   select case (actual_skip)
   case (1)
      pos = [start(1), start(2)+ind-2]
      if (pos(2) == 0) then
         pos(1) = pos(1) - 1
         pos(2) = serialize_page_size
      end if
   case (2)
      pos= [start(1), start(2)+ind-1]
   case (3)
      pos= [start(1), start(2)+ind+len(exp)-2]
   case (4)
      pos= [start(1), start(2)+ind+len(exp)-1]
      if (pos(2) > serialize_page_size) then
         pos(1) = pos(1) + 1
         pos(2) = pos(2) - serialize_page_size
      end if
   end select
else
   !!! Substring not found on first page. Is the next page already read?
   if ((start(1) >= limit(1)) .or. &
       (start(1)+1 > this%active_pages(2))) then
      !!! Either the limit is reached or the next page is not ready.
      pos = [0, 0]
   else
      !!! The next page is available.
      page2 = mod(start(1)+1, this%ring_size)
      !!! We concatenate the edges. When l is the length of exp,
      !!! then we want to concatenate the l-1 last characters of
      !!! page one and the first l characters of page two.
      ! print *,"overlap: |",this%ring(page) &
      !    (serialize_page_size-len(exp)+2:)//this%ring(page2) &
      !    (:len(exp)),"|"
      ind = index (this%ring(page) (serialize_page_size - &
           len(exp)+2:)//this%ring(page2) (:len(exp)),exp)
      if (ind > 0) then
         select case (actual_skip)
         case (1)
            pos = [start(1), serialize_page_size-len(exp)+ind]
         case (2)
            pos = [start(1), serialize_page_size-len(exp)+ind+1]
         case (3)
            pos = [start(1)+1, ind]
         case (4)
            pos = [start(1)+1, ind+1]
```

```
            end select
         else
            !!! EXP is not found in the overlap region.
            !!! We recursively search the next pages.
            pos = this%find_pure (exp, [start(i_one) + i_one, i_one], &
                 limit, skip)
         end if
      end if
   end if
else
   !!! Limit is before start
   pos = [0, 0]
end if
end function page_ring_find_pure
```

page_ring_t routines for positioning:

⟨*Muli base: page ring: TBP*⟩+≡

```
generic :: get_position => page_ring_get_position1, page_ring_get_position2
procedure, private :: page_ring_get_position1
procedure, private :: page_ring_get_position2
```

⟨*Muli base: procedures*⟩+≡

```
pure subroutine page_ring_get_position1 (this, pos)
  class(page_ring_t), intent(in) :: this
  integer(dik), intent(out) :: pos
  pos = page_ring_ordinal (this%position_stack%position)
end subroutine page_ring_get_position1
```

⟨*Muli base: procedures*⟩+≡

```
pure subroutine page_ring_get_position2 (this, pos)
  class(page_ring_t), intent(in) :: this
  integer(dik), dimension(2), intent(out) :: pos
  pos = this%position_stack%position
end subroutine page_ring_get_position2
```

⟨*Muli base: page ring: TBP*⟩+≡

```
generic :: pop_position => pop_actual_position, pop_given_position
procedure, private :: pop_actual_position => &
     page_ring_ring_pop_actual_position
procedure, private :: pop_given_position => &
     page_ring_ring_pop_given_position
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_ring_pop_actual_position (this)
  class(page_ring_t), intent(inout) :: this
  call this%position_stack%pop ()
end subroutine page_ring_ring_pop_actual_position
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_ring_pop_given_position (this, pos)
  class(page_ring_t), intent(inout) :: this
  integer(dik), dimension(2), intent(out) :: pos
  call this%position_stack%pop (pos)
```

```
end subroutine page_ring_ring_pop_given_position
```

⟨*Muli base: page ring: TBP*⟩+≡

```
generic :: push_position => push_actual_position, push_given_position
procedure, private :: push_actual_position => &
     page_ring_ring_push_actual_position
procedure, private :: push_given_position => &
     page_ring_ring_push_given_position
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_ring_push_actual_position (this)
  class(page_ring_t), intent(inout) :: this
  call this%position_stack%push ()
end subroutine page_ring_ring_push_actual_position
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_ring_push_given_position (this, pos)
  class(page_ring_t), intent(inout) :: this
  integer(dik), dimension(2), intent(in) :: pos
  call this%position_stack%push (pos)
end subroutine page_ring_ring_push_given_position
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: set_position => page_ring_set_position
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_set_position (this, pos)
  class(page_ring_t), intent(inout) :: this
  integer(dik), dimension(2), intent(in) :: pos
  this%position_stack%position = pos
end subroutine page_ring_set_position
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: turn_page => page_ring_turn_page
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_turn_page (this)
  class(page_ring_t), intent(inout) :: this
  this%position_stack%position(1) = this%position_stack%position(1) + 1
  this%position_stack%position(2) = 1
end subroutine page_ring_turn_page
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: proceed => page_ring_proceed
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_proceed (this, n, deactivate)
  class(page_ring_t), intent(inout) :: this
  integer(dik), intent(in) :: n
  logical, intent(in), optional :: deactivate
  integer(dik) :: offset
  offset = this%position_stack%position(2) + n
  do while (offset > serialize_page_size)
```

```
      if (this%position_stack%position(1) >= this%active_pages(2)) &
           call this%activate_next_page ()
      this%position_stack%position(1) = this%position_stack%position(1) + 1
      offset = offset - serialize_page_size
   end do
   this%position_stack%position(2) = offset
   if (present (deactivate)) then
      if (deactivate)this%active_pages(1) = this%actual_page ()
   end if
end subroutine page_ring_proceed
```

These are the page_ring_t routines for printing:

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: print_to_unit => page_ring_print_to_unit
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_print_to_unit (this, unit, parents, components, peers)
  class(page_ring_t), intent(in) :: this
  integer, intent(in) :: unit
  integer(dik), intent(in) :: parents, components, peers
  write (unit, "(1x,A)")   "Components of page_ring_t: "
  write (unit, "(3x,A,A)")  "asynchronous: ", this%asynchronous
  write (unit, "(3x,A,L1)") "eof reached:  ", this%eof_reached
  write (unit, "(3x,A,I0)") "ring_size:    ", this%ring_size
  write (unit, "(3x,A,I0)") "unit:         ", this%unit
  write (unit, "(3x,A,I0)") "action:       ", this%action
  write (unit, "(3x,A,I0,I0)") &
       "position:     ", this%position_stack%position
  write (unit, "(3x,A,I0)") "active_pages: ", this%active_pages
  write (unit, "(3x,A,I0)") "file size:    ", this%eof_int
  write (unit, "(3x,A,I0,I0)")  "eof position: ", this%eof_pos
  write (unit, "(3x,A,A)")  "eof string:   ", char(this%eof_string)
  if (allocated (this%ring)) then
     write (unit, "(3x,A)") "Ring is allocated."
     if (components > 0)  call this%print_ring (unit)
  else
     write (unit, "(3x,A)")  "Ring is not allocated."
  end if
end subroutine page_ring_print_to_unit
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: print_ring => page_ring_print_ring
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_print_ring (this, unit)
  class(page_ring_t), intent(in) :: this
  integer, intent(in) :: unit
  integer(dik) :: n
  write (unit, "(1x,A)")  "Begin of page ring"
  do n = this%active_pages(1), this%active_pages(2)
     write (unit, "(3x,A,I0,A,A)") &
          "(", n, ")", this%ring (mod(n, this%ring_size))
  end do
  write (unit, "(1x,A)")  "End of page ring"
```

```
end subroutine page_ring_print_ring
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: print_position => page_ring_print_position
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_print_position(this)
  class(page_ring_t), intent(inout) :: this
  print *, this%actual_position(), &
       this%ring(this%actual_index()) (:this%actual_offset() - 1), "|", &
       this%ring(this%actual_index()) (this%actual_offset():)
end subroutine page_ring_print_position
```

Here are the page_ring_t routines for writing:

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: put => page_ring_put
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_put (this)
  class(page_ring_t), intent(inout) :: this
end subroutine page_ring_put
```

⟨*Muli base: page ring: TBP*⟩+≡

```
generic :: push => push_string, push_integer, push_integer_dik, &
     push_real, push_integer_array, push_integer_array_dik, &
     push_real_array
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure, private :: push_string => page_ring_push_string
```

⟨*Muli base: procedures*⟩+≡

```
recursive subroutine page_ring_push_string (this, string)
  class(page_ring_t), intent(inout) :: this
  character(*), intent(in) :: string
  integer(dik) :: cut, l
  l = len(string)
  if (l <= serialize_page_size-this%actual_offset()+1) then
     this%ring(this%actual_index()) &
          (this%actual_offset():this%actual_offset()+l-1)=string
     if (l == serialize_page_size-this%actual_offset()+1) then
        call this%break()
        call this%flush()
     else
        call this%proceed(l)
     end if
  else
     cut = serialize_page_size-this%actual_offset() + 1
     call this%push_string(string(:cut))
     call this%push_string(string(cut+1:))
  end if
end subroutine page_ring_push_string
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure, private :: push_integer => page_ring_push_integer
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_push_integer (this, in)
  class(page_ring_t), intent(inout) :: this
  integer, intent(in) :: in
  call this%push_integer_dik (int(in,kind=dik))
end subroutine page_ring_push_integer
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure, private :: push_integer_dik => page_ring_push_integer_dik
```

⟨*Muli base: procedures*⟩+≡

```
recursive subroutine page_ring_push_integer_dik (this, int)
  class(page_ring_t), intent(inout) :: this
  integer(dik), intent(in) :: int
  integer(dik) :: int1
  if (int < 0) then
     call this%push ("-")
     call this%push_integer_dik (-int)
  else
     if (int > 9)  call this%push (int/10)
     int1 = mod(int, 10*i_one)
     select case (int1)
     case (0)
        call this%push ("0")
     case (1)
        call this%push ("1")
     case (2)
        call this%push ("2")
     case (3)
        call this%push ("3")
     case (4)
        call this%push ("4")
     case (5)
        call this%push ("5")
     case (6)
        call this%push ("6")
     case (7)
        call this%push ("7")
     case (8)
        call this%push ("8")
     case (9)
        call this%push ("9")
     end select
  end if
end subroutine page_ring_push_integer_dik
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure, private :: push_integer_array => page_ring_push_integer_array
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_push_integer_array(this,int)
  class(page_ring_t), intent(inout) :: this
  integer, dimension(:), intent(in) :: int
  integer :: n
```

```
    do n = 1, size(int)
       call this%push (int(n))
       call this%push (" ")
    end do
  end subroutine page_ring_push_integer_array
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: push_integer_array_dik => &
       page_ring_push_integer_array_dik
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine page_ring_push_integer_array_dik(this,int)
    class(page_ring_t), intent(inout) :: this
    integer(dik), dimension(:), intent(in) :: int
    integer(dik) :: n
    do n = 1, size(int)
       call this%push (int(n))
       call this%push (" ")
    end do
  end subroutine page_ring_push_integer_array_dik
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: push_real => page_ring_push_real
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine page_ring_push_real (this, dou)
    class(page_ring_t), intent(inout) :: this
    real(default), intent(in) :: dou
    integer(dik) :: f
    ! print *,"page_ring_push_real: ",dou
    if (dou == 0D0) then
       call this%push ("0")
    else
       f = int (scale (fraction(dou), digits(dou)), kind=dik)
       call this%push (digits(dou))
       call this%push (":")
       call this%push (f)
       call this%push (":")
       call this%push (exponent(dou))
    end if
    call this%push (" ")
  end subroutine page_ring_push_real
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: push_real_array => page_ring_push_real_array
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine page_ring_push_real_array (this, dou)
    class(page_ring_t), intent(inout) :: this
    real(default), dimension(:), intent(in) :: dou
    integer(dik) :: n
    do n=1, size(dou)
       call this%push (dou(n))
    end do
```

```
end subroutine page_ring_push_real_array
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: get_character => page_ring_get_character
```

⟨*Muli base: procedures*⟩+≡

```
elemental function  page_ring_get_character (this)
  class(page_ring_t), intent(in) :: this
  character :: page_ring_get_character
  page_ring_get_character = this%ring (this%actual_index()) &
       (this%actual_offset():this%actual_offset())
end function page_ring_get_character
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: allocate_substring => page_ring_allocate_substring
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_allocate_substring (this, p1, p2, string)
  class(page_ring_t), intent(in) :: this
  integer(dik), dimension(2), intent(in) :: p1, p2
  character(:), allocatable, intent(out) :: string
  string = this%substring (p1, p2)
end subroutine page_ring_allocate_substring
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: pop_character => page_ring_pop_character
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_pop_character (this, c)
  class(page_ring_t), intent(inout) :: this
  character, intent(out) :: c
  c = this%ring (this%actual_index()) &
       (this%actual_offset():this%actual_offset())
  if (this%actual_offset () == serialize_page_size)  call this%read_page
  call this%proceed (i_one)
end subroutine page_ring_pop_character
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure :: pop_by_keys => page_ring_pop_by_keys
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_pop_by_keys (this, start, stop, inclusive, res)
  class(page_ring_t), intent(inout) :: this
  character(*), intent(in), optional :: start
  character(*), intent(in) :: stop
  logical, optional, intent(in) :: inclusive
  character(len=*), intent(out) :: res
  integer(dik), dimension(2) :: i1, i2
  if (inclusive) then
     call this%find (start, 2, .true., i1)
     call this%find (stop, 3, .false., i2)
  else
     call this%find (start, 4, .true., i1)
     call this%find (stop, 1, .false., i2)
```

```
    end if
    res = this%substring (i1, i2)
    call this%set_position (i2)
  end subroutine page_ring_pop_by_keys
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  generic :: substring => page_ring_substring1, page_ring_substring2
  procedure, private :: page_ring_substring1
  procedure, private :: page_ring_substring2
```

⟨*Muli base: procedures*⟩+≡

```
  pure function page_ring_substring1 (this, i) result (res)
    class(page_ring_t), intent(in) :: this
    integer(dik), dimension(2,2), intent(in) :: i
    character(ring_position_metric1(i)) :: res
    integer(dik) :: page, pos
    if (i(1,1) == i(1,2)) then
       res = this%ring (mod(i(1,1), this%ring_size)) (i(2,1):i(2,2))
    else
       pos = serialize_page_size - i(2,1)
       res(1:pos+1) = this%ring (mod(i(1,1),this%ring_size)) (i(2,1):)
       do page = i(1,1) + 1, i(1,1) - 1
          res (pos+2:pos+2+serialize_page_size) = &
               this%ring (mod(page,this%ring_size))
          pos = pos + serialize_page_size
       end do
       res(pos+2:pos+1+i(2,2)) = &
            this%ring (mod(page,this%ring_size)) (1:i(2,2))
    end if
  end function page_ring_substring1
```

⟨*Muli base: procedures*⟩+≡

```
  pure function page_ring_substring2 (this, i1, i2) result (res)
    class(page_ring_t), intent(in) :: this
    integer(dik), dimension(2), intent(in) :: i1,i2
    character(ring_position_metric2(i1,i2)) :: res
    integer(dik) :: page, pos
    if (i1(1) == i2(1)) then
       res = this%ring(mod(i1(1),this%ring_size)) (i1(2):i2(2))
    else
       pos = serialize_page_size - i1(2)
       res(1:pos+1) = this%ring(mod(i1(1),this%ring_size)) (i1(2):)
       do page = i1(1)+1, i2(1)-1
          res(pos+2:pos+2+serialize_page_size) = &
               this%ring(mod(page, this%ring_size))
          pos = pos + serialize_page_size
       end do
       res(pos+2:pos+1+i2(2)) = this%ring(mod(page, this%ring_size)) (1:i2(2))
    end if
  end function page_ring_substring2
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  generic :: substring_by_keys => page_ring_character_by_keys, &
```

```
        page_ring_positions_by_keys
    procedure, private :: page_ring_character_by_keys
    procedure, private :: page_ring_positions_by_keys
```

⟨*Muli base: procedures*⟩+≡

```
    pure recursive subroutine page_ring_character_by_keys (this, exp1, &
        exp2, start, limit, inclusive, length, string)
      class(page_ring_t), intent(in) :: this
      character(*), intent(in) :: exp1, exp2
      integer(dik), dimension(2), intent(in) :: start, limit
      logical, optional, intent(in) :: inclusive
      integer(dik), intent(out), optional :: length
      character(:), allocatable, intent(out) :: string
      integer(dik), dimension(2,2) :: pos
      call this%substring_by_keys (exp1, exp2, start, limit, &
          inclusive, length, pos)
      string = this%substring (pos(:,1),pos(:,2))
    end subroutine page_ring_character_by_keys
```

⟨*Muli base: procedures*⟩+≡

```
    pure recursive subroutine page_ring_positions_by_keys (this, exp1, &
        exp2, start, limit, inclusive, length, pos)
      class(page_ring_t), intent(in) :: this
      character(*), intent(in) :: exp1, exp2
      integer(dik), dimension(2), intent(in) :: start, limit
      logical, optional, intent(in) :: inclusive
      integer(dik), intent(out), optional :: length
      integer(dik), dimension(2,2), intent(out) :: pos
      if (inclusive) then
         pos(1:2,1) = this%find_pure (exp1, start, limit, 2)
      else
         pos(1:2,1) = this%find_pure (exp1,start, limit, 4)
      end if
      ! print *,pos1
      if (present(length)) then
         length = 0
      end if
      if (pos(2,1) > 0) then
         if (inclusive) then
            pos(1:2,2) = this%find_pure (exp2, pos(1:2,1), limit, 3)
         else
            pos(1:2,2) = this%find_pure (exp2, pos(1:2,1), limit, 1)
         end if
         ! print *,pos2
         if (pos(2,2) > 0) then
            if (present (length)) then
               length = ring_position_metric1 (pos)
            end if
         end if
      end if
    end subroutine page_ring_positions_by_keys
```

⟨*Muli base: page ring: TBP*⟩+≡

```
    generic :: pop => pop_string, pop_integer, pop_integer_dik, &
```

```
      pop_real, pop_logical, pop_integer_array, &
      pop_integer_array_dik, pop_real_array
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure, private :: pop_string => page_ring_pop_string
```

⟨*Muli base: procedures*⟩+≡

```
recursive subroutine page_ring_pop_string (this, res)
  class(page_ring_t), intent(inout) :: this
  character(len=*), intent(out) :: res
  integer(dik) :: n, cut
  n = len(res)
  cut = serialize_page_size-this%actual_offset() + 1
  if (n <= cut) then
     res = this%ring (this%actual_index()) &
          (this%actual_offset():this%actual_offset()+n)
     if (n == cut) then
        call this%read_page
     end if
     call this%proceed (n)
  else
     call this%pop (res(:cut))
     call this%pop (res(cut+1:))
  end if
end subroutine page_ring_pop_string
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure, private :: pop_integer => page_ring_pop_integer
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_pop_integer (this,in)
  class(page_ring_t), intent(inout) :: this
  integer, intent(out) :: in
  integer(dik) :: in_dik
  call this%pop (in_dik)
  in = int(in_dik)
end subroutine page_ring_pop_integer
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure, private :: pop_integer_dik => page_ring_pop_integer_dik
```

⟨*Muli base: procedures*⟩+≡

```
subroutine page_ring_pop_integer_dik (this, int)
  class(page_ring_t), intent(inout) :: this
  integer(dik), intent(out) :: int
  integer(dik) :: int1
  integer(dik) :: sign
  character :: c
  int = 0
  sign = 1
  c = " "
  do while (scan (c, serialize_integer_characters) == 0)
     call this%pop_character (c)
  end do
  if (c == "-") then
```

```
      sign = -1
      call this%pop_character (c)
    end if
    do while (scan (c, serialize_integer_characters) > 0)
      int = int * 10
      select case (c)
      case ("1")
        int = int + 1
      case ("2")
        int = int + 2
      case ("3")
        int = int + 3
      case ("4")
        int = int + 4
      case ("5")
        int = int + 5
      case ("6")
        int = int + 6
      case ("7")
        int = int + 7
      case ("8")
        int = int + 8
      case ("9")
        int = int + 9
      end select
      call this%pop_character (c)
    end do
    int = int * sign
    if (c == "<")  call this%proceed (-i_one)
  end subroutine page_ring_pop_integer_dik
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: pop_integer_array => page_ring_pop_integer_array
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine page_ring_pop_integer_array (this, int)
    class(page_ring_t), intent(inout) :: this
    integer, dimension(:), intent(out) :: int
    integer :: n
    do n = 1, size(int)
      call this%pop (int(n))
    end do
  end subroutine page_ring_pop_integer_array
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: pop_integer_array_dik => &
      page_ring_pop_integer_array_dik
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine page_ring_pop_integer_array_dik (this, int)
    class(page_ring_t), intent(inout) :: this
    integer(dik), dimension(:), intent(out) :: int
    integer(dik) :: n
    do n = 1, size(int)
```

```
      call this%pop (int(n))
    end do
  end subroutine page_ring_pop_integer_array_dik
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: pop_logical => page_ring_pop_logical
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine page_ring_pop_logical (this, l)
    class(page_ring_t), intent(inout) :: this
    logical, intent(out) :: l
    character(1) :: lc
    call this%pop (lc)
    do while (scan (lc,"tTfF") == 0)
      call this%pop (lc)
    end do
    read (lc, "(L1)") l
  end subroutine page_ring_pop_logical
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: pop_real => page_ring_pop_real
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine page_ring_pop_real (this, def, skip)
    class(page_ring_t), intent(inout) :: this
    real(default), intent(out) :: def
    logical, optional, intent(in) :: skip
    integer(dik) :: d, f, e
    call this%pop (d)
    if (d == i_zero) then
      def = zero
    else
      call this%pop (f)
      call this%pop (e)
      def = set_exponent (scale (real(f, kind=default), -d), e)
    end if
    if (present (skip)) then
      if (.not. skip)  call this%proceed (-i_one)
    end if
  end subroutine page_ring_pop_real
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: pop_real_array => page_ring_pop_real_array
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine page_ring_pop_real_array (this, def, skip)
    class(page_ring_t), intent(inout) :: this
    real(default), dimension(:), intent(out) :: def
    logical, optional, intent(in) :: skip
    integer(dik) :: n
    call this%pop_real (def(1))
    do n = 2, size(def)
      call this%pop_real (def(n))
    end do
```

```
    if (present(skip)) then
       if (.not. skip)  call this%proceed (-i_one)
    end if
  end subroutine page_ring_pop_real_array
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure :: close => page_ring_close
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine page_ring_close (this)
    class(page_ring_t), intent(inout) :: this
    if (this%action == 1) then
       call this%flush ()
       ! call this%print_position()
       if (this%asynchronous) then
          write (this%unit, asynchronous="yes") &
               this%ring (this%actual_index()) (:this%actual_offset() - 1)
       else
          write (this%unit, asynchronous="no") &
               this%ring (this%actual_index()) (:this%actual_offset() - 1)
       end if
    end if
    close (this%unit)
  end subroutine page_ring_close
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure :: ring_index => page_ring_ring_index
```

⟨*Muli base: procedures*⟩+≡

```
  elemental integer(dik) function page_ring_ring_index (this, n)
    class(page_ring_t), intent(in) :: this
    integer(dik), intent(in) :: n
    page_ring_ring_index = mod(n, this%ring_size)
  end function page_ring_ring_index
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: activate_next_page => page_ring_activate_next_page
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine page_ring_activate_next_page (this)
    class(page_ring_t), intent(inout) :: this
    if (this%active_pages(2) - this%active_pages(1) + 1 >= &
         this%ring_size)  call this%enlarge
    this%active_pages(2) = this%active_pages(2) + 1
  end subroutine page_ring_activate_next_page
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: enlarge => page_ring_enlarge
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine page_ring_enlarge (this)
    class(page_ring_t), intent(inout) :: this
    character(serialize_page_size), dimension(:), allocatable :: tmp_ring
    integer(dik) :: n
```

```
    call move_alloc (this%ring, tmp_ring)
    allocate (this%ring(0:this%ring_size*2-1))
    do n = this%active_pages(1), this%active_pages(2)
       this%ring (mod(n,this%ring_size*2)) = tmp_ring (mod(n,this%ring_size))
    end do
    this%ring_size = this%ring_size * 2
  end subroutine page_ring_enlarge
```

These are specific implementations of generic procedures:

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: actual_index => page_ring_actual_index
```

⟨*Muli base: procedures*⟩+≡

```
  elemental integer(dik) function page_ring_actual_index (this)
    class(page_ring_t), intent(in) :: this
    page_ring_actual_index = &
         mod (this%position_stack%position(1), this%ring_size)
  end function page_ring_actual_index
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: actual_page => page_ring_actual_page
```

⟨*Muli base: procedures*⟩+≡

```
  elemental integer(dik) function page_ring_actual_page (this)
    class(page_ring_t), intent(in) :: this
    page_ring_actual_page = this%position_stack%position(1)
  end function page_ring_actual_page
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: actual_offset => page_ring_actual_offset
```

⟨*Muli base: procedures*⟩+≡

```
  elemental integer(kind=dik) function page_ring_actual_offset(this)
    class(page_ring_t),intent(in) :: this
    page_ring_actual_offset=this%position_stack%position(2)
  end function page_ring_actual_offset
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: actual_position => page_ring_actual_position
```

⟨*Muli base: procedures*⟩+≡

```
  pure function page_ring_actual_position(this)
    class(page_ring_t), intent(in) :: this
    integer(dik), dimension(2) :: page_ring_actual_position
    page_ring_actual_position = this%position_stack%position
  end function page_ring_actual_position
```

⟨*Muli base: page ring: TBP*⟩+≡

```
  procedure, private :: first_index => page_ring_first_index
```

⟨*Muli base: procedures*⟩+≡

```
elemental integer(dik) function page_ring_first_index (this)
  class(page_ring_t), intent(in) :: this
  page_ring_first_index = mod(this%active_pages(1), this%ring_size)
end function page_ring_first_index
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure, private :: first_page => page_ring_first_page
```

⟨*Muli base: procedures*⟩+≡

```
elemental integer(dik) function page_ring_first_page (this)
  class(page_ring_t), intent(in) :: this
  page_ring_first_page = this%active_pages(1)
end function page_ring_first_page
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure, private :: last_index => page_ring_last_index
```

⟨*Muli base: procedures*⟩+≡

```
elemental integer(dik) function page_ring_last_index (this)
  class(page_ring_t), intent(in) :: this
  page_ring_last_index = mod(this%active_pages(2), this%ring_size)
end function page_ring_last_index
```

⟨*Muli base: page ring: TBP*⟩+≡

```
procedure, private :: last_page => page_ring_last_page
```

⟨*Muli base: procedures*⟩+≡

```
elemental integer(dik) function page_ring_last_page (this)
  class(page_ring_t), intent(in) :: this
  page_ring_last_page = this%active_pages(2)
end function page_ring_last_page
```

⟨*Muli base: public*⟩+≡

```
public :: marker_t
```

⟨*Muli base: types*⟩+≡

```
type, extends (page_ring_t) :: marker_t
   private
   integer(dik) :: indentation=0
   integer(dik) :: n_instances=0
   logical :: do_break=.true.
   logical :: do_indent=.false.
   class(serializable_ref_type),pointer :: heap=>null()
   class(serializable_ref_type),pointer :: references=>null()
 contains
 ⟨Muli base: marker: TBP⟩
end type marker_t
```

⟨*Muli base: marker: TBP*⟩≡

```
procedure :: mark_begin => marker_mark_begin
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_begin (this, tag, type, name, target, pointer, shape)
  class(marker_t), intent(inout) :: this
  character(*), intent(in) :: tag
  character(*), intent(in), optional :: type, name
  integer(kind=dik), intent(in), optional :: target, pointer
  integer,intent(in), dimension(:), optional :: shape
  call this%indent ()
  call this%push ("<")
  call this%push (tag)
  if (present (type))  call this%push (' type="'//type//'"')
  if (present (name))  call this%push (' name="'//name//'"')
  if (present (target)) then
     call this%push (' target="')
     call this%push (target)
     call this%push ('"')
  end if
  if (present (pointer))then
     call this%push (' pointer="')
     call this%push (pointer)
     call this%push ('"')
  end if
  if (present (shape))then
     call this%push (' shape="')
     call this%push (shape)
     call this%push ('"')
  end if
  call this%push (">")
  this%indentation = this%indentation + 1
end subroutine marker_mark_begin
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_instance_begin => marker_mark_instance_begin
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_instance_begin &
     (this, ser, name, target, pointer, shape)
  class(marker_t), intent(inout) :: this
  class(ser_class_t), intent(in) :: ser
  character(*), intent(in) :: name
  integer(dik), intent(in), optional :: target, pointer
  integer, dimension(:), intent(in), optional :: shape
  character(:), allocatable :: this_type
  call ser%get_type (this_type)
  call this%mark_begin ("ser", this_type, name, target, pointer, shape)
end subroutine marker_mark_instance_begin
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_end => marker_mark_end
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_end (this, tag)
  class(marker_t), intent(inout) :: this
  character(*), intent(in), optional :: tag
```

```
    this%indentation = this%indentation - 1
    call this%indent ()
    if (present (tag)) then
       call this%push ("</"//tag//">")
    else
       call this%push ("</ser>")
    end if
  end subroutine marker_mark_end
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: mark_instance_end => marker_mark_instance_end
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_mark_instance_end (this)
    class(marker_t), intent(inout) :: this
    call this%mark_end ("ser")
  end subroutine marker_mark_instance_end
```

⟨*Muli base: marker: TBP*⟩+≡

```
  generic :: mark => mark_logical, &
       mark_integer, mark_integer_array, mark_integer_matrix, &
       mark_integer_dik, mark_integer_array_dik, mark_integer_matrix_dik, &
       mark_default, mark_default_array, mark_default_matrix, mark_string
  procedure, private :: mark_logical => marker_mark_logical
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_mark_logical (this, name, content)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    logical, intent(in) :: content
    call this%indent ()
    call this%push ("<"//name//">")
    if (content) then
       call this%push ("T")
    else
       call this%push ("F")
    end if
    call this%push ("</"//name//">")
  end subroutine marker_mark_logical
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: mark_integer => marker_mark_integer
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_mark_integer (this, name, content)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    integer, intent(in) :: content
    call this%indent ()
    call this%push ("<"//name//">")
    call this%push (content)
    call this%push ("</"//name//">")
  end subroutine marker_mark_integer
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_integer_array => marker_mark_integer_array
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_integer_array (this, name, content)
  class(marker_t), intent(inout) :: this
  character(*), intent(in) :: name
  integer, dimension(:), intent(in) :: content
  call this%indent ()
  call this%push ("<"//name//">")
  call this%push (content)
  call this%push ("</"//name//">")
end subroutine marker_mark_integer_array
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_integer_matrix => marker_mark_integer_matrix
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_integer_matrix (this, name, content)
  class(marker_t), intent(inout) :: this
  character(*), intent(in) :: name
  integer, dimension(:,:), intent(in) :: content
  integer :: n
  integer, dimension(2) :: s
  s= shape(content)
  call this%indent ()
  call this%push ("<"//name//">")
  do n = 1, s(2)
     call this%push (content(:,n))
     call this%push (" ")
  end do
  call this%push ("</"//name//">")
end subroutine marker_mark_integer_matrix
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_integer_dik => marker_mark_integer_dik
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_integer_dik (this, name, content)
  class(marker_t), intent(inout) :: this
  character(*), intent(in) :: name
  integer(dik), intent(in) :: content
  call this%indent ()
  call this%push ("<"//name//">")
  call this%push (content)
  call this%push ("</"//name//">")
end subroutine marker_mark_integer_dik
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_integer_array_dik => marker_mark_integer_array_dik
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_integer_array_dik (this, name, content)
  class(marker_t), intent(inout) :: this
  character(*), intent(in) :: name
  integer(dik), dimension(:), intent(in) :: content
  call this%indent ()
  call this%push ("<"//name//">")
  call this%push (content)
  call this%push ("</"//name//">")
end subroutine marker_mark_integer_array_dik
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_integer_matrix_dik => marker_mark_integer_matrix_dik
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_integer_matrix_dik (this, name, content)
  class(marker_t), intent(inout) :: this
  character(*), intent(in) :: name
  integer(dik), dimension(:,:), intent(in) :: content
  integer :: n
  integer, dimension(2) :: s
  call this%indent ()
  call this%push ("<"//name//">")
  do n = 1, s(2)
     call this%push (content(:,n))
     call this%push (" ")
  end do
  call this%push ("</"//name//">")
end subroutine marker_mark_integer_matrix_dik
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_default => marker_mark_default
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_default (this, name, content)
  class(marker_t), intent(inout) :: this
  character(*), intent(in) :: name
  real(default), intent(in) :: content
  call this%indent ()
  call this%push ("<"//name//">")
  call this%push (content)
  call this%push ("</"//name//">")
end subroutine marker_mark_default
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_default_array => marker_mark_default_array
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_default_array (this, name, content)
  class(marker_t), intent(inout) :: this
  character(*), intent(in) :: name
  real(default), dimension(:), intent(in) :: content
  call this%indent ()
  call this%push ("<"//name//">")
```

```
      call this%push (content)
      call this%push ("</"//name//">")
    end subroutine marker_mark_default_array
```

⟨*Muli base: marker: TBP*⟩+≡

```
    procedure :: mark_default_matrix => marker_mark_default_matrix
```

⟨*Muli base: procedures*⟩+≡

```
    subroutine marker_mark_default_matrix (this, name, content)
      class(marker_t), intent(inout) :: this
      character(*), intent(in) :: name
      real(default), dimension(:,:), intent(in) :: content
      integer :: n
      integer, dimension(2) :: s
      s = shape(content)
      call this%indent ()
      call this%push ("<"//name//">")
      do n = 1, s(2)
         call this%push (content(:,n))
         call this%push (" ")
      end do
      call this%push ("</"//name//">")
    end subroutine marker_mark_default_matrix
```

⟨*Muli base: marker: TBP*⟩+≡

```
    procedure :: mark_string => marker_mark_string
```

⟨*Muli base: procedures*⟩+≡

```
    subroutine marker_mark_string (this, name, content)
      class(marker_t), intent(inout) :: this
      character(*), intent(in) :: name, content
      call this%indent ()
      call this%push ("<"//name//">")
      call this%push (content)
      call this%push ("</"//name//">")
    end subroutine marker_mark_string
```

⟨*Muli base: marker: TBP*⟩+≡

```
    procedure :: mark_instance => marker_mark_instance
```

⟨*Muli base: procedures*⟩+≡

```
    recursive subroutine marker_mark_instance (this, ser, name, target, pointer)
      class(marker_t), intent(inout) :: this
      class(ser_class_t), intent(in) :: ser
      character(len=*), intent(in) :: name
      integer(dik), intent(in), optional :: target, pointer
      integer(dik) :: status
      call this%mark_instance_begin (ser, name, target, pointer)
      call ser%write_to_marker (this, status)
      call this%mark_end ("ser")
    end subroutine marker_mark_instance
```

⟨*Muli base: marker: TBP*⟩+≡

```
    procedure :: mark_target => marker_mark_target
```

⟨*Muli base: procedures*⟩+≡

```
recursive subroutine marker_mark_target (this, name, ser)
  class(marker_t), intent(inout) :: this
  class(ser_class_t), target, intent(in) :: ser
  character(len=*), intent(in) :: name
  this%n_instances = this%n_instances + 1
  call this%push_heap (ser, this%n_instances)
  call this%mark_instance (ser, name, target = this%n_instances)
end subroutine marker_mark_target
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_allocatable => marker_mark_allocatable
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_allocatable (this, name, ser)
  class(marker_t), intent(inout) :: this
  class(ser_class_t), allocatable, intent(in) :: ser
  character(len=*), intent(in) :: name
  if (allocated (ser)) then
     call this%mark_instance (ser, name)
  else
     call this%mark_null (name)
  end if
end subroutine marker_mark_allocatable
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_pointer => marker_mark_pointer
```

⟨*Muli base: procedures*⟩+≡

```
recursive subroutine marker_mark_pointer (this, name, ser)
  class(marker_t), intent(inout) :: this
  class(ser_class_t), pointer, intent(in) :: ser
  character(len=*), intent(in) :: name
  character(:), allocatable :: type
  integer(dik) :: p
  if (associated (ser)) then
     call this%search_heap (ser, p)
     if (p > 0) then
        call ser%get_type (type)
        call this%push ('<ser type="')
        call this%push (type)
        call this%push ('" name="')
        call this%push (name)
        call this%push ('" pointer="')
        call this%push (p)
        call this%push ('"/>')
     else
        call this%mark_target (name, ser)
     end if
  else
     call this%mark_null (name)
  end if
end subroutine marker_mark_pointer
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_null => marker_mark_null
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_null (this, name)
  class(marker_t), intent(inout) :: this
  character(*), intent(in) :: name
  call this%indent ()
  call this%push ('<ser type="null" name="')
  call this%push (name)
  call this%push ('"/>')
end subroutine marker_mark_null
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_nothing => marker_mark_nothing
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_nothing (this, name)
  class(marker_t), intent(inout) :: this
  character(*), intent(in) :: name
  call this%indent ()
  call this%push ('<')
  call this%push (name)
  call this%push ('/>')
end subroutine marker_mark_nothing
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: mark_empty => marker_mark_empty
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_mark_empty (this, tag, type, name, target, pointer, shape)
  class(marker_t), intent(inout) :: this
  character(*), intent(in) :: tag
  character(*), intent(in), optional :: type, name
  integer(dik), intent(in), optional :: target, pointer
  integer, dimension(:), intent(in), optional :: shape
  call this%push ("<")
  call this%push (tag)
  if (present (type))  call this%push (' type="'//type//'"')
  if (present (name))  call this%push (' name="'//name//'"')
  if (present (target)) then
     call this%push (' target="')
     call this%push (target)
     call this%push ('"')
  end if
  if (present (pointer)) then
     call this%push (' pointer="')
     call this%push (pointer)
     call this%push ('"')
  end if
  if (present (shape)) then
     call this%push (' shape="')
     call this%push (shape)
     call this%push ('"')
  end if
```

```
    call this%push ("/>")
  end subroutine marker_mark_empty
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_begin => marker_pick_begin
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_begin (this, tag, type, name, target, &
      pointer, shape, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: tag
    integer(dik), dimension(2,2),intent(out),optional :: type,name
    integer(dik), intent(out), optional :: target, pointer
    integer, dimension(:), allocatable, optional, intent(out) :: shape
    integer(dik), intent(out) :: status
    integer(dik), dimension(2) :: p1, p2, p3
    integer(dik) :: l
    call this%find ("<", skip=4, proceed=.true., pos=p1)
    call this%find (">", skip=1, proceed=.false., pos=p2)
    p3 = this%find_pure (" ",p1,p2,skip=1)
    if (p3(2) > 0) then
       if (this%substring(p1, p3) == tag) then
          status = serialize_ok
          if (present (type)) then
             call this%substring_by_keys &
                  ('type="','"', p3, p2, .false., l, type)
             if (l <= 0) then
                call msg_error ("marker_pick_begin: No type found")
                status = serialize_wrong_type
             end if
          end if
          if (present (name)) then
             call this%substring_by_keys &
                  ('name="','"', p3, p2, .false., l, name)
             if (l <= 0) then
                call msg_error ("marker_pick_begin: No name found")
                status = serialize_wrong_name
                call this%print_position ()
                stop
             end if
          end if
          if (present (target)) then
             p1 = this%find_pure ('target="', p3, p2, 4)
             if (p1(2) > 0) then
                call this%set_position (p1)
                call this%pop (target)
             else
                target = -1
                status = serialize_ok
             end if
          end if
          if (present (pointer)) then
             p1=this%find_pure ('pointer="', p3, p2, 4)
             if (p1(2) > 0)then
```

```
            call this%set_position (p1)
            call this%pop (pointer)
         else
            pointer = -1
            status = serialize_ok
         end if
      end if
      if (present (shape)) then
         p1 = this%find_pure ('shape="', p3, p2, 4)
         if (p1(2) > 0) then
            call this%set_position (p1)
            call this%pop (shape)
         else
            status = serialize_ok
         end if
      end if
   else
      call msg_error ("marker_pick_begin: Wrong tag. Expected: " // &
           tag // " Found: " // this%substring(p1, p3))
      status = serialize_wrong_tag
      call this%print_position ()
   end if
else
   if (this%substring(p1, p2) == tag) then
      status = serialize_ok
   else
      call msg_error ("marker_pick_begin: Wrong tag. Expected: " // &
           tag // " Found: " // this%substring(p1, p2))
      status = serialize_wrong_tag
   end if
end if
call this%set_position (p2)
call this%proceed (i_one*2, .true.)
end subroutine marker_pick_begin
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: query_instance_begin => marker_query_instance_begin
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_query_instance_begin &
     (this, type, name, target, pointer, shape,status)
  class(marker_t), intent(inout) :: this
  integer(dik), dimension(2,2), intent(out), optional :: type, name
  integer(dik), intent(out), optional :: target, pointer
  integer, dimension(:), allocatable, intent(out), optional :: shape
  integer(dik), intent(out) :: status
  call this%pick_begin ("ser", type, name, target, pointer, shape, status)
end subroutine marker_query_instance_begin
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: pick_instance_begin => marker_pick_instance_begin
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_pick_instance_begin &
```

```
      (this, name, type, target, pointer, shape, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    integer(dik), dimension(2,2), intent(out), optional :: type
    integer(dik), intent(out), optional :: target,pointer
    integer, dimension(:), allocatable,intent(out), optional :: shape
    integer(dik), intent(out) :: status
    integer(dik), dimension(2,2) :: read_name
    call this%query_instance_begin &
         (type, read_name, target, pointer, shape, status)
    if (status == serialize_ok) then
       if (.not. this%str_equal (name, read_name)) &
            status = serialize_wrong_name
    end if
  end subroutine marker_pick_instance_begin
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_end => marker_pick_end
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_end (this, tag, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: tag
    integer(dik), intent(out) :: status
    integer(dik), dimension(2) :: p1, p2
    call this%find ("</", skip=4, proceed=.true., pos=p1)
    call this%find (">", skip=1, proceed=.false., pos=p2)
    if (tag == this%substring (p1, p2)) then
       status = serialize_ok
    else
       call msg_error ("marker_pick_end: Wrong tag. Expected: " // tag &
            // " Found: " // this%substring (p1, p2))
       ! print *,"p1=",p1,"p2=",p2
       call this%print_position ()
    end if
    call this%set_position (p2)
    call this%proceed (i_one*2, .true.)
  end subroutine marker_pick_end
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_instance_end => marker_pick_instance_end
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_instance_end (this, status)
    class(marker_t), intent(inout) :: this
    integer(dik), intent(out) :: status
    call this%pick_end ("ser",status)
  end subroutine marker_pick_instance_end
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_instance => marker_pick_instance
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_pick_instance (this, name, ser, status)
  class(marker_t), intent(inout) :: this
  class(ser_class_t), intent(out) :: ser
  character(*), intent(in) :: name
  integer(dik), intent(out) :: status
  integer(dik), dimension(2,2) :: type, r_name
  call this%pick_begin ("ser", type, r_name, status=status)
  if (status == serialize_ok) then
     if (ser%verify_type (this%substring(type))) then
        if (this%str_equal (name, r_name)) then
           call ser%read_from_marker (this, status)
           call this%pick_end ("ser", status)
        else
           call msg_error ("marker_pick_instance: Name mismatch")
           write (*,*) "Expected: ", name, " Found: ", r_name
           status = serialize_wrong_name
           call this%print_position
        end if
     else
        call msg_error ("marker_pick_instance: Type mismatch: ")
        write (*,*) type
        call ser%write_type (output_unit)
        write (*,*)
        status = serialize_wrong_type
        call this%print_position
     end if
  end if
end subroutine marker_pick_instance
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: pick_target => marker_pick_target
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_pick_target (this, name, ser, status)
  class(marker_t), intent(inout) :: this
  class(ser_class_t), target, intent(out) :: ser
  character(*), intent(in) :: name
  integer(dik), intent(out) :: status
  integer(dik), dimension(2,2) :: type, r_name
  integer(dik) :: target
  call this%pick_begin ("ser", type, r_name, target, status=status)
  if (status == serialize_ok) then
     if (ser%verify_type (this%substring(type))) then
        if (this%str_equal (name, r_name)) then
           call ser%read_target_from_marker (this, status)
           if (target > 0)  call this%push_heap (ser, target)
        else
           call msg_error ("marker_pick_instance: Name mismatch: ")
           write (*,*) "Expected: ", name, " Found: ", r_name
           status = serialize_wrong_name
        end if
     else
        call msg_error ("marker_pick_instance: Type mismatch: ")
```

```
            write (*,*)  type
            status = serialize_wrong_type
         end if
      end if
      call this%pick_end ("ser", status)
   end subroutine marker_pick_target
```

⟨*Muli base: marker: TBP*⟩+≡

```
   procedure :: pick_allocatable => marker_pick_allocatable
```

⟨*Muli base: procedures*⟩+≡

```
   subroutine marker_pick_allocatable (this, name, ser)
      class(marker_t), intent(inout) :: this
      character(*), intent(in) :: name
      class(ser_class_t), allocatable, intent(out) :: ser
      class(ser_class_t), pointer :: ref
      integer(dik),dimension(2,2) :: type, r_name
      integer(dik) :: status
      call this%pick_begin ("ser", type, r_name, status=status)
      if (status == serialize_ok) then
         if (ser%verify_type (this%substring(type))) then
            if (this%str_equal (name, r_name)) then
               call this%search_reference (type, ref)
               if (associated (ref)) then
                  allocate (ser, source=ref)
                  call ser%read_from_marker (this, status)
               else
                  call msg_error ("marker_pick_allocatable:")
                  write (*,*) "Type ", type, " not found on reference stack."
               end if
            else
               call msg_error ("marker_pick_instance: Name mismatch: ")
               write (*,*)  "Expected: ",name," Found: ",r_name
               status = serialize_wrong_name
            end if
         else
            call msg_error ("marker_pick_instance: Type mismatch: ")
            write (*,*)  type
            status = serialize_wrong_type
         end if
      end if
      call this%pick_end ("ser", status)
   end subroutine marker_pick_allocatable
```

⟨*Muli base: marker: TBP*⟩+≡

```
   procedure :: pick_pointer => marker_pick_pointer
```

⟨*Muli base: procedures*⟩+≡

```
   recursive subroutine marker_pick_pointer (this, name, ser)
      class(marker_t), intent(inout) :: this
      character(*), intent(in) :: name
      class(ser_class_t), pointer, intent(out) :: ser
      class(ser_class_t), pointer :: ref
      integer(dik), dimension(2,2) :: type, r_name
```

```
    integer(dik) :: status, t, p
    nullify (ser)
    call this%pick_begin &
         ("ser", type, r_name, target=t, pointer=p, status=status)
    if (status == serialize_ok) then
       if (.not. this%str_equal ("null",type)) then
          if (p > 0) then
             call this%search_heap (p, ser)
          else
             call this%search_reference (type, ref)
             if (associated (ref))then
                allocate (ser, source=ref)
                call ser%read_target_from_marker (this, status)
                call this%pick_end ("ser", status)
                if (t > 0)  call this%push_heap (ser, t)
             else
                write (*,*) "marker_pick_pointer:&
                     & Type ",type," not found on reference stack."
             end if
          end if
       end if
    end if
  end subroutine marker_pick_pointer
```

⟨*Muli base: marker: TBP*⟩+≡

```
  generic :: pick => pick_logical, &
       pick_integer, pick_integer_array, pick_integer_matrix, &
       pick_integer_dik, pick_integer_array_dik, pick_integer_matrix_dik, &
       pick_default, pick_default_array, pick_default_matrix, pick_string
  procedure :: pick_logical => marker_pick_logical
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_logical (this, name, content, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    logical, intent(out) :: content
    integer(dik), intent(out) :: status
    call this%pick_begin (name, status=status)
    if (status == serialize_ok) then
       call this%pop (content)
       call this%pick_end (name, status)
    end if
  end subroutine marker_pick_logical
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_integer => marker_pick_integer
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_integer (this, name, content, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    integer, intent(out) :: content
    integer(dik), intent(out) :: status
    call this%pick_begin (name, status=status)
```

```
    if (status == serialize_ok) then
       call this%pop (content)
       call this%pick_end (name, status)
    end if
  end subroutine marker_pick_integer
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_integer_array => marker_pick_integer_array
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_integer_array (this, name, content, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    integer, dimension(:), intent(out) :: content
    integer(dik), intent(out) :: status
    call this%pick_begin (name, status=status)
    if (status == serialize_ok) then
       call this%pop (content)
       call this%pick_end (name, status)
    end if
  end subroutine marker_pick_integer_array
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_integer_matrix => marker_pick_integer_matrix
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_integer_matrix (this, name, content, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    integer, dimension(:,:), intent(out) :: content
    integer(dik), intent(out) :: status
    integer :: n
    integer, dimension(2) :: s
    s = shape(content)
    call this%pick_begin (name, status=status)
    if (status == serialize_ok) then
       do n = 1, s(2)
          call this%pop (content(:,n))
       end do
       call this%pick_end (name, status)
    end if
  end subroutine marker_pick_integer_matrix
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_integer_dik => marker_pick_integer_dik
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_integer_dik (this, name, content, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    integer(dik), intent(out) :: content
    integer(dik), intent(out) :: status
    call this%pick_begin (name, status=status)
    if (status == serialize_ok) then
```

```
      call this%pop (content)
      call this%pick_end (name,status)
    end if
  end subroutine marker_pick_integer_dik
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_integer_array_dik => marker_pick_integer_array_dik
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_integer_array_dik (this, name, content, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    integer(dik), dimension(:), intent(out) :: content
    integer(dik), intent(out) :: status
    call this%pick_begin (name, status=status)
    if (status == serialize_ok) then
       call this%pop (content)
       call this%pick_end (name, status)
    end if
  end subroutine marker_pick_integer_array_dik
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_integer_matrix_dik => marker_pick_integer_matrix_dik
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_integer_matrix_dik (this, name, content, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    integer(dik), dimension(:,:), intent(out) :: content
    integer(dik), intent(out) :: status
    integer :: n
    integer, dimension(2) :: s
    s = shape(content)
    call this%pick_begin (name, status=status)
    if (status == serialize_ok) then
       do n = 1, s(2)
          call this%pop (content(:,n))
       end do
       call this%pick_end (name,status)
    end if
  end subroutine marker_pick_integer_matrix_dik
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_default => marker_pick_default
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_default (this, name, content, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    real(default), intent(out) :: content
    integer(dik), intent(out) :: status
    call this%pick_begin (name, status=status)
    if (status == serialize_ok) then
       call this%pop (content)
```

```
      call this%pick_end (name,status)
    end if
  end subroutine marker_pick_default
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_default_array => marker_pick_default_array
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_default_array (this, name, content, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    real(default), dimension(:), intent(out) :: content
    integer(dik), intent(out) :: status
    call this%pick_begin (name, status=status)
    if (status == serialize_ok) then
       call this%pop (content)
       call this%pick_end (name, status)
    end if
  end subroutine marker_pick_default_array
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_default_matrix => marker_pick_default_matrix
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_default_matrix (this, name, content, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    real(default), dimension(:,:), intent(out) :: content
    integer(dik), intent(out) :: status
    integer :: n
    integer, dimension(2) :: s
    s = shape(content)
    call this%pick_begin (name, status=status)
    if (status == serialize_ok) then
       do n = 1, s(2)
          call this%pop (content(:,n))
       end do
       call this%pick_end (name,status)
    end if
  end subroutine marker_pick_default_matrix
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: pick_string => marker_pick_string
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_pick_string (this, name, content, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    character(:), allocatable, intent(out) :: content
    integer(dik), intent(out) :: status
    call this%pick_begin (name, status=status)
    if (status == serialize_ok) then
       call this%pop (content)
       call this%pick_end (name, status)
```

```
    end if
  end subroutine marker_pick_string
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: verify_nothing => marker_verify_nothing
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_verify_nothing (this, name, status)
    class(marker_t), intent(inout) :: this
    character(*), intent(in) :: name
    integer(dik) ,intent(out) :: status
    integer(dik), dimension(2) :: p1, p2
    call this%find ("<", skip=4, proceed=.false., pos=p1)
    call this%find (">", 1, .false., p2)
    if (name//"/" == this%substring(p1, p2)) then
       status = serialize_nothing
       call this%set_position (p2)
       call this%proceed (i_one*3, .true.)
    else
       if (name == this%substring(p1, p2)) then
          status = serialize_ok
       else
          status = serialize_wrong_tag
       end if
    end if
  end subroutine marker_verify_nothing
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: indent => marker_indent
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_indent (this, step)
    class(marker_t), intent(inout) :: this
    integer(dik), optional :: step
    if (this%do_break)  call this%push (new_line(" "))
    if (this%do_indent) then
       if (present(step)) this%indentation = this%indentation + step
       call this%push (repeat(" ", this%indentation))
    end if
    this%active_pages(1) = this%actual_page()
  end subroutine marker_indent
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: push_heap => marker_push_heap
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_push_heap (this, ser, id)
    class(marker_t), intent(inout) :: this
    class(ser_class_t), target, intent(in) :: ser
    integer(dik), intent(in) :: id
    class(serializable_ref_type), pointer :: new_ref
    allocate (new_ref)
    new_ref%next => this%heap
    new_ref%ref => ser
```

```
      new_ref%id = id
      this%heap => new_ref
    end subroutine marker_push_heap
```

⟨*Muli base: marker: TBP*⟩+≡

```
    procedure :: pop_heap => marker_pop_heap
```

⟨*Muli base: procedures*⟩+≡

```
    subroutine marker_pop_heap (this, ser)
      class(marker_t), intent(inout) :: this
      class(ser_class_t), pointer, intent(out) :: ser
      class(serializable_ref_type), pointer :: old_ref
      if (associated (this%heap)) then
         old_ref => this%heap
         ser => old_ref%ref
         this%heap => this%heap%next
         deallocate (old_ref)
      else
         call msg_error ("marker_pop_heap: heap_stack is not associated.")
      end if
    end subroutine marker_pop_heap
```

⟨*Muli base: marker: TBP*⟩+≡

```
    procedure :: push_reference => marker_push_reference
```

⟨*Muli base: procedures*⟩+≡

```
    subroutine marker_push_reference (this, ser, id)
      class(marker_t), intent(inout) :: this
      class(ser_class_t), target, intent(in) :: ser
      integer(kind=dik), intent(in), optional :: id
      class(serializable_ref_type), pointer :: new_ref
      allocate (new_ref)
      new_ref%next => this%references
      new_ref%ref => ser
      if (present(id)) then
         new_ref%id = id
      else
         new_ref%id = -1
      end if
      this%references => new_ref
    end subroutine marker_push_reference
```

⟨*Muli base: marker: TBP*⟩+≡

```
    procedure :: pop_reference => marker_pop_reference
```

⟨*Muli base: procedures*⟩+≡

```
    subroutine marker_pop_reference (this, ser)
      class(marker_t), intent(inout) :: this
      class(ser_class_t), pointer, intent(out) :: ser
      class(serializable_ref_type), pointer :: old_ref
      if (associated (this%references)) then
         old_ref => this%references
         ser => old_ref%ref
         this%references => this%references%next
```

```
      deallocate (old_ref)
    else
      call msg_error &
           ("marker_pop_reference: reference_stack is not associated.")
    end if
  end subroutine marker_pop_reference
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: reset_references => marker_reset_references
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_reset_references (this)
    class(marker_t), intent(inout) :: this
    if (associated (this%references)) then
      call this%references%finalize ()
      deallocate (this%references)
    end if
  end subroutine marker_reset_references
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: search_reference => marker_search_reference
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_search_reference (this, type, ser)
    class(marker_t), intent(in) :: this
    integer(dik), dimension(2,2), intent(in) :: type
    class(ser_class_t), pointer, intent(out) :: ser
    !!! !!! !!! NAG bug workaround
    class(ser_class_t), pointer :: tmp_ser
    class(serializable_ref_type), pointer :: ref
    ref => this%references
    nullify (ser)
    do while (associated (ref))
      tmp_ser => ref%ref
      if (tmp_ser%verify_type (this%substring(type))) then
        ser => tmp_ser
        exit
      end if
      ref => ref%next
    end do
  end subroutine marker_search_reference
```

⟨*Muli base: marker: TBP*⟩+≡

```
  procedure :: reset_heap => marker_reset_heap
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine marker_reset_heap (this)
    class(marker_t), intent(inout) :: this
    if (associated (this%heap)) then
      call this%heap%finalize ()
      deallocate (this%heap)
    end if
  end subroutine marker_reset_heap
```

⟨*Muli base: marker: TBP*⟩+≡

```
procedure :: finalize => marker_finalize
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_finalize (this)
  class(marker_t), intent(inout) :: this
  call this%reset_heap ()
  call this%reset_references ()
end subroutine marker_finalize
```

⟨*Muli base: marker: TBP*⟩+≡

```
generic :: search_heap => search_heap_by_id, search_heap_by_ref
procedure :: search_heap_by_id => marker_search_heap_by_id
procedure :: search_heap_by_ref => marker_search_heap_by_ref
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_search_heap_by_ref (this, ref, id)
  class(marker_t), intent(in) :: this
  class(ser_class_t), pointer, intent(in) :: ref
  integer(dik), intent(out) :: id
  class(serializable_ref_type), pointer :: ref_p
  ref_p => this%heap
  id = 0
  do while (associated (ref_p))
     if (associated (ref, ref_p%ref)) then
        id = ref_p%id
        exit
     end if
     ref_p => ref_p%next
  end do
end subroutine marker_search_heap_by_ref
```

⟨*Muli base: procedures*⟩+≡

```
subroutine marker_search_heap_by_id (this, id, ser)
  class(marker_t), intent(in) :: this
  integer(dik), intent(in) :: id
  class(ser_class_t), pointer, intent(out) :: ser
  class(serializable_ref_type), pointer :: ref
  ref => this%heap
  do while (associated (ref))
     if (id == ref%id) then
        ser => ref%ref
        exit
     end if
     ref => ref%next
  end do
end subroutine marker_search_heap_by_id
```

⟨*Muli base: procedures*⟩+≡

```
elemental function measurable_less_measurable (mea1, mea2)
  class(measure_class_t), intent(in) :: mea1, mea2
  logical :: measurable_less_measurable
  measurable_less_measurable = mea1%measure() < mea2%measure()
```

```
    end function measurable_less_measurable
```

⟨*Muli base: procedures*⟩+≡

```
    elemental function measurable_less_default (mea1, def)
      class(measure_class_t), intent(in) :: mea1
      real(default), intent(in) :: def
      logical :: measurable_less_default
      measurable_less_default = mea1%measure() < def
    end function measurable_less_default
```

⟨*Muli base: procedures*⟩+≡

```
    elemental function measurable_less_or_equal_measurable (mea1, mea2)
      class(measure_class_t), intent(in) :: mea1, mea2
      logical :: measurable_less_or_equal_measurable
      measurable_less_or_equal_measurable = mea1%measure() <= mea2%measure()
    end function measurable_less_or_equal_measurable
```

⟨*Muli base: procedures*⟩+≡

```
    elemental function measurable_less_or_equal_default (mea1, def)
      class(measure_class_t), intent(in) :: mea1
      real(default), intent(in) :: def
      logical :: measurable_less_or_equal_default
      measurable_less_or_equal_default = mea1%measure() <= def
    end function measurable_less_or_equal_default
```

⟨*Muli base: procedures*⟩+≡

```
   elemental function measurable_equal_measurable (mea1, mea2)
      class(measure_class_t), intent(in) :: mea1, mea2
      logical :: measurable_equal_measurable
      measurable_equal_measurable = mea1%measure() == mea2%measure()
    end function measurable_equal_measurable
```

⟨*Muli base: procedures*⟩+≡

```
    elemental function measurable_equal_default (mea1, def)
      class(measure_class_t), intent(in) :: mea1
      real(default), intent(in) :: def
      logical :: measurable_equal_default
      measurable_equal_default = mea1%measure() == def
    end function measurable_equal_default
```

⟨*Muli base: procedures*⟩+≡

```
    elemental function measurable_equal_or_greater_measurable (mea1, mea2)
      class(measure_class_t), intent(in) :: mea1, mea2
      logical :: measurable_equal_or_greater_measurable
      measurable_equal_or_greater_measurable = mea1%measure() >= mea2%measure()
    end function measurable_equal_or_greater_measurable
```

⟨*Muli base: procedures*⟩+≡

```
    elemental function measurable_equal_or_greater_default (mea1, def)
      class(measure_class_t), intent(in) :: mea1
```

```
    real(default), intent(in) :: def
    logical :: measurable_equal_or_greater_default
    measurable_equal_or_greater_default = mea1%measure() >= def
  end function measurable_equal_or_greater_default
```

⟨*Muli base: procedures*⟩+≡

```
  elemental function measurable_greater_measurable (mea1, mea2)
    class(measure_class_t), intent(in) :: mea1,mea2
    logical :: measurable_greater_measurable
    measurable_greater_measurable = mea1%measure() > mea2%measure()
  end function measurable_greater_measurable
```

⟨*Muli base: procedures*⟩+≡

```
  elemental function measurable_greater_default (mea1, def)
    class(measure_class_t), intent(in) :: mea1
    real(default), intent(in) :: def
    logical :: measurable_greater_default
    measurable_greater_default = mea1%measure() > def
  end function measurable_greater_default
```

⟨*Muli base: procedures*⟩+≡

```
  pure function page_ring_position (n)
    integer(dik), intent(in) :: n
    integer(dik), dimension(2) :: page_ring_position
    page_ring_position(2) = mod(n, serialize_page_size)
    page_ring_position(1) = (n-page_ring_position(2)) / serialize_page_size
  end function page_ring_position
```

⟨*Muli base: procedures*⟩+≡

```
  pure integer(dik) function page_ring_ordinal (pos)
    integer(dik), dimension(2), intent(in) :: pos
    page_ring_ordinal = pos(1) * serialize_page_size + pos(2)
  end function page_ring_ordinal
```

⟨*Muli base: procedures*⟩+≡

```
  pure logical function page_ring_position_is_before_int_pos (m, n)
    integer(dik), intent(in) :: m
    integer(dik), dimension(2), intent(in) :: n
    if (m < page_ring_ordinal(n)) then
       page_ring_position_is_before_int_pos = .true.
    else
       page_ring_position_is_before_int_pos = .false.
    end if
  end function page_ring_position_is_before_int_pos
```

⟨*Muli base: procedures*⟩+≡

```
  pure logical function page_ring_position_is_before_pos_int (m, n)
    integer(dik), dimension(2), intent(in) :: m
    integer(dik), intent(in) :: n
    if (page_ring_ordinal(m) < n) then
       page_ring_position_is_before_pos_int = .true.
```

```
    else
       page_ring_position_is_before_pos_int = .false.
    end if
  end function page_ring_position_is_before_pos_int
```

⟨*Muli base: procedures*⟩+≡

```
  pure logical function page_ring_position_is_before_pos_pos (m, n)
    integer(dik), dimension(2), intent(in) :: m,n
    if (m(1) < n(1)) then
       page_ring_position_is_before_pos_pos = .true.
    else
       if (m(1) > n(1)) then
          page_ring_position_is_before_pos_pos = .false.
       else
          if (m(2) < n(2)) then
             page_ring_position_is_before_pos_pos = .true.
          else
             page_ring_position_is_before_pos_pos = .false.
          end if
       end if
    end if
  end function page_ring_position_is_before_pos_pos
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine ring_position_increase (pos, n)
    integer(dik), dimension(2), intent(inout) :: pos
    integer(dik), intent(in) :: n
    pos = page_ring_position (page_ring_ordinal(pos) + n)
  end subroutine ring_position_increase
```

⟨*Muli base: procedures*⟩+≡

```
  pure integer(dik) function ring_position_metric1 (p)
    integer(dik), dimension(2,2), intent(in) :: p
    ring_position_metric1 = (p(1,2) - p(1,1)) * serialize_page_size + &
         p(2,2) - p(2,1) + 1
  end function ring_position_metric1

  pure integer(dik) function ring_position_metric2 (p1, p2)
    integer(dik), dimension(2), intent(in) :: p1, p2
    ring_position_metric2 = (p2(1) - p1(1)) * &
         serialize_page_size + p2(2) - p1(2) + 1
  end function ring_position_metric2
```

⟨*Muli base: public*⟩+≡

```
  public :: generate_unit
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine generate_unit (unit, min, max)
    integer, intent(out) :: unit
    integer, intent(in), optional :: min,max
    integer :: min_u, max_u
    logical :: is_open
    ! print *,"generate_unit"
```

```
    unit = -1
    if (present (min)) then
       min_u = min
    else
       min_u = 10
    end if
    if (present (max)) then
       max_u = max
    else
       max_u = huge (max_u)
    end if
    do unit = min_u, max_u
       !print *,"testing ",unit
       inquire (unit, opened = is_open)
       if (.not. is_open) then
          exit
       end if
    end do
  end subroutine generate_unit
```

⟨*Muli base: public*⟩+≡

```
  public :: ilog2
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine ilog2 (int, exp, rem)
    integer,intent(in) :: int
    integer,intent(out) :: exp, rem
    integer :: count
    count = 2
    exp = 1
    do while (count < int)
       exp = exp + 1
       count = ishft(count, 1)
    end do
    if (count > int) then
       rem = (int - ishft(count, -1))
    else
       rem = 0
    end if
  end subroutine ilog2
```

⟨*Muli base: public*⟩+≡

```
  public :: integer_with_leading_zeros
```

⟨*Muli base: procedures*⟩+≡

```
  subroutine integer_with_leading_zeros (number, length, string)
    integer, intent(in) :: number, length
    character(len=*), intent(out) :: string
    integer :: zeros
    character::sign
    if (number == 0) then
       string = repeat("0", length)
    else
       if (number > 0) then
```

```
            zeros = length -floor(log10 (real(number))) - 1
            if (zeros < 0) then
               string = repeat("*", length)
            else
               write (string, fmt="(A,I0)")  repeat("0", zeros), number
            end if
         else
            zeros = length - floor (log10 (real (-number))) - 2
            if (zeros < 0) then
               string = repeat("*", length)
            else
               write (string, fmt="(A,A,I0)") "-", repeat("0", zeros), &
                    abs(number)
            end if
         end if
      end if
   end subroutine integer_with_leading_zeros
```

⟨*Muli base: procedures*⟩+≡

```
   pure logical function character_is_in (c, array)
      character, intent(in) :: c
      character, dimension(:), intent(in) :: array
      integer(dik) :: n
      character_is_in = .false.
      do n=1,size(array)
         if (c == array(n)) then
            character_is_in = .true.
            exit
         end if
      end do
   end function character_is_in
```

22.2 Transverse momenta

This file contains the module `muli_momentum`. Its purpose is to store the actual value of the evolution parameter p_t^2 in a convenient way. We use the normalized value $p_t^2/p_{t,\max}^2$ for generating the next value of the scale, also need the square root of both p_t^2 and $p_t^2/p_{t,\max}^2$ for other purposes. That's why I store all four combinations together with $p_{t,\max}$ in an array.

⟨`muli_momentum.f90`⟩≡

```
  ⟨File header⟩

  module muli_momentum

  ⟨Use kinds⟩
    use constants
    use muli_base

  ⟨Standard module head⟩
```

```
⟨Muli mom: public⟩

⟨Muli mom: types⟩

⟨Muli mom: interfaces⟩

contains

⟨Muli mom: procedures⟩

end module muli_momentum
```

⟨*Muli mom: public*⟩≡
```
public :: transverse_mom_t
```

⟨*Muli mom: types*⟩≡
```
type, extends (ser_class_t) :: transverse_mom_t
   private
   real(default), dimension(0:4) :: momentum = [0, 0, 0, 0, 0]
 contains
 ⟨Muli mom: trans mom: TBP⟩
end type transverse_mom_t
```

⟨*Muli mom: trans mom: TBP*⟩≡
```
procedure :: mom_write_to_marker => transverse_mom_write_to_marker
procedure :: write_to_marker => transverse_mom_write_to_marker
```

⟨*Muli mom: procedures*⟩≡
```
subroutine transverse_mom_write_to_marker (this, marker, status)
  class(transverse_mom_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%mark_begin ("transverse_mom_t")
  call marker%mark ("gev_momenta", this%momentum(0:1))
  call marker%mark_end ("transverse_mom_t")
end subroutine transverse_mom_write_to_marker
```

⟨*Muli mom: trans mom: TBP*⟩+≡
```
procedure :: mom_read_from_marker => transverse_mom_read_from_marker
procedure :: read_from_marker => transverse_mom_read_from_marker
```

⟨*Muli mom: procedures*⟩+≡
```
subroutine transverse_mom_read_from_marker (this, marker, status)
  class(transverse_mom_t), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%pick_begin ("transverse_mom_t", status=status)
  call marker%pick ("gev_momenta", this%momentum(0:1), status)
  this%momentum(2:4) = [ this%momentum(1)**2, &
                         this%momentum(1) / this%momentum(0), &
                         (this%momentum(1)/this%momentum(0))**2 ]
  call marker%pick_end ("transverse_mom_t", status=status)
end subroutine transverse_mom_read_from_marker
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure :: mom_print_to_unit => transverse_mom_print_to_unit
procedure :: print_to_unit => transverse_mom_print_to_unit
```

⟨*Muli mom: procedures*⟩+≡

```
subroutine transverse_mom_print_to_unit &
     (this, unit, parents, components, peers)
  class(transverse_mom_t), intent(in) :: this
  integer, intent(in) :: unit
  integer(dik), intent(in) :: parents, components, peers
  write (unit, "(1x,A)")  "Components of transverse_mom_t:"
  write (unit, "(3x,A)")  "Actual energy scale:"
  write (unit, "(A,E20.10)")  "Max scale (MeV)   :", this%momentum(0)
  write (unit, "(A,E20.10)")  "Scale (MeV)       :", this%momentum(1)
  write (unit, "(A,E20.10)")  "Scale^2 (MeV^2)   :", this%momentum(2)
  write (unit, "(A,E20.10)")  "Scale normalized  :", this%momentum(3)
  write (unit, "(A,E20.10)")  "Scale^2 normalized:", this%momentum(4)
end subroutine transverse_mom_print_to_unit
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure, nopass :: get_type => transverse_mom_get_type
```

⟨*Muli mom: procedures*⟩+≡

```
pure subroutine transverse_mom_get_type (type)
  character(:), allocatable, intent(out) :: type
  allocate (type, source="transverse_mom_t")
end subroutine transverse_mom_get_type
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure :: get_gev_initial_cme => transverse_mom_get_gev_initial_cme
```

⟨*Muli mom: procedures*⟩+≡

```
elemental function transverse_mom_get_gev_initial_cme (this) result(scale)
  class(transverse_mom_t), intent(in) :: this
  real(default) :: scale
  scale = this%momentum(0) * 2D0
end function transverse_mom_get_gev_initial_cme
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure :: get_gev_max_scale => transverse_mom_get_gev_max_scale
```

⟨*Muli mom: procedures*⟩+≡

```
elemental function transverse_mom_get_gev_max_scale (this) result (scale)
  class(transverse_mom_t), intent(in) :: this
  real(default) :: scale
  scale = this%momentum(0)
end function transverse_mom_get_gev_max_scale
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure :: get_gev2_max_scale => transverse_mom_get_gev2_max_scale
```

⟨*Muli mom: procedures*⟩+≡

```
elemental function transverse_mom_get_gev2_max_scale (this) result (scale)
  class(transverse_mom_t), intent(in) :: this
  real(default) :: scale
  scale = this%momentum(0)**2
end function transverse_mom_get_gev2_max_scale
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure :: get_gev_scale => transverse_mom_get_gev_scale
```

⟨*Muli mom: procedures*⟩+≡

```
elemental function transverse_mom_get_gev_scale(this) result(scale)
  class(transverse_mom_t), intent(in) :: this
  real(default) :: scale
  scale = this%momentum(1)
end function transverse_mom_get_gev_scale
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure :: get_gev2_scale => transverse_mom_get_gev2_scale
```

⟨*Muli mom: procedures*⟩+≡

```
elemental function transverse_mom_get_gev2_scale (this) result (scale)
  class(transverse_mom_t), intent(in) :: this
  real(default) :: scale
  scale = this%momentum(2)
end function transverse_mom_get_gev2_scale
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure :: get_unit_scale => transverse_mom_get_unit_scale
```

⟨*Muli mom: procedures*⟩+≡

```
pure function transverse_mom_get_unit_scale (this) result (scale)
  class(transverse_mom_t), intent(in) :: this
  real(default) :: scale
  scale = this%momentum(3)
end function transverse_mom_get_unit_scale
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure :: get_unit2_scale => transverse_mom_get_unit2_scale
```

⟨*Muli mom: procedures*⟩+≡

```
pure function transverse_mom_get_unit2_scale (this) result (scale)
  class(transverse_mom_t), intent(in) :: this
  real(default) :: scale
  scale = this%momentum(4)
end function transverse_mom_get_unit2_scale
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure :: set_gev_initial_cme => transverse_mom_set_gev_initial_cme
```

⟨*Muli mom: procedures*⟩+≡

```
subroutine transverse_mom_set_gev_initial_cme (this, new_gev_initial_cme)
  class(transverse_mom_t), intent(inout) :: this
  real(default), intent(in)  ::  new_gev_initial_cme
  this%momentum(0) = new_gev_initial_cme / 2D0
  this%momentum(3) = this%momentum(1) / this%momentum(0)
  this%momentum(4) = this%momentum(3)**2
end subroutine transverse_mom_set_gev_initial_cme
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure :: set_gev_max_scale => transverse_mom_set_gev_max_scale
```

⟨*Muli mom: procedures*⟩+≡

```
subroutine transverse_mom_set_gev_max_scale (this, new_gev_max_scale)
  class(transverse_mom_t), intent(inout) :: this
  real(default), intent(in)  ::  new_gev_max_scale
  this%momentum(0) = new_gev_max_scale
  this%momentum(3) = this%momentum(1) / this%momentum(0)
  this%momentum(4) = this%momentum(3)**2
end subroutine transverse_mom_set_gev_max_scale
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure :: set_gev2_max_scale => transverse_mom_set_gev2_max_scale
```

⟨*Muli mom: procedures*⟩+≡

```
subroutine transverse_mom_set_gev2_max_scale (this, new_gev2_max_scale)
  class(transverse_mom_t), intent(inout) :: this
  real(default), intent(in)  ::  new_gev2_max_scale
  this%momentum(0) = sqrt (new_gev2_max_scale)
  this%momentum(3) = this%momentum(1) / this%momentum(0)
  this%momentum(4) = this%momentum(3)**2
end subroutine transverse_mom_set_gev2_max_scale
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure :: set_gev_scale => transverse_mom_set_gev_scale
```

⟨*Muli mom: procedures*⟩+≡

```
subroutine transverse_mom_set_gev_scale (this, new_gev_scale)
  class(transverse_mom_t), intent(inout) :: this
  real(default), intent(in)  ::  new_gev_scale
  this%momentum(1) = new_gev_scale
  this%momentum(2) = new_gev_scale**2
  this%momentum(3) = new_gev_scale / this%momentum(0)
  this%momentum(4) = this%momentum(3)**2
end subroutine transverse_mom_set_gev_scale
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
procedure :: set_gev2_scale => transverse_mom_set_gev2_scale
```

⟨*Muli mom: procedures*⟩+≡

```
subroutine transverse_mom_set_gev2_scale (this, new_gev2_scale)
  class(transverse_mom_t), intent(inout) :: this
  real(default), intent(in) :: new_gev2_scale
  this%momentum(1) = sqrt (new_gev2_scale)
```

```
    this%momentum(2) = new_gev2_scale
    this%momentum(3) = this%momentum(1) / this%momentum(0)
    this%momentum(4) = this%momentum(3)**2
  end subroutine transverse_mom_set_gev2_scale
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
  procedure :: set_unit_scale => transverse_mom_set_unit_scale
```

⟨*Muli mom: procedures*⟩+≡

```
  subroutine transverse_mom_set_unit_scale (this, new_unit_scale)
    class(transverse_mom_t), intent(inout)::this
    real(default), intent(in) :: new_unit_scale
    this%momentum(1) = new_unit_scale * this%momentum(0)
    this%momentum(2) = this%momentum(1)**2
    this%momentum(3) = new_unit_scale
    this%momentum(4) = this%momentum(3)**2
  end subroutine transverse_mom_set_unit_scale
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
  procedure :: set_unit2_scale => transverse_mom_set_unit2_scale
```

⟨*Muli mom: procedures*⟩+≡

```
  subroutine transverse_mom_set_unit2_scale (this, new_unit2_scale)
    class(transverse_mom_t), intent(inout)::this
    real(default), intent(in) :: new_unit2_scale
    this%momentum(3) = sqrt (new_unit2_scale)
    this%momentum(4) = new_unit2_scale
    this%momentum(1) = this%momentum(3) * this%momentum(0)
    this%momentum(2) = this%momentum(1)**2
  end subroutine transverse_mom_set_unit2_scale
```

⟨*Muli mom: trans mom: TBP*⟩+≡

```
  generic :: initialize => transverse_mom_initialize
  procedure :: transverse_mom_initialize
```

⟨*Muli mom: procedures*⟩+≡

```
  subroutine transverse_mom_initialize (this, gev2_s)
    class(transverse_mom_t), intent(out) :: this
    real(default), intent(in) :: gev2_s
    real(default) :: gev_s
    gev_s = sqrt (gev2_s)
    this%momentum = [gev_s/2, gev_s/2, gev2_s/4, one, one]
  end subroutine transverse_mom_initialize
```

⟨*Muli mom: public*⟩+≡

```
  public :: qcd_2_2_class
```

⟨*Muli mom: types*⟩+≡

```
  type, extends (transverse_mom_t), abstract :: qcd_2_2_class
   contains
   ⟨Muli mom: QCD22: TBP⟩
  end type qcd_2_2_class
```

⟨*Muli mom: QCD22: TBP*⟩≡
```
procedure(qcd_get_int), deferred :: get_process_id
```
⟨*Muli mom: QCD22: TBP*⟩+≡
```
procedure(qcd_get_int), deferred :: get_integrand_id
```
⟨*Muli mom: QCD22: TBP*⟩+≡
```
procedure(qcd_get_int), deferred :: get_diagram_kind
```
⟨*Muli mom: QCD22: TBP*⟩+≡
```
procedure(qcd_get_int_4), deferred :: get_lha_flavors
```
⟨*Muli mom: QCD22: TBP*⟩+≡
```
procedure(qcd_get_int_4), deferred :: get_pdg_flavors
```
⟨*Muli mom: QCD22: TBP*⟩+≡
```
procedure(qcd_get_int_by_int), deferred :: get_parton_id
```
⟨*Muli mom: QCD22: TBP*⟩+≡
```
procedure(qcd_get_int_2), deferred :: get_parton_kinds
```
⟨*Muli mom: QCD22: TBP*⟩+≡
```
procedure(qcd_get_int_2), deferred :: get_pdf_int_kinds
```
⟨*Muli mom: QCD22: TBP*⟩+≡
```
procedure(qcd_get_real), deferred :: get_momentum_boost
```
⟨*Muli mom: QCD22: TBP*⟩+≡
```
! procedure(qcd_get_real_3),deferred :: get_parton_in_momenta
```
⟨*Muli mom: QCD22: TBP*⟩+≡
```
procedure(qcd_get_real_2), deferred :: get_remnant_momentum_fractions
```
⟨*Muli mom: QCD22: TBP*⟩+≡
```
procedure(qcd_get_real_2), deferred :: get_total_momentum_fractions
```
⟨*Muli mom: interfaces*⟩≡
```
abstract interface
   subroutine qcd_none (this)
     import qcd_2_2_class
     class(qcd_2_2_class), target, intent(in) :: this
   end subroutine qcd_none
end interface
```
⟨*Muli mom: interfaces*⟩+≡
```
! abstract interface
!    subroutine qcd_get_beam (this, beam)
!      import qcd_2_2_class
!      import pp_remnant_class
!      class(qcd_2_2_class),target, intent(in) :: this
!      class(pp_remnant_class),pointer, intent(out) :: beam
!     end subroutine qcd_get_beam
! end interface
```
⟨*Muli mom: interfaces*⟩+≡
```
abstract interface
   elemental function qcd_get_real (this)
     import
     class(qcd_2_2_class), intent(in) :: this
     real(default) :: qcd_get_real
   end function qcd_get_real
end interface
```

⟨*Muli mom: interfaces*⟩+≡

```
abstract interface
   pure function qcd_get_real_2 (this)
     import
     class(qcd_2_2_class), intent(in) :: this
     real(default), dimension(2) :: qcd_get_real_2
   end function qcd_get_real_2
end interface
```

⟨*Muli mom: interfaces*⟩+≡

```
abstract interface
   pure function qcd_get_real_3 (this)
     import
     class(qcd_2_2_class), intent(in) :: this
     real(default), dimension(3) :: qcd_get_real_3
   end function qcd_get_real_3
end interface
```

⟨*Muli mom: interfaces*⟩+≡

```
abstract interface
   elemental function qcd_get_int (this)
     import
     class(qcd_2_2_class), intent(in) :: this
     integer :: qcd_get_int
   end function qcd_get_int
end interface
```

⟨*Muli mom: interfaces*⟩+≡

```
abstract interface
   pure function qcd_get_int_by_int (this, n)
     import
     class(qcd_2_2_class), intent(in) :: this
     integer, intent(in) :: n
     integer :: qcd_get_int_by_int
   end function qcd_get_int_by_int
end interface
```

⟨*Muli mom: interfaces*⟩+≡

```
abstract interface
   pure function qcd_get_int_2 (this)
     import
     class(qcd_2_2_class), intent(in) :: this
     integer, dimension(2) :: qcd_get_int_2
   end function qcd_get_int_2
end interface
```

⟨*Muli mom: interfaces*⟩+≡

```
abstract interface
   pure function qcd_get_int_4 (this)
     import
     class(qcd_2_2_class), intent(in) :: this
     integer, dimension(4) :: qcd_get_int_4
   end function qcd_get_int_4
end interface
```

22.3 Multi parton interactions

This file contains the module `muli_interactions`. The allowed interactions and their cross sections are defined here. Additionaly, some coordinate transformations which annihilate divergencies of the cross sections are defined. Since the phase space border is hyperbolic, this transformations are also hyperbolic. That's why all interactions are named `x_cart` for cartesian or `x_hyp` for hyperbolic to avoid confusion.

⟨`muli_interactions.f90`⟩≡

```
⟨File header⟩

module muli_interactions

⟨Use kinds with double⟩
  use constants
  use muli_momentum

⟨Standard module head⟩

⟨Muli interactions: public⟩

⟨Muli interactions: variables⟩

⟨Muli interactions: interfaces⟩

contains

⟨Muli interactions: procedures⟩

end module muli_interactions
```

⟨*Muli interactions: variables*⟩≡

```
    character(len=2), dimension(-6:6), parameter :: integer_parton_names = &
         ["-6", "-5", "-4", "-3", "-2", "-1", "00", &
          "+1", "+2", "+3", "+4", "+5", "+6" ]
    character, dimension(-6:6),parameter :: traditional_parton_names = &
         ["T", "B", "C", "S", "U", "D", "g", "d", "u", "s", "c", "b", "t"]
```

These are the phase space coefficients of the polynomial mappings, the evolution variable is `pt2s/(x1*x2)`.

⟨*Muli interactions: variables*⟩+≡

```
    real(default), dimension(1:4,1:5), parameter :: &
         phase_space_coefficients_in = reshape (source = &
         [ 6144, -4608,  +384,    0, &
           6144, -5120,  +384,    0, &
           6144, -2048,  +128, -576, &
          13824, -9600, +1056,    0, &
          31104,-19872, +2160, +486 ], shape=[4,5])
```

⟨*Muli interactions: variables*⟩+≡

```
    integer, parameter :: hadron_A_kind =  2212
    integer, parameter :: hadron_B_kind = -2212
    integer, dimension(4), parameter, public :: &
```

```
     parton_kind_of_int_kind = [1, 1, 2, 2]
real(default), parameter :: b_sigma_tot_all = 100 !mb PDG
real(default), parameter :: &
     b_sigma_tot_nd = 0.5*b_sigma_tot_all !!! PRD 49 n5 1994
real(default), parameter, public :: &
     gev_cme_tot = 14000 ! total center of mass energie
real(default), parameter :: gev2_cme_tot = gev_cme_tot**2 !!! s
real(default), parameter :: gev_pt_max = gev_cme_tot/2D0
real(default), parameter :: gev2_pt_max = gev2_cme_tot/4D0
!model parameters
real(default), parameter :: gev_pt_min = 8E-1_default
real(default), parameter :: gev2_pt_min = gev_pt_min**2
real(default), parameter :: pts_min = gev_pt_min / gev_pt_max
real(default), parameter :: pts2_min = gev2_pt_min / gev2_pt_max
real(default), parameter :: gev_p_t_0 = 2.0
real(default), parameter :: gev2_p_t_0 = gev_p_t_0**2
real(default), parameter :: norm_p_t_0 = gev_p_t_0 / gev_pt_max
real(default), parameter :: norm2_p_t_0 = gev2_p_t_0 / gev2_pt_max
!mathematical constants
real(default), parameter, public :: euler = exp(one)
!physical constants
real(default), parameter :: gev2_mbarn = 0.389379304_default
real(default), parameter :: const_pref = pi * gev2_mbarn / &
     (gev2_cme_tot * b_sigma_tot_nd)
```

⟨*Muli interactions: variables*⟩+≡

```
integer, parameter, public :: LHA_FLAVOR_AT = -6
integer, parameter, public :: LHA_FLAVOR_AB = -5
integer, parameter, public :: LHA_FLAVOR_AC = -4
integer, parameter, public :: LHA_FLAVOR_AS = -3
integer, parameter, public :: LHA_FLAVOR_AU = -2
integer, parameter, public :: LHA_FLAVOR_AD = -1
integer, parameter, public :: LHA_FLAVOR_G = 0
integer, parameter, public :: LHA_FLAVOR_D = 1
integer, parameter, public :: LHA_FLAVOR_U = 2
integer, parameter, public :: LHA_FLAVOR_S = 3
integer, parameter, public :: LHA_FLAVOR_C = 4
integer, parameter, public :: LHA_FLAVOR_B = 5
integer, parameter, public :: LHA_FLAVOR_T = 6
```

⟨*Muli interactions: variables*⟩+≡

```
integer, parameter, public :: PDG_FLAVOR_AT = -6
integer, parameter, public :: PDG_FLAVOR_AB = -5
integer, parameter, public :: PDG_FLAVOR_AC = -4
integer, parameter, public :: PDG_FLAVOR_AS = -3
integer, parameter, public :: PDG_FLAVOR_AU = -2
integer, parameter, public :: PDG_FLAVOR_AD = -1
integer, parameter, public :: PDG_FLAVOR_G = 21
integer, parameter, public :: PDG_FLAVOR_D = 1
integer, parameter, public :: PDG_FLAVOR_U = 2
integer, parameter, public :: PDG_FLAVOR_S = 3
integer, parameter, public :: PDG_FLAVOR_C = 4
integer, parameter, public :: PDG_FLAVOR_B = 5
```

```
integer, parameter, public :: PDG_FLAVOR_T = 6
```

⟨*Muli interactions: variables*⟩+≡

```
integer, parameter, public :: PARTON_SEA = 1
integer, parameter, public :: PARTON_VALENCE = 2
integer, parameter, public :: PARTON_SEA_AND_VALENCE = 3
integer, parameter, public :: PARTON_TWIN = 4
integer, parameter, public :: PARTON_SEA_AND_TWIN = 5
integer, parameter, public :: PARTON_VALENCE_AND_TWIN = 6
integer, parameter, public :: PARTON_ALL = 7
```

⟨*Muli interactions: variables*⟩+≡

```
integer, parameter, public :: PDF_UNDEFINED = 0
integer, parameter, public :: PDF_GLUON = 1
integer, parameter, public :: PDF_SEA = 2
integer, parameter, public :: PDF_VALENCE_DOWN = 3
integer, parameter, public :: PDF_VALENCE_UP = 4
integer, parameter, public :: PDF_TWIN = 5
```

Evolution variable is `pt2s/(x1*x2)`.

⟨*Muli interactions: variables*⟩+≡

```
real(default), dimension(1:4,1:8),parameter :: &
     phase_space_coefficients_inout = reshape(source=[ &
     3072,  -2304,  +192,    0,  &
     6144,  -5120,  +384,    0,  &
     0,         0,   192,  -96,  &
     3072,  -2048,  +192,  -96,  &
     0,      2048, -2176, +576,  &
     0,       288,  -306,  +81,  &
     6912,  -4800,  +528,    0,  &
     31104,-23328, +5832, -486], &
     shape=[4,8])
```

⟨*Muli interactions: variables*⟩+≡

```
integer, dimension(1:4,0:8), parameter :: inout_signatures = &
     reshape (source = [ &
      1, 1, 1, 1, &   !1a
     -1, 1,-1, 1, &   !1b
      1, 1, 1, 1, &   !2
      1,-1, 1,-1, &   !3
      1,-1, 1,-1, &   !4
      1,-1, 0, 0, &   !5
      0, 0, 1,-1, &   !6
      1, 0, 1, 0, &   !7
      0, 0, 0, 0  ], &
      shape = [4,9])
```

⟨*Muli interactions: variables*⟩+≡

```
integer, dimension(6,-234:234), save, public :: valid_processes
data valid_processes (:,-234)  / -6,  -6,  -6,  -6,   2,   2 /
data valid_processes (:,-233)  / -6,  -5,  -6,  -5,   1,   1 /
```

```
data valid_processes (:,-232)  / -6,  -5,  -5,  -6,   1,   1 /
data valid_processes (:,-231)  / -6,  -4,  -6,  -4,   1,   1 /
data valid_processes (:,-230)  / -6,  -4,  -4,  -6,   1,   1 /
data valid_processes (:,-229)  / -6,  -3,  -6,  -3,   1,   1 /
data valid_processes (:,-228)  / -6,  -3,  -3,  -6,   1,   1 /
data valid_processes (:,-227)  / -6,  -2,  -6,  -2,   1,   1 /
data valid_processes (:,-226)  / -6,  -2,  -2,  -6,   1,   1 /
data valid_processes (:,-225)  / -6,  -1,  -6,  -1,   1,   1 /
data valid_processes (:,-224)  / -6,  -1,  -1,  -6,   1,   1 /
data valid_processes (:,-223)  / -6,   0,  -6,   0,   4,   7 /
data valid_processes (:,-222)  / -6,   0,   0,  -6,   4,   7 /
data valid_processes (:,-221)  / -6,   1,  -6,   1,   1,   1 /
data valid_processes (:,-220)  / -6,   1,   1,  -6,   1,   1 /
data valid_processes (:,-219)  / -6,   2,  -6,   2,   1,   1 /
data valid_processes (:,-218)  / -6,   2,   2,  -6,   1,   1 /
data valid_processes (:,-217)  / -6,   3,  -6,   3,   1,   1 /
data valid_processes (:,-216)  / -6,   3,   3,  -6,   1,   1 /
data valid_processes (:,-215)  / -6,   4,  -6,   4,   1,   1 /
data valid_processes (:,-214)  / -6,   4,   4,  -6,   1,   1 /
data valid_processes (:,-213)  / -6,   5,  -6,   5,   1,   1 /
data valid_processes (:,-212)  / -6,   5,   5,  -6,   1,   1 /
data valid_processes (:,-211)  / -6,   6,  -6,   6,   3,   4 /
data valid_processes (:,-210)  / -6,   6,  -5,   5,   3,   3 /
data valid_processes (:,-209)  / -6,   6,  -4,   4,   3,   3 /
data valid_processes (:,-208)  / -6,   6,  -3,   3,   3,   3 /
data valid_processes (:,-207)  / -6,   6,  -2,   2,   3,   3 /
data valid_processes (:,-206)  / -6,   6,  -1,   1,   3,   3 /
data valid_processes (:,-205)  / -6,   6,   0,   0,   3,   5 /
data valid_processes (:,-204)  / -6,   6,   1,  -1,   3,   3 /
data valid_processes (:,-203)  / -6,   6,   2,  -2,   3,   3 /
data valid_processes (:,-202)  / -6,   6,   3,  -3,   3,   3 /
data valid_processes (:,-201)  / -6,   6,   4,  -4,   3,   3 /
data valid_processes (:,-200)  / -6,   6,   5,  -5,   3,   3 /
data valid_processes (:,-199)  / -6,   6,   6,  -6,   3,   4 /
data valid_processes (:,-198)  / -5,  -6,  -6,  -5,   1,   1 /
data valid_processes (:,-197)  / -5,  -6,  -5,  -6,   1,   1 /
data valid_processes (:,-196)  / -5,  -5,  -5,  -5,   2,   2 /
data valid_processes (:,-195)  / -5,  -4,  -5,  -4,   1,   1 /
data valid_processes (:,-194)  / -5,  -4,  -4,  -5,   1,   1 /
data valid_processes (:,-193)  / -5,  -3,  -5,  -3,   1,   1 /
data valid_processes (:,-192)  / -5,  -3,  -3,  -5,   1,   1 /
data valid_processes (:,-191)  / -5,  -2,  -5,  -2,   1,   1 /
data valid_processes (:,-190)  / -5,  -2,  -2,  -5,   1,   1 /
data valid_processes (:,-189)  / -5,  -1,  -5,  -1,   1,   1 /
data valid_processes (:,-188)  / -5,  -1,  -1,  -5,   1,   1 /
data valid_processes (:,-187)  / -5,   0,  -5,   0,   4,   7 /
data valid_processes (:,-186)  / -5,   0,   0,  -5,   4,   7 /
data valid_processes (:,-185)  / -5,   1,  -5,   1,   1,   1 /
data valid_processes (:,-184)  / -5,   1,   1,  -5,   1,   1 /
data valid_processes (:,-183)  / -5,   2,  -5,   2,   1,   1 /
data valid_processes (:,-182)  / -5,   2,   2,  -5,   1,   1 /
data valid_processes (:,-181)  / -5,   3,  -5,   3,   1,   1 /
data valid_processes (:,-180)  / -5,   3,   3,  -5,   1,   1 /
data valid_processes (:,-179)  / -5,   4,  -5,   4,   1,   1 /
```

```
data valid_processes (:,-178)  / -5,   4,   4,  -5,   1,   1 /
data valid_processes (:,-177)  / -5,   5,  -6,   6,   3,   3 /
data valid_processes (:,-176)  / -5,   5,  -5,   5,   3,   4 /
data valid_processes (:,-175)  / -5,   5,  -4,   4,   3,   3 /
data valid_processes (:,-174)  / -5,   5,  -3,   3,   3,   3 /
data valid_processes (:,-173)  / -5,   5,  -2,   2,   3,   3 /
data valid_processes (:,-172)  / -5,   5,  -1,   1,   3,   3 /
data valid_processes (:,-171)  / -5,   5,   0,   0,   3,   5 /
data valid_processes (:,-170)  / -5,   5,   1,  -1,   3,   3 /
data valid_processes (:,-169)  / -5,   5,   2,  -2,   3,   3 /
data valid_processes (:,-168)  / -5,   5,   3,  -3,   3,   3 /
data valid_processes (:,-167)  / -5,   5,   4,  -4,   3,   3 /
data valid_processes (:,-166)  / -5,   5,   5,  -5,   3,   4 /
data valid_processes (:,-165)  / -5,   5,   6,  -6,   3,   3 /
data valid_processes (:,-164)  / -5,   6,  -5,   6,   1,   1 /
data valid_processes (:,-163)  / -5,   6,   6,  -5,   1,   1 /
data valid_processes (:,-162)  / -4,  -6,  -6,  -4,   1,   1 /
data valid_processes (:,-161)  / -4,  -6,  -4,  -6,   1,   1 /
data valid_processes (:,-160)  / -4,  -5,  -5,  -4,   1,   1 /
data valid_processes (:,-159)  / -4,  -5,  -4,  -5,   1,   1 /
data valid_processes (:,-158)  / -4,  -4,  -4,  -4,   2,   2 /
data valid_processes (:,-157)  / -4,  -3,  -4,  -3,   1,   1 /
data valid_processes (:,-156)  / -4,  -3,  -3,  -4,   1,   1 /
data valid_processes (:,-155)  / -4,  -2,  -4,  -2,   1,   1 /
data valid_processes (:,-154)  / -4,  -2,  -2,  -4,   1,   1 /
data valid_processes (:,-153)  / -4,  -1,  -4,  -1,   1,   1 /
data valid_processes (:,-152)  / -4,  -1,  -1,  -4,   1,   1 /
data valid_processes (:,-151)  / -4,   0,  -4,   0,   4,   7 /
data valid_processes (:,-150)  / -4,   0,   0,  -4,   4,   7 /
data valid_processes (:,-149)  / -4,   1,  -4,   1,   1,   1 /
data valid_processes (:,-148)  / -4,   1,   1,  -4,   1,   1 /
data valid_processes (:,-147)  / -4,   2,  -4,   2,   1,   1 /
data valid_processes (:,-146)  / -4,   2,   2,  -4,   1,   1 /
data valid_processes (:,-145)  / -4,   3,  -4,   3,   1,   1 /
data valid_processes (:,-144)  / -4,   3,   3,  -4,   1,   1 /
data valid_processes (:,-143)  / -4,   4,  -6,   6,   3,   3 /
data valid_processes (:,-142)  / -4,   4,  -5,   5,   3,   3 /
data valid_processes (:,-141)  / -4,   4,  -4,   4,   3,   4 /
data valid_processes (:,-140)  / -4,   4,  -3,   3,   3,   3 /
data valid_processes (:,-139)  / -4,   4,  -2,   2,   3,   3 /
data valid_processes (:,-138)  / -4,   4,  -1,   1,   3,   3 /
data valid_processes (:,-137)  / -4,   4,   0,   0,   3,   5 /
data valid_processes (:,-136)  / -4,   4,   1,  -1,   3,   3 /
data valid_processes (:,-135)  / -4,   4,   2,  -2,   3,   3 /
data valid_processes (:,-134)  / -4,   4,   3,  -3,   3,   3 /
data valid_processes (:,-133)  / -4,   4,   4,  -4,   3,   4 /
data valid_processes (:,-132)  / -4,   4,   5,  -5,   3,   3 /
data valid_processes (:,-131)  / -4,   4,   6,  -6,   3,   3 /
data valid_processes (:,-130)  / -4,   5,  -4,   5,   1,   1 /
data valid_processes (:,-129)  / -4,   5,   5,  -4,   1,   1 /
data valid_processes (:,-128)  / -4,   6,  -4,   6,   1,   1 /
data valid_processes (:,-127)  / -4,   6,   6,  -4,   1,   1 /
data valid_processes (:,-126)  / -3,  -6,  -6,  -3,   1,   1 /
data valid_processes (:,-125)  / -3,  -6,  -3,  -6,   1,   1 /
```

```
data valid_processes (:,-124)  / -3,  -5,  -5,  -3,   1,   1 /
data valid_processes (:,-123)  / -3,  -5,  -3,  -5,   1,   1 /
data valid_processes (:,-122)  / -3,  -4,  -4,  -3,   1,   1 /
data valid_processes (:,-121)  / -3,  -4,  -3,  -4,   1,   1 /
data valid_processes (:,-120)  / -3,  -3,  -3,  -3,   2,   2 /
data valid_processes (:,-119)  / -3,  -2,  -3,  -2,   1,   1 /
data valid_processes (:,-118)  / -3,  -2,  -2,  -3,   1,   1 /
data valid_processes (:,-117)  / -3,  -1,  -3,  -1,   1,   1 /
data valid_processes (:,-116)  / -3,  -1,  -1,  -3,   1,   1 /
data valid_processes (:,-115)  / -3,   0,  -3,   0,   4,   7 /
data valid_processes (:,-114)  / -3,   0,   0,  -3,   4,   7 /
data valid_processes (:,-113)  / -3,   1,  -3,   1,   1,   1 /
data valid_processes (:,-112)  / -3,   1,   1,  -3,   1,   1 /
data valid_processes (:,-111)  / -3,   2,  -3,   2,   1,   1 /
data valid_processes (:,-110)  / -3,   2,   2,  -3,   1,   1 /
data valid_processes (:,-109)  / -3,   3,  -6,   6,   3,   3 /
data valid_processes (:,-108)  / -3,   3,  -5,   5,   3,   3 /
data valid_processes (:,-107)  / -3,   3,  -4,   4,   3,   3 /
data valid_processes (:,-106)  / -3,   3,  -3,   3,   3,   4 /
data valid_processes (:,-105)  / -3,   3,  -2,   2,   3,   3 /
data valid_processes (:,-104)  / -3,   3,  -1,   1,   3,   3 /
data valid_processes (:,-103)  / -3,   3,   0,   0,   3,   5 /
data valid_processes (:,-102)  / -3,   3,   1,  -1,   3,   3 /
data valid_processes (:,-101)  / -3,   3,   2,  -2,   3,   3 /
data valid_processes (:,-100)  / -3,   3,   3,  -3,   3,   4 /
data valid_processes (:, -99)  / -3,   3,   4,  -4,   3,   3 /
data valid_processes (:, -98)  / -3,   3,   5,  -5,   3,   3 /
data valid_processes (:, -97)  / -3,   3,   6,  -6,   3,   3 /
data valid_processes (:, -96)  / -3,   4,  -3,   4,   1,   1 /
data valid_processes (:, -95)  / -3,   4,   4,  -3,   1,   1 /
data valid_processes (:, -94)  / -3,   5,  -3,   5,   1,   1 /
data valid_processes (:, -93)  / -3,   5,   5,  -3,   1,   1 /
data valid_processes (:, -92)  / -3,   6,  -3,   6,   1,   1 /
data valid_processes (:, -91)  / -3,   6,   6,  -3,   1,   1 /
data valid_processes (:, -90)  / -2,  -6,  -6,  -2,   1,   1 /
data valid_processes (:, -89)  / -2,  -6,  -2,  -6,   1,   1 /
data valid_processes (:, -88)  / -2,  -5,  -5,  -2,   1,   1 /
data valid_processes (:, -87)  / -2,  -5,  -2,  -5,   1,   1 /
data valid_processes (:, -86)  / -2,  -4,  -4,  -2,   1,   1 /
data valid_processes (:, -85)  / -2,  -4,  -2,  -4,   1,   1 /
data valid_processes (:, -84)  / -2,  -3,  -3,  -2,   1,   1 /
data valid_processes (:, -83)  / -2,  -3,  -2,  -3,   1,   1 /
data valid_processes (:, -82)  / -2,  -2,  -2,  -2,   2,   2 /
data valid_processes (:, -81)  / -2,  -1,  -2,  -1,   1,   1 /
data valid_processes (:, -80)  / -2,  -1,  -1,  -2,   1,   1 /
data valid_processes (:, -79)  / -2,   0,  -2,   0,   4,   7 /
data valid_processes (:, -78)  / -2,   0,   0,  -2,   4,   7 /
data valid_processes (:, -77)  / -2,   1,  -2,   1,   1,   1 /
data valid_processes (:, -76)  / -2,   1,   1,  -2,   1,   1 /
data valid_processes (:, -75)  / -2,   2,  -6,   6,   3,   3 /
data valid_processes (:, -74)  / -2,   2,  -5,   5,   3,   3 /
data valid_processes (:, -73)  / -2,   2,  -4,   4,   3,   3 /
data valid_processes (:, -72)  / -2,   2,  -3,   3,   3,   3 /
data valid_processes (:, -71)  / -2,   2,  -2,   2,   3,   4 /
```

```
data valid_processes (:, -70)  / -2,   2,  -1,   1,   3,   3 /
data valid_processes (:, -69)  / -2,   2,   0,   0,   3,   5 /
data valid_processes (:, -68)  / -2,   2,   1,  -1,   3,   3 /
data valid_processes (:, -67)  / -2,   2,   2,  -2,   3,   4 /
data valid_processes (:, -66)  / -2,   2,   3,  -3,   3,   3 /
data valid_processes (:, -65)  / -2,   2,   4,  -4,   3,   3 /
data valid_processes (:, -64)  / -2,   2,   5,  -5,   3,   3 /
data valid_processes (:, -63)  / -2,   2,   6,  -6,   3,   3 /
data valid_processes (:, -62)  / -2,   3,  -2,   3,   1,   1 /
data valid_processes (:, -61)  / -2,   3,   3,  -2,   1,   1 /
data valid_processes (:, -60)  / -2,   4,  -2,   4,   1,   1 /
data valid_processes (:, -59)  / -2,   4,   4,  -2,   1,   1 /
data valid_processes (:, -58)  / -2,   5,  -2,   5,   1,   1 /
data valid_processes (:, -57)  / -2,   5,   5,  -2,   1,   1 /
data valid_processes (:, -56)  / -2,   6,  -2,   6,   1,   1 /
data valid_processes (:, -55)  / -2,   6,   6,  -2,   1,   1 /
data valid_processes (:, -54)  / -1,  -6,  -6,  -1,   1,   1 /
data valid_processes (:, -53)  / -1,  -6,  -1,  -6,   1,   1 /
data valid_processes (:, -52)  / -1,  -5,  -5,  -1,   1,   1 /
data valid_processes (:, -51)  / -1,  -5,  -1,  -5,   1,   1 /
data valid_processes (:, -50)  / -1,  -4,  -4,  -1,   1,   1 /
data valid_processes (:, -49)  / -1,  -4,  -1,  -4,   1,   1 /
data valid_processes (:, -48)  / -1,  -3,  -3,  -1,   1,   1 /
data valid_processes (:, -47)  / -1,  -3,  -1,  -3,   1,   1 /
data valid_processes (:, -46)  / -1,  -2,  -2,  -1,   1,   1 /
data valid_processes (:, -45)  / -1,  -2,  -1,  -2,   1,   1 /
data valid_processes (:, -44)  / -1,  -1,  -1,  -1,   2,   2 /
data valid_processes (:, -43)  / -1,   0,  -1,   0,   4,   7 /
data valid_processes (:, -42)  / -1,   0,   0,  -1,   4,   7 /
data valid_processes (:, -41)  / -1,   1,  -6,   6,   3,   3 /
data valid_processes (:, -40)  / -1,   1,  -5,   5,   3,   3 /
data valid_processes (:, -39)  / -1,   1,  -4,   4,   3,   3 /
data valid_processes (:, -38)  / -1,   1,  -3,   3,   3,   3 /
data valid_processes (:, -37)  / -1,   1,  -2,   2,   3,   3 /
data valid_processes (:, -36)  / -1,   1,  -1,   1,   3,   4 /
data valid_processes (:, -35)  / -1,   1,   0,   0,   3,   5 /
data valid_processes (:, -34)  / -1,   1,   1,  -1,   3,   4 /
data valid_processes (:, -33)  / -1,   1,   2,  -2,   3,   3 /
data valid_processes (:, -32)  / -1,   1,   3,  -3,   3,   3 /
data valid_processes (:, -31)  / -1,   1,   4,  -4,   3,   3 /
data valid_processes (:, -30)  / -1,   1,   5,  -5,   3,   3 /
data valid_processes (:, -29)  / -1,   1,   6,  -6,   3,   3 /
data valid_processes (:, -28)  / -1,   2,  -1,   2,   1,   1 /
data valid_processes (:, -27)  / -1,   2,   2,  -1,   1,   1 /
data valid_processes (:, -26)  / -1,   3,  -1,   3,   1,   1 /
data valid_processes (:, -25)  / -1,   3,   3,  -1,   1,   1 /
data valid_processes (:, -24)  / -1,   4,  -1,   4,   1,   1 /
data valid_processes (:, -23)  / -1,   4,   4,  -1,   1,   1 /
data valid_processes (:, -22)  / -1,   5,  -1,   5,   1,   1 /
data valid_processes (:, -21)  / -1,   5,   5,  -1,   1,   1 /
data valid_processes (:, -20)  / -1,   6,  -1,   6,   1,   1 /
data valid_processes (:, -19)  / -1,   6,   6,  -1,   1,   1 /
data valid_processes (:, -18)  /  0,  -6,  -6,   0,   4,   7 /
data valid_processes (:, -17)  /  0,  -6,   0,  -6,   4,   7 /
```

```
data valid_processes (:, -16)  /  0,  -5,  -5,   0,   4,   7 /
data valid_processes (:, -15)  /  0,  -5,   0,  -5,   4,   7 /
data valid_processes (:, -14)  /  0,  -4,  -4,   0,   4,   7 /
data valid_processes (:, -13)  /  0,  -4,   0,  -4,   4,   7 /
data valid_processes (:, -12)  /  0,  -3,  -3,   0,   4,   7 /
data valid_processes (:, -11)  /  0,  -3,   0,  -3,   4,   7 /
data valid_processes (:, -10)  /  0,  -2,  -2,   0,   4,   7 /
data valid_processes (:,  -9)  /  0,  -2,   0,  -2,   4,   7 /
data valid_processes (:,  -8)  /  0,  -1,  -1,   0,   4,   7 /
data valid_processes (:,  -7)  /  0,  -1,   0,  -1,   4,   7 /
data valid_processes (:,  -6)  /  0,   0,  -6,   6,   5,   6 /
data valid_processes (:,  -5)  /  0,   0,  -5,   5,   5,   6 /
data valid_processes (:,  -4)  /  0,   0,  -4,   4,   5,   6 /
data valid_processes (:,  -3)  /  0,   0,  -3,   3,   5,   6 /
data valid_processes (:,  -2)  /  0,   0,  -2,   2,   5,   6 /
data valid_processes (:,  -1)  /  0,   0,  -1,   1,   5,   6 /
data valid_processes (:,   0)  /  0,   0,   0,   0,   5,   8 /
data valid_processes (:,   1)  /  0,   0,   1,  -1,   5,   6 /
data valid_processes (:,   2)  /  0,   0,   2,  -2,   5,   6 /
data valid_processes (:,   3)  /  0,   0,   3,  -3,   5,   6 /
data valid_processes (:,   4)  /  0,   0,   4,  -4,   5,   6 /
data valid_processes (:,   5)  /  0,   0,   5,  -5,   5,   6 /
data valid_processes (:,   6)  /  0,   0,   6,  -6,   5,   6 /
data valid_processes (:,   7)  /  0,   1,   0,   1,   4,   7 /
data valid_processes (:,   8)  /  0,   1,   1,   0,   4,   7 /
data valid_processes (:,   9)  /  0,   2,   0,   2,   4,   7 /
data valid_processes (:,  10)  /  0,   2,   2,   0,   4,   7 /
data valid_processes (:,  11)  /  0,   3,   0,   3,   4,   7 /
data valid_processes (:,  12)  /  0,   3,   3,   0,   4,   7 /
data valid_processes (:,  13)  /  0,   4,   0,   4,   4,   7 /
data valid_processes (:,  14)  /  0,   4,   4,   0,   4,   7 /
data valid_processes (:,  15)  /  0,   5,   0,   5,   4,   7 /
data valid_processes (:,  16)  /  0,   5,   5,   0,   4,   7 /
data valid_processes (:,  17)  /  0,   6,   0,   6,   4,   7 /
data valid_processes (:,  18)  /  0,   6,   6,   0,   4,   7 /
data valid_processes (:,  19)  /  1,  -6,  -6,   1,   1,   1 /
data valid_processes (:,  20)  /  1,  -6,   1,  -6,   1,   1 /
data valid_processes (:,  21)  /  1,  -5,  -5,   1,   1,   1 /
data valid_processes (:,  22)  /  1,  -5,   1,  -5,   1,   1 /
data valid_processes (:,  23)  /  1,  -4,  -4,   1,   1,   1 /
data valid_processes (:,  24)  /  1,  -4,   1,  -4,   1,   1 /
data valid_processes (:,  25)  /  1,  -3,  -3,   1,   1,   1 /
data valid_processes (:,  26)  /  1,  -3,   1,  -3,   1,   1 /
data valid_processes (:,  27)  /  1,  -2,  -2,   1,   1,   1 /
data valid_processes (:,  28)  /  1,  -2,   1,  -2,   1,   1 /
data valid_processes (:,  29)  /  1,  -1,  -6,   6,   3,   3 /
data valid_processes (:,  30)  /  1,  -1,  -5,   5,   3,   3 /
data valid_processes (:,  31)  /  1,  -1,  -4,   4,   3,   3 /
data valid_processes (:,  32)  /  1,  -1,  -3,   3,   3,   3 /
data valid_processes (:,  33)  /  1,  -1,  -2,   2,   3,   3 /
data valid_processes (:,  34)  /  1,  -1,  -1,   1,   3,   4 /
data valid_processes (:,  35)  /  1,  -1,   0,   0,   3,   5 /
data valid_processes (:,  36)  /  1,  -1,   1,  -1,   3,   4 /
data valid_processes (:,  37)  /  1,  -1,   2,  -2,   3,   3 /
```

```
data valid_processes (:,  38)  /  1,  -1,   3,  -3,   3,   3 /
data valid_processes (:,  39)  /  1,  -1,   4,  -4,   3,   3 /
data valid_processes (:,  40)  /  1,  -1,   5,  -5,   3,   3 /
data valid_processes (:,  41)  /  1,  -1,   6,  -6,   3,   3 /
data valid_processes (:,  42)  /  1,   0,   0,   1,   4,   7 /
data valid_processes (:,  43)  /  1,   0,   1,   0,   4,   7 /
data valid_processes (:,  44)  /  1,   1,   1,   1,   2,   2 /
data valid_processes (:,  45)  /  1,   2,   1,   2,   1,   1 /
data valid_processes (:,  46)  /  1,   2,   2,   1,   1,   1 /
data valid_processes (:,  47)  /  1,   3,   1,   3,   1,   1 /
data valid_processes (:,  48)  /  1,   3,   3,   1,   1,   1 /
data valid_processes (:,  49)  /  1,   4,   1,   4,   1,   1 /
data valid_processes (:,  50)  /  1,   4,   4,   1,   1,   1 /
data valid_processes (:,  51)  /  1,   5,   1,   5,   1,   1 /
data valid_processes (:,  52)  /  1,   5,   5,   1,   1,   1 /
data valid_processes (:,  53)  /  1,   6,   1,   6,   1,   1 /
data valid_processes (:,  54)  /  1,   6,   6,   1,   1,   1 /
data valid_processes (:,  55)  /  2,  -6,  -6,   2,   1,   1 /
data valid_processes (:,  56)  /  2,  -6,   2,  -6,   1,   1 /
data valid_processes (:,  57)  /  2,  -5,  -5,   2,   1,   1 /
data valid_processes (:,  58)  /  2,  -5,   2,  -5,   1,   1 /
data valid_processes (:,  59)  /  2,  -4,  -4,   2,   1,   1 /
data valid_processes (:,  60)  /  2,  -4,   2,  -4,   1,   1 /
data valid_processes (:,  61)  /  2,  -3,  -3,   2,   1,   1 /
data valid_processes (:,  62)  /  2,  -3,   2,  -3,   1,   1 /
data valid_processes (:,  63)  /  2,  -2,  -6,   6,   3,   3 /
data valid_processes (:,  64)  /  2,  -2,  -5,   5,   3,   3 /
data valid_processes (:,  65)  /  2,  -2,  -4,   4,   3,   3 /
data valid_processes (:,  66)  /  2,  -2,  -3,   3,   3,   3 /
data valid_processes (:,  67)  /  2,  -2,  -2,   2,   3,   4 /
data valid_processes (:,  68)  /  2,  -2,  -1,   1,   3,   3 /
data valid_processes (:,  69)  /  2,  -2,   0,   0,   3,   5 /
data valid_processes (:,  70)  /  2,  -2,   1,  -1,   3,   3 /
data valid_processes (:,  71)  /  2,  -2,   2,  -2,   3,   4 /
data valid_processes (:,  72)  /  2,  -2,   3,  -3,   3,   3 /
data valid_processes (:,  73)  /  2,  -2,   4,  -4,   3,   3 /
data valid_processes (:,  74)  /  2,  -2,   5,  -5,   3,   3 /
data valid_processes (:,  75)  /  2,  -2,   6,  -6,   3,   3 /
data valid_processes (:,  76)  /  2,  -1,  -1,   2,   1,   1 /
data valid_processes (:,  77)  /  2,  -1,   2,  -1,   1,   1 /
data valid_processes (:,  78)  /  2,   0,   0,   2,   4,   7 /
data valid_processes (:,  79)  /  2,   0,   2,   0,   4,   7 /
data valid_processes (:,  80)  /  2,   1,   1,   2,   1,   1 /
data valid_processes (:,  81)  /  2,   1,   2,   1,   1,   1 /
data valid_processes (:,  82)  /  2,   2,   2,   2,   2,   2 /
data valid_processes (:,  83)  /  2,   3,   2,   3,   1,   1 /
data valid_processes (:,  84)  /  2,   3,   3,   2,   1,   1 /
data valid_processes (:,  85)  /  2,   4,   2,   4,   1,   1 /
data valid_processes (:,  86)  /  2,   4,   4,   2,   1,   1 /
data valid_processes (:,  87)  /  2,   5,   2,   5,   1,   1 /
data valid_processes (:,  88)  /  2,   5,   5,   2,   1,   1 /
data valid_processes (:,  89)  /  2,   6,   2,   6,   1,   1 /
data valid_processes (:,  90)  /  2,   6,   6,   2,   1,   1 /
data valid_processes (:,  91)  /  3,  -6,  -6,   3,   1,   1 /
```

```
data valid_processes (:,  92)  /  3,  -6,   3,  -6,   1,   1 /
data valid_processes (:,  93)  /  3,  -5,  -5,   3,   1,   1 /
data valid_processes (:,  94)  /  3,  -5,   3,  -5,   1,   1 /
data valid_processes (:,  95)  /  3,  -4,  -4,   3,   1,   1 /
data valid_processes (:,  96)  /  3,  -4,   3,  -4,   1,   1 /
data valid_processes (:,  97)  /  3,  -3,  -6,   6,   3,   3 /
data valid_processes (:,  98)  /  3,  -3,  -5,   5,   3,   3 /
data valid_processes (:,  99)  /  3,  -3,  -4,   4,   3,   3 /
data valid_processes (:, 100)  /  3,  -3,  -3,   3,   3,   4 /
data valid_processes (:, 101)  /  3,  -3,  -2,   2,   3,   3 /
data valid_processes (:, 102)  /  3,  -3,  -1,   1,   3,   3 /
data valid_processes (:, 103)  /  3,  -3,   0,   0,   3,   5 /
data valid_processes (:, 104)  /  3,  -3,   1,  -1,   3,   3 /
data valid_processes (:, 105)  /  3,  -3,   2,  -2,   3,   3 /
data valid_processes (:, 106)  /  3,  -3,   3,  -3,   3,   4 /
data valid_processes (:, 107)  /  3,  -3,   4,  -4,   3,   3 /
data valid_processes (:, 108)  /  3,  -3,   5,  -5,   3,   3 /
data valid_processes (:, 109)  /  3,  -3,   6,  -6,   3,   3 /
data valid_processes (:, 110)  /  3,  -2,  -2,   3,   1,   1 /
data valid_processes (:, 111)  /  3,  -2,   3,  -2,   1,   1 /
data valid_processes (:, 112)  /  3,  -1,  -1,   3,   1,   1 /
data valid_processes (:, 113)  /  3,  -1,   3,  -1,   1,   1 /
data valid_processes (:, 114)  /  3,   0,   0,   3,   4,   7 /
data valid_processes (:, 115)  /  3,   0,   3,   0,   4,   7 /
data valid_processes (:, 116)  /  3,   1,   1,   3,   1,   1 /
data valid_processes (:, 117)  /  3,   1,   3,   1,   1,   1 /
data valid_processes (:, 118)  /  3,   2,   2,   3,   1,   1 /
data valid_processes (:, 119)  /  3,   2,   3,   2,   1,   1 /
data valid_processes (:, 120)  /  3,   3,   3,   3,   2,   2 /
data valid_processes (:, 121)  /  3,   4,   3,   4,   1,   1 /
data valid_processes (:, 122)  /  3,   4,   4,   3,   1,   1 /
data valid_processes (:, 123)  /  3,   5,   3,   5,   1,   1 /
data valid_processes (:, 124)  /  3,   5,   5,   3,   1,   1 /
data valid_processes (:, 125)  /  3,   6,   3,   6,   1,   1 /
data valid_processes (:, 126)  /  3,   6,   6,   3,   1,   1 /
data valid_processes (:, 127)  /  4,  -6,  -6,   4,   1,   1 /
data valid_processes (:, 128)  /  4,  -6,   4,  -6,   1,   1 /
data valid_processes (:, 129)  /  4,  -5,  -5,   4,   1,   1 /
data valid_processes (:, 130)  /  4,  -5,   4,  -5,   1,   1 /
data valid_processes (:, 131)  /  4,  -4,  -6,   6,   3,   3 /
data valid_processes (:, 132)  /  4,  -4,  -5,   5,   3,   3 /
data valid_processes (:, 133)  /  4,  -4,  -4,   4,   3,   4 /
data valid_processes (:, 134)  /  4,  -4,  -3,   3,   3,   3 /
data valid_processes (:, 135)  /  4,  -4,  -2,   2,   3,   3 /
data valid_processes (:, 136)  /  4,  -4,  -1,   1,   3,   3 /
data valid_processes (:, 137)  /  4,  -4,   0,   0,   3,   5 /
data valid_processes (:, 138)  /  4,  -4,   1,  -1,   3,   3 /
data valid_processes (:, 139)  /  4,  -4,   2,  -2,   3,   3 /
data valid_processes (:, 140)  /  4,  -4,   3,  -3,   3,   3 /
data valid_processes (:, 141)  /  4,  -4,   4,  -4,   3,   4 /
data valid_processes (:, 142)  /  4,  -4,   5,  -5,   3,   3 /
data valid_processes (:, 143)  /  4,  -4,   6,  -6,   3,   3 /
data valid_processes (:, 144)  /  4,  -3,  -3,   4,   1,   1 /
data valid_processes (:, 145)  /  4,  -3,   4,  -3,   1,   1 /
```

```
data valid_processes (:, 146)  /  4,  -2,  -2,   4,   1,   1 /
data valid_processes (:, 147)  /  4,  -2,   4,  -2,   1,   1 /
data valid_processes (:, 148)  /  4,  -1,  -1,   4,   1,   1 /
data valid_processes (:, 149)  /  4,  -1,   4,  -1,   1,   1 /
data valid_processes (:, 150)  /  4,   0,   0,   4,   4,   7 /
data valid_processes (:, 151)  /  4,   0,   4,   0,   4,   7 /
data valid_processes (:, 152)  /  4,   1,   1,   4,   1,   1 /
data valid_processes (:, 153)  /  4,   1,   4,   1,   1,   1 /
data valid_processes (:, 154)  /  4,   2,   2,   4,   1,   1 /
data valid_processes (:, 155)  /  4,   2,   4,   2,   1,   1 /
data valid_processes (:, 156)  /  4,   3,   3,   4,   1,   1 /
data valid_processes (:, 157)  /  4,   3,   4,   3,   1,   1 /
data valid_processes (:, 158)  /  4,   4,   4,   4,   2,   2 /
data valid_processes (:, 159)  /  4,   5,   4,   5,   1,   1 /
data valid_processes (:, 160)  /  4,   5,   5,   4,   1,   1 /
data valid_processes (:, 161)  /  4,   6,   4,   6,   1,   1 /
data valid_processes (:, 162)  /  4,   6,   6,   4,   1,   1 /
data valid_processes (:, 163)  /  5,  -6,  -6,   5,   1,   1 /
data valid_processes (:, 164)  /  5,  -6,   5,  -6,   1,   1 /
data valid_processes (:, 165)  /  5,  -5,  -6,   6,   3,   3 /
data valid_processes (:, 166)  /  5,  -5,  -5,   5,   3,   4 /
data valid_processes (:, 167)  /  5,  -5,  -4,   4,   3,   3 /
data valid_processes (:, 168)  /  5,  -5,  -3,   3,   3,   3 /
data valid_processes (:, 169)  /  5,  -5,  -2,   2,   3,   3 /
data valid_processes (:, 170)  /  5,  -5,  -1,   1,   3,   3 /
data valid_processes (:, 171)  /  5,  -5,   0,   0,   3,   5 /
data valid_processes (:, 172)  /  5,  -5,   1,  -1,   3,   3 /
data valid_processes (:, 173)  /  5,  -5,   2,  -2,   3,   3 /
data valid_processes (:, 174)  /  5,  -5,   3,  -3,   3,   3 /
data valid_processes (:, 175)  /  5,  -5,   4,  -4,   3,   3 /
data valid_processes (:, 176)  /  5,  -5,   5,  -5,   3,   4 /
data valid_processes (:, 177)  /  5,  -5,   6,  -6,   3,   3 /
data valid_processes (:, 178)  /  5,  -4,  -4,   5,   1,   1 /
data valid_processes (:, 179)  /  5,  -4,   5,  -4,   1,   1 /
data valid_processes (:, 180)  /  5,  -3,  -3,   5,   1,   1 /
data valid_processes (:, 181)  /  5,  -3,   5,  -3,   1,   1 /
data valid_processes (:, 182)  /  5,  -2,  -2,   5,   1,   1 /
data valid_processes (:, 183)  /  5,  -2,   5,  -2,   1,   1 /
data valid_processes (:, 184)  /  5,  -1,  -1,   5,   1,   1 /
data valid_processes (:, 185)  /  5,  -1,   5,  -1,   1,   1 /
data valid_processes (:, 186)  /  5,   0,   0,   5,   4,   7 /
data valid_processes (:, 187)  /  5,   0,   5,   0,   4,   7 /
data valid_processes (:, 188)  /  5,   1,   1,   5,   1,   1 /
data valid_processes (:, 189)  /  5,   1,   5,   1,   1,   1 /
data valid_processes (:, 190)  /  5,   2,   2,   5,   1,   1 /
data valid_processes (:, 191)  /  5,   2,   5,   2,   1,   1 /
data valid_processes (:, 192)  /  5,   3,   3,   5,   1,   1 /
data valid_processes (:, 193)  /  5,   3,   5,   3,   1,   1 /
data valid_processes (:, 194)  /  5,   4,   4,   5,   1,   1 /
data valid_processes (:, 195)  /  5,   4,   5,   4,   1,   1 /
data valid_processes (:, 196)  /  5,   5,   5,   5,   2,   2 /
data valid_processes (:, 197)  /  5,   6,   5,   6,   1,   1 /
data valid_processes (:, 198)  /  5,   6,   6,   5,   1,   1 /
data valid_processes (:, 199)  /  6,  -6,  -6,   6,   3,   4 /
```

```
data valid_processes (:, 200)  /  6,  -6,  -5,   5,   3,   3 /
data valid_processes (:, 201)  /  6,  -6,  -4,   4,   3,   3 /
data valid_processes (:, 202)  /  6,  -6,  -3,   3,   3,   3 /
data valid_processes (:, 203)  /  6,  -6,  -2,   2,   3,   3 /
data valid_processes (:, 204)  /  6,  -6,  -1,   1,   3,   3 /
data valid_processes (:, 205)  /  6,  -6,   0,   0,   3,   5 /
data valid_processes (:, 206)  /  6,  -6,   1,  -1,   3,   3 /
data valid_processes (:, 207)  /  6,  -6,   2,  -2,   3,   3 /
data valid_processes (:, 208)  /  6,  -6,   3,  -3,   3,   3 /
data valid_processes (:, 209)  /  6,  -6,   4,  -4,   3,   3 /
data valid_processes (:, 210)  /  6,  -6,   5,  -5,   3,   3 /
data valid_processes (:, 211)  /  6,  -6,   6,  -6,   3,   4 /
data valid_processes (:, 212)  /  6,  -5,  -5,   6,   1,   1 /
data valid_processes (:, 213)  /  6,  -5,   6,  -5,   1,   1 /
data valid_processes (:, 214)  /  6,  -4,  -4,   6,   1,   1 /
data valid_processes (:, 215)  /  6,  -4,   6,  -4,   1,   1 /
data valid_processes (:, 216)  /  6,  -3,  -3,   6,   1,   1 /
data valid_processes (:, 217)  /  6,  -3,   6,  -3,   1,   1 /
data valid_processes (:, 218)  /  6,  -2,  -2,   6,   1,   1 /
data valid_processes (:, 219)  /  6,  -2,   6,  -2,   1,   1 /
data valid_processes (:, 220)  /  6,  -1,  -1,   6,   1,   1 /
data valid_processes (:, 221)  /  6,  -1,   6,  -1,   1,   1 /
data valid_processes (:, 222)  /  6,   0,   0,   6,   4,   7 /
data valid_processes (:, 223)  /  6,   0,   6,   0,   4,   7 /
data valid_processes (:, 224)  /  6,   1,   1,   6,   1,   1 /
data valid_processes (:, 225)  /  6,   1,   6,   1,   1,   1 /
data valid_processes (:, 226)  /  6,   2,   2,   6,   1,   1 /
data valid_processes (:, 227)  /  6,   2,   6,   2,   1,   1 /
data valid_processes (:, 228)  /  6,   3,   3,   6,   1,   1 /
data valid_processes (:, 229)  /  6,   3,   6,   3,   1,   1 /
data valid_processes (:, 230)  /  6,   4,   4,   6,   1,   1 /
data valid_processes (:, 231)  /  6,   4,   6,   4,   1,   1 /
data valid_processes (:, 232)  /  6,   5,   5,   6,   1,   1 /
data valid_processes (:, 233)  /  6,   5,   6,   5,   1,   1 /
data valid_processes (:, 234)  /  6,   6,   6,   6,   2,   2 /
```

⟨*Muli interactions: variables*⟩+≡

```
integer, dimension(2,0:16), parameter, public :: &
     double_pdf_kinds = reshape ( [ &
     0, 0, &
     1, 1, &
     1, 2, &
     1, 3, &
     1, 4, &
     2, 1, &
     2, 2, &
     2, 3, &
     2, 4, &
     3, 1, &
     3, 2, &
     3, 3, &
     3, 4, &
     4, 1, &
     4, 2, &
```

```
        4, 3, &
        4, 4], [2, 17])
```

⟨*Muli interactions: variables*⟩+≡

```
    integer, parameter, dimension(371), public :: int_all = [ &
           -6,  -5,  -4,  -3,  -2,  -1,   0,  1 ,   2, &
            3,   4,   5,   6, -14, -13, -12, -11, -10, &
            9,  -8,  -7,   7,   8,   9,  10,  11,  12, &
           13,  14,   7,   8,   9,  10,-151,-150,-115, &
         -114, -79, -78, -43, -42,  42,  43,  78,  79, &
          114, 115, 150, 151,-158,-157,-156,-155,-154, &
         -153,-152,-149,-148,-147,-146,-145,-144,-143, &
         -142,-141,-140,-139,-138,-137,-136,-135,-134, &
         -133,-132,-131,-122,-121,-120,-119,-118,-117, &
         -116,-113,-112,-111,-110,-109,-108,-107,-106, &
         -105,-104,-103,-102,-101,-100, -99, -98, -97, &
          -96, -95, -86, -85, -84, -83, -82, -81, -80, &
          -77, -76, -75, -74, -73, -72, -71, -70, -69, &
          -68, -67, -66, -65, -64, -63, -62, -61, -60, &
          -59, -50, -49, -48, -47, -46, -45, -44, -41, &
          -40, -39, -38, -37, -36, -35, -34, -33, -32, &
          -31, -30, -29, -28, -27, -26, -25, -24, -23, &
           23,  24,  25,  26,  27,  28,  29,  30,  31, &
           32,  33,  34,  35,  36,  37,  38,  39,  40, &
           41,  44,  45,  46,  47,  48,  49,  50,  59, &
           60,  61,  62,  63,  64,  65,  66,  67,  68, &
           69,  70,  71,  72,  73,  74,  75,  76,  77, &
           80,  81,  82,  83,  84,  85,  86,  95,  96, &
           97,  98,  99, 100, 101, 102, 103, 104, 105, &
          106, 107, 108, 109, 110, 111, 112, 113, 116, &
          117, 118, 119, 120, 121, 122, 131, 132, 133, &
          134, 135, 136, 137, 138, 139, 140, 141, 142, &
          143, 144, 145, 146, 147, 148, 149, 152, 153, &
          154, 155, 156, 157, 158,-149,-148,-113,-112, &
          -77, -76, -41, -40, -39, -38, -37, -36, -35, &
          -34, -33, -32, -31, -30, -29,  44,  80,  81, &
          116, 117, 152, 153,-147,-146,-111,-110, -75, &
          -74, -73, -72, -71, -70, -69, -68, -67, -66, &
          -65, -64, -63, -28, -27,  45,  46,  82, 118, &
          119, 154, 155,  42,  43,  23,  24,  25,  26, &
           27,  28,  29,  30,  31,  32,  33,  34,  35, &
           36,  37,  38,  39,  40,  41,  44,  45,  46, &
           47,  48,  49,  50,  44,  45,  46,  78,  79, &
           59,  60,  61,  62,  63,  64,  65,  66,  67, &
           68,  69,  70,  71,  72,  73,  74,  75,  76, &
           77,  80,  81,  82,  83,  84,  85,  86,  80, &
           81,  82 ]
```

⟨*Muli interactions: variables*⟩+≡

```
    integer, parameter, dimension(16), public :: int_sizes_all = &
        [13, 16, 2, 2, 16, 208, 26, 26, 2, 26, 1, 2, 2, 26, 2, 1]
```

⟨*Muli interactions: variables*⟩+≡

```
integer, parameter, dimension(3,0:8), public :: muli_flow_stats = &
     reshape( [ &
         1,  2,  4,   &
         3,  4,  4,   &
         5,  6,  8,   &
         7,  8,  4,   &
         9, 10,  8,   &
        11, 16, 16,   &
        17, 22, 16,   &
        23, 28, 16,   &
        29, 52, 96 ], &
        [3,9])
```

⟨*Muli interactions: variables*⟩+≡

```
integer, parameter, dimension(0:4,52), public :: muli_flows = &
     reshape( [ &
     3, 0, 0, 1, 2, &   !1a
     1, 0, 0, 2, 1, &
     1, 2, 0, 0, 3, &   !1b
     3, 3, 0, 0, 2, &
     4, 0, 0, 1, 2, &   !2
     4, 0, 0, 2, 1, &
     3, 2, 0, 0, 3, &   !3
     1, 3, 0, 0, 2, &
     4, 2, 0, 0, 3, &   !4
     4, 3, 0, 0, 2, &
     4, 0, 1, 3, 4, &   !5
     4, 0, 1, 4, 3, &
     2, 0, 3, 1, 4, &
     2, 0, 4, 1, 3, &
     2, 0, 3, 4, 1, &
     2, 0, 4, 3, 1, &
     4, 1, 2, 4, 0, &    !6
     2, 1, 4, 2, 0, &
     4, 2, 1, 4, 0, &
     2, 4, 1, 2, 0, &
     2, 2, 4, 1, 0, &
     2, 4, 2, 1, 0, &
     2, 0, 1, 2, 4, &    !7
     2, 0, 1, 4, 2, &
     4, 0, 2, 1, 4, &
     4, 0, 4, 1, 2, &
     2, 0, 2, 4, 1, &
     2, 0, 4, 2, 1, &
     9, 1, 2, 3, 4, &    !8
     5, 1, 2, 4, 3, &
     5, 1, 3, 2, 4, &
     3, 1, 4, 2, 3, &
     3, 1, 3, 4, 2, &
     5, 1, 4, 3, 2, &
     5, 2, 1, 3, 4, &
     5, 2, 1, 4, 3, &
     3, 3, 1, 2, 4, &
     3, 4, 1, 2, 3, &
```

```
      3, 3, 1, 4, 2, &
      3, 4, 1, 3, 2, &
      3, 2, 3, 1, 4, &
      3, 2, 4, 1, 3, &
      5, 3, 2, 1, 4, &
      3, 4, 2, 1, 3, &
      5, 3, 4, 1, 2, &
      3, 4, 3, 1, 2, &
      3, 2, 3, 4, 1, &
      3, 2, 4, 3, 1, &
      3, 3, 2, 4, 1, &
      5, 4, 2, 3, 1, &
      3, 3, 4, 2, 1, &
      5, 4, 3, 2, 1], [5, 52])
```

This value, `pts2_scale`, seems to be nowhere set in the code.

⟨*Muli interactions: variables*⟩+≡
```
real(default) :: pts2_scale
```

⟨*Muli interactions: interfaces*⟩≡
```
abstract interface
   function trafo_in (in)
     use kinds !NODEP!
     real(default), dimension(3) :: trafo_in
     real(default), dimension(3), intent(in) :: in
   end function trafo_in
end interface
```

⟨*Muli interactions: interfaces*⟩+≡
```
abstract interface
   pure function coord_scalar_in (hyp)
     use kinds !NODEP!
     real(default) :: coord_scalar_in
     real(double), dimension(3), intent(in) :: hyp
   end function coord_scalar_in
end interface
```

⟨*Muli interactions: interfaces*⟩+≡
```
abstract interface
   subroutine coord_hcd_in (hyp, cart, denom)
     use kinds !NODEP!
     real(default), dimension(3), intent(in) :: hyp
     real(default), dimension(3), intent(out) :: cart
     real(default), intent(out) :: denom
   end subroutine coord_hcd_in
end interface
```

This is the interface to the routines `alphasPDF` and `evolvePDF` from LHAPDF or internal PDFs which therefore need to be explicitly in `double` precision.

⟨*Muli interactions: interfaces*⟩+≡
```
interface
   pure function alphaspdf (Q)
     use kinds !NODEP!
     real(double) :: alphaspdf
     real(double), intent(in) :: Q
```

```
      end function alphaspdf
   end interface
```

⟨*Muli interactions: interfaces*⟩+≡

```
   interface
      pure subroutine evolvepdf (x, q, f)
        use kinds !NODEP!
        real(double), intent(in) :: x, q
        real(double), intent(out), dimension(-6:6) :: f
      end subroutine evolvepdf
   end interface
```

⟨*Muli interactions: public*⟩≡

```
   public :: muli_get_state_transformations
```

⟨*Muli interactions: procedures*⟩≡

```
   pure function muli_get_state_transformations &
        (inout_kind, lha_flavors) result (transformations)
     integer, intent(in) :: inout_kind
     integer, dimension(4), intent(in) :: lha_flavors
     integer, dimension(4) :: signature
     logical, dimension(3) :: transformations
     where (lha_flavors > 0)
        signature = 1
     elsewhere (lha_flavors < 0)
        signature = -1
     elsewhere
        signature = 0
     end where
     ! print *,"inout_kind=",inout_kind
     ! print *,"lha_flavors=",lha_flavors
     ! print *,"signature",signature
     if ((sum(inout_signatures(1:2,inout_kind)) == sum(signature(1:2))) .and. &
         (sum(inout_signatures(3:4,inout_kind)) == sum(signature(3:4)))) then
            transformations(1) = .false.
         else
            transformations(1) = .true.
            signature = -signature
         end if
     if (all (inout_signatures(1:2,inout_kind) == signature(1:2))) then
        transformations(2) = .false.
     else
        transformations(2) = .true.
     end if
     if (all(inout_signatures(3:4,inout_kind) == signature(3:4))) then
        transformations(3) = .false.
     else
        transformations(3) = .true.
     end if
     ! print *,"signature",signature
     ! print *,"transformations=",transformations
   end function muli_get_state_transformations
```

⟨*Muli interactions: public*⟩+≡

```
   public :: h_to_c_param
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function h_to_c_param (hyp)
  real(default), dimension(3) :: h_to_c_param
  real(default), dimension(3), intent(in) :: hyp
  h_to_c_param = [sqrt (sqrt ((((hyp(1)**4) * (one-hyp(3))) + &
       hyp(3))**2 + (((hyp(2)-(5E-1_default))**3)*4)**2) - &
       ((hyp(2)-(5E-1_default))**3)*4), &
       sqrt (sqrt ((((hyp(1)**4)*(one-hyp(3))) + hyp(3))**2 + &
       (((hyp(2)-(5E-1_default))**3)*4)**2) + &
       ((hyp(2)-(5E-1_default))**3)*4), hyp(3)]
end function h_to_c_param
```

⟨*Muli interactions: public*⟩+≡

```
public :: c_to_h_param
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function c_to_h_param (cart)
  real(default), dimension(3) :: c_to_h_param
  real(default), dimension(3), intent(in)::cart
  c_to_h_param= [ (((cart(1)*cart(2)) - cart(3)) / &
       (one - cart(3)))**(1/four), (one + sign(abs((cart(2)**2) - &
       (cart(1)**2))**(1/three), cart(2) - cart(1))) / two, cart(3) ]
end function c_to_h_param
```

⟨*Muli interactions: public*⟩+≡

```
public :: h_to_c_param_def
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function h_to_c_param_def (hyp)
  real(default), dimension(3) :: h_to_c_param_def
  real(default), dimension(3), intent(in) :: hyp
  h_to_c_param_def = h_to_c_param ([hyp(1), hyp(2), pts2_scale])
end function h_to_c_param_def
```

⟨*Muli interactions: public*⟩+≡

```
public :: h_to_c_ort
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function h_to_c_ort (hyp)
  real(default), dimension(3) :: h_to_c_ort
  real(default), dimension(3), intent(in) :: hyp
  h_to_c_ort = [sqrt (sqrt (((hyp(1) * (one - hyp(3))) + hyp(3))**2 + &
       (hyp(2) - (5E-1_default))**2) - (hyp(2) - (5E-1_default))), &
       sqrt (sqrt (((hyp(1) * (one - hyp(3))) + hyp(3))**2 + &
       (hyp(2)-(5E-1_default))**2) + (hyp(2) - (5E-1_default))), hyp(3)]
end function h_to_c_ort
```

⟨*Muli interactions: public*⟩+≡

```
public :: c_to_h_ort
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function c_to_h_ort (cart)
  real(default), dimension(3) :: c_to_h_ort
  real(default), dimension(3), intent(in) :: cart
```

```
    c_to_h_ort = [ (cart(3) - (cart(1)*cart(2))) / (cart(3) - one), &
         (one - cart(1)**2 + cart(2)**2) / two, cart(3)]
  end function c_to_h_ort
```

⟨*Muli interactions: public*⟩+≡

```
  public :: h_to_c_ort_def
```

⟨*Muli interactions: procedures*⟩+≡

```
  pure function h_to_c_ort_def (hyp)
    real(default), dimension(3) :: h_to_c_ort_def
    real(default), dimension(3), intent(in) :: hyp
    h_to_c_ort_def = h_to_c_ort ([hyp(1), hyp(2), pts2_scale])
  end function h_to_c_ort_def
```

⟨*Muli interactions: public*⟩+≡

```
  public :: c_to_h_ort_def
```

⟨*Muli interactions: procedures*⟩+≡

```
  pure function c_to_h_ort_def (cart)
    real(default), dimension(3) :: c_to_h_ort_def
    real(default), dimension(3), intent(in) :: cart
    c_to_h_ort_def = c_to_h_ort ([ cart(1), cart(2), pts2_scale])
  end function c_to_h_ort_def
```

⟨*Muli interactions: public*⟩+≡

```
  public :: h_to_c_noparam
```

⟨*Muli interactions: procedures*⟩+≡

```
  pure function h_to_c_noparam (hyp)
    real(default), dimension(2) :: h_to_c_noparam
    real(default), dimension(2), intent(in) :: hyp
    h_to_c_noparam = [sqrt (sqrt (hyp(1)**8 + (((hyp(2) - &
         (5E-1_default))**3)*4)**2) - ((hyp(2)-(5E-1_default))**3)*4), &
         sqrt (sqrt (hyp(1)**8 + (((hyp(2)-(5E-1_default))**3)*4)**2) + &
         ((hyp(2)-(5E-1_default))**3)*4)]
  end function h_to_c_noparam
```

⟨*Muli interactions: public*⟩+≡

```
  public :: c_to_h_noparam
```

⟨*Muli interactions: procedures*⟩+≡

```
  pure function c_to_h_noparam (cart)
    real(default), dimension(2) :: c_to_h_noparam
    real(default), dimension(2), intent(in) :: cart
    c_to_h_noparam = [sqrt (sqrt (cart(1)*cart(2))), &
         (one + sign(abs((cart(2)**2) - (cart(1)**2))**(one/three), &
         cart(2)-cart(1)))/two]
  end function c_to_h_noparam
```

⟨*Muli interactions: public*⟩+≡

```
  public :: c_to_h_param_def
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function c_to_h_param_def (cart)
  real(default), dimension(3) :: c_to_h_param_def
  real(default), dimension(3), intent(in) :: cart
  if (product (cart(1:2)) >= pts2_scale) then
     c_to_h_param_def = c_to_h_param ([cart(1), cart(2), pts2_scale])
  else
     c_to_h_param_def = [-one, -one, -one]
  end if
end function c_to_h_param_def
```

⟨*Muli interactions: public*⟩+≡

```
public :: h_to_c_smooth
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function h_to_c_smooth (hyp)
  real(default), dimension(3) :: h_to_c_smooth
  real(default), dimension(3), intent(in) :: hyp
  real(default) :: h2
  h2 = (((hyp(2) - 5E-1_default)**3) * 4._default + hyp(2)-5E-1_default) &
       / two
  h_to_c_smooth = &
      [sqrt (sqrt((((hyp(1)**4)*(one-hyp(3)))+hyp(3))**2+h2**2) - h2), &
       sqrt (sqrt((((hyp(1)**4)*(one-hyp(3)))+hyp(3))**2+h2**2) + h2), &
       hyp(3)]
end function h_to_c_smooth
```

⟨*Muli interactions: public*⟩+≡

```
public :: c_to_h_smooth
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function c_to_h_smooth (cart)
  real(default), dimension(3) :: c_to_h_smooth
  real(default), dimension(3), intent(in) :: cart
  c_to_h_smooth = &
       [((product (cart(1:2)) - cart(3)) / (one - cart(3)))**(1/four), &
        (three-three**(two/3) / (-9._default * cart(1)**2 + &
         9._default * cart(2)**2 + sqrt (three + 81._default * &
         (cart(1)**2 - cart(2)**2)**2))**(one/three)&
       + 3**(one/3)*(-9._default * cart(1)**2 + 9._default*cart(2)**2 &
       + sqrt(three + 81._default*(cart(1)**2&
       - cart(2)**2)**2))**(one/3))/6._default,cart(3)]
end function c_to_h_smooth
```

⟨*Muli interactions: public*⟩+≡

```
public :: h_to_c_smooth_def
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function h_to_c_smooth_def (hyp)
  real(default), dimension(3) :: h_to_c_smooth_def
  real(default), dimension(3), intent(in) :: hyp
  h_to_c_smooth_def = h_to_c_smooth ([hyp(1), hyp(2), pts2_scale])
end function h_to_c_smooth_def
```

⟨*Muli interactions: public*⟩+≡

⟨*Muli interactions: procedures*⟩+≡

```
pure function c_to_h_smooth_def (cart)
  real(default), dimension(3)::c_to_h_smooth_def
  real(default), dimension(3), intent(in) :: cart
  if (product (cart(1:2)) >= pts2_scale) then
     c_to_h_smooth_def = c_to_h_smooth ([cart(1), cart(2), pts2_scale])
  else
     c_to_h_smooth_def = [-one, -one, -one]
  end if
end function c_to_h_smooth_def
```

⟨*Muli interactions: public*⟩+≡

```
public :: voxel_h_to_c_ort
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function voxel_h_to_c_ort (hyp)
  real(default) :: voxel_h_to_c_ort
  real(default), dimension(3), intent(in) :: hyp
  real(default) :: T, TH1
  T = one - hyp(3)
  TH1 = T * (one - hyp(1))
  voxel_h_to_c_ort = sqrt (T**2 / (five - four*(one-hyp(2))*hyp(2) - &
       four*(two-TH1)*TH1))
end function voxel_h_to_c_ort
```

⟨*Muli interactions: public*⟩+≡

```
public :: voxel_c_to_h_ort
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function voxel_c_to_h_ort(cart)
  real(default) :: voxel_c_to_h_ort
  real(default), dimension(3), intent(in) :: cart
  real(default) :: P
  P = product (cart(1:2))
  if (P > cart(3)) then
     voxel_c_to_h_ort = (cart(1)**2 + cart(2)**2) / (one -cart(3))
  else
     voxel_c_to_h_ort = zero
  end if
end function voxel_c_to_h_ort
```

⟨*Muli interactions: public*⟩+≡

```
public :: voxel_h_to_c_noparam
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function voxel_h_to_c_noparam (hyp)
  real(default) :: voxel_h_to_c_noparam
  real(default), dimension(3), intent(in) :: hyp
  voxel_h_to_c_noparam = 12._default * sqrt ((hyp(1)**6 * &
       (one - two*hyp(2))**4) / (4*hyp(1)**8 + (one - two*hyp(2))**6))
end function voxel_h_to_c_noparam
```

⟨*Muli interactions: public*⟩+≡

```
public :: voxel_c_to_h_noparam
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function voxel_c_to_h_noparam (cart)
  real(default) :: voxel_c_to_h_noparam
  real(default), dimension(3), intent(in) :: cart
  real(default) :: P
  voxel_c_to_h_noparam = (cart(1)**2 + cart(2)**2) / (12._default * &
       (cart(1)*cart(2))**(three/four) * &
       (cart(2)**2 + cart(1)**2)**(two/three))
end function voxel_c_to_h_noparam
```

⟨*Muli interactions: public*⟩+≡

```
public :: voxel_h_to_c_param
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function voxel_h_to_c_param (hyp)
  real(default) :: voxel_h_to_c_param
  real(default), dimension(3), intent(in) :: hyp
  voxel_h_to_c_param = 12*Sqrt((hyp(1)**6 * &
       (one - 2._default*hyp(2))**4 * (hyp(3) - one)**2) / &
       ((one - two * hyp(2))**6 + four * &
       (hyp(3)-(hyp(1)**4*(hyp(3)-one)))**2))
end function voxel_h_to_c_param
```

⟨*Muli interactions: public*⟩+≡

```
public :: voxel_c_to_h_param
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function voxel_c_to_h_param (cart)
  real(default)::voxel_c_to_h_param
  real(default), dimension(3), intent(in) :: cart
  real(default) :: P, T, CP, CM
  P = product (cart(1:2))
  if (P > cart(3)) then
     P = P - cart(3)
     CP = cart(1)**2 + cart(2)**2
     CM = abs(cart(2)**2 - cart(1)**2)
     T = 1 - cart(3)
     voxel_c_to_h_param = (Cp*sqrt(sqrt(P/T))) / (12*Cm**(two/three)*P)
  else
     voxel_c_to_h_param = zero
  end if
end function voxel_c_to_h_param
```

⟨*Muli interactions: public*⟩+≡

```
public :: voxel_h_to_c_smooth
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function voxel_h_to_c_smooth (hyp)
  real(default) :: voxel_h_to_c_smooth
  real(default), dimension(3), intent(in) :: hyp
  real(default) :: T
```

```
    T = one - hyp(3)
    voxel_h_to_c_smooth = 8._default * (hyp(1)**3 * (one + three * &
         (hyp(2) - one)*hyp(2))*T) / sqrt ((one - two*hyp(2) * (two + &
         hyp(2)*(two*hyp(2)-three)))**2 + &
         four * (one + (hyp(1)**4 - one)*T)**2)
  end function voxel_h_to_c_smooth
```

⟨*Muli interactions: public*⟩+≡

```
  public :: voxel_c_to_h_smooth
```

⟨*Muli interactions: procedures*⟩+≡

```
  pure function voxel_c_to_h_smooth (cart)
    real(default) :: voxel_c_to_h_smooth
    real(default), dimension(3), intent(in) :: cart
    real(default) :: P, S, T, CM, CP
    P = product (cart(1:2))
    if (P > cart(3)) then
       P = P - cart(3)
       CP = cart(1)**2 + cart(2)**2
       CM = cart(2)**2 - cart(1)**2
       T = 1 - cart(3)
       S = sqrt(three + 81._default*cm**2)
       voxel_c_to_h_smooth = (three**(one/three) * Cp*(three**(one/three) + &
            (9._default*Cm + S)**(two/three)) * sqrt (sqrt (P/T))) / &
            (four * P * S * (9._default * Cm + S)**(one/three))
    else
       voxel_c_to_h_smooth = zero
    end if
  end function voxel_c_to_h_smooth
```

⟨*Muli interactions: public*⟩+≡

```
  public :: voxel_h_to_c_ort_def
```

⟨*Muli interactions: procedures*⟩+≡

```
  pure function voxel_h_to_c_ort_def (hyp)
    real(default) :: voxel_h_to_c_ort_def
    real(default), dimension(3), intent(in) :: hyp
    voxel_h_to_c_ort_def = voxel_h_to_c_ort (hyp)
  end function voxel_h_to_c_ort_def
```

⟨*Muli interactions: public*⟩+≡

```
  public :: voxel_c_to_h_ort_def
```

⟨*Muli interactions: procedures*⟩+≡

```
  pure function voxel_c_to_h_ort_def (cart)
    real(default) :: voxel_c_to_h_ort_def
    real(default), dimension(3), intent(in) :: cart
    voxel_c_to_h_ort_def = voxel_c_to_h_ort (cart)
  end function voxel_c_to_h_ort_def
```

⟨*Muli interactions: public*⟩+≡

```
  public :: voxel_h_to_c_param_def
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function voxel_h_to_c_param_def (hyp)
  real(default) :: voxel_h_to_c_param_def
  real(default), dimension(3), intent(in) :: hyp
  voxel_h_to_c_param_def = voxel_h_to_c_param (hyp)
end function voxel_h_to_c_param_def
```

⟨*Muli interactions: public*⟩+≡

```
public :: voxel_c_to_h_param_def
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function voxel_c_to_h_param_def (cart)
  real(default) :: voxel_c_to_h_param_def
  real(default), dimension(3), intent(in) :: cart
  voxel_c_to_h_param_def = voxel_c_to_h_param (cart)
end function voxel_c_to_h_param_def
```

⟨*Muli interactions: public*⟩+≡

```
public :: voxel_h_to_c_smooth_def
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function voxel_h_to_c_smooth_def (hyp)
  real(default) :: voxel_h_to_c_smooth_def
  real(default), dimension(3), intent(in) :: hyp
  voxel_h_to_c_smooth_def = voxel_h_to_c_smooth (hyp)
end function voxel_h_to_c_smooth_def
```

⟨*Muli interactions: public*⟩+≡

```
public :: voxel_c_to_h_smooth_def
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function voxel_c_to_h_smooth_def (cart)
  real(default) :: voxel_c_to_h_smooth_def
  real(default), dimension(3), intent(in) :: cart
  voxel_c_to_h_smooth_def = voxel_c_to_h_smooth (cart)
end function voxel_c_to_h_smooth_def
```

⟨*Muli interactions: public*⟩+≡

```
public :: denom_cart
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function denom_cart (cart)
  real(default) :: denom_cart
  real(default), dimension(3), intent(in) :: cart
  denom_cart = 1._default / (864._default * sqrt (cart(3)**3 * &
       (1._default - cart(3) / product(cart(1:2)))))
end function denom_cart
```

⟨*Muli interactions: public*⟩+≡

```
public :: denom_ort
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function denom_ort (hyp)
  real(default) :: denom_ort
  real(default), dimension(3), intent(in) :: hyp
  real(default) :: Y, P
  Y = (one - two * hyp(2))**2
  P = one - hyp(3)
  if (hyp(1) > zero .and. hyp(3) > zero) then
     denom_ort = sqrt ((P + (-1 + Hyp(1))*P**2) / &
          (746496*hyp(1)*hyp(3)**3 * (4*(1 + (-1 + hyp(1))*P)**2 + Y)))
  else
     denom_ort = zero
  end if
end function denom_ort
```

⟨*Muli interactions: public*⟩+≡

```
public :: denom_param
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function denom_param (hyp)
  real(default) :: denom_param
  real(default), dimension(3), intent(in) :: hyp
  real(default) :: X, Y, P
  X = hyp(1)**4
  Y = 1._default - 2._default * hyp(2)
  P = 1._default - hyp(3)
  if (hyp(3) > 0._default) then
     denom_param = sqrt ((P * (1+P*(X-1)) * Sqrt(X)*Y**4) / &
          (5184*(4*(1+P*(X-1))**2+Y**6)*hyp(3)**3))
  else
     denom_param = zero
  end if
end function denom_param
```

⟨*Muli interactions: public*⟩+≡

```
public :: denom_param_reg
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function denom_param_reg (hyp)
  real(default) :: denom_param_reg
  real(default), dimension(3), intent(in) :: hyp
  real(default) :: X, Y, P
  X = hyp(1)**4
  Y = one - two * hyp(2)
  P = one - hyp(3)
  if (hyp(3) > zero) then
     denom_param_reg = sqrt ((P*(1+P*(X-1)) * Sqrt(X)*Y**4) / &
          (5184*(4*(1+P*(X-1))**2+Y**6) * (hyp(3) + norm2_p_t_0)**3))
  else
     denom_param_reg = zero
  end if
end function denom_param_reg
```

⟨*Muli interactions: public*⟩+≡

```
public :: denom_smooth
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function denom_smooth (hyp)
  real(default) :: denom_smooth
  real(default), dimension(3), intent(in) :: hyp
  real(default) :: X, Y, P
  X = hyp(1)**2
  Y = (one - two * hyp(2))**2
  P = one - hyp(3)
  if (hyp(3) > zero) then
     denom_smooth = sqrt ((P * X * (one + P*(-one + X**2)) * &
          (1 + three*Y)**2)/(46656*hyp(3)**3 &
          *(16*(1 + P*(-1 + X**2))**2 + Y + 2*Y**2 + Y**3)))
  else
     denom_smooth = zero
  end if
end function denom_smooth
```

⟨*Muli interactions: public*⟩+≡

```
public :: denom_smooth_reg
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function denom_smooth_reg (hyp)
  real(default) :: denom_smooth_reg
  real(default), dimension(3), intent(in) :: hyp
  real(default) :: X, Y, P
  X = hyp(1)**2
  Y = (one - two * hyp(2))**2
  P = one - hyp(3)
  if (hyp(3) > zero) then
     denom_smooth_reg = sqrt ((P * X * (1 + P*(-1 + X**2)) * &
          (1 + 3*Y)**2)/(46656*(hyp(3) + norm2_p_t_0)**3 * &
          (16 * (1 + P*(-1 + X**2))**2 + Y + 2*Y**2 + Y**3)))
  else
     denom_smooth_reg = zero
  end if
end function denom_smooth_reg
```

⟨*Muli interactions: public*⟩+≡

```
public :: denom_cart_save
```

⟨*Muli interactions: procedures*⟩+≡

```
 pure function denom_cart_save (cart)
  real(default) :: denom_cart_save
  real(default), dimension(3), intent(in) :: cart
  if (product(cart(1:2)) > cart(3)) then
     denom_cart_save = denom_cart (cart)
  else
     denom_cart_save = zero
  end if
end function denom_cart_save
```

⟨*Muli interactions: public*⟩+≡
```
public :: denom_ort_save
```

⟨*Muli interactions: procedures*⟩+≡
```
pure function denom_ort_save (hyp)
  real(default) :: denom_ort_save
  real(default), dimension(3), intent(in) :: hyp
  real(default) :: Y, Z, W
  real(default), dimension(3) :: cart
  cart = h_to_c_ort (hyp)
  if (cart(1) > one .or. cart(2) > one) then
     denom_ort_save = zero
  else
     denom_ort_save = denom_ort (hyp)
  end if
end function denom_ort_save
```

⟨*Muli interactions: public*⟩+≡
```
public :: denom_param_save
```

⟨*Muli interactions: procedures*⟩+≡
```
pure function denom_param_save (hyp)
  real(default) :: denom_param_save
  real(default), dimension(3), intent(in) :: hyp
  real(default) :: Y, Z, W
  real(default), dimension(3) :: cart
  cart=h_to_c_param (hyp)
  if (cart(1) > one .or. cart(2) > one) then
     denom_param_save = zero
  else
     denom_param_save = denom_param (hyp)
  end if
end function denom_param_save
```

⟨*Muli interactions: public*⟩+≡
```
public :: denom_smooth_save
```

⟨*Muli interactions: procedures*⟩+≡
```
pure function denom_smooth_save (hyp)
  real(default) :: denom_smooth_save
  real(default), dimension(3), intent(in) :: hyp
  real(default) :: Y, Z, W
  real(default), dimension(3) :: cart
  cart = h_to_c_smooth (hyp)
  if (cart(1) > one .or. cart(2) > one) then
     denom_smooth_save = zero
  else
     denom_smooth_save = denom_smooth (hyp)
  end if
end function denom_smooth_save
```

⟨*Muli interactions: public*⟩+≡
```
public :: denom_cart_cuba_int
```

⟨*Muli interactions: procedures*⟩+≡

```
subroutine denom_cart_cuba_int (d_cart, cart, d_denom, denom, pt2s)
  real(default), dimension(3), intent(in) :: cart
  real(default), dimension(1), intent(out) :: denom
  real(default), intent(in) :: pt2s
  integer, intent(in) :: d_cart, d_denom
  denom(1) = denom_cart_save ([cart(1), cart(2), pt2s])
end subroutine denom_cart_cuba_int
```

⟨*Muli interactions: public*⟩+≡

```
public :: denom_ort_cuba_int
```

⟨*Muli interactions: procedures*⟩+≡

```
subroutine denom_ort_cuba_int (d_hyp, hyp, d_denom, denom, pt2s)
  real(default), dimension(3), intent(in) :: hyp
  real(default), dimension(1), intent(out) :: denom
  real(default), intent(in) :: pt2s
  integer, intent(in) :: d_hyp, d_denom
  denom(1) = denom_ort_save ([hyp(1), hyp(2), pt2s])
end subroutine denom_ort_cuba_int
```

⟨*Muli interactions: public*⟩+≡

```
public :: denom_param_cuba_int
```

⟨*Muli interactions: procedures*⟩+≡

```
subroutine denom_param_cuba_int (d_hyp, hyp, d_denom, denom, pt2s)
  real(default), dimension(3), intent(in) :: hyp
  real(default), dimension(1), intent(out) :: denom
  real(default), intent(in) :: pt2s
  integer, intent(in) :: d_hyp, d_denom
  denom(1) = denom_param_save ([hyp(1), hyp(2), pt2s])
end subroutine denom_param_cuba_int
```

⟨*Muli interactions: public*⟩+≡

```
public :: denom_smooth_cuba_int
```

⟨*Muli interactions: procedures*⟩+≡

```
subroutine denom_smooth_cuba_int (d_hyp, hyp, d_denom, denom, pt2s)
  real(default), dimension(3), intent(in) :: hyp
  real(default), dimension(1), intent(out) :: denom
  real(default), intent(in) :: pt2s
  integer, intent(in) :: d_hyp, d_denom
  denom(1) = denom_smooth_save ([hyp(1), hyp(2), pt2s])
end subroutine denom_smooth_cuba_int
```

⟨*Muli interactions: public*⟩+≡

```
public :: coordinates_hcd_cart
```

⟨*Muli interactions: procedures*⟩+≡

```
subroutine coordinates_hcd_cart (hyp, cart, denom)
  real(default), dimension(3), intent(in) :: hyp
  real(default), dimension(3), intent(out) :: cart
  real(default), intent(out) :: denom
  cart = hyp
```

```
    denom = denom_cart_save (cart)
  end subroutine coordinates_hcd_cart
```

⟨*Muli interactions: public*⟩+≡

```
  public :: coordinates_hcd_ort
```

⟨*Muli interactions: procedures*⟩+≡

```
  subroutine coordinates_hcd_ort (hyp, cart, denom)
    real(default), dimension(3), intent(in) :: hyp
    real(default), dimension(3), intent(out) :: cart
    real(default), intent(out)::denom
    cart = h_to_c_ort (hyp)
    denom = denom_ort (hyp)
  end subroutine coordinates_hcd_ort
```

⟨*Muli interactions: public*⟩+≡

```
  public :: coordinates_hcd_param
```

⟨*Muli interactions: procedures*⟩+≡

```
  subroutine coordinates_hcd_param (hyp, cart, denom)
    real(default), dimension(3), intent(in) :: hyp
    real(default), dimension(3), intent(out) :: cart
    real(default), intent(out) :: denom
    cart = h_to_c_param (hyp)
    denom = denom_param (hyp)
  end subroutine coordinates_hcd_param
```

⟨*Muli interactions: public*⟩+≡

```
  public :: coordinates_hcd_param_reg
```

⟨*Muli interactions: procedures*⟩+≡

```
  subroutine coordinates_hcd_param_reg (hyp, cart, denom)
    real(default), dimension(3), intent(in) :: hyp
    real(default), dimension(3), intent(out) :: cart
    real(default), intent(out) :: denom
    cart = h_to_c_param (hyp)
    denom = denom_param_reg (hyp)
  end subroutine coordinates_hcd_param_reg
```

⟨*Muli interactions: public*⟩+≡

```
  public :: coordinates_hcd_smooth
```

⟨*Muli interactions: procedures*⟩+≡

```
  subroutine coordinates_hcd_smooth (hyp, cart, denom)
    real(default), dimension(3), intent(in) :: hyp
    real(default), dimension(3), intent(out) :: cart
    real(default), intent(out) :: denom
    cart = h_to_c_smooth (hyp)
    denom = denom_smooth (hyp)
  end subroutine coordinates_hcd_smooth
```

⟨*Muli interactions: public*⟩+≡

```
  public :: coordinates_hcd_smooth_reg
```

⟨*Muli interactions: procedures*⟩+≡

```
subroutine coordinates_hcd_smooth_reg (hyp, cart, denom)
  real(default), dimension(3), intent(in) :: hyp
  real(default), dimension(3), intent(out) :: cart
  real(default), intent(out) :: denom
  cart = h_to_c_smooth (hyp)
  denom = denom_smooth_reg (hyp)
end subroutine coordinates_hcd_smooth_reg
```

⟨*Muli interactions: public*⟩+≡

```
public :: interactions_dddsigma_reg
```

⟨*Muli interactions: procedures*⟩+≡

```
pure subroutine interactions_dddsigma_reg &
     (process_id, double_pdf_id, hyp, cart, dddsigma)
  real(default), intent(out) :: dddsigma
  integer, intent(in) :: process_id, double_pdf_id
  real(default), dimension(3), intent(in) :: hyp
  real(default), dimension(3), intent(out) :: cart
  real(default) :: a, pt2shat, gev_pt, gev2_pt
  cart = h_to_c_param (hyp)
  a = product (cart(1:2))
  if (cart(1) <= 1D0 .and. cart(2) <= 1D0) then
     pt2shat = hyp(3) / a
     gev_pt = sqrt(hyp(3)) * gev_pt_max
     gev2_pt = hyp(3) * gev2_pt_max
     ! print *,process_id,pt2shat
     dddsigma = &
          const_pref &
          * alphasPDF (dble (sqrt (gev2_pt+gev2_p_t_0)))**2 &
          * ps_io_pol (process_id, pt2shat) &
          * pdf_in_in_kind &
             (process_id, double_pdf_id, cart(1), cart(2), gev_pt) &
          * denom_param_reg (hyp) / a
  else
     dddsigma = zero
  end if
end subroutine interactions_dddsigma_reg
```

⟨*Muli interactions: public*⟩+≡

```
public :: pdf_in_in_kind
```

⟨*Muli interactions: procedures*⟩+≡

```
pure function pdf_in_in_kind (process_id, double_pdf_id, c1, c2, gev_pt)
  real(default) :: pdf_in_in_kind
  real(default), intent(in) :: c1, c2, gev_pt
  integer, intent(in) :: process_id, double_pdf_id
  real(default) :: pdf1, pdf2
  call single_pdf (valid_processes(1, process_id), &
       double_pdf_kinds(1, double_pdf_id), c1, gev_pt, pdf1)
  call single_pdf (valid_processes(2, process_id), &
       double_pdf_kinds(2, double_pdf_id), c2, gev_pt, pdf2)
  pdf_in_in_kind = pdf1 * pdf2
contains
```

```
  pure subroutine single_pdf (flavor, pdf_kind, c, gev_pt, pdf)
    integer, intent(in) :: flavor, pdf_kind
    real(default), intent(in) :: c, gev_pt
    real(default), intent(out) :: pdf
    real(double), dimension(-6:6) :: lha_pdf
    call evolvePDF (dble (c), dble (gev_pt), lha_pdf)
    select case (pdf_kind)
    case (1)
       pdf = lha_pdf (0)
    case (2)
       if (flavor==1 .or. flavor==2) then
          pdf = lha_pdf (-flavor)
       else
          pdf = lha_pdf (flavor)
       end if
    case (3)
       pdf = lha_pdf(1) - lha_pdf(-1)
    case (4)
       pdf = lha_pdf(2) - lha_pdf(-2)
    end select
  end subroutine single_pdf
end function pdf_in_in_kind
```

⟨*Muli interactions: public*⟩+≡

```
public :: ps_io_pol
```

⟨*Muli interactions: procedures*⟩+≡

```
elemental function ps_io_pol (process_io_id, pt2shat)
  real(default) :: ps_io_pol
  integer, intent(in) :: process_io_id
  real(default), intent(in) :: pt2shat
  ps_io_pol = dot_product([1._default, pt2shat, pt2shat**2, pt2shat**3], &
       phase_space_coefficients_inout (1:4, &
       valid_processes (6, process_io_id)))
end function ps_io_pol
```

⟨*Muli interactions: public*⟩+≡

```
public :: interactions_dddsigma
```

⟨*Muli interactions: procedures*⟩+≡

```
pure subroutine interactions_dddsigma &
     (process_id, double_pdf_id, hyp, cart, dddsigma)
  real(default), intent(out) :: dddsigma
  integer, intent(in) :: process_id, double_pdf_id
  real(default), dimension(3), intent(in) :: hyp
  real(default), dimension(3), intent(out) :: cart
  real(default) :: a, pt2shat, gev_pt
  cart = h_to_c_param (hyp)
  a = product (cart(1:2))
  if (cart(1) <= 1._default .and. cart(2) <= 1._default) then
     pt2shat = hyp(3) / a
     gev_pt = sqrt(hyp(3)) * gev_pt_max
     ! print *,process_id,pt2shat
     dddsigma = const_pref * &
```

```
            alphasPDF (dble (gev_pt))**2 * &
            ps_io_pol (process_id, pt2shat) * &
            pdf_in_in_kind &
               (process_id, double_pdf_id, cart(1), cart(2), gev_pt) * &
            denom_param(hyp) / a
      else
         dddsigma = zero
      end if
   end subroutine interactions_dddsigma
```

⟨*Muli interactions: public*⟩+≡

```
   public :: interactions_dddsigma_print
```

⟨*Muli interactions: procedures*⟩+≡

```
   subroutine interactions_dddsigma_print &
        (process_id, double_pdf_id, hyp, cart, dddsigma)
      real(default), intent(out) :: dddsigma
      integer, intent(in) :: process_id, double_pdf_id
      real(default), dimension(3), intent(in) :: hyp
      real(default), dimension(3), intent(out) :: cart
      real(default) :: a, pt2shat, gev_pt
      cart = h_to_c_param (hyp)
      a = product (cart(1:2))
      if (cart(1) <= 1._default .and. cart(2) <= 1._default) then
         pt2shat = hyp(3) / a
         gev_pt=sqrt(hyp(3))*gev_pt_max
         ! print *,process_id,pt2shat
         dddsigma = const_pref * &
              ! alphasPDF(dble (gev_pt))**2 * &
              ps_io_pol (process_id, pt2shat) * &
              pdf_in_in_kind &
                 (process_id, double_pdf_id, cart(1), cart(2), gev_pt) * &
              denom_param (hyp) / a
      else
         dddsigma = zero
      end if
      write(11, *) dddsigma, pt2shat, &
           pdf_in_in_kind (process_id, double_pdf_id, cart(1), cart(2), &
           gev_pt), ps_io_pol (process_id, pt2shat), const_pref, &
           denom_param(hyp), a
      flush(11)
   end subroutine interactions_dddsigma_print
```

⟨*Muli interactions: public*⟩+≡

```
   public :: interactions_dddsigma_cart
```

⟨*Muli interactions: procedures*⟩+≡

```
   pure subroutine interactions_dddsigma_cart &
        (process_id, double_pdf_id, cart, dddsigma)
      real(default), intent(out) :: dddsigma
      integer, intent(in) :: process_id, double_pdf_id
      real(default), dimension(3), intent(in) :: cart
      real(default) :: a, pt2shat, gev_pt
      a = product (cart(1:2))
```

```
    if (cart(1) <= one .and. cart(2) <= one) then
       pt2shat = cart(3) / a
       gev_pt = sqrt(cart(3)) * gev_pt_max
       ! print *,process_id,pt2shat
       dddsigma = const_pref * &
            alphasPDF (dble (gev_pt))**2 * &
            ps_io_pol (process_id, pt2shat) * &
            pdf_in_in_kind &
               (process_id, double_pdf_id, cart(1), cart(2), gev_pt) * &
            denom_cart (cart) / a
    else
       dddsigma = zero
    end if
  end subroutine interactions_dddsigma_cart
```

⟨*Muli interactions: public*⟩+≡

```
  public :: cuba_gg_me_smooth
```

⟨*Muli interactions: procedures*⟩+≡

```
  subroutine cuba_gg_me_smooth (d_hyp, hyp, d_me, me, pt2s)
    integer, intent(in) :: d_hyp, d_me
    real(default), dimension(d_hyp), intent(in) :: hyp
    real(default), dimension(1), intent(out) :: me
    real(default), dimension(3) :: cart
    real(default), intent(in) :: pt2s
    real(default) :: p, p2
    if (d_hyp == 3) then
       p = hyp(3)
       p2 = hyp(3)**2
    else
       if (d_hyp == 2) then
          p = sqrt (pt2s)
          p2 = pt2s
       end if
    end if
    cart = h_to_c_smooth ([hyp(1), hyp(2), p2])
    if (p > pts_min .and. product (cart(1:2)) > p2) then
       me(1) = const_pref * &
            alphasPDF (dble (p*gev_pt_max))**2 * &
            ps_io_pol (109, p2) * &
            pdf_in_in_kind (109, 11, cart(1), cart(2), p2) * &
            denom_smooth ([hyp(1), hyp(2), p2]) / product (cart(1:2))
    else
       me(1) = zero
    end if
  end subroutine cuba_gg_me_smooth
```

⟨*Muli interactions: public*⟩+≡

```
  public :: cuba_gg_me_param
```

⟨*Muli interactions: procedures*⟩+≡

```
  subroutine cuba_gg_me_param (d_hyp, hyp, d_me, me, pt2s)
    integer, intent(in)::d_hyp,d_me
    real(default), dimension(d_hyp), intent(in) :: hyp
```

```
    real(default), dimension(1), intent(out) :: me
    real(default), dimension(3) :: cart
    real(default), intent(in) :: pt2s
    real(default) :: p, p2
    if (d_hyp == 3) then
       p = hyp(3)
       p2 = hyp(3)**2
    else
       if (d_hyp == 2) then
          p = sqrt (pt2s)
          p2 = pt2s
       end if
    end if
    cart = h_to_c_param ([hyp(1), hyp(2), p2])
    if (p>pts_min .and. product (cart(1:2))>p2) then
       me(1) = const_pref * &
            alphasPDF(dble (p*gev_pt_max))**2 * &
            ps_io_pol (109, p2) * &
            pdf_in_in_kind (109, 11, cart(1), cart(2), p2) * &
            denom_param ([hyp(1), hyp(2), p2]) / product(cart(1:2))
    else
       me(1) = zero
    end if
  end subroutine cuba_gg_me_param
```

⟨*Muli interactions: public*⟩+≡

```
  public :: cuba_gg_me_ort
```

⟨*Muli interactions: procedures*⟩+≡

```
  subroutine cuba_gg_me_ort (d_hyp, hyp, d_me, me, pt2s)
    integer, intent(in) :: d_hyp, d_me
    real(default), dimension(d_hyp), intent(in) :: hyp
    real(default), dimension(1), intent(out) :: me
    real(default), dimension(3) :: cart
    real(default), intent(in) :: pt2s
    real(default) :: p, p2
    if (d_hyp == 3) then
       p = hyp(3)
       p2 = hyp(3)**2
    else
       if (d_hyp == 2) then
          p = sqrt(pt2s)
          p2 = pt2s
       end if
    end if
    cart = h_to_c_ort ([hyp(1), cart(2), p2])
    if (p > pts_min .and. product (cart(1:2)) > p2) then
       me(1) = const_pref * &
            alphasPDF(dble (p*gev_pt_max))**2 * &
            ps_io_pol (109, p2) * &
            pdf_in_in_kind (109, 11, cart(1), cart(2), p2) * &
            denom_ort ([hyp(1), hyp(2), p2]) / product (cart(1:2))
    else
       me(1) = zero
```

```
    end if
  end subroutine cuba_gg_me_ort
```

⟨*Muli interactions: public*⟩+≡

```
  public :: cuba_gg_me_cart
```

⟨*Muli interactions: procedures*⟩+≡

```
  subroutine cuba_gg_me_cart (d_cart, cart, d_me, me, pt2s)
    integer, intent(in) :: d_cart, d_me
    real(default), dimension(d_cart), intent(in) :: cart
    real(default), dimension(1), intent(out) :: me
    real(default), intent(in) :: pt2s
    real(default) :: a, p, p2
    if (d_cart == 3) then
       p = cart(3)
       p2 = cart(3)**2
    else
       if (d_cart == 2) then
          p = sqrt (pt2s)
          p2 = pt2s
       end if
    end if
    a = product (cart(1:2))
    if (p > pts_min .and. a > p2) then
       me(1) = const_pref * &
            alphasPDF (dble (p*gev_pt_max))**2 * &
            ps_io_pol (109, p2) * &
            pdf_in_in_kind (109, 11, cart(1), cart(2), p2) * &
            denom_cart ([cart(1), cart(2), p2]) / a
    else
       me(1) = zero
    end if
  end subroutine cuba_gg_me_cart
```

⟨*Muli interactions: public*⟩+≡

```
  public :: interactions_proton_proton_integrand_generic_17_reg
```

⟨*Muli interactions: procedures*⟩+≡

```
  subroutine interactions_proton_proton_integrand_generic_17_reg &
       (hyp_2, trafo, f, pt)
    real(default), dimension(2), intent(in) :: hyp_2
    procedure(coord_hcd_in) :: trafo
    real(default), dimension(17), intent(out) :: f
    class(transverse_mom_t), intent(in) :: pt
    real(default), dimension(3) :: cart, hyp_3
    real(default), dimension(5) :: psin
    real(double), dimension(-6:6) :: c_dble, d_dble
    real(default), dimension(-6:6) :: c, d
    real(default) :: gev_pt, gev2_pt, pts, pt2s, pt2shat, a, &
         pdf_seaquark_seaquark, pdf_seaquark_gluon, pdf_gluon_gluon, &
         pdf_up_seaquark, pdf_up_gluon, pdf_down_seaquark, pdf_down_gluon, &
         v1u, v1d, v2u, v2d, denom

    pts = pt%get_unit_scale()
```

```
pt2s = pt%get_unit2_scale()
gev_pt = pt%get_gev_scale()
gev2_pt = pt%get_gev2_scale()

hyp_3(1:2) = hyp_2
hyp_3(3) = pt2s
call trafo (hyp_3, cart, denom)
a = product (cart(1:2))
if (cart(1) <= one .and. cart(2) <= one .and. a > pt2s) then
   pt2shat = pt2s / a
   ! phase space polynom
   psin = matmul ([one, pt2shat, pt2shat**2, pt2shat**3], &
        phase_space_coefficients_in)
   ! pdf
   call evolvepdf (dble (cart(1)), dble (gev_pt), c_dble)
   call evolvepdf (dble (cart(2)), dble (gev_pt), d_dble)
   c = c_dble
   d = d_dble
   ! c = [1,1,1,1,1,1,1,1,1,1,1,1]*1D0
   ! d = c
   v1d = c(1) - c(-1)
   v1u = c(2) - c(-2)
   v2d = d(1) - d(-1)
   v2u = d(2) - d(-2)
   c(1) = c(-1)
   c(2) = c(-2)
   d(1) = d(-1)
   d(2) = d(-2)
   f(1) = zero
   !!! gluon_gluon
   f( 2) = (c(0)*d(0)) * psin(5)
        !!! type5
   !!! gluon_seaquark
   f( 3) = (c(0)*d(-4) + c(0)*d(-3) + c(0)*d(-2) + c(0)*d(-1) + &
            c(0)*d(1)  + c(0)*d(2)  + c(0)*d(3)  + c(0)*d(4)) * psin(4)
        !!! type4
   !!! gluon_down
   f( 4) = (c(0)*v2d) * psin(4)
        !!! type4
   !!! gluon_up
   f( 5) = (c(0)*v2u) * psin(4)
        !!! type4
   !!! seaquark_gluon
   f( 6) = (c(-4)*d(0) + c(-3)*d(0) + c(-2)*d(0) + c(-1)*d(0) + &
            c(1)*d(0)  + c(2)*d(0)  + c(3)*d(0)  + c(4)*d(0)) * psin(4)
        !!! type4
   !!! seaquark_seaquark
   f( 7) = &
        !!! type1
           (c(-4)*d(-3) + c(-4)*d(-2) + c(-4)*d(-1) + c(-4)*d( 1) + &
            c(-4)*d( 2) + c(-4)*d( 3) + c(-3)*d(-4) + c(-3)*d(-2) + &
            c(-3)*d(-1) + c(-3)*d( 1) + c(-3)*d( 2) + c(-3)*d( 4) + &
            c(-2)*d(-4) + c(-2)*d(-3) + c(-2)*d(-1) + c(-2)*d( 1) + &
            c(-2)*d( 3) + c(-2)*d( 4) + c(-1)*d(-4) + c(-1)*d(-3) + &
```

```
         c(-1)*d(-2) + c(-1)*d( 2) + c(-1)*d( 3) + c(-1)*d( 4) + &
         c( 1)*d(-4) + c( 1)*d(-3) + c( 1)*d(-2) + c( 1)*d( 2) + &
         c( 1)*d( 3) + c( 1)*d( 4) + c( 2)*d(-4) + c( 2)*d(-3) + &
         c( 2)*d(-1) + c( 2)*d( 1) + c( 2)*d( 3) + c( 2)*d( 4) + &
         c( 3)*d(-4) + c( 3)*d(-2) + c( 3)*d(-1) + c( 3)*d( 1) + &
         c( 3)*d( 2) + c( 3)*d( 4) + c( 4)*d(-3) + c( 4)*d(-2) + &
         c( 4)*d(-1) + c( 4)*d( 1) + c( 4)*d( 2) + c( 4)*d( 3)) * &
         psin(1) + &
     !!! type2
        (c(-4)*d(-4) + c(-3)*d(-3) + c(-2)*d(-2) + c(-1)*d(-1) + &
         c( 4)*d( 4) + c( 3)*d( 3) + c(2)*d( 2)  + c(1)*d( 1)) * &
         psin(2) + &
     !!! type3
        (c(-4)*d( 4) + c(-3)*d( 3) + c(-2)*d( 2) + c(-1)*d( 1) + &
         c( 4)*d(-4) + c( 3)*d(-3) + c(2)*d(-2)  + c(1)*d(-1)) * &
         psin(3)
!!! seaquark_down
f( 8) = &
     !!! type1
        (c(-4)*v2d + c(-3)*v2d + c(-2)*v2d + c( 2)*v2d + &
         c( 3)*v2d + c( 4)*v2d) * psin(1) + &
     !!! type2
         c( 1)*v2d * psin(2) + &
     !!! type3
         c(-1)*v2d * psin(3)
!!! seaquark_up
f( 9) = &
     !!! type1
        (c(-4)*v2u + c(-3)*v2u + c(-1)*v2u + c( 1)*v2u + &
         c( 3)*v2u + c( 4)*v2u) * psin(1) + &
     !!! type2
         c(2)*v2u * psin(2) + &
     !!! type3
         c(-2)*v2u * psin(3)
!!! down_gluon
f(10) = (v1d*d( 0)) * psin(4)
     !!! type4
!!! down_seaquark
f(11) = &
     !!! type1
        (v1d*d(-4) + v1d*d(-3) + v1d*d(-2) + v1d*d( 2) + &
         v1d*d( 3) + v1d*d( 4)) * psin(1) + &
     !!! type2
         v1d*d( 1) * psin(2) + &
     !!! type3
         v1d*d(-1) * psin(3)
!!! down_down
f(12) = v1d*v2d * psin(2)
!!! down_up
f(13) = v1d*v2u * psin(1)
!!! up_gluon
f(14) = (v1u*d(0)) * psin(4)
     !!! type4
!!! up_seaquark
```

```
      f(15) = &
           !!! type1
              (v1u*d(-4) + v1u*d(-3) + v1u*d(-1) + v1u*d( 1) + &
               v1u*d( 3) + v1u*d( 4)) * psin(1) + &
           !!! type2
               v1u*d(2) * psin(2) + &
           !!! type3
               v1u*d(-2) * psin(3)
      !!! up_down
      f(16) = v1u * v2d * psin(1)
      !!! up_up
      f(17) = v1u * v2u * psin(2)
      f=f * const_pref &
          * alphasPDF (dble (sqrt(gev2_pt+gev2_p_t_0)))**2 &
          * denom / a
      ! print *, const_pref, alphasPDF(gev_pt)**2, denom_smooth (hyp), a
   else
      f = [zero, zero, zero, zero, zero, zero, zero, zero, zero, &
           zero, zero, zero, zero, zero, zero, zero, zero]
   end if
   ! print *, pt2shat, c(0)*d(0), psin(5), const_pref, &
   !     alphasPDF(gev_pt)**2, denom, a
end subroutine interactions_proton_proton_integrand_generic_17_reg
```

⟨*Muli interactions: procedures*⟩+≡

```
!  subroutine coordinates_proton_proton_integrand_cart_11 &
!       (d_hyp, hyp_2, d_f, f)
!    integer, intent(in) :: d_hyp, d_f
!    real(default), dimension(2), intent(in) :: hyp_2
!    real(default), dimension(11), intent(out) :: f
!    call coordinates_proton_proton_integrand_generic_11 &
!       (hyp_2, coordinates_hcd_cart, f)
!    ! write (51,*) hyp_2, momentum_get_pts_scale(), f
!  end subroutine coordinates_proton_proton_integrand_cart_11
```

⟨*Muli interactions: procedures*⟩+≡

```
!  subroutine coordinates_proton_proton_integrand_ort_11 &
!       (d_hyp, hyp_2, d_f, f)
!    integer, intent(in) :: d_hyp, d_f
!    real(default), dimension(2), intent(in) :: hyp_2
!    real(default), dimension(11), intent(out) :: f
!    call coordinates_proton_proton_integrand_generic_11 &
!       (hyp_2, coordinates_hcd_ort, f)
!    ! write (52,*) hyp_2, momentum_get_pts_scale(), f
!  end subroutine coordinates_proton_proton_integrand_ort_11
```

⟨*Muli interactions: procedures*⟩+≡

```
!  subroutine coordinates_proton_proton_integrand_param_11 &
!       (d_hyp, hyp_2, d_f, f)
!    integer, intent(in) :: d_hyp, d_f
!    real(default), dimension(2), intent(in) :: hyp_2
!    real(default), dimension(11), intent(out) :: f
```

```
!    call coordinates_proton_proton_integrand_generic_11 &
!       (hyp_2, coordinates_hcd_param, f)
!    ! write(53,*) hyp_2, momentum_get_pts_scale(), f
!  end subroutine coordinates_proton_proton_integrand_param_11
```

⟨*Muli interactions: procedures*⟩+≡

```
!  subroutine coordinates_proton_proton_integrand_smooth_11 &
!       (d_hyp, hyp_2, d_f, f)
!    integer, intent(in)::d_hyp,d_f
!    real(default), dimension(2), intent(in) :: hyp_2
!    real(default), dimension(11), intent(out) :: f
!    call coordinates_proton_proton_integrand_generic_11 (hyp_2, &
!       coordinates_hcd_smooth, f)
!    ! write (54,*) hyp_2, momentum_get_pts_scale(), f
!  end subroutine coordinates_proton_proton_integrand_smooth_11
```

⟨*Muli interactions: public*⟩+≡

```
public :: interactions_proton_proton_integrand_param_17_reg
```

⟨*Muli interactions: procedures*⟩+≡

```
subroutine interactions_proton_proton_integrand_param_17_reg &
     (d_hyp, hyp_2, d_f, f, pt)
  integer, intent(in) :: d_hyp, d_f
  real(default), dimension(2), intent(in) :: hyp_2
  real(default), dimension(17), intent(out) :: f
  class(transverse_mom_t), intent(in) :: pt
  call interactions_proton_proton_integrand_generic_17_reg &
       (hyp_2, coordinates_hcd_param_reg, f, pt)
  ! write (53,*)  hyp_2,momentum_get_pts_scale(),f
end subroutine interactions_proton_proton_integrand_param_17_reg
```

⟨*Muli interactions: public*⟩+≡

```
public :: interactions_proton_proton_integrand_smooth_17_reg
```

⟨*Muli interactions: procedures*⟩+≡

```
subroutine interactions_proton_proton_integrand_smooth_17_reg &
     (d_hyp, hyp_2, d_f, f, pt)
  integer, intent(in) :: d_hyp, d_f
  real(default), dimension(2), intent(in) :: hyp_2
  real(default), dimension(17), intent(out) :: f
  class(transverse_mom_t), intent(in) :: pt
  call interactions_proton_proton_integrand_generic_17_reg &
       (hyp_2, coordinates_hcd_smooth_reg, f, pt)
  ! write (53,*)hyp_2,momentum_get_pts_scale(), f
end subroutine interactions_proton_proton_integrand_smooth_17_reg
```

22.4 VEGAS and CUBA integration routines

This file contains the module `muli_cuba`, a wrapper for the CUBA integration library. Different algorithms and settings have been tried out for the integration, including VEGAS, and this wrapper has been mainly written for that purpose.

⟨`muli_cuba.f90`⟩≡

⟨*File header*⟩

```
module muli_cuba
```

⟨*Use kinds*⟩

```
  use constants
  use diagnostics
  use muli_base
  use muli_momentum
```

⟨*Standard module head*⟩

⟨*Muli CUBA: public*⟩

⟨*Muli CUBA: variables*⟩

⟨*Muli CUBA: types*⟩

⟨*Muli CUBA: interfaces*⟩

```
contains
```

⟨*Muli CUBA: procedures*⟩

```
end module muli_cuba
```

⟨*Muli CUBA: variables*⟩≡

```
  integer, parameter :: max_maxeval = huge(1)
```

⟨*Muli CUBA: public*⟩≡

```
  public :: cuba_class
```

⟨*Muli CUBA: types*⟩≡

```
  type, extends (ser_class_t), abstract :: cuba_class
    real(default) :: start_time = zero
    real(default) :: stop_time = zero
    real(default) :: run_time = zero
    integer :: dim_x = 2
    integer :: dim_f = 1
    type(transverse_mom_t) :: userdata
    real(default) :: eps_rel = 1.E-3_default
    real(default) :: eps_abs = 0._default
    integer :: flags = 0
    integer :: seed = 1
    integer :: min_eval = 0
    integer :: max_eval = max_maxeval
    integer :: neval = 0
    integer, public :: fail = -1
    integer :: nregions = 0
    real(default), dimension(:), allocatable :: integral
    real(default), dimension(:), allocatable :: error
    real(default), dimension(:), allocatable :: prob
    procedure(integrand_interface), nopass, pointer :: integrand
  contains
```

⟨*Muli CUBA: cuba class: TBP*⟩

```
end type cuba_class
```

⟨*Muli CUBA: types*⟩+≡

```
type, extends (cuba_class) :: cuba_cuhre_t
   private
   integer :: key = 13
 contains
 ⟨Muli CUBA: cuba cuhre: TBP⟩
end type cuba_cuhre_t
```

⟨*Muli CUBA: types*⟩+≡

```
type, extends (cuba_class) :: cuba_suave_t
   private
   integer :: nnew = 10000    !1000
   integer :: flatness = 5   !50
 contains
 ⟨Muli CUBA: cuba suave: TBP⟩
end type cuba_suave_t
```

⟨*Muli CUBA: public*⟩+≡

```
public :: cuba_divonne_t
```

⟨*Muli CUBA: types*⟩+≡

```
type, extends (cuba_class) :: cuba_divonne_t
   private
   integer :: key1 = 13
   integer :: key2 = 13
   integer :: key3 = 13
   integer :: maxpass = 2
   real(default) :: border = zero
   real(default) :: maxchisq = 10._default
   real(default) :: mindeviation = .25_default
   integer :: ngiven = 0
   integer :: ldxgiven = 0
   ! real(default), dimension(ldxgiven,ngiven) :: &
   !    xgiven = reshape( source = [ 0.0,0.0 ], shape = [2,1])
   real(default), dimension(:,:), allocatable :: xgiven
   ! real(default), dimension(2) :: xgiven = [1E-1_default, 5E-1_default]
   integer :: nextra = 0
 contains
 ⟨Muli CUBA: cuba divonne: TBP⟩
end type cuba_divonne_t
```

⟨*Muli CUBA: types*⟩+≡

```
type, extends (cuba_class) :: cuba_vegas_t
   private
   integer :: nstart = 500
   integer :: nincrease = 1000
   integer :: nbatch = 1000
   integer :: gridno = 0
   character(len=8), pointer :: statefile => null()
 contains
```

⟨*Muli CUBA: cuba vegas: TBP*⟩
```
end type cuba_vegas_t
```

⟨*Muli CUBA: cuba class: TBP*⟩≡
```
procedure :: write_to_marker => cuba_write_to_marker
```

⟨*Muli CUBA: procedures*⟩≡
```
subroutine cuba_write_to_marker (this, marker, status)
  class(cuba_class), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%mark_begin ("cuba_class")
  call marker%mark ("dim_x", this%dim_x)
  call marker%mark ("dim_f", this%dim_f)
  call marker%mark ("eps_rel", this%eps_rel)
  call marker%mark ("eps_abs", this%eps_abs)
  call marker%mark ("flags", this%flags)
  call marker%mark ("min_eval", this%min_eval)
  call marker%mark ("max_eval", this%max_eval)
  call marker%mark ("neval", this%neval)
  call marker%mark ("fail", this%fail)
  call marker%mark ("nregions", this%nregions)
  if (allocated (this%integral)) then
     call marker%mark ("integral", this%integral)
  else
     call marker%mark_null ("integral")
  end if
  if (allocated(this%error)) then
     call marker%mark ("error", this%error)
  else
     call marker%mark_null ("error")

  end if
  if (allocated (this%prob)) then
     call marker%mark ("prob", this%prob)
  else
     call marker%mark_null ("prob")
  end if
  call marker%mark_null ("cuba_class")
end subroutine cuba_write_to_marker
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡
```
procedure :: read_from_marker => cuba_read_from_marker
```

⟨*Muli CUBA: procedures*⟩+≡
```
subroutine cuba_read_from_marker (this, marker, status)
  class(cuba_class), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out)::status
  call marker%pick_begin ("CUBA_CLASS", status=status)
  call marker%pick ("dim_x", this%dim_x,status)
  call marker%pick ("dim_f", this%dim_f,status)
  call marker%pick ("eps_rel", this%eps_rel,status)
  call marker%pick ("eps_abs", this%eps_abs,status)
```

```
    call marker%pick ("flags", this%flags,status)
    call marker%pick ("min_eval", this%min_eval,status)
    call marker%pick ("max_eval", this%max_eval,status)
    call marker%pick ("neval", this%neval,status)
    call marker%pick ("fail", this%fail,status)
    call marker%pick ("nregions", this%nregions,status)
    call marker%verify_nothing("integral",status)
    if (allocated (this%integral))  deallocate (this%integral)
    if (status == serialize_ok) then
       allocate (this%integral (this%dim_f))
       call marker%pick ("integral", this%integral, status)
    end if
    call marker%verify_nothing ("error", status)
    if (allocated (this%error))  deallocate (this%error)
    if (status == serialize_ok) then
       allocate (this%error (this%dim_f))
       call marker%pick ("error", this%error, status)
    end if
    call marker%verify_nothing ("prob", status)
    if (allocated (this%prob))  deallocate (this%prob)
    if (status == serialize_ok) then
       allocate (this%prob (this%dim_f))
       call marker%pick ("prob", this%prob, status)
    end if
    call marker%pick_end ("cuba_class", status)
  end subroutine cuba_read_from_marker
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
  procedure :: print_to_unit => cuba_print_to_unit
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine cuba_print_to_unit (this, unit, parents, components, peers)
    class(cuba_class), intent(in) :: this
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    character(11) :: n
    write (n, '("(",I2,"(E12.4))")')  this%dim_f
    write (unit, "(1x,A)")  "Components of cuba_class:"
    write (unit, "(3x,A)")  "Parameters:"
    write (unit, "(3x,A,I10)")   "dim_f:     ", this%dim_f
    write (unit, "(3x,A,I10)")   "dim_x:     ", this%dim_x
    call this%userdata%print_to_unit (unit, parents, components-1, peers)
    write (unit, "(3x,A,E10.4)") "eps_rel:   ", this%eps_rel
    write (unit, "(3x,A,E10.4)") "eps_abs:   ", this%eps_abs
    write (unit, "(3x,A,I10)")   "flags:     ", this%flags
    write (unit, "(3x,A,I10)")   "seed:      ", this%seed
    write (unit, "(3x,A,I10)")   "min_eval:  ", this%min_eval
    write (unit, "(3x,A,I10)")   "max_eval:  ", this%max_eval
    write (unit, "(3x,A)")  "Results:"
    write (unit, "(3x,A,I10)")   "neval:     ", this%neval
    write (unit, "(3x,A,I10)")   "fail:      ", this%fail
    write (unit, "(3x,A)", advance="no") "integral:  "
    write (unit, fmt=n)  this%integral
    write (unit, "(3x,A)", advance="no")  "error:     "
```

```
    write (unit, fmt=n)  this%error
    write (unit, "(3x,A)", advance="no")  "prob:      "
    write (unit, fmt=n)  this%prob
    write (unit, "(3x,A,E10.4)")  "time:      ", &
         this%stop_time - this%start_time
    ! write(unit,'("time:      ",E10.4)') this%run_time
  end subroutine cuba_print_to_unit
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
  generic   :: get_integral => get_integral_array, get_integral_1
  procedure :: get_integral_array => cuba_get_integral_array
  procedure :: get_integral_1 => cuba_get_integral_1
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine cuba_get_integral_array (this, integral)
    class(cuba_class) :: this
    real(default), intent(out), dimension(:) :: integral
    integral = this%integral
  end subroutine cuba_get_integral_array
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine cuba_get_integral_1 (this, integral)
    class(cuba_class) :: this
    real(default), intent(out) :: integral
    integral = this%integral(1)
  end subroutine cuba_get_integral_1
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
  procedure :: copy_common => cuba_copy_common
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine cuba_copy_common (this, source)
    class(cuba_class), intent(out) :: this
    class(cuba_class), intent(in) :: source
    this%dim_x = source%dim_x
    this%dim_f = source%dim_f
    this%eps_rel = source%eps_rel
    this%eps_abs = source%eps_abs
    this%flags = source%flags
    this%min_eval = source%min_eval
    this%max_eval = source%max_eval
    call this%alloc()
  end subroutine cuba_copy_common
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
  procedure :: set_common => cuba_set_common
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine cuba_set_common (this, dim_x, dim_f, eps_rel, eps_abs, &
       flags, seed, min_eval, max_eval, integrand, userdata)
    class(cuba_class), intent(inout) :: this
    integer, intent(in), optional :: dim_x, dim_f, flags, min_eval, &
         max_eval, seed
```

```
      real(default), intent(in), optional :: eps_rel,eps_abs
      type(transverse_mom_t), intent(in), optional :: userdata
      procedure(integrand_interface), optional :: integrand
      if (present (dim_x)) then
         call this%set_dim_x (dim_x)
      end if
      if (present (dim_f)) then
         call this%set_dim_f (dim_f)
      end if
      if (present (flags)) then
         this%flags = flags
      end if
      if (present (seed)) then
         this%seed = seed
      end if
      if (present (min_eval)) then
         this%min_eval = min_eval
      end if
      if (present (max_eval)) then
         if (max_eval < max_maxeval) then
            this%max_eval = max_eval
         else
            call msg_warning &
                 ("cuba_set_common: Value of max_eval is too large.")
            this%max_eval = max_maxeval
         end if
      end if
      if (present (eps_rel)) then
         this%eps_rel = eps_rel
      end if
      if (present (eps_abs)) then
         this%eps_abs = eps_abs
      end if
      if (present (integrand))  this%integrand=>integrand
      if (present (userdata))  this%userdata=userdata
    end subroutine cuba_set_common
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
    procedure :: set_dim_f => cuba_set_dim_f
```

⟨*Muli CUBA: procedures*⟩+≡

```
    subroutine cuba_set_dim_f (this, new_dim_f)
      class(cuba_class) :: this
      integer, intent(in) :: new_dim_f
      if (new_dim_f > 0) then
         this%dim_f = new_dim_f
         call this%alloc_dim_f
      else
         call msg_error ("cuba_set_dim_f: New value for dim_f is " &
              // "negative. dim_f is not set.")
      end if
    end subroutine cuba_set_dim_f
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
procedure :: set_dim_x => cuba_set_dim_x
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_set_dim_x (this, new_dim_x)
  class(cuba_class) :: this
  integer, intent(in) :: new_dim_x
  if (new_dim_x > 0) then
     this%dim_x = new_dim_x
  else
     call msg_error ("cuba_set_dim_x: New value for dim_x is " &
          // "negative. dim_x is not set.")
  end if
end subroutine cuba_set_dim_x
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
procedure :: reset_timer => cuba_reset_timer
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_reset_timer (this)
  class(cuba_class), intent(inout) :: this
  this%start_time = zero
  this%stop_time = zero
  this%run_time = zero
end subroutine cuba_reset_timer
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
procedure :: integrate_with_timer => cuba_integrate_with_timer
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_integrate_with_timer (this, integrand)
  class(cuba_class), intent(inout) :: this
  procedure(integrand_interface) :: integrand
  call cpu_time (this%start_time)
  call this%integrate (integrand)
  call cpu_time (this%stop_time)
  this%run_time = this%run_time + this%stop_time - this%start_time
end subroutine cuba_integrate_with_timer
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
procedure :: integrate_associated => cuba_integrate_associated
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_integrate_associated (this)
  class(cuba_class), intent(inout) :: this
  call this%integrate_with_timer (this%integrand)
end subroutine cuba_integrate_associated
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
generic :: integrate => integrate_nd, integrate_userdata
procedure(integrate_interface), deferred :: integrate_nd
procedure(integrate_userdata_interface), deferred :: integrate_userdata
procedure(cuba_copy_interface), deferred :: copy
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
procedure :: dealloc_dim_f => cuba_dealloc_dim_f
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_dealloc_dim_f(this)
  class(cuba_class) :: this
  ! print '("cuba_dealloc_dim_f...")'
  if (allocated (this%integral)) then
     deallocate (this%integral)
  end if
  if (allocated (this%error)) then
     deallocate (this%error)
  end if
  if (allocated (this%prob)) then
     deallocate (this%prob)
  end if
  ! print '("done")'
end subroutine cuba_dealloc_dim_f
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
procedure :: alloc_dim_f => cuba_alloc_dim_f
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_alloc_dim_f (this)
  class(cuba_class) :: this
  call this%dealloc_dim_f ()
  allocate (this%integral (this%dim_f))
  allocate (this%error (this%dim_f))
  allocate (this%prob (this%dim_f))
end subroutine cuba_alloc_dim_f
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
procedure :: dealloc => cuba_dealloc
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_dealloc (this)
  class(cuba_class) :: this
  call this%dealloc_dim_f
end subroutine cuba_dealloc
```

⟨*Muli CUBA: cuba class: TBP*⟩+≡

```
procedure :: alloc => cuba_alloc
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_alloc (this)
  class(cuba_class) :: this
  call this%alloc_dim_f
end subroutine cuba_alloc
```

⟨*Muli CUBA: cuba vegas: TBP*⟩≡

```
procedure :: write_to_marker => cuba_vegas_write_to_marker
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_vegas_write_to_marker (this, marker, status)
  class(cuba_vegas_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
```

```
    call marker%mark_begin ("cuba_vegas_t")
    call cuba_write_to_marker (this, marker, status)
    call marker%mark("nstart", this%nstart)
    call marker%mark("nincrease", this%nincrease)
    call marker%mark_null ("cuba_vegas_t")
  end subroutine cuba_vegas_write_to_marker
```

⟨*Muli CUBA: cuba vegas: TBP*⟩+≡

```
  procedure :: read_from_marker => cuba_vegas_read_from_marker
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine cuba_vegas_read_from_marker (this, marker, status)
    class(cuba_vegas_t), intent(out) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    call marker%pick_begin ("cuba_vegas_t", status=status)
    call cuba_read_from_marker (this, marker, status)
    call marker%pick ("nstart", this%nstart, status)
    call marker%pick ("nincrease", this%nincrease, status)
    call marker%pick_end ("cuba_vegas_t", status)
  end subroutine cuba_vegas_read_from_marker
```

⟨*Muli CUBA: cuba vegas: TBP*⟩+≡

```
  procedure :: print_to_unit => cuba_vegas_print_to_unit
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine cuba_vegas_print_to_unit(this,unit,parents,components,peers)
    class(cuba_vegas_t), intent(in) :: this
    INTEGER, INTENT(IN) :: unit
    integer(dik), intent(in)::parents,components,peers
    if (parents>0)call cuba_print_to_unit(this,unit,parents-1,components,peers)
    write (unit, "(1x,A)")       "Components of cuba_vegas_t:"
    write (unit, "(3x,A,I10)")   "nstart:    ", this%nstart
    write (unit, "(3x,A,I10)")   "nincrease: ", this%nincrease
    write (unit, "(3x,A,I10)")   "nbatch:    ", this%nbatch
    write (unit, "(3x,A,I10)")   "gridno:    ", this%gridno
    if (associated (this%statefile)) then
       write (unit, "(3x,A,A)")  "statefile: ", this%statefile
    else
       write (unit, "(3x,A)")    "statefile:    not associated"
    end if
  end subroutine cuba_vegas_print_to_unit
```

⟨*Muli CUBA: cuba vegas: TBP*⟩+≡

```
  procedure, nopass :: get_type => cuba_vegas_get_type
```

⟨*Muli CUBA: procedures*⟩+≡

```
  pure subroutine cuba_vegas_get_type (type)
    character(:), allocatable, intent(out) :: type
    allocate (type, source="cuba_vegas_t")
  end subroutine cuba_vegas_get_type
```

⟨*Muli CUBA: cuba vegas: TBP*⟩+≡

```
  procedure :: integrate_nd => integrate_vegas
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine integrate_vegas (this, integrand)
  class(cuba_vegas_t), intent(inout) :: this
  procedure(integrand_interface) :: integrand
  ! print '("vegas")'
  ! call vegas (this%dim_x, this%dim_f, integrand, this%userdata, &
  !      this%eps_rel, this%eps_abs, this%flags, this%seed, &
  !      this%min_eval, this%max_eval, this%nstart, this%nincrease, &
  !      this%nbatch, this%gridno, this%statefile, this%neval, &
  !      this%fail, this%integral, this%error, this%prob)
end subroutine integrate_vegas
```

⟨*Muli CUBA: cuba vegas: TBP*⟩+≡

```
procedure :: integrate_userdata => integrate_vegas_userdata
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine integrate_vegas_userdata (this, integrand, userdata)
  class(cuba_vegas_t), intent(inout) :: this
  procedure(integrand_interface) :: integrand
  class(transverse_mom_t), intent(in) :: userdata
  ! print '("vegas")'
  ! call vegas(this%dim_x, this%dim_f, integrand, userdata, this%eps_rel, &
  !      this%eps_abs, this%flags, this%seed, this%min_eval, this%max_eval, &
  !      this%nstart, this%nincrease, this%nbatch, this%gridno, &
  !      this%statefile, this%neval, this%fail, this%integral, &
  !      this%error, this%prob)
end subroutine integrate_vegas_userdata
```

⟨*Muli CUBA: cuba vegas: TBP*⟩+≡

```
procedure :: copy => cuba_vegas_copy
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_vegas_copy (this, source)
  class(cuba_vegas_t), intent(out) :: this
  class(cuba_class), intent(in) :: source
  select type (source)
  class is (cuba_vegas_t)
     call this%copy_common (source)
     this%nstart = source%nstart
     this%nincrease = source%nincrease
  class default
     call msg_error ("cuba_vegas_copy: type of source is not type " &
          // "compatible with cuba_vegas_t.")
  end select
end subroutine cuba_vegas_copy
```

⟨*Muli CUBA: cuba vegas: TBP*⟩+≡

```
procedure :: set_deferred => cuba_vegas_set_deferred
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_vegas_set_deferred &
     (this, n_start, n_increase, nbatch, gridno, statefile)
  class(cuba_vegas_t), intent(inout) :: this
  integer, intent(in), optional :: n_start, n_increase, nbatch, gridno
```

```
    character(len=*), intent(in), target, optional :: statefile
    if (present (n_start))  this%nstart = n_start
    if (present (n_increase))  this%nincrease = n_increase
    if (present (nbatch))  this%nbatch = nbatch
    if (present (gridno))  this%gridno = gridno
    if (present (statefile))  this%statefile => statefile
  end subroutine cuba_vegas_set_deferred
```

⟨*Muli CUBA: cuba divonne: TBP*⟩≡

```
  procedure :: write_to_marker => cuba_divonne_write_to_marker
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine cuba_divonne_write_to_marker (this, marker, status)
    class(cuba_divonne_t), intent(in) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    call marker%mark_begin ("cuba_divonne_t")
    call cuba_write_to_marker (this, marker, status)
    call marker%mark ("key1", this%key1)
    call marker%mark ("key2", this%key2)
    call marker%mark ("key3", this%key3)
    call marker%mark ("maxpass", this%maxpass)
    call marker%mark ("border", this%border)
    call marker%mark ("maxchisq", this%maxchisq)
    call marker%mark ("mindeviation", this%mindeviation)
    call marker%mark ("ngiven", this%ngiven)
    call marker%mark ("ldxgiven", this%ldxgiven)
    call marker%mark ("nextra", this%nextra)
    call marker%mark ("xgiven", this%xgiven)
    call marker%mark_null ("cuba_divonne_t")
  end subroutine cuba_divonne_write_to_marker
```

⟨*Muli CUBA: cuba divonne: TBP*⟩+≡

```
  procedure :: read_from_marker => cuba_divonne_read_from_marker
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine cuba_divonne_read_from_marker (this, marker, status)
    class(cuba_divonne_t), intent(out) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    call marker%pick_begin ("cuba_divonne_t", status=status)
    call cuba_read_from_marker (this, marker, status)
    call marker%pick ("key1", this%key1, status)
    call marker%pick ("key2", this%key2, status)
    call marker%pick ("key3", this%key3, status)
    call marker%pick ("maxpass", this%maxpass, status)
    call marker%pick ("border", this%border, status)
    call marker%pick ("maxchisq", this%maxchisq, status)
    call marker%pick ("mindeviation", this%mindeviation, status)
    call marker%pick ("ngiven", this%ngiven, status)
    call marker%pick ("ldxgiven", this%ldxgiven, status)
    call marker%pick ("nextra", this%nextra, status)
    if (allocated (this%xgiven))  deallocate (this%xgiven)
    allocate (this%xgiven (this%ldxgiven, this%ngiven))
```

```
    call marker%pick ("xgiven", this%xgiven, status)
    call marker%pick_end ("cuba_divonne_t", status)
  end subroutine cuba_divonne_read_from_marker
```

⟨*Muli CUBA: cuba divonne: TBP*⟩+≡

```
  procedure :: print_to_unit => cuba_divonne_print_to_unit
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine cuba_divonne_print_to_unit (this, unit, parents, components, peers)
    class(cuba_divonne_t), intent(in) :: this
    INTEGER, INTENT(IN) :: unit
    integer(dik), intent(in) :: parents, components, peers
    if (parents > 0)  &
         call cuba_print_to_unit (this, unit, parents-1, components, peers)
    write (unit, "(1x,A)")          "Components of cuba_divonne_t:"
    write (unit, "(3x,A,I10)")      "key1:        ", this%key1
    write (unit, "(3x,A,I10)")      "key2:        ", this%key2
    write (unit, "(3x,A,I10)")      "key3:        ", this%key3
    write (unit, "(3x,A,I10)")      "maxpass:     ", this%maxpass
    write (unit, "(3x,A,I10)")      "ngiven:      ", this%ngiven
    write (unit, "(3x,A,I10)")      "ldxgiven:    ", this%ldxgiven
    write (unit, "(3x,A,I10)")      "nextra:      ", this%nextra
    write (unit, "(3x,A,E10.4)")    "border:      ", this%border
    write (unit, "(3x,A,E10.4)")    "maxchisq:    ", this%maxchisq
    write (unit, "(3x,A,E10.4)")    "mindeviation:", this%mindeviation
    write (unit, "(3x,A,2(E10.4))") "xgiven:      ", this%xgiven
  end subroutine cuba_divonne_print_to_unit
```

⟨*Muli CUBA: cuba divonne: TBP*⟩+≡

```
  procedure, nopass :: get_type => cuba_divonne_get_type
```

⟨*Muli CUBA: procedures*⟩+≡

```
  pure subroutine cuba_divonne_get_type (type)
    character(:), allocatable, intent(out) :: type
    allocate(type, source="cuba_divonne_t")
  end subroutine cuba_divonne_get_type
```

⟨*Muli CUBA: cuba divonne: TBP*⟩+≡

```
  procedure :: integrate_nd => integrate_divonne
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine integrate_divonne (this, integrand)
    class(cuba_divonne_t), intent(inout) :: this
    procedure(integrand_interface) :: integrand
    ! call this%reset_output()
    ! print '("divonne")'
    ! call divonne(this%dim_x, this%dim_f, integrand, this%userdata, &
    !      this%eps_rel, this%eps_abs, this%flags, this%seed, this%min_eval, &
    !      this%max_eval, this%key1, this%key2, this%key3, this%maxpass, &
    !      this%border, this%maxchisq, this%mindeviation, this%ngiven, &
    !      this%ldxgiven, this%xgiven, this%nextra, &
    !      ! this%peakfinder, &
    !      0, this%nregions, this%neval, this%fail, this%integral, &
    !      this%error, this%prob)
```

```
end subroutine integrate_divonne
```

⟨*Muli CUBA: cuba divonne: TBP*⟩+≡

```
procedure :: integrate_userdata => integrate_divonne_userdata
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine integrate_divonne_userdata (this, integrand, userdata)
  class(cuba_divonne_t), intent(inout) :: this
  procedure(integrand_interface) :: integrand
  class(transverse_mom_t), intent(in) :: userdata
  ! call this%reset_output()
  ! print '("divonne")'
  ! call divonne (this%dim_x, this%dim_f, integrand, userdata, &
  !      this%eps_rel, this%eps_abs, this%flags, this%seed, this%min_eval, &
  !      this%max_eval, this%key1, this%key2, this%key3, this%maxpass, &
  !      this%border, this%maxchisq, this%mindeviation, this%ngiven, &
  !      this%ldxgiven, this%xgiven, this%nextra, &
  !      ! this%peakfinder, &
  !      0, this%nregions, this%neval, this%fail, this%integral, &
  !      this%error, this%prob)
end subroutine integrate_divonne_userdata
```

⟨*Muli CUBA: cuba divonne: TBP*⟩+≡

```
procedure :: copy => cuba_divonne_copy
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_divonne_copy (this, source)
  class(cuba_divonne_t), intent(out) :: this
  class(cuba_class), intent(in) :: source
  select type (source)
  class is (cuba_divonne_t)
     call this%copy_common(source)
     call this%set_deferred (source%key1, source%key2, source%key3, &
          source%maxpass, source%border, source%maxchisq, &
          source%mindeviation, source%xgiven)
  class default
     call msg_error ("cuba_divonne_copy: type of source is not " &
          // "type compatible with cuba_divonne_t.")
  end select
end subroutine cuba_divonne_copy
```

⟨*Muli CUBA: cuba divonne: TBP*⟩+≡

```
procedure :: set_deferred => cuba_divonne_set_deferred
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_divonne_set_deferred (this, key1, key2, key3, maxpass, &
     border, maxchisq, mindeviation, xgiven, xgiven_flat)
  class(cuba_divonne_t) :: this
  integer, optional, intent(in) :: key1, key2, key3, maxpass
  real(default), optional, intent(in) :: border, maxchisq, mindeviation
  real(default), dimension(:,:), optional, intent(in) :: xgiven
  real(default), dimension(:), optional, intent(in) :: xgiven_flat
  integer, dimension(2) :: s
  if (present (key1))  this%key1 = key1
```

```
    if (present (key2))  this%key2 = key2
    if (present (key3))  this%key3 = key3
    if (present (maxpass))  this%maxpass = maxpass
    if (present (border))  this%border = border
    if (present (maxchisq))  this%maxchisq = maxchisq
    if (present (mindeviation))  this%mindeviation = mindeviation
    if (present (xgiven)) then
       if (allocated (this%xgiven))  deallocate (this%xgiven)
       s = shape(xgiven)
       if (s(1) == this%dim_x) then
          allocate (this%xgiven (s(1), s(2)), source=xgiven)
          this%ldxgiven = s(1)
          this%ngiven = s(2)
       else
          call msg_error ("cuba_divonne_set_deferred: shape of xgiven " &
               // "is not [dim_x,:].")
          this%ngiven = 0
       end if
    end if
    if (present (xgiven_flat)) then
       if (allocated (this%xgiven))  deallocate (this%xgiven)
       if (mod(size(xgiven_flat), this%dim_x) == 0) then
          this%ngiven = size(xgiven_flat) / this%dim_x
          this%ldxgiven = this%dim_x
          allocate (this%xgiven (this%ldxgiven,this%ngiven))
          this%xgiven = reshape(xgiven_flat, [this%ldxgiven, this%ngiven])
       else
          call msg_error ("cuba_divonne_set_deferred: size of xgiven_flat " &
               // "is no multiple of dim_x.")
          this%ngiven = 0
       end if
    end if
  end subroutine cuba_divonne_set_deferred
```

⟨*Muli CUBA: cuba cuhre: TBP*⟩≡

```
  procedure :: write_to_marker => cuba_cuhre_write_to_marker
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine cuba_cuhre_write_to_marker (this, marker, status)
    class(cuba_cuhre_t), intent(in) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    call marker%mark_begin ("cuba_cuhre_t")
    call cuba_write_to_marker (this, marker, status)
    call marker%mark ("key", this%key)
    call marker%pick_end ("cuba_cuhre_t", status)
  end subroutine cuba_cuhre_write_to_marker
```

⟨*Muli CUBA: cuba cuhre: TBP*⟩+≡

```
  procedure :: read_from_marker => cuba_cuhre_read_from_marker
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine cuba_cuhre_read_from_marker (this, marker, status)
    class(cuba_cuhre_t), intent(out) :: this
```

```
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    call marker%pick_begin ("cuba_cuhre_t", status=status)
    call cuba_read_from_marker (this, marker, status)
    call marker%pick ("key",this%key, status)
    call marker%pick_end ("cuba_cuhre_t", status)
  end subroutine cuba_cuhre_read_from_marker
```

⟨*Muli CUBA: cuba cuhre: TBP*⟩+≡

```
  procedure :: print_to_unit => cuba_cuhre_print_to_unit
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine cuba_cuhre_print_to_unit (this, unit, parents, components, peers)
    class(cuba_cuhre_t), intent(in) :: this
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    if (parents > 0)  &
         call cuba_print_to_unit (this, unit, parents-1, components, peers)
    write (unit, "(1x,A)")      "Components of cuba_cuhre_t:"
    write (unit, "(3x,A,I10)")  "key:       ", this%key
  end subroutine cuba_cuhre_print_to_unit
```

⟨*Muli CUBA: cuba cuhre: TBP*⟩+≡

```
  procedure, nopass :: get_type => cuba_cuhre_get_type
```

⟨*Muli CUBA: procedures*⟩+≡

```
  pure subroutine cuba_cuhre_get_type (type)
    character(:), allocatable, intent(out) :: type
    allocate (type, source="cuba_cuhre_t")
  end subroutine cuba_cuhre_get_type
```

⟨*Muli CUBA: cuba cuhre: TBP*⟩+≡

```
  procedure :: integrate_nd => integrate_cuhre
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine integrate_cuhre (this, integrand)
    class(cuba_cuhre_t), intent(inout) :: this
    procedure(integrand_interface) :: integrand
    ! print '("cuhre")'
    ! call cuhre(this%dim_x, this%dim_f, integrand, this%userdata, &
    !      this%eps_rel, this%eps_abs, this%flags, &
    !      ! this%seed, &
    !      this%min_eval, this%max_eval, this%key, this%nregions, &
    !      this%neval, this%fail, this%integral, this%error, this%prob)
  end subroutine integrate_cuhre
```

⟨*Muli CUBA: cuba cuhre: TBP*⟩+≡

```
  procedure :: integrate_userdata => integrate_cuhre_userdata
```

⟨*Muli CUBA: procedures*⟩+≡

```
  subroutine integrate_cuhre_userdata (this, integrand, userdata)
    class(cuba_cuhre_t), intent(inout) :: this
    procedure(integrand_interface) :: integrand
    class(transverse_mom_t), intent(in) :: userdata
```

```
  ! print '("cuhre")'
  ! call cuhre(this%dim_x, this%dim_f, integrand, userdata, this%eps_rel, &
  !      this%eps_abs, this%flags, &
  !      ! this%seed, &
  !      this%min_eval, this%max_eval, this%key, this%nregions, &
  !      this%neval, this%fail, this%integral, this%error, this%prob)
end subroutine integrate_cuhre_userdata
```

⟨*Muli CUBA: cuba cuhre: TBP*⟩+≡

```
procedure :: copy => cuba_cuhre_copy
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_cuhre_copy (this, source)
  class(cuba_cuhre_t), intent(out) :: this
  class(cuba_class), intent(in) :: source
  select type (source)
  class is (cuba_cuhre_t)
     call this%copy_common (source)
     this%key = source%key
  class default
     call msg_error ("cuba_cuhre_copy: type of source is not type " &
          // "compatible with cuba_cuhre_t.")
  end select
end subroutine cuba_cuhre_copy
```

⟨*Muli CUBA: cuba cuhre: TBP*⟩+≡

```
procedure :: set_deferred => cuba_cuhre_set_deferred
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_cuhre_set_deferred (this, key)
  class(cuba_cuhre_t), intent(inout) :: this
  integer, intent(in) :: key
  this%key = key
end subroutine cuba_cuhre_set_deferred
```

⟨*Muli CUBA: cuba suave: TBP*⟩≡

```
procedure :: write_to_marker => cuba_suave_write_to_marker
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_suave_write_to_marker (this, marker, status)
  class(cuba_suave_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%mark_begin ("cuba_suave_t")
  call cuba_write_to_marker (this, marker, status)
  call marker%mark ("nnew", this%nnew)
  call marker%mark ("flatness", this%flatness)
  call marker%mark_null ("cuba_suave_t")
end subroutine cuba_suave_write_to_marker
```

⟨*Muli CUBA: cuba suave: TBP*⟩+≡

```
procedure :: read_from_marker => cuba_suave_read_from_marker
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_suave_read_from_marker (this, marker, status)
  class(cuba_suave_t), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%pick_begin ("cuba_suave_t", status=status)
  call cuba_read_from_marker (this, marker, status)
  call marker%pick ("nnew", this%nnew, status)
  call marker%pick ("flatnes", this%flatness, status)
  call marker%pick_end ("cuba_suave_t", status)
end subroutine cuba_suave_read_from_marker
```

⟨*Muli CUBA: cuba suave: TBP*⟩+≡

```
procedure::print_to_unit=>cuba_suave_print_to_unit
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_suave_print_to_unit (this, unit, parents, components, peers)
  class(cuba_suave_t), intent(in) :: this
  integer, intent(in) :: unit
  integer(dik), intent(in) :: parents, components, peers
  if (parents > 0) &
       call cuba_print_to_unit (this, unit, parents-1, components, peers)
  write (unit, "(1x,A)")     "Components of cuba_suave_t:"
  write (unit, "(3x,A,I10)") "nnew:      ", this%nnew
  write (unit, "(3x,A,I10)") "flatness:  ", this%flatness
end subroutine cuba_suave_print_to_unit
```

⟨*Muli CUBA: cuba suave: TBP*⟩+≡

```
procedure, nopass :: get_type => cuba_suave_get_type
```

⟨*Muli CUBA: procedures*⟩+≡

```
pure subroutine cuba_suave_get_type (type)
  character(:), allocatable, intent(out) :: type
  allocate (type, source="cuba_suave_t")
end subroutine cuba_suave_get_type
```

⟨*Muli CUBA: cuba suave: TBP*⟩+≡

```
procedure :: integrate_nd => integrate_suave
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine integrate_suave (this, integrand)
  class(cuba_suave_t), intent(inout) :: this
  procedure(integrand_interface) :: integrand
  ! print '("suave")'
  ! call suave(this%dim_x, this%dim_f, integrand, this%userdata, &
  !      this%eps_rel, this%eps_abs, this%flags, this%seed, &
  !      this%min_eval, this%max_eval, this%nnew, this%flatness, &
  !      this%nregions, this%neval, this%fail, this%integral, &
  !      this%error, this%prob)
end subroutine integrate_suave
```

⟨*Muli CUBA: cuba suave: TBP*⟩+≡

```
procedure :: integrate_userdata => integrate_suave_userdata
```

⟨*Muli CUBA: procedures*⟩+≡

```
 subroutine integrate_suave_userdata (this, integrand, userdata)
  class(cuba_suave_t), intent(inout) :: this
  procedure(integrand_interface) :: integrand
  class(transverse_mom_t), intent(in) :: userdata
  ! print '("suave")'
  ! call suave (this%dim_x, this%dim_f, integrand, userdata, &
  !      this%eps_rel, this%eps_abs, this%flags, this%seed, &
  !      this%min_eval, this%max_eval, this%nnew, this%flatness, &
  !      this%nregions, this%neval, this%fail, this%integral, &
  !      this%error, this%prob)
end subroutine integrate_suave_userdata
```

⟨*Muli CUBA: cuba suave: TBP*⟩+≡

```
procedure :: copy => cuba_suave_copy
```

⟨*Muli CUBA: procedures*⟩+≡

```
subroutine cuba_suave_copy (this, source)
  class(cuba_suave_t), intent(out) :: this
  class(cuba_class), intent(in) :: source
  select type (source)
  class is (cuba_suave_t)
     call this%copy_common (source)
     this%nnew = source%nnew
     this%flatness = source%flatness
  class default
     call msg_error ("cuba_suave_copy: type of source is not type " &
          // "compatible with cuba_suave_t.")
  end select
end subroutine cuba_suave_copy
```

⟨*Muli CUBA: interfaces*⟩≡

```
interface
   subroutine integrand_interface (dim_x, x, dim_f, f,userdata)
   ⟨Use kinds⟩
     use muli_momentum
     integer, intent(in) :: dim_x, dim_f
     real(default), dimension(dim_x), intent(in) :: x
     real(default), dimension(dim_f), intent(out) :: f
     class(transverse_mom_t), intent(in) :: userdata
   end subroutine integrand_interface
end interface
```

⟨*Muli CUBA: interfaces*⟩+≡

```
interface
   subroutine cuba_copy_interface (this, source)
     import :: cuba_class
     class(cuba_class), intent(out) :: this
     class(cuba_class), intent(in) :: source
   end subroutine cuba_copy_interface
end interface
```

⟨*Muli CUBA: interfaces*⟩+≡

```
interface
```

```
      subroutine ca_plain (this)
        import :: cuba_class
        class(cuba_class) :: this
      end subroutine ca_plain
   end interface
```

⟨*Muli CUBA: interfaces*⟩+≡

```
   interface
      subroutine integrate_interface (this, integrand)
        import :: cuba_class
        class(cuba_class), intent(inout) :: this
        interface
           subroutine integrand (dim_x, x, dim_f, f,userdata)
           ⟨Use kinds⟩
             use muli_momentum
             integer, intent(in) :: dim_x, dim_f
             real(default), dimension(dim_x), intent(in) :: x
             real(default), dimension(dim_f), intent(out) :: f
             class(transverse_mom_t), intent(in) :: userdata
           end subroutine integrand
        end interface
      end subroutine integrate_interface
   end interface
```

⟨*Muli CUBA: interfaces*⟩+≡

```
   interface
      subroutine integrate_userdata_interface (this, integrand,userdata)
        use muli_momentum
        import :: cuba_class
        class(cuba_class), intent(inout) :: this
        interface
           subroutine integrand (dim_x, x, dim_f, f,userdata)
           ⟨Use kinds⟩
             use muli_momentum
             integer, intent(in) :: dim_x, dim_f
             real(default), dimension(dim_x), intent(in) :: x
             real(default), dimension(dim_f), intent(out) :: f
             class(transverse_mom_t), intent(in) :: userdata
           end subroutine integrand
        end interface
        class(transverse_mom_t), intent(in) :: userdata
      end subroutine integrate_userdata_interface
   end interface
```

22.5 Trapezoidal integration routines

This file contains the module "muli_trapezium". The name is derived from the trapezoidal integration rule. The purpose of this module is to define a binary tree muli_trapezium_tree_t which holds a probability function in terms of trapezoidal segments. Its leaves of type muli_trapezium_list_t are connected to form a list, so you can either walk the root function back and forth or pick a certain segment in logarithmic time by walking down the tree. All nodes extend

muli_trapezium_t which holds the actual values. Those are the values of the density function, the integral from this segment to the end of the integration area and a probability function calculated from these values. All values are taken at the upper "right" bound of the segment. Additionally the differences of these values to the values of the left neighbor is stored.

⟨muli_trapezium.f90⟩≡
```
  ⟨File header⟩

  module muli_trapezium
    use, intrinsic :: iso_fortran_env, only: output_unit
  ⟨Use kinds⟩
    use constants
    use diagnostics
    use muli_base

  ⟨Standard module head⟩

  ⟨Muli trapezium: variables⟩

  ⟨Muli trapezium: public⟩

  ⟨Muli trapezium: types⟩

  ⟨Muli trapezium: interfaces⟩

  contains

  ⟨Muli trapezium: procedures⟩

  end module muli_trapezium
```

⟨Muli trapezium: variables⟩≡
```
    integer, private, parameter :: value_dimension = 7
    integer, private, parameter :: r_value_index = 1
    integer, private, parameter :: d_value_index = 2
    integer, private, parameter :: r_integral_index = 3
    integer, private, parameter :: d_integral_index = 4
    integer, private, parameter :: r_probability_index = 5
    integer, private, parameter :: d_probability_index = 6
    integer, private, parameter :: error_index = 7
```

⟨Muli trapezium: interfaces⟩≡
```
    abstract interface
       subroutine muli_trapezium_append_interface (this, right)
         import muli_trapezium_node_class_t
         class(muli_trapezium_node_class_t), intent(inout), target :: this, right
       end subroutine muli_trapezium_append_interface
    end interface
```

⟨Muli trapezium: interfaces⟩+≡
```
    abstract interface
       subroutine muli_trapezium_final_interface (this)
         import muli_trapezium_node_class_t
         class(muli_trapezium_node_class_t), intent(inout) :: this
```

```
      end subroutine muli_trapezium_final_interface
   end interface
```

This is the base type `muli_trapezium_t`. Its component `values` has a first index is in $\{0, \ldots, \dim - 1\}$, while the second index distingiushes between `r_value`, `d_value`, `r_integral`, `d_integral`, `r_probability`, `d_probability`.

⟨*Muli trapezium: public*⟩≡
```
   public :: muli_trapezium_t
```

⟨*Muli trapezium: types*⟩≡
```
   type, extends (measure_class_t) :: muli_trapezium_t
      private
      integer :: dim = 0
      real(default) :: r_position = 0
      real(default) :: d_position = 0
      real(default) :: measure_comp = 0
      real(default), dimension(:,:), allocatable :: values
    contains
    ⟨Muli trapezium: trapezium: TBP⟩
   end type muli_trapezium_t
```

⟨*Muli trapezium: public*⟩+≡
```
   public :: muli_trapezium_node_class_t
```

⟨*Muli trapezium: types*⟩+≡
```
   type, extends (muli_trapezium_t), abstract :: muli_trapezium_node_class_t
      private
      class(muli_trapezium_node_class_t), pointer :: left => null()
      class(muli_trapezium_node_class_t), pointer :: right => null()
      ! real(default) :: criterion
    contains
    ⟨Muli trapezium: node: TBP⟩
   end type muli_trapezium_node_class_t
```

⟨*Muli trapezium: public*⟩+≡
```
   public :: muli_trapezium_tree_t
```

⟨*Muli trapezium: types*⟩+≡
```
   type, extends(muli_trapezium_node_class_t) :: muli_trapezium_tree_t
      class(muli_trapezium_node_class_t), pointer :: down => null()
    contains
    ⟨Muli trapezium: tree: TBP⟩
   end type muli_trapezium_tree_t
```

⟨*Muli trapezium: public*⟩+≡
```
   public :: muli_trapezium_list_t
```

⟨*Muli trapezium: types*⟩+≡
```
   type, extends (muli_trapezium_node_class_t) :: muli_trapezium_list_t
    contains
    ⟨Muli trapezium: list: TBP⟩
   end type muli_trapezium_list_t
```

⟨*Muli trapezium: trapezium: TBP*⟩≡

```
procedure :: write_to_marker => muli_trapezium_write_to_marker
```

⟨*Muli trapezium: procedures*⟩≡

```
  subroutine muli_trapezium_write_to_marker (this,marker,status)
  class(muli_trapezium_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  integer::dim
  call marker%mark_begin ("muli_trapezium_t")
  call marker%mark ("dim", this%dim)
  call marker%mark ("r_position", this%r_position)
  call marker%mark ("d_position", this%d_position)
  if (allocated(this%values)) then
     call marker%mark ("values", this%values)
  else
     call marker%mark_null ("values")
  end if
  call marker%mark_end ("muli_trapezium_t")
end subroutine muli_trapezium_write_to_marker
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: read_from_marker => muli_trapezium_read_from_marker
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_read_from_marker (this,marker,status)
  class(muli_trapezium_t), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  integer :: dim
  call marker%pick_begin ("muli_trapezium_t", status=status)
  call marker%pick ("dim", this%dim,status)
  call marker%pick ("r_position", this%r_position, status)
  call marker%pick ("d_position", this%d_position, status)
  if (allocated (this%values))  deallocate (this%values)
  call marker%verify_nothing ("values", status)
  if (status == serialize_ok) then
     allocate(this%values(0:this%dim-1,7))
     call marker%pick ("values",this%values, status)
  end if
  call marker%pick_end("muli_trapezium_t",status)
end subroutine muli_trapezium_read_from_marker
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: print_to_unit => muli_trapezium_print_to_unit
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_print_to_unit (this, unit, parents, components, peers)
  class(muli_trapezium_t), intent(in) :: this
  integer, intent(in) :: unit
  integer(dik), intent(in) :: parents, components, peers
  write (unit, "(1x,A)") "Components of muli_trapezium_t:"
  write (unit, fmt=*)"Dimension:        ",this%dim
  write (unit,fmt=*)"Right position:   ",this%r_position
  write (unit,fmt=*)"Position step:    ",this%d_position
```

```
    if (allocated(this%values)) then
       if (components>0) then
          write (unit,fmt=*)"Right values:     ",muli_trapezium_get_r_value_array(this)
          write (unit,fmt=*) "Value step:       ", this%get_d_value()
          write (unit,fmt=*)"Right integrals:  ",this%get_r_integral()
          write (unit,fmt=*)"Integral step:    ",this%get_d_integral()
          write (unit,fmt=*)"Right propabilities:",this%get_r_probability()
          write (unit,fmt=*)"Probability step: ",this%get_d_probability()
          write (unit,fmt=*)"Errors:           ",this%get_error()
       else
          write (unit, "(3x,A)") "Values are allocated."
       end if
    else
       write (unit, "(3x,A)") "Values are not allocated."
    end if
  end subroutine muli_trapezium_print_to_unit
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  procedure, nopass :: get_type => muli_trapezium_get_type
```

⟨*Muli trapezium: procedures*⟩+≡

```
  pure subroutine muli_trapezium_get_type (type)
    character(:),allocatable, intent(out) :: type
    allocate (type, source="muli_trapezium_t")
  end subroutine muli_trapezium_get_type
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  procedure, nopass :: verify_type => muli_trapezium_verify_type
```

⟨*Muli trapezium: procedures*⟩+≡

```
  elemental logical function muli_trapezium_verify_type (type) result (match)
    character(*), intent(in) :: type
    match = type == "muli_trapezium_t"
  end function muli_trapezium_verify_type
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  procedure :: measure => muli_trapezium_measure
```

⟨*Muli trapezium: procedures*⟩+≡

```
  elemental function muli_trapezium_measure (this)
    class(muli_trapezium_t), intent(in) :: this
    real(default) :: muli_trapezium_measure
    muli_trapezium_measure = this%measure_comp
  end function muli_trapezium_measure
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  procedure :: initialize => muli_trapezium_initialize
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_initialize (this, dim, r_position, d_position)
    class(muli_trapezium_t), intent(inout) :: this
    integer, intent(in) :: dim
    real(default), intent(in) :: r_position, d_position
    integer :: dim1, dim2
```

```
    this%dim = dim
    this%r_position = r_position
    this%d_position = d_position
    if (allocated (this%values))  deallocate (this%values)
    allocate (this%values(0:dim-1,value_dimension))
    do dim2 = 1, value_dimension-1
       do dim1 = 0, dim-1
          this%values(dim1,dim2) = zero
       end do
    end do
    do dim1 = 0, dim-1
       this%values(dim1, value_dimension) = huge(one)
    end do
    this%measure_comp = huge(one)
  end subroutine muli_trapezium_initialize
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  procedure :: get_dimension => muli_trapezium_get_dimension
```

⟨*Muli trapezium: procedures*⟩+≡

```
  elemental function muli_trapezium_get_dimension (this) result (dim)
    class(muli_trapezium_t), intent(in) :: this
    integer :: dim
    dim = this%dim
  end function muli_trapezium_get_dimension
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  procedure :: get_l_position => muli_trapezium_get_l_position
```

⟨*Muli trapezium: procedures*⟩+≡

```
  pure function muli_trapezium_get_l_position (this) result (pos)
    class(muli_trapezium_t), intent(in) :: this
    real(default) :: pos
    pos = this%r_position - this%d_position
  end function muli_trapezium_get_l_position
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  procedure :: get_r_position => muli_trapezium_get_r_position
```

⟨*Muli trapezium: procedures*⟩+≡

```
  pure function muli_trapezium_get_r_position (this) result (pos)
    class(muli_trapezium_t), intent(in) :: this
    real(default) :: pos
    pos = this%r_position
  end function muli_trapezium_get_r_position
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  procedure :: get_d_position => muli_trapezium_get_d_position
```

⟨*Muli trapezium: procedures*⟩+≡

```
  pure function muli_trapezium_get_d_position (this) result (pos)
    class(muli_trapezium_t), intent(in) :: this
    real(default) :: pos
    pos = this%d_position
```

```
end function muli_trapezium_get_d_position
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
generic :: get_l_value => get_l_value_array, get_l_value_element
procedure :: get_l_value_array => muli_trapezium_get_l_value_array
procedure :: get_l_value_element => muli_trapezium_get_l_value_element
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_get_l_value_array (this) result (subarray)
  class(muli_trapezium_t), intent(in) :: this
  real(default), dimension(this%dim) :: subarray
  subarray = this%values(0:this%dim-1, r_value_index) - &
       this%values(0:this%dim-1, d_value_index)
end function muli_trapezium_get_l_value_array
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_get_l_value_element (this, set) result (element)
  class(muli_trapezium_t), intent(in) :: this
  integer, intent(in) :: set
  real(default) :: element
  element = this%values(set, r_value_index) - this%values(set, d_value_index)
end function muli_trapezium_get_l_value_element
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
generic :: get_r_value => get_r_value_array, get_r_value_element
procedure :: get_r_value_array => muli_trapezium_get_r_value_array
procedure :: get_r_value_element => muli_trapezium_get_r_value_element
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_get_r_value_element (this, set) result (element)
  class(muli_trapezium_t), intent(in) :: this
  integer, intent(in) :: set
  real(default) :: element
  element = this%values (set, r_value_index)
end function muli_trapezium_get_r_value_element
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_get_r_value_array (this) result (subarray)
  class(muli_trapezium_t), intent(in) :: this
  real(default), dimension(this%dim) :: subarray
  subarray = this%values(0:this%dim-1, r_value_index)
end function muli_trapezium_get_r_value_array
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
generic :: get_d_value => get_d_value_array, get_d_value_element
procedure :: get_d_value_array => muli_trapezium_get_d_value_array
procedure :: get_d_value_element => muli_trapezium_get_d_value_element
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_get_d_value_element (this, set) result (element)
  class(muli_trapezium_t), intent(in) :: this
  integer, intent(in) :: set
  real(default) :: element
```

```
    element=this%values (set, d_value_index)
  end function muli_trapezium_get_d_value_element
```

⟨*Muli trapezium: procedures*⟩+≡

```
  pure function muli_trapezium_get_d_value_array (this) result (subarray)
    class(muli_trapezium_t), intent(in) :: this
    real(default), dimension(this%dim) :: subarray
    subarray = this%values(0:this%dim-1, d_value_index)
  end function muli_trapezium_get_d_value_array
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  generic :: get_l_integral => get_l_integral_array, get_l_integral_element
  procedure :: get_l_integral_array => muli_trapezium_get_l_integral_array
  procedure :: get_l_integral_element => muli_trapezium_get_l_integral_element
```

⟨*Muli trapezium: procedures*⟩+≡

```
  pure function muli_trapezium_get_l_integral_element &
       (this, set) result (element)
    class(muli_trapezium_t), intent(in) :: this
    integer, intent(in) :: set
    real(default) :: element
    element = this%values (set, r_integral_index) - &
         this%values (set, d_integral_index)
  end function muli_trapezium_get_l_integral_element
```

⟨*Muli trapezium: procedures*⟩+≡

```
  pure function muli_trapezium_get_l_integral_array (this) result (subarray)
    class(muli_trapezium_t), intent(in) :: this
    real(default), dimension(this%dim) :: subarray
    subarray = this%values (0:this%dim-1, r_integral_index) - &
         this%values (0:this%dim-1, d_integral_index)
  end function muli_trapezium_get_l_integral_array
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  generic :: get_r_integral => get_r_integral_array, get_r_integral_element
  procedure :: get_r_integral_array => muli_trapezium_get_r_integral_array
  procedure :: get_r_integral_element => muli_trapezium_get_r_integral_element
```

⟨*Muli trapezium: procedures*⟩+≡

```
  pure function muli_trapezium_get_r_integral_element (this, set) result (element)
    class(muli_trapezium_t), intent(in) :: this
    integer, intent(in) :: set
    real(default) :: element
    element = this%values (set, r_integral_index)
  end function muli_trapezium_get_r_integral_element
```

⟨*Muli trapezium: procedures*⟩+≡

```
  pure function muli_trapezium_get_r_integral_array (this) result (subarray)
    class(muli_trapezium_t), intent(in) :: this
    real(default), dimension(this%dim) :: subarray
    subarray = this%values (0:this%dim-1, r_integral_index)
  end function muli_trapezium_get_r_integral_array
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡
```
generic :: get_d_integral => get_d_integral_array, get_d_integral_element
procedure :: get_d_integral_array => muli_trapezium_get_d_integral_array
procedure :: get_d_integral_element => muli_trapezium_get_d_integral_element
```
⟨*Muli trapezium: procedures*⟩+≡
```
pure function muli_trapezium_get_d_integral_element &
     (this, set) result (element)
  class(muli_trapezium_t), intent(in) :: this
  integer, intent(in) :: set
  real(default) :: element
  element = this%values (set, d_integral_index)
end function muli_trapezium_get_d_integral_element
```

⟨*Muli trapezium: procedures*⟩+≡
```
pure function muli_trapezium_get_d_integral_array (this) result (subarray)
  class(muli_trapezium_t), intent(in) :: this
  real(default), dimension(this%dim) :: subarray
  subarray = this%values (0:this%dim-1, d_integral_index)
end function muli_trapezium_get_d_integral_array
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡
```
generic :: get_l_probability => &
     get_l_probability_array, get_l_probability_element
procedure :: get_l_probability_element => &
     muli_trapezium_get_l_probability_element
procedure :: get_l_probability_array => &
     muli_trapezium_get_l_probability_array
```
⟨*Muli trapezium: procedures*⟩+≡
```
pure function muli_trapezium_get_l_probability_element &
     (this, set) result (element)
  class(muli_trapezium_t), intent(in) :: this
  integer, intent(in) :: set
  real(default) :: element
  element = this%values (set, r_probability_index) - &
       this%values (set, d_probability_index)
end function muli_trapezium_get_l_probability_element
```

⟨*Muli trapezium: procedures*⟩+≡
```
pure function muli_trapezium_get_l_probability_array (this) result (subarray)
  class(muli_trapezium_t), intent(in) :: this
  real(default), dimension(this%dim) :: subarray
  subarray = this%values (0:this%dim-1, r_probability_index) - &
       this%values (0:this%dim-1, d_probability_index)
end function muli_trapezium_get_l_probability_array
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡
```
generic :: get_r_probability => &
     get_r_probability_array, get_r_probability_element
procedure :: get_r_probability_element => &
     muli_trapezium_get_r_probability_element
procedure :: get_r_probability_array => &
     muli_trapezium_get_r_probability_array
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_get_r_probability_element &
     (this, set) result (element)
  class(muli_trapezium_t), intent(in) :: this
  integer, intent(in) :: set
  real(default) :: element
  element = this%values (set, r_probability_index)
end function muli_trapezium_get_r_probability_element
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_get_r_probability_array (this) result (subarray)
  class(muli_trapezium_t), intent(in) :: this
  real(default), dimension(this%dim) :: subarray
  subarray = this%values (0:this%dim-1, r_probability_index)
end function muli_trapezium_get_r_probability_array
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
generic :: get_d_probability => &
     get_d_probability_array, get_d_probability_element
procedure :: get_d_probability_element => &
     muli_trapezium_get_d_probability_element
procedure :: get_d_probability_array => &
     muli_trapezium_get_d_probability_array
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_get_d_probability_array (this) result (subarray)
  class(muli_trapezium_t), intent(in) :: this
  real(default), dimension(this%dim) :: subarray
  subarray = this%values (0:this%dim-1, d_probability_index)
end function muli_trapezium_get_d_probability_array
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_get_d_probability_element &
     (this, set) result (element)
  class(muli_trapezium_t), intent(in) :: this
  integer, intent(in) :: set
  real(default) :: element
  element = this%values (set, d_probability_index)
end function muli_trapezium_get_d_probability_element
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: get_error => muli_trapezium_get_error
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_get_error_sum (this) result (error)
  class(muli_trapezium_t), intent(in) :: this
  real(default) :: error
  error = sum (this%values (0:this%dim-1, error_index))
end function muli_trapezium_get_error_sum
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: get_error_sum => muli_trapezium_get_error_sum
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_get_error (this) result (error)
  class(muli_trapezium_t), intent(in) :: this
  real(default), dimension(this%dim) :: error
  error = this%values (0:this%dim-1, error_index)
end function muli_trapezium_get_error
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: get_integral_sum => muli_trapezium_get_integral_sum
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_get_integral_sum (this) result (error)
  class(muli_trapezium_t), intent(in) :: this
  real(default) :: error
  error = sum (this%values (0:this%dim-1, d_integral_index))
end function muli_trapezium_get_integral_sum
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: get_value_at_position => muli_trapezium_get_value_at_position
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_get_value_at_position (this, pos, subarray)
  class(muli_trapezium_t), intent(in) :: this
  real(default), intent(in) :: pos
  real(default), dimension(this%dim), intent(out) :: subarray
  subarray = this%get_r_value_array() - this%get_d_value() * &
       this%d_position / (this%r_position-pos)
end subroutine muli_trapezium_get_value_at_position
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: set_r_value => muli_trapezium_set_r_value
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_set_r_value (this, subarray)
  class(muli_trapezium_t), intent(inout) :: this
  real(default), intent(in), dimension(0:this%dim-1) :: subarray
  this%values(0:this%dim-1, r_value_index) = subarray
end subroutine muli_trapezium_set_r_value
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: set_d_value => muli_trapezium_set_d_value
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_set_d_value (this, subarray)
  class(muli_trapezium_t), intent(inout) :: this
  real(default), intent(in), dimension(0:this%dim-1) :: subarray
  this%values(0:this%dim-1,d_value_index) = subarray
end subroutine muli_trapezium_set_d_value
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: set_r_integral => muli_trapezium_set_r_integral
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_set_r_integral (this, subarray)
  class(muli_trapezium_t), intent(inout) :: this
  real(default), intent(in), dimension(0:this%dim-1) :: subarray
  this%values(0:this%dim-1,r_integral_index) = subarray
end subroutine muli_trapezium_set_r_integral
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: set_d_integral => muli_trapezium_set_d_integral
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_set_d_integral (this, subarray)
  class(muli_trapezium_t), intent(inout) :: this
  real(default), intent(in), dimension(0:this%dim-1) :: subarray
  this%values (0:this%dim-1, d_integral_index) = subarray
end subroutine muli_trapezium_set_d_integral
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: set_r_probability => muli_trapezium_set_r_probability
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_set_r_probability (this, subarray)
  class(muli_trapezium_t), intent(inout) :: this
  real(default), intent(in), dimension(0:this%dim-1) :: subarray
  this%values (0:this%dim-1,r_probability_index) = subarray
end subroutine muli_trapezium_set_r_probability
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: set_d_probability => muli_trapezium_set_d_probability
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_set_d_probability (this, subarray)
  class(muli_trapezium_t), intent(inout) :: this
  real(default), intent(in), dimension(0:this%dim-1) :: subarray
  this%values (0:this%dim-1,d_probability_index) = subarray
end subroutine muli_trapezium_set_d_probability
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: set_error => muli_trapezium_set_error
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_set_error (this, subarray)
  class(muli_trapezium_t), intent(inout) :: this
  real(default), intent(in), dimension(0:this%dim-1) :: subarray
  this%values (0:this%dim-1, error_index) = subarray
  this%measure_comp = sum (subarray)
end subroutine muli_trapezium_set_error
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: is_left_of => muli_trapezium_is_left_of
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_is_left_of (this, that) result (is_left)
  logical :: is_left
  class(muli_trapezium_t), intent(in) :: this, that
  is_left = this%r_position <= that%r_position   !-that%d_position
  ! if (is_left.and.that%r_position < this%r_position) then
  !    print *,"!"
  !    STOP
  ! end if
end function muli_trapezium_is_left_of
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: includes => muli_trapezium_includes
```

⟨*Muli trapezium: procedures*⟩+≡

```
elemental logical function muli_trapezium_includes &
     (this, dim, position, value, integral, probability) result (includes)
  class(muli_trapezium_t), intent(in) :: this
  integer, intent(in) :: dim
  real(default), intent(in),optional :: position, value, integral, probability
  includes = .true.
  if (present (position)) then
     if (this%get_l_position() > position .or. &
         position >= this%get_r_position())  includes = .false.
  end if
  if (present (value)) then
     if (this%get_l_value(dim) > value .or. value >= &
          this%get_r_value(dim))  includes = .false.
  end if
  if (present (integral)) then
     if (this%get_l_integral(dim) > integral .or. integral >= &
          this%get_r_integral(dim))  includes = .false.
  end if
  if (present (probability)) then
     if (this%get_l_probability(dim) > probability .or. &
          probability >= this%get_r_probability(dim))  includes = .false.
  end if
end function muli_trapezium_includes
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: to_node => muli_trapezium_to_node
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_to_node (this, value, list, tree)
  class(muli_trapezium_t), intent(in) :: this
  real(default), intent(in) :: value
  ! class(muli_trapezium_node_class_t), optional, pointer, intent(out) :: node
  class(muli_trapezium_list_t), optional, pointer, intent(out) :: list
  class(muli_trapezium_tree_t), optional, pointer, intent(out) :: tree
  ! if (present (node)) then
  !    allocate (node)
  !    node%dim = this%dim
  !    node%r_position = this%r_position
  !    node%d_position = this%d_position
```

```
    !    allocate (node%values (this%dim, value_dimension), source=this%values)
    ! end if
    if (present (list)) then
       allocate (list)
       list%dim = this%dim
       list%r_position = this%r_position
       list%d_position = this%d_position
       allocate (list%values (0:this%dim-1, value_dimension), source=this%values)
    end if
    if (present (tree)) then
       allocate (tree)
       tree%dim = this%dim
       tree%r_position = this%r_position
       tree%d_position = this%d_position
       allocate (tree%values (0:this%dim-1, value_dimension), source=this%values)
    end if
  end subroutine muli_trapezium_to_node
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  procedure :: sum_up => muli_trapezium_sum_up
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_sum_up (this)
    class(muli_trapezium_t), intent(inout) :: this
    integer :: i
    if (allocated (this%values)) then
       do i = 1, 7
          this%values (0,i) = sum (this%values (1:this%dim-1,i))
       end do
    end if
  end subroutine muli_trapezium_sum_up
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  procedure :: approx_value => muli_trapezium_approx_value
```

⟨*Muli trapezium: procedures*⟩+≡

```
  pure function muli_trapezium_approx_value (this, x) result (val)
    ! returns the values at x
    class(muli_trapezium_t), intent(in) :: this
    real(default), dimension(this%dim) :: val
    real(default), intent(in) :: x
    val = this%get_r_value_array() + (x - this%r_position) * &
         this%get_d_value() / this%d_position
  end function muli_trapezium_approx_value
```

This function returns the value at `x`.

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  procedure :: approx_value_n => muli_trapezium_approx_value_n
```

⟨*Muli trapezium: procedures*⟩+≡

```
  elemental function muli_trapezium_approx_value_n (this, x, n) result (val)
    class(muli_trapezium_t), intent(in) :: this
    real(default) :: val
    real(default), intent(in) :: x
```

```
    integer, intent(in) :: n
    val = this%get_r_value_element(n) + (x - this%r_position) * &
         this%get_d_value_element(n) / this%d_position
end function muli_trapezium_approx_value_n
```

This function returns the integral from x to r_position.

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: approx_integral => muli_trapezium_approx_integral
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure function muli_trapezium_approx_integral (this, x)
  class(muli_trapezium_t), intent(in) :: this
  real(default), dimension(this%dim) :: muli_trapezium_approx_integral
  real(default), intent(in) :: x
  muli_trapezium_approx_integral = &
       ! this%get_r_integral()+&
       ! (this%r_position-x)*this%get_r_value()+&
       ! (x**2-this%r_position**2)*this%get_d_integral()/(this%d_position*2D0)
       this%get_r_integral() + &
       ((this%r_position - x) * &
       (-this%get_d_value() * (this%r_position - x) + 2 * &
       this%d_position*this%get_r_value_array())) / &
       (2 * this%d_position)
end function muli_trapezium_approx_integral
```

This function returns the integral from x to r_position.

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: approx_integral_n => muli_trapezium_approx_integral_n
```

⟨*Muli trapezium: procedures*⟩+≡

```
elemental function muli_trapezium_approx_integral_n (this, x, n) result (val)
  class(muli_trapezium_t), intent(in) :: this
  real(default) :: val
  real(default), intent(in) :: x
  integer, intent(in) :: n
  val = this%get_r_integral_element (n) + ((this%r_position - x) * &
       (-this%get_d_value_element (n) * (this%r_position - x) + 2 * &
       this%d_position * this%get_r_value_element (n))) / &
       (2 * this%d_position)
end function muli_trapezium_approx_integral_n
```

This function returns the values at x.

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
procedure :: approx_probability => muli_trapezium_approx_probability
```

⟨*Muli trapezium: procedures*⟩+≡

```
 pure function muli_trapezium_approx_probability (this, x) result (prop)
  class(muli_trapezium_t), intent(in) :: this
  real(default), dimension(this%dim) :: prop
  real(default), intent(in) :: x
  prop = exp (- this%approx_integral (x))
end function muli_trapezium_approx_probability
```

This function returns the integral from x to r_position.

⟨*Muli trapezium: trapezium: TBP*⟩+≡
```
procedure :: approx_probability_n => muli_trapezium_approx_probability_n
```

⟨*Muli trapezium: procedures*⟩+≡
```
elemental function muli_trapezium_approx_probability_n (this, x, n) result (val)
  class(muli_trapezium_t), intent(in) :: this
  real(default) :: val
  real(default), intent(in) :: x
  integer, intent(in) :: n
  val = exp (- this%approx_integral_n (x, n))
end function muli_trapezium_approx_probability_n
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡
```
procedure :: approx_position_by_integral => &
     muli_trapezium_approx_position_by_integral
```

⟨*Muli trapezium: procedures*⟩+≡
```
elemental function muli_trapezium_approx_position_by_integral &
     (this, dim, int) result (val)
  class(muli_trapezium_t), intent(in) :: this
  real(default) :: val
  integer, intent(in) :: dim
  real(default), intent(in) :: int
  real(default) :: dpdv
  dpdv = (this%d_position / this%values (dim,d_value_index))
  val = this%r_position - dpdv * (this%values (dim, r_value_index) - &
         sqrt (((this%values (dim, r_integral_index) - int) * two / dpdv) + &
         this%values (dim, r_value_index)**2))
end function muli_trapezium_approx_position_by_integral
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡
```
! procedure :: choose_partons => muli_trapezium_choose_partons
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡
```
procedure :: split => muli_trapezium_split
```

⟨*Muli trapezium: procedures*⟩+≡
```
subroutine muli_trapezium_split (this, c_value, c_position, new_node)
  class(muli_trapezium_t), intent(inout) :: this
  real(default), intent(in) :: c_position
  real(default), intent(in), dimension(this%dim) :: c_value
  class(muli_trapezium_t), intent(out), pointer :: new_node
  real(default) :: ndpr, ndpl
  real(default), dimension(:), allocatable :: ov, edv
  ndpr = this%r_position - c_position
  ndpl = this%d_position - ndpr
  allocate (ov (0:this%dim-1), source=this%get_r_value_array() - ndpr * &
       this%get_d_value() / this%d_position)
  allocate (edv (0:this%dim-1), source=c_value-ov)
  allocate (new_node)
  call new_node%initialize (dim=this%dim, r_position=c_position, &
       d_position=ndpl)
  call new_node%set_r_value (c_value)
  call new_node%set_d_value (this%get_d_value() + &
```

```
        c_value-this%get_r_value_array())
    call new_node%set_d_integral (ndpl*(this%get_d_value() - &
        this%get_r_value_array() - c_value) / two)
    call new_node%set_error (abs((edv*ndpl) / two))
    ! new_node%measure_comp = sum (abs((edv*ndpl) / two))
    this%d_position = ndpr
    call this%set_d_value (this%get_r_value_array() - c_value)
    call this%set_d_integral (- (ndpr*(this%get_r_value_array() + c_value) / two))
    call this%set_error (abs(edv*ndpr / two))
    ! this%measure_comp = sum (abs(edv*ndpr / two))
    ! write (*, "(1x,A)") "muli_trapezium_split: new errors:"
    ! write (*, "(3x,ES14.7)")  this%get_error()
    ! write (*, "(3x,ES14.7)")  new_node%get_error()
    ! write (*, "(3x,11(ES20.10)")  new_node%get_d_integral()
    ! write (*, "(3x,11(ES20.10)")  this%get_d_integral()
  end subroutine muli_trapezium_split
```

⟨*Muli trapezium: trapezium: TBP*⟩+≡

```
  procedure :: update => muli_trapezium_update
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_update (this)
    class(muli_trapezium_t), intent(inout) :: this
    real(default), dimension(:), allocatable :: integral
    real(default), dimension(0:this%dim-1) :: d_int
    !!! !!! !!! Workaround for gfortran 5.0 ICE
    d_int = this%get_d_integral ()
    allocate (integral (0:this%dim-1), source=d_int)
    call this%set_d_integral (-this%d_position * (this%get_r_value_array() &
        - this%get_d_value() / 2))
    call this%set_error (abs (this%get_d_integral() - integral))
    ! write (*, "(3x,11(ES20.10)")  this%get_d_integral()
  end subroutine muli_trapezium_update
```

⟨*Muli trapezium: node: TBP*⟩≡

```
  procedure :: deserialize_from_marker => &
      muli_trapezium_node_deserialize_from_marker
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_node_deserialize_from_marker (this, name, marker)
    class(muli_trapezium_node_class_t), intent(out) :: this
    character(*), intent(in) :: name
    class(marker_t), intent(inout) :: marker
    integer(dik) :: status
    class(ser_class_t), pointer :: ser
    allocate (muli_trapezium_tree_t :: ser)
    call marker%push_reference (ser)
    allocate (muli_trapezium_list_t::ser)
    call marker%push_reference (ser)
    call serializable_deserialize_from_marker (this, name, marker)
    call marker%pop_reference (ser)
    deallocate (ser)
    call marker%pop_reference (ser)
    deallocate (ser)
```

```
    end subroutine muli_trapezium_node_deserialize_from_marker
```

⟨*Muli trapezium: node: TBP*⟩+≡

```
    procedure(muli_trapezium_append_interface), deferred :: append
```

⟨*Muli trapezium: node: TBP*⟩+≡

```
    procedure(muli_trapezium_final_interface), deferred :: finalize
```

⟨*Muli trapezium: list: TBP*⟩≡

```
    procedure :: append => muli_trapezium_list_append
```

⟨*Muli trapezium: procedures*⟩+≡

```
    subroutine muli_trapezium_list_append (this, right)
      class(muli_trapezium_list_t), intent(inout), target :: this
      class(muli_trapezium_node_class_t), intent(inout), target :: right
      this%right => right
      right%left => this
    end subroutine muli_trapezium_list_append
```

⟨*Muli trapezium: node: TBP*⟩+≡

```
    procedure :: nullify => muli_trapezium_node_nullify
```

⟨*Muli trapezium: procedures*⟩+≡

```
    subroutine muli_trapezium_node_nullify (this)
      class(muli_trapezium_node_class_t), intent(out) :: this
      nullify (this%left)
      nullify (this%right)
    end subroutine muli_trapezium_node_nullify
```

⟨*Muli trapezium: node: TBP*⟩+≡

```
    procedure :: get_left => muli_trapezium_node_get_left
```

⟨*Muli trapezium: procedures*⟩+≡

```
    subroutine muli_trapezium_node_get_left (this, left)
      class(muli_trapezium_node_class_t), intent(in) :: this
      class(muli_trapezium_node_class_t), pointer, intent(out) :: left
      left => this%left
    end subroutine muli_trapezium_node_get_left
```

⟨*Muli trapezium: node: TBP*⟩+≡

```
    procedure :: get_right => muli_trapezium_node_get_right
```

⟨*Muli trapezium: procedures*⟩+≡

```
    subroutine muli_trapezium_node_get_right (this, right)
      class(muli_trapezium_node_class_t), intent(in) :: this
      class(muli_trapezium_node_class_t), pointer, intent(out) :: right
      right => this%right
    end subroutine muli_trapezium_node_get_right
```

⟨*Muli trapezium: node: TBP*⟩+≡

```
    procedure :: get_leftmost => muli_trapezium_node_get_leftmost
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_node_get_leftmost (this, node)
  class(muli_trapezium_node_class_t), intent(in) :: this
  class(muli_trapezium_node_class_t), pointer, intent(out) :: node
  if (associated (this%left)) then
     node => this%left
     do while (associated (node%left))
        node => node%left
     end do
  else
     nullify (node)
  end if
end subroutine muli_trapezium_node_get_leftmost
```

⟨*Muli trapezium: node: TBP*⟩+≡

```
procedure :: get_rightmost => muli_trapezium_node_get_rightmost
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_node_get_rightmost (this, right)
  class(muli_trapezium_node_class_t), intent(in) :: this
  class(muli_trapezium_node_class_t), pointer, intent(out) :: right
  if (associated (this%right)) then
     right => this%right
     do while (associated (right%right))
        right => right%right
     end do
  else
     nullify (right)
  end if
end subroutine muli_trapezium_node_get_rightmost
```

⟨*Muli trapezium: node: TBP*⟩+≡

```
generic :: decide => decide_by_value, decide_by_position
procedure :: decide_by_value => muli_trapezium_node_decide_by_value
procedure :: decide_by_position => muli_trapezium_node_decide_by_position
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_node_decide_by_value (this, value, dim, record, node)
  class(muli_trapezium_node_class_t), intent(in) :: this
  real(default), intent(in) :: value
  integer, intent(in) :: record, dim
  class(muli_trapezium_node_class_t), pointer, intent(out) :: node
  if (this%values (dim, record) > value) then
     node => this%left
  else
     node => this%right
  end if
end subroutine muli_trapezium_node_decide_by_value
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_node_decide_by_position (this, position, node)
  class(muli_trapezium_node_class_t), intent(in) :: this
  real(default), intent(in) :: position
```

```
    class(muli_trapezium_node_class_t), pointer, intent(out) :: node
    if (this%r_position > position) then
       node => this%left
    else
       node => this%right
    end if
  end subroutine muli_trapezium_node_decide_by_position
```

⟨*Muli trapezium: node: TBP*⟩+≡

```
  procedure :: decide_decreasing => muli_trapezium_node_decide_decreasing
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_node_decide_decreasing &
       (this, value, dim, record, node)
    class(muli_trapezium_node_class_t), intent(in) :: this
    real(default), intent(in) :: value
    integer, intent(in) :: record, dim
    class(muli_trapezium_node_class_t), pointer, intent(out) :: node
    if (this%values (dim, record) <= value) then
       node => this%left
    else
       node => this%right
    end if
  end subroutine muli_trapezium_node_decide_decreasing
```

⟨*Muli trapezium: node: TBP*⟩+≡

```
  procedure :: to_tree => muli_trapezium_node_to_tree
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_node_to_tree (this, out_tree)
    class(muli_trapezium_node_class_t), target, intent(in) :: this
    class(muli_trapezium_tree_t), intent(out) :: out_tree
    out_tree%left => this%left
    out_tree%right => this%right
  end subroutine muli_trapezium_node_to_tree
```

⟨*Muli trapezium: node: TBP*⟩+≡

```
  procedure :: untangle => muli_trapezium_node_untangle
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_node_untangle(this)
    class(muli_trapezium_node_class_t), intent(inout), target :: this
    if (associated (this%left)) then
       if (associated (this%left%right, this)) then
          nullify (this%left%right)
          nullify (this%left)
       end if
    end if
  end subroutine muli_trapezium_node_untangle
```

⟨*Muli trapezium: node: TBP*⟩+≡

```
  procedure :: apply => muli_trapezium_node_apply
```

⟨*Muli trapezium: procedures*⟩+≡

```
recursive subroutine muli_trapezium_node_apply(this,proc)
  class(muli_trapezium_node_class_t), intent(inout) :: this
  interface
     subroutine proc(this)
       import muli_trapezium_node_class_t
       class(muli_trapezium_node_class_t), intent(inout) :: this
     end subroutine proc
  end interface
  if (associated(this%right))call proc(this%right)
  if (associated(this%left))call proc(this%left)
  call proc(this)
end subroutine muli_trapezium_node_apply
```

⟨*Muli trapezium: node: TBP*⟩+≡

```
! procedure :: copy => muli_trapezium_node_copy
! generic :: assignment(=) => copy
! procedure, deferred :: approx => muli_trapezium_node_approx
```

⟨*Muli trapezium: tree: TBP*⟩≡

```
procedure :: write_to_marker => muli_trapezium_tree_write_to_marker
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_tree_write_to_marker (this, marker, status)
  class(muli_trapezium_tree_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  class(muli_trapezium_list_t), pointer :: list
  class(ser_class_t), pointer :: ser
  call marker%mark_begin ("muli_trapezium_tree_t")
  call this%get_left_list (list)
  ser => list
  call marker%mark_pointer ("list", ser)
  call marker%mark_end ("muli_trapezium_tree_t")
end subroutine muli_trapezium_tree_write_to_marker
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
procedure :: read_from_marker => muli_trapezium_tree_read_from_marker
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_tree_read_from_marker (this, marker, status)
  class(muli_trapezium_tree_t), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  class(ser_class_t), pointer :: ser
  call marker%pick_begin ("muli_trapezium_tree_t", status=status)
  call marker%pick_pointer ("list", ser)
  if (associated (ser)) then
     select type (ser)
     class is (muli_trapezium_list_t)
        call ser%to_tree (this)
     class default
        nullify (this%left)
        nullify (this%right)
```

```
         nullify (this%down)
      end select
   else
      nullify (this%left)
      nullify (this%right)
      nullify (this%down)
   end if
   call marker%pick_end ("muli_trapezium_tree_t", status)
end subroutine muli_trapezium_tree_read_from_marker
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
procedure :: print_to_unit => muli_trapezium_tree_print_to_unit
```

⟨*Muli trapezium: procedures*⟩+≡

```
recursive subroutine muli_trapezium_tree_print_to_unit &
     (this, unit, parents, components, peers)
   class(muli_trapezium_tree_t), intent(in) :: this
   integer, intent(in) :: unit
   integer(dik), intent(in) :: parents, components, peers
   class(ser_class_t), pointer :: ser
   if (parents > 0)  call muli_trapezium_print_to_unit &
        (this, unit, parents-1, components, peers)
   ser => this%down
   call serialize_print_peer_pointer (ser, unit, i_one, i_zero, i_one, "DOWN")
   if (associated (this%left)) then
      select type (sertmp => this%left)
      class is (muli_trapezium_list_t)
         ser => sertmp
         call serialize_print_peer_pointer &
              (ser, unit, parents, components, i_zero, "LEFT")
      class default
         call serialize_print_peer_pointer &
              (ser, unit, parents, components, peers, "LEFT")
      end select
   else
      write (unit, "(1x,A)")  "Left is not associated."
   end if
   if (associated (this%right)) then
      select type (sertmp => this%right)
      class is (muli_trapezium_list_t)
         ser => sertmp
         call serialize_print_peer_pointer &
              (ser, unit, parents, components, i_zero, "RIGHT")
      class default
         call serialize_print_peer_pointer &
              (ser, unit, parents, components, peers, "RIGHT")
      end select
   else
      write (unit, "(1x,A)")  "Right is not associated."
   end if
end subroutine muli_trapezium_tree_print_to_unit
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
procedure, nopass :: get_type => muli_trapezium_tree_get_type
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure subroutine muli_trapezium_tree_get_type (type)
  character(:),allocatable, intent(out) :: type
  allocate (type, source="muli_trapezium_tree_t")
end subroutine muli_trapezium_tree_get_type
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
procedure, nopass :: verify_type => muli_trapezium_tree_verify_type
```

⟨*Muli trapezium: procedures*⟩+≡

```
elemental logical function muli_trapezium_tree_verify_type (type) result (match)
  character(*), intent(in) :: type
  match = type == "muli_trapezium_tree_t"
end function muli_trapezium_tree_verify_type
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
procedure :: nullify => muli_trapezium_tree_nullify
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_tree_nullify (this)
  class(muli_trapezium_tree_t), intent(out) :: this
  call muli_trapezium_node_nullify (this)
  nullify (this%down)
end subroutine muli_trapezium_tree_nullify
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
procedure :: finalize => muli_trapezium_tree_finalize
```

⟨*Muli trapezium: procedures*⟩+≡

```
recursive subroutine muli_trapezium_tree_finalize (this)
  class(muli_trapezium_tree_t), intent(inout) :: this
  if (associated (this%right)) then
     call this%right%untangle ()
     call this%right%finalize ()
     deallocate (this%right)
  end if
  if (associated (this%left)) then
     call this%left%untangle ()
     call this%left%finalize ()
     deallocate (this%left)
  end if
  this%dim = 0
end subroutine muli_trapezium_tree_finalize
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
procedure :: decide_by_value => muli_trapezium_tree_decide_by_value
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_tree_decide_by_value (this, value, dim, record, node)
  class(muli_trapezium_tree_t), intent(in) :: this
  real(default), intent(in) :: value
  integer, intent(in) :: record, dim
  class(muli_trapezium_node_class_t), pointer, intent(out) :: node
  if (this%down%values (dim, record) > value) then
```

```
      node => this%left
    else
      node => this%right
    end if
  end subroutine muli_trapezium_tree_decide_by_value
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
  procedure :: decide_by_position => muli_trapezium_tree_decide_by_position
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_tree_decide_by_position (this, position, node)
    class(muli_trapezium_tree_t), intent(in) :: this
    real(default), intent(in) :: position
    class(muli_trapezium_node_class_t), pointer, intent(out) :: node
    if (this%down%r_position > position) then
      node => this%left
    else
      node => this%right
    end if
  end subroutine muli_trapezium_tree_decide_by_position
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
  procedure :: decide_decreasing => muli_trapezium_tree_decide_decreasing
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_tree_decide_decreasing &
      (this, value, dim, record, node)
    class(muli_trapezium_tree_t), intent(in) :: this
    real(default), intent(in) :: value
    integer, intent(in) :: record, dim
    ! integer, save :: count=0
    class(muli_trapezium_node_class_t), pointer, intent(out) :: node
    ! count = count + 1
    if (this%down%values (dim, record) <= value) then
      ! print ('("Decide: value(",I2,",",I1,")=",E20.7," > ",E20.7, &
      !     ": go left.")'), dim, record, this%down%values(dim, record), value
      node => this%left
    else
      ! print ('("Decide: value(",I2,",",I1,")=",E20.7," <= ", &
      !   E20.7,": go right.")'), &
      !   dim, record, this%down%values(dim, record), value
      node => this%right
    end if
  end subroutine muli_trapezium_tree_decide_decreasing
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
  procedure :: get_left_list => muli_trapezium_tree_get_left_list
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_tree_get_left_list (this, list)
    class(muli_trapezium_tree_t), intent(in) :: this
    class(muli_trapezium_list_t), pointer, intent(out) :: list
    class(muli_trapezium_node_class_t), pointer::node
    call this%get_leftmost (node)
```

```
    if (associated (node)) then
       select type (node)
       class is (muli_trapezium_list_t)
          list => node
       class default
          nullify (list)
       end select
    else
       nullify (list)
    end if
  end subroutine muli_trapezium_tree_get_left_list
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
  procedure :: get_right_list => muli_trapezium_tree_get_right_list
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_tree_get_right_list (this, list)
    class(muli_trapezium_tree_t), intent(in) :: this
    class(muli_trapezium_list_t), pointer, intent(out) :: list
    class(muli_trapezium_node_class_t), pointer::node
    call this%get_rightmost (node)
    if (associated (node)) then
       select type (node)
       class is (muli_trapezium_list_t)
          list => node
       class default
          nullify (list)
       end select
    else
       nullify (list)
    end if
  end subroutine muli_trapezium_tree_get_right_list
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
  generic :: find => find_by_value, find_by_position
  procedure :: find_by_value => muli_trapezium_tree_find_by_value
  procedure :: find_by_position => muli_trapezium_tree_find_by_position
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_tree_find_by_value (this, value, dim, record, node)
    class(muli_trapezium_tree_t), intent(in), target :: this
    real(default), intent(in) :: value
    integer, intent(in) :: record, dim
    class(muli_trapezium_node_class_t), pointer, intent(out) :: node
    node => this
    do while (.not. allocated (node%values))
       call node%decide (value, dim, record, node)
    end do
  end subroutine muli_trapezium_tree_find_by_value
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_tree_find_by_position (this, position, node)
    class(muli_trapezium_tree_t), intent(in), target :: this
```

```
    real(default), intent(in) :: position
    class(muli_trapezium_node_class_t), pointer, intent(out) :: node
    node => this
    do while (.not. allocated (node%values))
       call node%decide (position, node)
    end do
  end subroutine muli_trapezium_tree_find_by_position
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
  procedure :: find_decreasing => muli_trapezium_tree_find_decreasing
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_tree_find_decreasing (this, value, dim, node)
    class(muli_trapezium_tree_t), intent(in), target :: this
    real(default), intent(in) :: value
    integer, intent(in) :: dim
    class(muli_trapezium_node_class_t), pointer, intent(out) :: node
    node => this
    do while (.not. allocated (node%values))
       call node%decide_decreasing (value, dim, r_integral_index, node)
    end do
  end subroutine muli_trapezium_tree_find_decreasing
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
  procedure :: approx_by_integral => muli_trapezium_tree_approx_by_integral
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_tree_approx_by_integral &
       (this, int, dim, in_range, position, value, integral, content)
    class(muli_trapezium_tree_t), intent(in), target :: this
    real(default), intent(in) :: int
    integer, intent(in) :: dim
    logical, intent(out) :: in_range
    class(muli_trapezium_node_class_t), pointer, intent(out), optional :: content
    real(default), intent(out), optional :: position, value, integral
    integer :: i
    real(default) :: DINT    !,l_prop,r_prop,d_prop
    real(default) :: RP, DP, RV, DV, RI    !FC = gfortran
    class(muli_trapezium_node_class_t), pointer :: node
    node => this
    do while (.not. allocated (node%values))
       call node%decide_decreasing(INT, dim, r_integral_index, node)
    end do
    if (   int<=node%values(dim,r_integral_index)-node%values(dim,d_integral_index)&
         &.and.&
         &int>=node%values(dim,r_integral_index)) then
       in_range=.true.
!       associate(&!FC = nagfor
!            &RP=>node%r_position,&!FC = nagfor
!            &DP=>node%d_position,&!FC = nagfor
!            &RV=>node%values(dim,r_value_index),&!FC = nagfor
!            &DV=>node%values(dim,d_value_index),&!FC = nagfor
!            &RI=>node%values(dim,r_integral_index))!FC = nagfor
         RP=node%r_position!FC = gfortran
```

```
         DP=node%d_position!FC = gfortran
         RV=node%values(dim,r_value_index)!FC = gfortran
         DV=node%values(dim,d_value_index)!FC = gfortran
         RI=node%values(dim,r_integral_index)!FC = gfortran
         if (present(position)) then
            DINT=(ri-int)*2D0*dv/dp
            position=rp-(dp/dv)*(rv-sqrt(dint+rv**2))
         end if
         if (present(value)) then
            value=Sqrt(dp*(-2*dv*int + 2*dv*ri + dp*rv**2))/dp
         end if
         if (present(integral)) then
            integral=int
         end if
         if (present(content)) then
            content=>node
         end if
!       end associate!FC = nagfor
     else
        in_range=.false.
     end if
   end subroutine muli_trapezium_tree_approx_by_integral
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
   procedure :: approx_by_probability => muli_trapezium_tree_approx_by_probability
```

⟨*Muli trapezium: procedures*⟩+≡

```
   subroutine muli_trapezium_tree_approx_by_probability &
        (this, prop, dim, in_range, position, value, integral, content)
     class(muli_trapezium_tree_t), intent(in), target :: this
     real(default), intent(in) :: prop
     integer, intent(in) :: dim
     logical, intent(out) :: in_range
     class(muli_trapezium_node_class_t), pointer, intent(out), optional :: content
     real(default), intent(out), optional :: position, value, integral
     integer :: i
     real(default) :: int
     class(muli_trapezium_node_class_t), pointer :: node
     if (zero < prop .and. prop < one) then
        node => this
        int = -log (prop)
        call muli_trapezium_tree_approx_by_integral &
             (this, int, dim, in_range, position, value, integral, content)
     else
        in_range = .false.
     end if
   end subroutine muli_trapezium_tree_approx_by_probability
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
   procedure :: to_tree => muli_trapezium_tree_to_tree
```

⟨*Muli trapezium: procedures*⟩+≡

```
   subroutine muli_trapezium_tree_to_tree (this, out_tree)
     class(muli_trapezium_tree_t), target, intent(in) :: this
```

```
    class(muli_trapezium_tree_t), intent(out) :: out_tree
    out_tree%left => this%left
    out_tree%right => this%right
    out_tree%down => this%down
  end subroutine muli_trapezium_tree_to_tree
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
  procedure :: append => muli_trapezium_tree_append
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_tree_append(this,right)
    class(muli_trapezium_tree_t), intent(inout), target :: this
    class(muli_trapezium_node_class_t), intent(inout), target :: right
    call msg_error ("muli_trapezium_tree_append: Not yet implemented.")
  end subroutine muli_trapezium_tree_append
```

⟨*Muli trapezium: tree: TBP*⟩+≡

```
  procedure :: gnuplot => muli_trapezium_tree_gnuplot
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_tree_gnuplot (this, dir)
    class(muli_trapezium_tree_t), intent(in) :: this
    character(len=*), intent(in) :: dir
    class(muli_trapezium_list_t), pointer :: list
    call this%get_left_list (list)
    call list%gnuplot (dir)
  end subroutine muli_trapezium_tree_gnuplot
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
  procedure :: write_to_marker => muli_trapezium_list_write_to_marker
```

⟨*Muli trapezium: procedures*⟩+≡

```
  recursive subroutine muli_trapezium_list_write_to_marker (this, marker, status)
    class(muli_trapezium_list_t), intent(in) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    class(ser_class_t), pointer :: ser
    call marker%mark_begin ("muli_trapezium_list_t")
    call muli_trapezium_write_to_marker (this, marker, status)
    ser => this%right
    call marker%mark_pointer ("right", ser)
    call marker%mark_end ("muli_trapezium_list_t")
  end subroutine muli_trapezium_list_write_to_marker
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
  procedure :: read_from_marker => muli_trapezium_list_read_from_marker
```

⟨*Muli trapezium: procedures*⟩+≡

```
  recursive subroutine muli_trapezium_list_read_from_marker (this, marker, status)
    class(muli_trapezium_list_t), intent(out) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    call msg_warning ("muli_trapezium_list_read_from_marker: " // &
         "You cannot deserialize a list with this subroutine.")
```

```
    call msg_error ("Use muli_trapezium_list_read_target_from_marker instead.")
  end subroutine muli_trapezium_list_read_from_marker
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
  procedure :: read_target_from_marker => &
      muli_trapezium_list_read_target_from_marker
```

⟨*Muli trapezium: procedures*⟩+≡

```
  recursive subroutine muli_trapezium_list_read_target_from_marker &
      (this, marker, status)
    class(muli_trapezium_list_t), target, intent(out) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    class(ser_class_t), pointer :: ser
    call marker%pick_begin ("muli_trapezium_list_t", status=status)
    call muli_trapezium_read_from_marker (this, marker, status)
    call marker%pick_pointer ("right", ser)
    if (associated (ser)) then
       select type (ser)
       class is (muli_trapezium_list_t)
          this%right => ser
          ser%left => this
       class default
          nullify (this%right)
          call msg_error ("muli_trapezium_list_read_target_from_marker: " &
               // "Unexpected type for right component.")
       end select
    else
       nullify (this%right)
    end if
    call marker%pick_end ("muli_trapezium_list_t", status)
  end subroutine muli_trapezium_list_read_target_from_marker
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
  procedure :: print_to_unit => muli_trapezium_list_print_to_unit
```

⟨*Muli trapezium: procedures*⟩+≡

```
  recursive subroutine muli_trapezium_list_print_to_unit &
      (this, unit, parents, components, peers)
    class(muli_trapezium_list_t), intent(in) :: this
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    class(ser_class_t), pointer :: ser
    if (parents > 0)  call muli_trapezium_print_to_unit &
         (this, unit, parents-1, components, peers)
    ser => this%left
    call serialize_print_peer_pointer &
         (ser, unit, -i_one, -i_one, -i_one, "LEFT")
    ser => this%right
    call serialize_print_peer_pointer &
         (ser, unit, parents, components, peers, "RIGHT")
  end subroutine muli_trapezium_list_print_to_unit
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
procedure, nopass :: get_type => muli_trapezium_list_get_type
```

⟨*Muli trapezium: procedures*⟩+≡

```
pure subroutine muli_trapezium_list_get_type (type)
  character(:), allocatable, intent(out) :: type
  allocate (type, source="muli_trapezium_list_t")
end subroutine muli_trapezium_list_get_type
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
procedure, nopass :: verify_type => muli_trapezium_list_verify_type
```

⟨*Muli trapezium: procedures*⟩+≡

```
elemental logical function muli_trapezium_list_verify_type (type) result (match)
  character(*), intent(in) :: type
  match = type == "muli_trapezium_list_t"
end function muli_trapezium_list_verify_type
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
procedure :: finalize => muli_trapezium_list_finalize
```

⟨*Muli trapezium: procedures*⟩+≡

```
recursive subroutine muli_trapezium_list_finalize(this)
  class(muli_trapezium_list_t), intent(inout)::this
  if (associated(this%right)) then
     call this%right%finalize()
     deallocate(this%right)
  end if
  this%dim=0
end subroutine muli_trapezium_list_finalize
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
generic :: insert_right => insert_right_a   !, insert_right_b
procedure :: insert_right_a => muli_trapezium_list_insert_right_a
! procedure :: insert_right_b => muli_trapezium_list_insert_right_b
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_list_insert_right_a (this, value, content, new_node)
  class(muli_trapezium_list_t), intent(inout), target :: this
  real(default), intent(in) :: value
  class(muli_trapezium_t), intent(in) :: content
  class(muli_trapezium_list_t), pointer, intent(out) :: new_node
  class(muli_trapezium_list_t), pointer :: tmp_list
  call content%to_node (value, list=tmp_list)
  if (associated (this%right)) then
     this%right%left => tmp_list
     tmp_list%right => this%right
  else
     nullify (tmp_list%right)
  end if
  this%right => tmp_list
  tmp_list%left => this
  new_node => tmp_list
end subroutine muli_trapezium_list_insert_right_a
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
generic :: insert_left => insert_left_a   !, insert_left_b
procedure :: insert_left_a => muli_trapezium_list_insert_left_a
! procedure :: insert_left_b => muli_trapezium_list_insert_left_b
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_list_insert_left_a (this, value, content, new_node)
  class(muli_trapezium_list_t), intent(inout), target :: this
  real(default), intent(in) :: value
  class(muli_trapezium_t), intent(in) :: content
  class(muli_trapezium_list_t), pointer, intent(out) :: new_node
  call content%to_node (value, list=new_node)
  new_node%right => this
  if (associated (this%left)) then
     new_node%left => this%left
     this%left%right => new_node
  else
     nullify (new_node%left)
  end if
  this%left => new_node
end subroutine muli_trapezium_list_insert_left_a
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
procedure :: to_tree => muli_trapezium_list_to_tree
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_list_to_tree (this, out_tree)
  class(muli_trapezium_list_t), target, intent(in) :: this
  class(muli_trapezium_tree_t), intent(out) :: out_tree
  type(muli_trapezium_tree_t),target :: do_list
  class(muli_trapezium_node_class_t),pointer :: this_entry,do_list_entry,node
  class(muli_trapezium_tree_t),pointer :: tree1,tree2
  integer :: ite,log,n_deep,n_leaves
  n_leaves=0
  this_entry => this
  count: do while(associated(this_entry))
     n_leaves=n_leaves+1
     this_entry=>this_entry%right
  end do count
  call ilog2(n_leaves,log,n_deep)
  this_entry => this
  do_list_entry => do_list
  deep: do ite=0,n_deep-1
     allocate(tree1)
     tree1%down=>this_entry%right
     allocate(tree2)
     tree2%down=>this_entry
     tree2%left=>this_entry
     tree2%right=>this_entry%right
     tree1%left=>tree2
     this_entry => this_entry%right%right
     do_list_entry%right=>tree1
     do_list_entry=>tree1
  end do deep
  rest: do while(associated(this_entry))
```

```
      allocate(tree1)
      tree1%down=>this_entry
      tree1%left=>this_entry
      do_list_entry%right => tree1
      do_list_entry => tree1
      this_entry => this_entry%right
      ite=ite+1
   end do rest
   tree: do while(ite>2)
      do_list_entry => do_list%right
      node=>do_list
      level: do while(associated(do_list_entry))
         node%right=>do_list_entry%right
         node=>do_list_entry%right
         do_list_entry%right=>node%left
         node%left=>do_list_entry
         do_list_entry=>node%right
         ite=ite-1
      end do level
   end do tree
   node=>do_list%right
   select type(node)
   type is (muli_trapezium_tree_t)
      call node%to_tree(out_tree)
   class default
      print *,"muli_trapezium_list_to_tree"
      print *,"unexpeted type for do_list%right"
   end select
   out_tree%right=>out_tree%right%left
   if (allocated(out_tree%values)) then
      deallocate(out_tree%values)
   end if
   deallocate(do_list%right%right)
   deallocate(do_list%right)
end subroutine muli_trapezium_list_to_tree
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
procedure :: gnuplot => muli_trapezium_list_gnuplot
```

⟨*Muli trapezium: procedures*⟩+≡

```
subroutine muli_trapezium_list_gnuplot (this, dir)
   class(muli_trapezium_list_t), intent(in), target :: this
   character(len=*), intent(in) :: dir
   character(len=*), parameter :: val_file = "/value.plot"
   character(len=*), parameter :: int_file = "/integral.plot"
   character(len=*), parameter :: err_file = "/integral_error.plot"
   character(len=*), parameter :: pro_file = "/probability.plot"
   character(len=*), parameter :: den_file = "/density.plot"
   character(len=*), parameter :: fmt = "(ES20.10)"
   class(muli_trapezium_node_class_t), pointer :: list
   integer :: val_unit, err_unit, int_unit, pro_unit, den_unit
   list => this
   call generate_unit (val_unit, 100, 1000)
   open (val_unit, file = dir // val_file)
```

```
    call generate_unit (int_unit, 100, 1000)
    open (int_unit, file = dir // int_file)
    call generate_unit (err_unit, 100, 1000)
    open (err_unit, file = dir // err_file)
    call generate_unit (pro_unit, 100, 1000)
    open (pro_unit, file = dir // pro_file)
    call generate_unit (den_unit, 100, 1000)
    open (den_unit, file = dir // den_file)
    do while (associated (list))
       ! print *,list%r_position,list%get_r_value()
       !!! !!! !!! gfortran 5.0.0 ICE
       ! write (val_unit, fmt, advance="no")  list%r_position
       ! call write_array (val_unit, list%get_r_value_array(), fmt)
       ! write (int_unit,fmt,advance="no")  list%r_position
       ! call write_array (int_unit, list%get_r_integral(), fmt)
       ! write (err_unit, fmt, advance="no")  list%r_position
       ! call write_array (err_unit, list%get_error(), fmt)
       ! write (pro_unit, fmt, advance="no")  list%r_position
       ! call write_array (pro_unit, list%get_r_probability(), fmt)
       ! write (den_unit, fmt, advance="no")  list%r_position
       ! call write_array (den_unit, list%get_r_probability() * &
       !      list%get_r_value_array(), fmt)
       list => list%right
    end do
    close (val_unit)
    close (int_unit)
    close (err_unit)
    close (pro_unit)
    close (den_unit)
    contains
      subroutine write_array (unit, array, form)
        integer, intent(in) :: unit
        real(default), dimension(:), intent(in) :: array
        character(len=*), intent(in) :: form
        integer :: n
        do n = 1, size(array)
           write (unit, form, advance="no")  array(n)
           flush (unit)
        end do
        write (unit, *)
      end subroutine write_array
  end subroutine muli_trapezium_list_gnuplot
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
  procedure :: integrate => muli_trapezium_list_integrate
```

⟨*Muli trapezium: procedures*⟩+≡

```
  subroutine muli_trapezium_list_integrate (this, integral_sum, error_sum)
    class(muli_trapezium_list_t), intent(in), target :: this
    real(default), intent(out) :: error_sum, integral_sum
    real(default), dimension(:), allocatable :: integral
    class(muli_trapezium_node_class_t), pointer :: node
    allocate (integral (0:this%dim-1))
    call this%get_rightmost (node)
```

```
    integral = 0._default
    integral_sum = 0._default
    error_sum = 0._default
    integrate: do while (associated (node))
       node%values(1,r_value_index) = sum(node%values(1:this%dim-1,r_value_index))
       node%values(1,d_value_index) = sum(node%values(1:this%dim-1,d_value_index))
       ! node%values (1, r_integral_index) = &
       !    sum (node%values (1:this%dim-1, r_integral_index))
       ! node%values (1, d_integral_index) = &
       !    sum (node%values (1:this%dim-1, d_integral_index))
       node%values(1, error_index) = sum (node%values(1:this%dim-1, error_index))
       error_sum = error_sum + node%values (1, error_index)
       !!! !!! !!! gfortran 5.0.0 ICE
       ! call node%set_d_integral (node%get_d_position() * &
       !      (node%get_d_value() / 2 - node%get_r_value_array ()))
       call node%set_r_probability (exp (-integral))
       call node%set_r_integral (integral)
       !!! !!! !!! gfortran 5.0.0 ICE
       ! integral = integral - node%get_d_integral()
       ! call node%set_d_probability (node%get_r_probability() - exp(-integral))
       ! call muli_trapezium_write (node, output_unit)
       call node%get_left (node)
    end do integrate
    integral_sum = integral (1)
  end subroutine muli_trapezium_list_integrate
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
  procedure :: check => muli_trapezium_list_check
```

⟨*Muli trapezium: procedures*⟩+≡

```
  recursive subroutine muli_trapezium_list_check (this)
    class(muli_trapezium_list_t), intent(in),target :: this
    class(muli_trapezium_node_class_t), pointer :: tn, next
    real(default), parameter :: eps = 1E-10_default
    logical::test
    if (associated(this%right)) then
       next=>this%right
       test=(this%r_position.le.this%right%get_l_position()+eps)
       print *,"position check:  ",test
       if (.not.test) then
          call this%print_parents()
          call next%print_parents()
       end if
       select type (next)
       class is (muli_trapezium_list_t)
          tn=>this
          print *,"structure check: ",associated(tn,next%left)
          print *,"class check:    T"
          call next%check()
       class default
          print *,"class check:    F"
       end select
    else
       print *,"end of list at ",this%r_position
```

```
      end if
    end subroutine muli_trapezium_list_check
```

⟨*Muli trapezium: list: TBP*⟩+≡

```
    procedure :: apply => muli_trapezium_list_apply
```

⟨*Muli trapezium: procedures*⟩+≡

```
    recursive subroutine muli_trapezium_list_apply (this, proc)
      class(muli_trapezium_list_t), intent(inout) :: this
      interface
         subroutine proc (this)
           import muli_trapezium_node_class_t
           class(muli_trapezium_node_class_t), intent(inout) :: this
         end subroutine proc
      end interface
      if (associated (this%right))call this%right%apply (proc)
      call proc (this)
    end subroutine muli_trapezium_list_apply
```

⟨*Muli trapezium: procedures*⟩+≡

```
  !  subroutine muli_trapezium_list_insert_right_old &
  !       (this, value, content, new_node)
  !    class(muli_trapezium_list_t), intent(inout), target :: this
  !    real(default), intent(in) :: value
  !    class(muli_trapezium_t), intent(in) :: content
  !    class(muli_trapezium_list_t), pointer, intent(out) :: new_node
  !    call content%to_node (value, list=new_node)
  !    new_node%left => this
  !    if (associated (this%right)) then
  !       new_node%right => this%right
  !       this%right%left => new_node
  !    else
  !       nullify (new_node%right)
  !    end if
  !    this%right => new_node
  !  end subroutine muli_trapezium_list_insert_right_old
```

⟨*Muli trapezium: procedures*⟩+≡

```
  !  subroutine muli_trapezium_node_error_no_content (this)
  !    class(muli_trapezium_node_class_t), intent(in) :: this
  !!    print ("muli_trapezium_node: Trying to access unallocated content.")
  !!    call this%print()
  !  end subroutine muli_trapezium_node_error_no_content
```

22.6 Fibonacci trees

This file contains the module `muli_fibonacci_tree`. A fibonacci tree is a self-balancing binary tree. "Balanced" means that the depth of the left successor may differ from the depth of the right successor in any node by maximally one. The maximally unbalanced tree is maximally unbalanced in every node. So the

number of leaves of a maximally unbalanced fibonacci tree of depth n is precisely the nth fibonacci number.

We use this tree for the adaptive quadrature. In every iteration of the algorithm, we have to pick the segment with largest integration error, cut it into pieces and calculate the new integration error for the pieces. A binary tree is obvioulsly a good choice for soring these segments. The problem is, that we always pick leaves from the same side of the tree. So the tree must decline at one side and grow at the other side. Sorting gets less efficient and finally most of the overall CPU time gets wasted for sorting. This tree outperforms a naive binary tree significantly in this particular job.

⟨*File header*⟩+≡

⟨muli_fibonacci_tree.f90⟩≡

```
module muli_fibonacci_tree

⟨Use kinds⟩
  use diagnostics
  use muli_base

⟨Standard module head⟩

⟨Muli fibonacci: public⟩

⟨Muli fibonacci: variables⟩

⟨Muli fibonacci: types⟩

contains

⟨Muli fibonacci: procedures⟩

end module muli_fibonacci_tree
```

⟨*Muli fibonacci: variables*⟩≡

```
character(*), parameter :: no_par = "edge=\noparent"
character(*), parameter :: no_ret = "edge=\noreturn"
character(*), parameter :: no_kid = "edge=\nochild"
character(*), parameter :: le_kid = "edge=\childofleave"
```

⟨*Muli fibonacci: public*⟩≡

```
public :: fibonacci_node_t
```

⟨*Muli fibonacci: types*⟩≡

```
type, extends (measure_class_t) :: fibonacci_node_t
   ! private
   class(fibonacci_node_t), pointer :: up => null()
   class(measure_class_t), pointer :: down => null()
   class(fibonacci_node_t), pointer :: left => null()
   class(fibonacci_node_t), pointer :: right => null()
   integer :: depth = 0
   ! real(default) :: value
 contains
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩
```
end type fibonacci_node_t
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩≡
```
procedure :: write_to_marker => fibonacci_node_write_to_marker
```

⟨*Muli fibonacci: procedures*⟩≡
```
recursive  subroutine fibonacci_node_write_to_marker (this, marker, status)
  class(fibonacci_node_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  class(ser_class_t), pointer :: ser
  call marker%mark_begin ("fibonacci_node_t")
  ser => this%left
  call marker%mark_pointer ("left", ser)
  ser => this%right
  call marker%mark_pointer ("right", ser)
  ser => this%down
  call marker%mark_pointer ("down", ser)
  call marker%mark_end ("fibonacci_node_t")
end subroutine fibonacci_node_write_to_marker
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡
```
procedure :: read_from_marker => fibonacci_node_read_from_marker
```

⟨*Muli fibonacci: procedures*⟩+≡
```
recursive subroutine fibonacci_node_read_from_marker (this, marker, status)
  class(fibonacci_node_t), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call msg_warning ("fibonacci_node_read_from_marker: You cannot " // &
       "deserialize a list with this subroutine.")
  call msg_error ("Use fibonacci_node_read_target_from_marker instead.")
end subroutine fibonacci_node_read_from_marker
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡
```
procedure :: read_target_from_marker => fibonacci_node_read_target_from_marker
```

⟨*Muli fibonacci: procedures*⟩+≡
```
recursive subroutine fibonacci_node_read_target_from_marker &
     (this, marker, status)
  class(fibonacci_node_t), target, intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  class(ser_class_t), pointer :: ser
  call marker%pick_begin ("fibonacci_node_t", status=status)
  call marker%pick_pointer ("left", ser)
  if (status == 0) then
     select type (ser)
     class is (fibonacci_node_t)
        this%left => ser
        this%left%up => this
     end select
  end if
```

```
    call marker%pick_pointer ("right", ser)
    if (status == 0) then
       select type (ser)
       class is (fibonacci_node_t)
          this%right => ser
          this%right%up => this
       end select
    end if
    call marker%pick_pointer ("down", ser)
    if (status == 0) then
       select type (ser)
       class is (measure_class_t)
          this%down => ser
       end select
    end if
    call marker%pick_end ("fibonacci_node_t", status)
  end subroutine fibonacci_node_read_target_from_marker
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: print_to_unit => fibonacci_node_print_to_unit
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  recursive subroutine fibonacci_node_print_to_unit &
       (this, unit, parents, components, peers)
    class(fibonacci_node_t), intent(in) :: this
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    class(ser_class_t), pointer :: ser
    write (unit, "(1x,A)")        "Components of fibonacci_node_t:"
    write (unit, "(3x,A,I22)")    "Depth:   ", this%depth
    write (unit, "(3x,A,E23.16)") "Value:   ", this%measure ()
    ser => this%up
    call serialize_print_comp_pointer &
         (ser, unit, parents, -i_one, -i_one, "Up:     ")
    ser => this%left
    call serialize_print_peer_pointer &
         (ser, unit, parents, components, peers, "Left:   ")
    ser => this%right
    call serialize_print_peer_pointer &
         (ser, unit, parents, components, peers, "Right:  ")
  end subroutine fibonacci_node_print_to_unit
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure, nopass :: get_type => fibonacci_node_get_type
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  pure subroutine fibonacci_node_get_type (type)
    character(:), allocatable, intent(out) :: type
    allocate (type, source="fibonacci_node_t")
  end subroutine fibonacci_node_get_type
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: deserialize_from_marker => fibonacci_node_deserialize_from_marker
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_deserialize_from_marker (this, name, marker)
  class(fibonacci_node_t), intent(out) :: this
  character(*), intent(in) :: name
  class(marker_t), intent(inout) :: marker
  class(ser_class_t), pointer :: ser
  allocate (fibonacci_leave_t :: ser)
  call marker%push_reference (ser)
  allocate (fibonacci_node_t :: ser)
  call marker%push_reference (ser)
  call serializable_deserialize_from_marker (this, name, marker)
  call marker%pop_reference (ser)
  deallocate (ser)
  call marker%pop_reference (ser)
  deallocate (ser)
end subroutine fibonacci_node_deserialize_from_marker
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: measure => fibonacci_node_measure
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental function fibonacci_node_measure (this)
  class(fibonacci_node_t), intent(in) :: this
  real(default) :: fibonacci_node_measure
  fibonacci_node_measure = this%down%measure ()
end function fibonacci_node_measure
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: deallocate_tree => fibonacci_node_deallocate_tree
```

⟨*Muli fibonacci: procedures*⟩+≡

```
recursive subroutine fibonacci_node_deallocate_tree (this)
  class(fibonacci_node_t), intent(inout) :: this
  if (associated (this%left)) then
     call this%left%deallocate_tree ()
     deallocate (this%left)
  end if
  if (associated (this%right)) then
     call this%right%deallocate_tree ()
     deallocate (this%right)
  end if
  call this%set_depth (0)
end subroutine fibonacci_node_deallocate_tree
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: deallocate_all => fibonacci_node_deallocate_all
```

⟨*Muli fibonacci: procedures*⟩+≡

```
recursive subroutine fibonacci_node_deallocate_all (this)
  class(fibonacci_node_t), intent(inout) :: this
  if (associated (this%left)) then
     call this%left%deallocate_all ()
     deallocate (this%left)
  end if
```

```
    if (associated (this%right)) then
       call this%right%deallocate_all ()
       deallocate (this%right)
    end if
    call this%set_depth (0)
  end subroutine fibonacci_node_deallocate_all
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: get_depth => fibonacci_node_get_depth
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  elemental function fibonacci_node_get_depth (this)
    class(fibonacci_node_t), intent(in) :: this
    integer :: fibonacci_node_get_depth
    fibonacci_node_get_depth = this%depth
  end function fibonacci_node_get_depth
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: count_leaves => fibonacci_node_count_leaves
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  recursive subroutine fibonacci_node_count_leaves (this, n)
    class(fibonacci_node_t), intent(in) :: this
    integer, intent(out) :: n
    integer :: n1, n2
    if (associated (this%left) .and. associated (this%right)) then
       call fibonacci_node_count_leaves (this%left, n1)
       call fibonacci_node_count_leaves (this%right, n2)
       n = n1 + n2
    else
       n = 1
    end if
  end subroutine fibonacci_node_count_leaves
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure,public,nopass :: is_leave => fibonacci_node_is_leave
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  elemental function fibonacci_node_is_leave ()
    logical :: fibonacci_node_is_leave
    fibonacci_node_is_leave = .false.
  end function fibonacci_node_is_leave
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure,public,nopass :: is_root => fibonacci_node_is_root
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  elemental function fibonacci_node_is_root ()
    logical :: fibonacci_node_is_root
    fibonacci_node_is_root = .false.
  end function fibonacci_node_is_root
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure,public,nopass :: is_inner => fibonacci_node_is_inner
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental function fibonacci_node_is_inner ()
  logical :: fibonacci_node_is_inner
  fibonacci_node_is_inner = .true.
end function fibonacci_node_is_inner
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: write_association => fibonacci_node_write_association
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_write_association (this, that)
  class(fibonacci_node_t), intent(in), target :: this
  class(fibonacci_node_t), intent(in), target :: that
  if (associated (that%left, this)) then
     write(*, "(A)")  "This is left child of that"
  end if
  if (associated (that%right, this)) then
     write(*, "(A)")  "This is right child of that"
  end if
  if (associated (that%up, this)) then
     write(*, "(A)")  "This is parent of that"
  end if
  if (associated (this%left, that)) then
     write(*, "(A)")  "That is left child of this"
  end if
  if (associated (this%right, that)) then
     write(*, "(A)")  "That is right child of this"
  end if
  if (associated (this%up, that)) then
     write(*, "(A)")  "That is parent of this"
  end if
end subroutine fibonacci_node_write_association
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: write_contents => fibonacci_node_write_contents
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_write_contents (this, unit)
  class(fibonacci_node_t), intent(in), target :: this
  integer, intent(in), optional :: unit
  call this%apply_to_leaves (fibonacci_leave_write_content, unit)
end subroutine fibonacci_node_write_contents
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: write_values => fibonacci_node_write_values
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_write_values (this, unit)
  class(fibonacci_node_t), intent(in), target :: this
  integer, intent(in), optional :: unit
  call this%apply_to_leaves (fibonacci_leave_write_value, unit)
end subroutine fibonacci_node_write_values
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: write_leaves => fibonacci_node_write_leaves
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_write_leaves (this, unit)
  class(fibonacci_node_t), intent(in), target :: this
  integer, intent(in),optional :: unit
  call this%apply_to_leaves (fibonacci_leave_write, unit)
end subroutine fibonacci_node_write_leaves
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
! procedure :: write => fibonacci_node_write_contents
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: write_pstricks => fibonacci_node_write_pstricks
```

⟨*Muli fibonacci: procedures*⟩+≡

```
recursive subroutine fibonacci_node_write_pstricks (this, unitnr)
  class(fibonacci_node_t), intent(in), target :: this
  integer, intent(in) :: unitnr
  if (associated (this%up)) then
     if (associated (this%up%left, this) .neqv. &
        (associated (this%up%right, this))) then
        ! write (unitnr,'("\begin{psTree}{\Toval{$",i3,"$}}")') int(this%depth)
        write (unitnr, &
             '("\begin{psTree}{\Toval{\node{",i3,"}{",f9.3,"}}}")') &
             int(this%depth), this%measure()
     else
        write (unitnr, &
             '("\begin{psTree}{\Toval[",a,"]{\node{",i3,"}{",f9.3,"}}}")') &
             no_ret, int(this%depth), this%measure()
     end if
  else
     write (unitnr, &
          '("\begin{psTree}{\Toval[",a,"]{\node{",i3,"}{",f9.3,"}}}")') &
          no_par, int(this%depth), this%measure()
  end if
  if (associated (this%left)) then
     call this%left%write_pstricks (unitnr)
  else
     write (unitnr,'("\Tr[edge=brokenline]{}")')
  end if
  if (associated (this%right)) then
     call this%right%write_pstricks (unitnr)
  else
     write (unitnr, '("\Tr[edge=brokenline]{}")')
  end if
  write (unitnr, '("\end{psTree}")')
end subroutine fibonacci_node_write_pstricks
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: copy_node => fibonacci_node_copy_node
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_copy_node (this, primitive)
  class(fibonacci_node_t), intent(out) :: this
  class(fibonacci_node_t), intent(in) :: primitive
  this%up => primitive%up
  this%left => primitive%left
  this%right => primitive%right
  this%depth = primitive%depth
  this%down => primitive%down
end subroutine fibonacci_node_copy_node
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: find_root => fibonacci_node_find_root
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_find_root (this, root)
  class(fibonacci_node_t), intent(in), target :: this
  class(fibonacci_root_t), pointer, intent(out) :: root
  class(fibonacci_node_t), pointer :: node
  node => this
  do while (associated (node%up))
     node => node%up
  end do
  select type (node)
  class is (fibonacci_root_t)
     root => node
  class default
     nullify (root)
     call msg_error ("fibonacci_node_find_root: root is not type " // &
          "compatible to fibonacci_root_t. Retured NULL().")
  end select
end subroutine fibonacci_node_find_root
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: find_leftmost => fibonacci_node_find_leftmost
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_find_leftmost (this, leave)
  class(fibonacci_node_t), intent(in), target :: this
  class(fibonacci_leave_t), pointer, intent(out) :: leave
  class(fibonacci_node_t),  pointer :: node
  node => this
  do while (associated (node%left))
     node => node%left
  end do
  select type (node)
  class is (fibonacci_leave_t)
     leave => node
  class default
     leave => null()
  end select
end subroutine fibonacci_node_find_leftmost
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: find_rightmost => fibonacci_node_find_rightmost
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_find_rightmost (this, leave)
  class(fibonacci_node_t), intent(in), target :: this
  class(fibonacci_leave_t), pointer, intent(out) :: leave
  class(fibonacci_node_t), pointer :: node
  node => this
  do while (associated (node%right))
     node => node%right
  end do
  select type (node)
  class is (fibonacci_leave_t)
     leave => node
  class default
     leave => null()
  end select
end subroutine fibonacci_node_find_rightmost
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: find => fibonacci_node_find
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_find (this, value, leave)
  class(fibonacci_node_t), intent(in), target :: this
  real(default), intent(in) :: value
  class(fibonacci_leave_t), pointer, intent(out) :: leave
  class(fibonacci_node_t), pointer :: node
  node => this
  do
     if (node >= value) then
        if (associated (node%left)) then
           node => node%left
        else
           call msg_warning ("fibonacci_node_find: broken tree!")
           leave => null()
           return
        end if
     else
        if (associated (node%right)) then
           node => node%right
        else
           call msg_warning ("fibonacci_node_find: broken tree!")
           leave => null()
           return
        end if
     end if
     select type (node)
     class is (fibonacci_leave_t)
        leave => node
        exit
     end select
  end do
end subroutine fibonacci_node_find
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: find_left_leave => fibonacci_node_find_left_leave
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_find_left_leave (this, leave)
  class(fibonacci_node_t), intent(in), target :: this
  class(fibonacci_node_t), pointer :: node
  class(fibonacci_leave_t), pointer, intent(out) :: leave
  nullify(leave)
  node => this
  do while (associated (node%up))
     if (associated (node%up%right, node)) then
        node => node%up%left
        do while (associated (node%right))
           node => node%right
        end do
        select type (node)
        class is (fibonacci_leave_t)
        leave => node
        end select
        exit
     end if
     node => node%up
  end do
end subroutine fibonacci_node_find_left_leave
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: find_right_leave => fibonacci_node_find_right_leave
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_find_right_leave (this, leave)
  class(fibonacci_node_t), intent(in), target :: this
  class(fibonacci_node_t), pointer :: node
  class(fibonacci_leave_t), pointer, intent(out) :: leave
  nullify (leave)
  node => this
  do while (associated (node%up))
     if (associated (node%up%left, node)) then
        node => node%up%right
        do while (associated (node%left))
           node => node%left
        end do
        select type (node)
        class is (fibonacci_leave_t)
        leave => node
        end select
        exit
     end if
     node => node%up
  end do
end subroutine fibonacci_node_find_right_leave
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: apply_to_leaves => fibonacci_node_apply_to_leaves
```

⟨*Muli fibonacci: procedures*⟩+≡

```
recursive subroutine fibonacci_node_apply_to_leaves (node, func, unit)
  class(fibonacci_node_t), intent(in), target :: node
  interface
     subroutine func (this, unit)
       import fibonacci_leave_t
       class(fibonacci_leave_t), intent(in), target :: this
       integer, intent(in), optional :: unit
     end subroutine func
  end interface
  integer, intent(in), optional :: unit
  select type (node)
  class is (fibonacci_leave_t)
     call func (node, unit)
  class default
     call node%left%apply_to_leaves (func, unit)
     call node%right%apply_to_leaves (func, unit)
  end select
end subroutine fibonacci_node_apply_to_leaves
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: apply_to_leaves_rl => fibonacci_node_apply_to_leaves_rl
```

⟨*Muli fibonacci: procedures*⟩+≡

```
recursive subroutine fibonacci_node_apply_to_leaves_rl (node, func, unit)
  class(fibonacci_node_t), intent(in), target :: node
  interface
     subroutine func (this, unit)
       import fibonacci_leave_t
       class(fibonacci_leave_t), intent(in), target :: this
       integer, intent(in), optional :: unit
     end subroutine func
  end interface
  integer, intent(in), optional :: unit
  select type (node)
  class is (fibonacci_leave_t)
     call func (node, unit)
  class default
     call node%right%apply_to_leaves_rl (func, unit)
     call node%left%apply_to_leaves_rl (func, unit)
  end select
end subroutine fibonacci_node_apply_to_leaves_rl
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: set_depth => fibonacci_node_set_depth
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_set_depth (this, depth)
  class(fibonacci_node_t), intent(inout) :: this
  integer, intent(in) :: depth
  this%depth = depth
```

```
end subroutine fibonacci_node_set_depth
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: append_left => fibonacci_node_append_left
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_append_left(this,new_branch)
  class(fibonacci_node_t),target :: this
  class(fibonacci_node_t),target :: new_branch
  this%left => new_branch
  new_branch%up => this
end subroutine fibonacci_node_append_left
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: append_right => fibonacci_node_append_right
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_append_right (this, new_branch)
  class(fibonacci_node_t), intent(inout), target :: this
  class(fibonacci_node_t), target :: new_branch
  this%right => new_branch
  new_branch%up => this
end subroutine fibonacci_node_append_right
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: replace => fibonacci_node_replace
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_replace (this, old_node)
  class(fibonacci_node_t), intent(inout), target :: this
  class(fibonacci_node_t), target :: old_node
  if (associated (old_node%up)) then
     if (old_node%is_left_child ()) then
        old_node%up%left => this
     else
        if (old_node%is_right_child ()) then
           old_node%up%right => this
        end if
     end if
     this%up => old_node%up
  else
     nullify (this%up)
  end if
end subroutine fibonacci_node_replace
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: swap => fibonacci_node_swap_nodes
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_swap_nodes (left, right)
  class(fibonacci_node_t), target, intent(inout) :: left, right
  class(fibonacci_node_t), pointer :: left_left, right_right
  class(measure_class_t), pointer::down
  ! swap branches
```

```
      left_left => left%left
      right_right => right%right
      left%left => right%right
      right%right => left_left
      ! repair up components
      right_right%up => left
      left_left%up => right
      ! repair down components
      down => left%down
      left%down => right%down
      right%down => down
    end subroutine fibonacci_node_swap_nodes

  !  subroutine fibonacci_node_swap_nodes (this, that)
  !    class(fibonacci_node_t),target :: this
  !    class(fibonacci_node_t), pointer, intent(in) :: that
  !    class(fibonacci_node_t), pointer :: par_i, par_a
  !    par_i => this%up
  !    par_a => that%up
  !    if (associated (par_i%left, this)) then
  !       par_i%left => that
  !    else
  !       par_i%right => that
  !    end if
  !    if (associated (par_a%left, that)) then
  !       par_a%left => this
  !    else
  !       par_a%right => this
  !    end if
  !    this%up => par_a
  !    that%up => par_i
  !  end subroutine fibonacci_node_swap_nodes
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
    procedure :: flip => fibonacci_node_flip_children
```

⟨*Muli fibonacci: procedures*⟩+≡

```
    subroutine fibonacci_node_flip_children (this)
      class(fibonacci_node_t), intent(inout) :: this
      class(fibonacci_node_t), pointer :: child
      child => this%left
      this%left => this%right
      this%right => child
    end subroutine fibonacci_node_flip_children
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
    procedure :: rip => fibonacci_node_rip
```

⟨*Muli fibonacci: procedures*⟩+≡

```
    subroutine fibonacci_node_rip (this)
      class(fibonacci_node_t), intent(inout), target :: this
      if (this%is_left_child ()) then
         nullify (this%up%left)
      end if
```

```
    if (this%is_right_child ()) then
       nullify (this%up%right)
    end if
    nullify (this%up)
  end subroutine fibonacci_node_rip
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: remove_and_keep_parent => fibonacci_node_remove_and_keep_parent
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  subroutine fibonacci_node_remove_and_keep_parent (this, pa)
    class(fibonacci_node_t), intent(inout), target :: this
    class(fibonacci_node_t), intent(out), pointer :: pa
    class(fibonacci_node_t), pointer :: twin
    if (.not. (this%is_root ())) then
       pa => this%up
       if (this%is_left_child ()) then
          twin => pa%right
       else
          twin => pa%left
       end if
       twin%up => pa%up
       if (associated (twin%left)) then
          twin%left%up => pa
       end if
       if (associated (twin%right)) then
          twin%right%up => pa
       end if
       call pa%copy_node (twin)
       select type (pa)
       class is (fibonacci_root_t)
          call pa%set_leftmost ()
          call pa%set_rightmost ()
       end select
       if (associated (this%right)) then
          this%right%left => this%left
       end if
       if (associated (this%left)) then
          this%left%right => this%right
       end if
       nullify (this%left)
       nullify (this%right)
       nullify (this%up)
       deallocate (twin)
    else
       pa => this
    end if
  end subroutine fibonacci_node_remove_and_keep_parent
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: remove_and_keep_twin => fibonacci_node_remove_and_keep_twin
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  subroutine fibonacci_node_remove_and_keep_twin (this, twin)
```

```
    class(fibonacci_node_t), intent(inout), target :: this
    class(fibonacci_node_t), intent(out), pointer :: twin
    class(fibonacci_node_t), pointer :: pa
    if (.not. (this%is_root ())) then
       pa => this%up
       if (.not. pa%is_root ()) then
          if (this%is_left_child ()) then
             twin => pa%right
          else
             twin => pa%left
          end if
          if (pa%is_left_child ()) then
             pa%up%left => twin
          else
             pa%up%right => twin
          end if
       end if
       twin%up => pa%up
       if (associated (this%right)) then
          this%right%left => this%left
       end if
       if (associated (this%left)) then
          this%left%right => this%right
       end if
       nullify (this%left)
       nullify (this%right)
       nullify (this%up)
       deallocate (pa)
    end if
  end subroutine fibonacci_node_remove_and_keep_twin
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: rotate_left => fibonacci_node_rotate_left
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  subroutine fibonacci_node_rotate_left (this)
    class(fibonacci_node_t), intent(inout), target :: this
    call this%swap (this%right)
    call this%right%flip ()
    call this%right%update_depth_unsave ()
    call this%flip ()
    ! value = this%value
    ! this%value = this%left%value
    ! this%left%value = value
  end subroutine fibonacci_node_rotate_left
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: rotate_right => fibonacci_node_rotate_right
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  subroutine fibonacci_node_rotate_right (this)
    class(fibonacci_node_t), intent(inout), target :: this
    call this%left%swap (this)
    call this%left%flip ()
```

```
    call this%left%update_depth_unsave ()
    call this%flip ()
    ! value = this%value
    ! this%value = this%right%value
    ! this%right%value = value
  end subroutine fibonacci_node_rotate_right
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: rotate => fibonacci_node_rotate
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  subroutine fibonacci_node_rotate (this)
    class(fibonacci_node_t), intent(inout), target :: this
    if (this%is_left_short ()) then
       call this%rotate_left ()
    else
       if (this%is_right_short ()) then
          call this%rotate_right ()
       end if
    end if
  end subroutine fibonacci_node_rotate
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: balance_node => fibonacci_node_balance_node
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  subroutine fibonacci_node_balance_node (this, changed)
    class(fibonacci_node_t), intent(inout), target :: this
    logical, intent(out) :: changed
    changed = .false.
    if (this%is_left_too_short ()) then
       if (this%right%is_right_short ()) then
          call this%right%rotate_right
       end if
       call this%rotate_left ()
       changed = .true.
    else
       if (this%is_right_too_short ()) then
          if (this%left%is_left_short ()) then
             call this%left%rotate_left
          end if
          call this%rotate_right ()
          changed = .true.
       end if
    end if
  end subroutine fibonacci_node_balance_node
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: update_depth_save => fibonacci_node_update_depth_save
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  subroutine fibonacci_node_update_depth_save (this, updated)
    class(fibonacci_node_t), intent(inout) :: this
    logical, intent(out) :: updated
```

```
    integer :: left, right, new_depth
    if (associated (this%left)) then
       left = this%left%depth + 1
    else
       left = -1
    end if
    if (associated (this%right)) then
       right = this%right%depth + 1
    else
       right = -1
    end if
    new_depth = max(left, right)
    if (this%depth == new_depth) then
       updated = .false.
    else
       this%depth = new_depth
       updated = .true.
    end if
  end subroutine fibonacci_node_update_depth_save
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: update_depth_unsave => fibonacci_node_update_depth_unsave
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  subroutine fibonacci_node_update_depth_unsave (this)
    class(fibonacci_node_t), intent(inout) :: this
    this%depth = max (this%left%depth+1, this%right%depth+1)
  end subroutine fibonacci_node_update_depth_unsave
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: repair => fibonacci_node_repair
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  subroutine fibonacci_node_repair (this)
    class(fibonacci_node_t), intent(inout), target :: this
    class(fibonacci_node_t), pointer:: node
    logical :: new_depth, new_balance
    new_depth = .true.
    node => this
    do while ((new_depth .or. new_balance) .and. (associated (node)))
       call node%balance_node (new_balance)
       call node%update_depth_save (new_depth)
       node => node%up
    end do
  end subroutine fibonacci_node_repair
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
  procedure :: is_left_short => fibonacci_node_is_left_short
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  elemental logical function fibonacci_node_is_left_short(this)
    class(fibonacci_node_t), intent(in) :: this
    fibonacci_node_is_left_short = (this%left%depth < this%right%depth)
  end function fibonacci_node_is_left_short
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: is_right_short => fibonacci_node_is_right_short
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_node_is_right_short (this)
  class(fibonacci_node_t), intent(in) :: this
  fibonacci_node_is_right_short = (this%right%depth < this%left%depth)
end function fibonacci_node_is_right_short
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: is_unbalanced => fibonacci_node_is_unbalanced
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_node_is_unbalanced (this)
  class(fibonacci_node_t), intent(in) :: this
  fibonacci_node_is_unbalanced = &
       (this%is_left_short () .or. this%is_right_short ())
end function fibonacci_node_is_unbalanced
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: is_left_too_short => fibonacci_node_is_left_too_short
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_node_is_left_too_short (this)
  class(fibonacci_node_t), intent(in) :: this
  fibonacci_node_is_left_too_short = (this%left%depth+1 < this%right%depth)
end function fibonacci_node_is_left_too_short
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: is_right_too_short => fibonacci_node_is_right_too_short
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_node_is_right_too_short (this)
  class(fibonacci_node_t), intent(in) :: this
  fibonacci_node_is_right_too_short = (this%right%depth+1 < this%left%depth)
end function fibonacci_node_is_right_too_short
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: is_too_unbalanced => fibonacci_node_is_too_unbalanced
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_node_is_too_unbalanced (this)
  class(fibonacci_node_t), intent(in) :: this
  fibonacci_node_is_too_unbalanced = &
       (this%is_left_too_short() .or. this%is_right_too_short())
end function fibonacci_node_is_too_unbalanced
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: is_left_child => fibonacci_node_is_left_child
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_node_is_left_child (this)
  class(fibonacci_node_t), intent(in),target :: this
  fibonacci_node_is_left_child = associated (this%up%left, this)
end function fibonacci_node_is_left_child
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
procedure :: is_right_child => fibonacci_node_is_right_child
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_node_is_right_child (this)
  class(fibonacci_node_t), intent(in),target :: this
  fibonacci_node_is_right_child = associated (this%up%right, this)
end function fibonacci_node_is_right_child
```

⟨*Muli fibonacci: fibonacci node: TBP*⟩+≡

```
! user
! node
! tree
! procedure :: balance
! procedure :: sort
! procedure :: merge
! procedure :: split
```

⟨*Muli fibonacci: public*⟩+≡

```
public :: fibonacci_leave_t
```

⟨*Muli fibonacci: types*⟩+≡

```
type, extends (fibonacci_node_t) :: fibonacci_leave_t
   ! class(measure_class_t), pointer :: content
contains
⟨Muli fibonacci: fibonacci leave: TBP⟩
end type fibonacci_leave_t
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩≡

```
! procedure :: write_to_marker => fibonacci_leave_write_to_marker
! procedure :: read_from_marker => fibonacci_leave_read_from_marker
procedure :: print_to_unit => fibonacci_leave_print_to_unit
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_leave_print_to_unit &
     (this, unit, parents, components, peers)
  class(fibonacci_leave_t), intent(in) :: this
  integer, intent(in) :: unit
  integer(dik), intent(in) :: parents, components, peers
  class(ser_class_t), pointer :: ser
  if (parents > 0)  call fibonacci_node_print_to_unit &
       (this, unit, parents-i_one, components, -i_one)
  write(unit, "(A)")  "Components of fibonacci_leave_t:"
  ser => this%down
  call serialize_print_comp_pointer &
       (ser, unit, parents, components, peers, "Content:")
end subroutine fibonacci_leave_print_to_unit
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure, nopass :: get_type => fibonacci_leave_get_type
```

⟨*Muli fibonacci: procedures*⟩+≡

```
pure subroutine fibonacci_leave_get_type (type)
  character(:), allocatable, intent(out) :: type
  allocate (type, source="fibonacci_leave_t")
```

```
end subroutine fibonacci_leave_get_type
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: deallocate_all => fibonacci_leave_deallocate_all
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_leave_deallocate_all (this)
  class(fibonacci_leave_t), intent(inout) :: this
  if (associated (this%down)) then
     deallocate (this%down)
  end if
end subroutine fibonacci_leave_deallocate_all
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: pick => fibonacci_leave_pick
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_leave_pick (this)
  class(fibonacci_leave_t), target, intent(inout) :: this
  class(fibonacci_node_t), pointer :: other
  class(fibonacci_root_t), pointer :: root
  ! call this%up%print_parents()
  call this%find_root (root)
  if (associated (this%up, root)) then
     if (this%up%depth < 2) then
        call msg_error ("fibonacci_leave_pick: Cannot pick leave. " // &
             "Tree must have at least three leaves.")
     else
        call this%remove_and_keep_parent (other)
        call other%repair ()
     end if
  else
     call this%remove_and_keep_twin (other)
     call other%up%repair ()
  end if
  if (associated (root%leftmost, this))  call root%set_leftmost ()
  if (associated (root%rightmost, this))  call root%set_rightmost ()
end subroutine fibonacci_leave_pick
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: get_left => fibonacci_leave_get_left
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_leave_get_left (this, leave)
  class(fibonacci_leave_t), intent(in) :: this
  class(fibonacci_leave_t), intent(out), pointer :: leave
  class(fibonacci_node_t), pointer :: node
  node => this%left
  select type (node)
  class is (fibonacci_leave_t)
     leave => node
  end select
end subroutine fibonacci_leave_get_left
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: get_right => fibonacci_leave_get_right
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_leave_get_right (this, leave)
  class(fibonacci_leave_t), intent(in) :: this
  class(fibonacci_leave_t), intent(out), pointer :: leave
  class(fibonacci_node_t), pointer :: node
  ! print *,"fibonacci_leave_get_right"
  ! call this%down%print_little
  if (associated (this%right)) then
     node => this%right
     ! call node%down%print_little
     select type (node)
     class is (fibonacci_leave_t)
        leave => node
     end select
  else
     ! print *,"no right leave"
     nullify (leave)
  end if
end subroutine fibonacci_leave_get_right
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: write_pstricks => fibonacci_leave_write_pstricks
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_leave_write_pstricks (this, unitnr)
  class(fibonacci_leave_t), intent(in), target :: this
  integer, intent(in) :: unitnr
  write (unitnr, "(A,I3,A,F9.3,A)")  &
       "\begin{psTree}{\Toval[linecolor=green]{\node{", this%depth, "}{", &
       this%measure(), "}}}"
  if (associated (this%left)) then
     write (unitnr, "(A,A,A)")  "\Tr[", le_kid, "]{}"
  end if
  if (associated (this%right)) then
     write (unitnr, "(A,A,A)")  "\Tr[", le_kid, "]{}"
  end if
  write (unitnr, "(A)")  "\end{psTree}"
end subroutine fibonacci_leave_write_pstricks
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: copy_content => fibonacci_leave_copy_content
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_leave_copy_content (this, content)
  class(fibonacci_leave_t) :: this
  class(measure_class_t), intent(in) :: content
  allocate (this%down, source=content)
end subroutine fibonacci_leave_copy_content
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: set_content => fibonacci_leave_set_content
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_leave_set_content (this, content)
  class(fibonacci_leave_t) :: this
  class(measure_class_t), target, intent(in) :: content
  this%down => content
end subroutine fibonacci_leave_set_content
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: get_content => fibonacci_leave_get_content
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_leave_get_content (this, content)
  class(fibonacci_leave_t), intent(in) :: this
  class(measure_class_t), pointer :: content
  content => this%down
end subroutine fibonacci_leave_get_content
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure, nopass :: is_inner => fibonacci_leave_is_inner
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_leave_is_inner ()
  fibonacci_leave_is_inner = .false.
end function fibonacci_leave_is_inner
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure, nopass :: is_leave => fibonacci_leave_is_leave
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_leave_is_leave ()
  fibonacci_leave_is_leave = .true.
end function fibonacci_leave_is_leave
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: insert_leave_by_node => fibonacci_leave_insert_leave_by_node
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_leave_insert_leave_by_node (this, new_leave)
  class(fibonacci_leave_t), target, intent(inout) :: this,new_leave
  class(fibonacci_node_t), pointer :: parent, new_node
  parent => this%up
  !print *, associated (this%left), associated (this%right)
  if (this < new_leave) then
     call fibonacci_node_spawn (new_node, this, new_leave, this%left, this%right)
     ! print *,"Repair! ",this%measure(),new_leave%measure()
  else
     call fibonacci_node_spawn (new_node, new_leave, this, this%left, this%right)
  end if
  if (associated (parent%left, this)) then
     call parent%append_left (new_node)
  else
     call parent%append_right (new_node)
  end if
  call parent%repair ()
```

```
end subroutine fibonacci_leave_insert_leave_by_node
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: is_left_short => fibonacci_leave_is_left_short
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_leave_is_left_short (this)
  class(fibonacci_leave_t), intent(in) :: this
  fibonacci_leave_is_left_short = .false.
end function fibonacci_leave_is_left_short
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: is_right_short => fibonacci_leave_is_right_short
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_leave_is_right_short (this)
  class(fibonacci_leave_t), intent(in) :: this
  fibonacci_leave_is_right_short = .false.
end function fibonacci_leave_is_right_short
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: is_unbalanced => fibonacci_leave_is_unbalanced
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_leave_is_unbalanced (this)
  class(fibonacci_leave_t), intent(in) :: this
  fibonacci_leave_is_unbalanced = .false.
end function fibonacci_leave_is_unbalanced
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: is_left_too_short => fibonacci_leave_is_left_too_short
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_leave_is_left_too_short (this)
  class(fibonacci_leave_t), intent(in) :: this
  fibonacci_leave_is_left_too_short = .false.
end function fibonacci_leave_is_left_too_short
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: is_right_too_short => fibonacci_leave_is_right_too_short
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_leave_is_right_too_short (this)
  class(fibonacci_leave_t), intent(in) :: this
  fibonacci_leave_is_right_too_short = .false.
end function fibonacci_leave_is_right_too_short
```

⟨*Muli fibonacci: fibonacci leave: TBP*⟩+≡

```
procedure :: is_too_unbalanced => fibonacci_leave_is_too_unbalanced
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_leave_is_too_unbalanced (this)
  class(fibonacci_leave_t), intent(in) :: this
  fibonacci_leave_is_too_unbalanced = .false.
end function fibonacci_leave_is_too_unbalanced
```

⟨*Muli fibonacci: public*⟩+≡

```
public :: fibonacci_root_t
```

⟨*Muli fibonacci: types*⟩+≡

```
type, extends (fibonacci_node_t) :: fibonacci_root_t
   logical::is_valid_c=.false.
   class(fibonacci_leave_t),pointer :: leftmost => null()
   class(fibonacci_leave_t),pointer :: rightmost => null()
contains
⟨Muli fibonacci: fibonacci root: TBP⟩
   procedure :: is_left_child => fibonacci_root_is_left_child
   procedure :: is_right_child => fibonacci_root_is_right_child
end type fibonacci_root_t
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩≡

```
procedure :: write_to_marker => fibonacci_root_write_to_marker
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_write_to_marker (this, marker, status)
  class(fibonacci_root_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  ! call marker%mark_begin ("fibonacci_root_t")
  call fibonacci_node_write_to_marker (this, marker, status)
  ! marker%mark_end ("fibonacci_root_t")
end subroutine fibonacci_root_write_to_marker
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: read_target_from_marker => fibonacci_root_read_target_from_marker
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_read_target_from_marker (this, marker, status)
  class(fibonacci_root_t), target, intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  ! call marker%pick_begin ("fibonacci_root_t", status)
  call fibonacci_node_read_from_marker (this, marker, status)
  call this%find_leftmost (this%leftmost)
  call this%find_rightmost (this%rightmost)
  ! call marker%pick_end ("fibonacci_root_t", status)
end subroutine fibonacci_root_read_target_from_marker
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: print_to_unit => fibonacci_root_print_to_unit
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_print_to_unit (this, unit, parents, components, peers)
  class(fibonacci_root_t), intent(in) :: this
  integer, intent(in) :: unit
  integer(dik), intent(in) :: parents, components, peers
  class(ser_class_t), pointer :: ser
  if (parents > 0)  call fibonacci_node_print_to_unit &
       (this, unit, parents-1, components, peers)
  write (unit, "(A)")  "Components of fibonacci_root_t:"
  ser => this%leftmost
  call serialize_print_peer_pointer &
       (ser, unit, parents, components, min(peers, i_one), "Leftmost: ")
  ser => this%rightmost
  call serialize_print_peer_pointer &
       (ser, unit, parents, components, min(peers, i_one), "Rightmost:")
end subroutine fibonacci_root_print_to_unit
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_root_is_left_child (this)
  class(fibonacci_root_t),target, intent(in) :: this
  fibonacci_root_is_left_child = .false.
end function fibonacci_root_is_left_child
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental logical function fibonacci_root_is_right_child (this)
  class(fibonacci_root_t),target, intent(in) :: this
  fibonacci_root_is_right_child = .false.
end function fibonacci_root_is_right_child
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure, nopass :: get_type => fibonacci_root_get_type
```

⟨*Muli fibonacci: procedures*⟩+≡

```
pure subroutine fibonacci_root_get_type (type)
  character(:),allocatable, intent(out) :: type
  allocate (type, source="fibonacci_root_t")
end subroutine fibonacci_root_get_type
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: get_leftmost=>fibonacci_root_get_leftmost
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_get_leftmost (this, leftmost)
  class(fibonacci_root_t), intent(in) :: this
  class(fibonacci_leave_t), pointer :: leftmost
  leftmost => this%leftmost
end subroutine fibonacci_root_get_leftmost
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: get_rightmost=>fibonacci_root_get_rightmost
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_get_rightmost (this, rightmost)
  class(fibonacci_root_t), intent(in) :: this
  class(fibonacci_leave_t),pointer :: rightmost
  rightmost => this%rightmost
end subroutine fibonacci_root_get_rightmost
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure, nopass :: is_root => fibonacci_root_is_root
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental function fibonacci_root_is_root ()
  logical::fibonacci_root_is_root
  fibonacci_root_is_root = .true.
end function fibonacci_root_is_root
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure, nopass :: is_inner => fibonacci_root_is_inner
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental function fibonacci_root_is_inner ()
  logical::fibonacci_root_is_inner
  fibonacci_root_is_inner = .false.
end function fibonacci_root_is_inner
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: is_valid => fibonacci_root_is_valid
```

⟨*Muli fibonacci: procedures*⟩+≡

```
elemental function fibonacci_root_is_valid (this)
  class(fibonacci_root_t), intent(in) :: this
  logical :: fibonacci_root_is_valid
  fibonacci_root_is_valid = this%is_valid_c
end function fibonacci_root_is_valid
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: count_leaves => fibonacci_root_count_leaves
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_count_leaves (this, n)
  class(fibonacci_root_t), intent(in) :: this
  integer, intent(out) :: n
  n = 0
  call fibonacci_node_count_leaves (this, n)
end subroutine fibonacci_root_count_leaves
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: write_pstricks => fibonacci_root_write_pstricks
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_write_pstricks (this, unitnr)
  class(fibonacci_root_t), intent(in), target :: this
  integer, intent(in) :: unitnr
  logical :: is_opened
  character :: is_sequential, is_formatted, is_writeable
  print *,"pstricks"
  inquire (unitnr, opened=is_opened, sequential=is_sequential, &
       formatted=is_formatted, write=is_writeable)
  if (is_opened) then
     if (is_sequential == "Y" .and. is_formatted == "Y " &
          .and. is_writeable == "Y") then
        ! write (unitnr, "(A,I3,A)")  &
        !     "\begin{psTree}{\Toval[linecolor=blue]{$", int(this%depth), "$}}"
        write (unitnr, "(A,I3,A,F9.3,A)")  &
             "\begin{psTree}{\Toval[linecolor=blue]{\node{", this%depth, &
             "}{", this%measure(), "}}}"
        if (associated (this%leftmost)) then
           call this%leftmost%write_pstricks (unitnr)
        else
           write (unitnr, "(A,A,A)") "\Tr[", no_kid, "]{}"
        end if
        if (associated (this%left)) then
           call this%left%write_pstricks (unitnr)
        else
           write (unitnr, "(A,A,A)") "\Tr[", no_kid, "]{}"
        end if
        if (associated (this%right)) then
           call this%right%write_pstricks (unitnr)
        else
           write (unitnr, "(A,A,A)") "\Tr[", no_kid, "]{}"
        end if
        if (associated (this%rightmost)) then
           call this%rightmost%write_pstricks (unitnr)
        else
           write(unitnr,'("\Tr[",a,"]{}")') no_kid
        end if
        write (unitnr, "(A)")  "\end{psTree}"
        write (unitnr, "(A)")  "\\"
     else
        write (*, "(A,I2,A)") &
           "fibonacci_node_write_pstricks: Unit ", unitnr, &
             " is not opened properly."
        write (*, "(A)")  "No output is written to unit."
     end if
  else
     write (*, "(A,I2,A)") &
        "fibonacci_node_write_pstricks: Unit ", unitnr, &
          " is not opened."
     write (*, "(A)")  "No output is written to unit."
  end if
end subroutine fibonacci_root_write_pstricks
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: copy_root => fibonacci_root_copy_root
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_copy_root (this, primitive)
  class(fibonacci_root_t), intent(out) :: this
  class(fibonacci_root_t), intent(in) :: primitive
  call fibonacci_node_copy_node (this, primitive)
  this%leftmost => primitive%leftmost
  this%rightmost => primitive%rightmost
end subroutine fibonacci_root_copy_root
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: push_by_content => fibonacci_root_push_by_content
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_push_by_content (this, content)
  class(fibonacci_root_t), target, intent(inout) :: this
  class(measure_class_t), target, intent(in) :: content
  class(fibonacci_leave_t), pointer :: node
  ! print *,"fibonacci_root_push_by_content: ",content%measure()
  allocate (node)
  node%down => content
  call this%push_by_leave (node)
end subroutine fibonacci_root_push_by_content
```

This is a workaround for gfortran bug 44696. This subroutine is a merge of `fibonacci_tree_push_by_node`, `fibonacci_node_find`, and `fibonacci_leave_insert_leave_by_node`.

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: push_by_leave => fibonacci_root_push_by_leave
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_push_by_leave (this, new_leave)
  class(fibonacci_root_t), target, intent(inout) :: this
  class(fibonacci_leave_t), pointer, intent(inout) :: new_leave
  class(fibonacci_leave_t), pointer :: old_leave
  class(fibonacci_node_t), pointer :: node, new_node, leave_c
  ! write (11, fmt=*)  "push by leave(", new_leave%measure(), ")\\" !PSTRICKS
  ! flush(11)  !PSTRICKS
  if (new_leave <= this%leftmost) then
     old_leave => this%leftmost
     this%leftmost => new_leave
     node => old_leave%up
     call fibonacci_node_spawn &
          (new_node, new_leave, old_leave, old_leave%left, old_leave%right)
     call node%append_left (new_node)
  else
     if (new_leave > this%rightmost) then
        old_leave => this%rightmost
        this%rightmost => new_leave
        node => old_leave%up
        call fibonacci_node_spawn &
             (new_node, old_leave, new_leave, old_leave%left, old_leave%right)
        call node%append_right (new_node)
     else
```

```
         node => this
         do
            if (new_leave <= node) then
               leave_c => node%left
               select type (leave_c)
               class is (fibonacci_leave_t)
                  if (new_leave <= leave_c) then
                     ! print *,"left left"
                     call fibonacci_node_spawn (new_node, new_leave, &
                          leave_c, leave_c%left, leave_c%right)
                  else
                     ! print *,"left right"
                     call fibonacci_node_spawn (new_node, leave_c, &
                          new_leave, leave_c%left, leave_c%right)
                  end if
                  call node%append_left (new_node)
                  exit
               class default
                  ! print *,"left"
                  node => node%left
               end select
            else
               leave_c => node%right
               select type (leave_c)
               class is (fibonacci_leave_t)
                  if (new_leave <= leave_c) then
                     ! print *,"right left"
                     call fibonacci_node_spawn (new_node, new_leave, &
                          leave_c, leave_c%left, leave_c%right)
                  else
                     ! print *,"right right"
                     call fibonacci_node_spawn (new_node, leave_c, &
                          new_leave, leave_c%left, leave_c%right)
                  end if
                  call node%append_right (new_node)
                  exit
               class default
                  ! print *,"right"
                  node => node%right
               end select
            end if
         end do
      end if
   end if
   ! call this%write_pstricks(11) ! PSTRICKS
   ! flush(11) ! PSTRICKS
   ! write(11,fmt=*)"repair\\" ! PSTRICKS
   call node%repair ()
   ! call this%write_pstricks (11) !PSTRICKS
   ! flush(11) !PSTRICKS
   ! call node%update_value (right_value)
   ! call this%write_pstricks (11)
   ! print *, new_node%value, new_node%left%value, new_node%right%value
end subroutine fibonacci_root_push_by_leave
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: pop_left => fibonacci_root_pop_left
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_pop_left (this, leave)
  class(fibonacci_root_t), intent(inout), target :: this
  class(fibonacci_leave_t), pointer, intent(out) :: leave
  class(fibonacci_node_t), pointer :: parent, grand
  ! write (11,fmt=*) "fibonacci root pop left\\"  ! PSTRICKS
  ! flush (11)   ! PSTRICKS
  leave => this%leftmost
  if (this%left%depth >= 1) then
     parent => leave%up
     grand => parent%up
     grand%left => parent%right
     parent%right%up => grand
     deallocate (parent)
     parent => grand%left
     if (.not. parent%is_leave ()) then
        parent => parent%left
     end if
     select type (parent)
     class is (fibonacci_leave_t)
        this%leftmost => parent
     class default
        call parent%print_all()
        call msg_fatal ("fibonacci_root_pop_left: ERROR: leftmost is no leave.")
     end select
     ! call this%write_pstricks (11)   ! PSTRICKS
     ! flush (11)   ! PSTRICKS
     ! write (11,fmt=*)  "fibonacci node repair\\"   ! PSTRICKS
     ! flush (11)   ! PSTRICKS
     call grand%repair ()
  else
     if (this%left%depth == 0 .and. this%right%depth == 1) then
        parent => this%right
        parent%right%up => this
        parent%left%up => this
        this%left => parent%left
        this%right => parent%right
        this%depth = 1
        deallocate (parent)
        parent => this%left
        select type (parent)
        class is (fibonacci_leave_t)
        this%leftmost => parent
        end select
        this%down => this%leftmost%down
     end if
  end if
  nullify (leave%right%left)
  nullify (leave%up)
  nullify (leave%right)
```

```
    nullify (this%leftmost%left)
    ! call this%write_pstricks (11)   ! PSTRICKS
    ! flush (11)   ! PSTRICKS
  end subroutine fibonacci_root_pop_left
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
  procedure :: pop_right => fibonacci_root_pop_right
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  subroutine fibonacci_root_pop_right (this, leave)
    class(fibonacci_root_t), intent(inout), target :: this
    class(fibonacci_leave_t), pointer, intent(out) :: leave
    class(fibonacci_node_t), pointer :: parent, grand
    leave => this%rightmost
    if (this%right%depth >= 1) then
       parent => leave%up
       grand => parent%up
       grand%right => parent%left
       parent%left%up => grand
       deallocate (parent)
       parent => grand%right
       if (.not. parent%is_leave ()) then
          parent => parent%right
       end if
       select type (parent)
       class is (fibonacci_leave_t)
          this%rightmost => parent
       class default
          call parent%print_all()
          call msg_fatal ("fibonacci_root_pop_left: ERROR: leftmost is no leave.")
       end select
       call grand%repair ()
    else
       if (this%right%depth == 0 .and. this%left%depth == 1) then
          parent => this%left
          parent%left%up => this
          parent%right%up => this
          this%right => parent%right
          this%left => parent%left
          this%depth = 1
          deallocate (parent)
          parent => this%right
          select type (parent)
          class is (fibonacci_leave_t)
          this%rightmost => parent
          end select
          this%down => this%rightmost%down
       end if
    end if
  end subroutine fibonacci_root_pop_right
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
  procedure :: list_to_tree => fibonacci_root_list_to_tree
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_list_to_tree (this, n_leaves, leave_list_target)
  class(fibonacci_root_t), target, intent(inout) :: this
  integer, intent(in) :: n_leaves
  type(fibonacci_leave_list_t), target, intent(in) :: leave_list_target
  ! class(fibonacci_root_t), pointer, intent(out) :: tree
  integer :: depth, n_deep, n_merge
  class(fibonacci_node_t), pointer :: node
  class(fibonacci_leave_list_t), pointer :: leave_list
  class(fibonacci_leave_t), pointer :: content
  real(default) :: up_value
  leave_list => leave_list_target
  call ilog2 (n_leaves, depth, n_deep)
  n_deep = n_deep * 2
  n_merge = 0
  this%depth = depth
  node => this
  OUTER: do
     do while (depth > 1)
        depth = depth - 1
        allocate (node%left)
        node%left%up => node
        node => node%left
        node%depth = depth
     end do
     node%left => leave_list%leave
     node%down => leave_list%leave%down
     leave_list => leave_list%next
     node%right => leave_list%leave
     content => leave_list%leave
     leave_list => leave_list%next
     n_merge = n_merge + 2
     INNER: do
        if (associated (node%up)) then
           if (node%is_left_child ()) then
              if (n_merge == n_deep .and. depth == 1) then
                 node => node%up
                 node%right => leave_list%leave
                 node%right%up => node
                 node%down => content%down
                 content => leave_list%leave
                 leave_list => leave_list%next
                 n_merge = n_merge + 1
                 cycle
              end if
              exit INNER
           else
              node => node%up
              depth = depth + 1
           end if
        else
           exit OUTER
        end if
     end do INNER
```

```
      node => node%up
      node%down => content%down
      allocate (node%right)
      node%right%up => node
      node => node%right
      if (n_deep == n_merge) then
         depth = depth - 1
      end if
      node%depth = depth
   end do OUTER
   call this%set_leftmost
   call this%set_rightmost
end subroutine fibonacci_root_list_to_tree
```

This subroutine has neither been used nor revised for a long time, so it might be broken.

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: merge => fibonacci_root_merge
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_merge(this_tree,that_tree,merge_tree)
  class(fibonacci_root_t), intent(in) :: this_tree
  class(fibonacci_root_t), intent(in) :: that_tree
  class(fibonacci_root_t), pointer, intent(out) :: merge_tree
  class(fibonacci_leave_t), pointer :: this_leave, that_leave, old_leave
  type(fibonacci_leave_list_t), target :: leave_list
  class(fibonacci_leave_list_t), pointer :: last_leave
  integer :: n_leaves
  if (associated (this_tree%leftmost) .and. associated (that_tree%leftmost)) then
     n_leaves = 1
     this_leave => this_tree%leftmost
     that_leave => that_tree%leftmost
     if (this_leave < that_leave) then
        allocate (leave_list%leave, source=this_leave)
        call this_leave%find_right_leave (this_leave)
     else
        allocate (leave_list%leave, source=that_leave)
        call that_leave%find_right_leave (that_leave)
     end if
     last_leave => leave_list
     do while (associated (this_leave) .and. associated (that_leave))
        if (this_leave < that_leave) then
           old_leave => this_leave
           call this_leave%find_right_leave (this_leave)
        else
           old_leave=>that_leave
           call that_leave%find_right_leave (that_leave)
        end if
        allocate (last_leave%next)
        last_leave => last_leave%next
        allocate (last_leave%leave, source=old_leave)
        n_leaves = n_leaves + 1
     end do
     if (associated (this_leave)) then
```

```
            old_leave => this_leave
         else
            old_leave => that_leave
         end if
         do while (associated (old_leave))
            allocate (last_leave%next)
            last_leave => last_leave%next
            allocate (last_leave%leave, source=old_leave)
            n_leaves = n_leaves + 1
            call old_leave%find_right_leave (old_leave)
         end do
         allocate (merge_tree)
         call merge_tree%list_to_tree (n_leaves, leave_list)
      else
         n_leaves = 0
      end if
      if (associated (leave_list%next)) then
         last_leave => leave_list%next
         do while (associated (last_leave%next))
            leave_list%next => last_leave%next
            deallocate (last_leave)
            last_leave => leave_list%next
         end do
         deallocate (last_leave)
      end if
   end subroutine fibonacci_root_merge
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
   procedure :: set_leftmost => fibonacci_root_set_leftmost
```

⟨*Muli fibonacci: procedures*⟩+≡

```
   subroutine fibonacci_root_set_leftmost (this)
      class(fibonacci_root_t) :: this
      call this%find_leftmost (this%leftmost)
   end subroutine fibonacci_root_set_leftmost
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
   procedure :: set_rightmost => fibonacci_root_set_rightmost
```

⟨*Muli fibonacci: procedures*⟩+≡

```
   subroutine fibonacci_root_set_rightmost (this)
      class(fibonacci_root_t) :: this
      call this%find_rightmost (this%rightmost)
   end subroutine fibonacci_root_set_rightmost
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
   procedure :: init_by_leave => fibonacci_root_init_by_leave
```

⟨*Muli fibonacci: procedures*⟩+≡

```
   subroutine fibonacci_root_init_by_leave (this, left_leave, right_leave)
      class(fibonacci_root_t), target, intent(out) :: this
      class(fibonacci_leave_t), target, intent(in) :: left_leave, right_leave
      if (left_leave <= right_leave) then
         this%left => left_leave
```

```
      this%right => right_leave
      this%leftmost => left_leave
      this%rightmost => right_leave
   else
      this%left => right_leave
      this%right => left_leave
      this%leftmost => right_leave
      this%rightmost => left_leave
   end if
   this%left%up => this
   this%right%up => this
   this%down => this%leftmost%down
   this%depth = 1
   this%leftmost%right => this%rightmost
   this%rightmost%left => this%leftmost
   this%is_valid_c = .true.
end subroutine fibonacci_root_init_by_leave
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: init_by_content => fibonacci_root_init_by_content
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_init_by_content (this, left_content, right_content)
  class(fibonacci_root_t), target, intent(out) :: this
  class(measure_class_t), intent(in), target :: left_content, right_content
  call this%reset ()
  print *, "fibonacci_root_init_by_content: ", left_content%measure (), &
       right_content%measure ()
  if (left_content < right_content) then
     call this%leftmost%set_content (left_content)
     call this%rightmost%set_content (right_content)
  else
     call this%leftmost%set_content (right_content)
     call this%rightmost%set_content (left_content)
  end if
  this%down => this%leftmost%down
  this%is_valid_c = .true.
end subroutine fibonacci_root_init_by_content
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
procedure :: reset => fibonacci_root_reset
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_root_reset (this)
  class(fibonacci_root_t), target, intent(inout) :: this
  call this%deallocate_tree ()
  allocate (this%leftmost)
  allocate (this%rightmost)
  this%depth = 1
  this%leftmost%depth = 0
  this%rightmost%depth = 0
  this%left => this%leftmost
  this%right => this%rightmost
  this%left%up => this
```

```
    this%right%up => this
    this%leftmost%right => this%rightmost
    this%rightmost%left => this%leftmost
  end subroutine fibonacci_root_reset
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
  procedure :: deallocate_tree => fibonacci_root_deallocate_tree
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  recursive subroutine fibonacci_root_deallocate_tree (this)
    class(fibonacci_root_t), intent(inout) :: this
    call this%deallocate_tree ()
    nullify (this%leftmost)
    nullify (this%rightmost)
  end subroutine fibonacci_root_deallocate_tree
```

⟨*Muli fibonacci: fibonacci root: TBP*⟩+≡

```
  procedure :: deallocate_all => fibonacci_root_deallocate_all
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  recursive subroutine fibonacci_root_deallocate_all (this)
    class(fibonacci_root_t), intent(inout) :: this
    call this%deallocate_all ()
    nullify (this%leftmost)
    nullify (this%rightmost)
  end subroutine fibonacci_root_deallocate_all
```

⟨*Muli fibonacci: types*⟩+≡

```
  ! class(serializable_ref_type), pointer :: ref_list
```

⟨*Muli fibonacci: types*⟩+≡

```
  type, extends (fibonacci_root_t) :: fibonacci_stub_t
   contains
   ⟨Muli fibonacci: fibonacci stub: TBP⟩
  end type fibonacci_stub_t
```

⟨*Muli fibonacci: fibonacci stub: TBP*⟩≡

```
  procedure :: write_to_marker => fibonacci_stub_write_to_marker
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  subroutine fibonacci_stub_write_to_marker (this, marker, status)
    class(fibonacci_stub_t), intent(in) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
  end subroutine fibonacci_stub_write_to_marker
```

⟨*Muli fibonacci: fibonacci stub: TBP*⟩+≡

```
  procedure :: read_target_from_marker => fibonacci_stub_read_target_from_marker
```

⟨*Muli fibonacci: procedures*⟩+≡

```
  subroutine fibonacci_stub_read_target_from_marker (this, marker, status)
    class(fibonacci_stub_t), target, intent(out) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
```

```
end subroutine fibonacci_stub_read_target_from_marker
```

⟨*Muli fibonacci: fibonacci stub: TBP*⟩+≡

```
! procedure :: print_to_unit => fibonacci_stub_print_to_unit
```

⟨*Muli fibonacci: fibonacci stub: TBP*⟩+≡

```
procedure, nopass :: get_type => fibonacci_stub_get_type
```

⟨*Muli fibonacci: procedures*⟩+≡

```
pure subroutine fibonacci_stub_get_type (type)
  character(:), allocatable, intent(out) :: type
  allocate (type, source="fibonacci_stub_t")
end subroutine fibonacci_stub_get_type
```

⟨*Muli fibonacci: fibonacci stub: TBP*⟩+≡

```
procedure :: push_by_content => fibonacci_stub_push_by_content
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_stub_push_by_content (this, content)
  class(fibonacci_stub_t), target, intent(inout) :: this
  class(measure_class_t), target, intent(in) :: content
  class(fibonacci_leave_t), pointer :: leave
  allocate (leave)
  call leave%set_content (content)
  call this%push_by_leave (leave)
end subroutine fibonacci_stub_push_by_content
```

⟨*Muli fibonacci: fibonacci stub: TBP*⟩+≡

```
procedure :: push_by_leave => fibonacci_stub_push_by_leave
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_stub_push_by_leave (this, new_leave)
  class(fibonacci_stub_t), target, intent(inout) :: this
  class(fibonacci_leave_t), pointer, intent(inout) :: new_leave
  class(fibonacci_leave_t), pointer :: old_leave
  if (this%depth < 1) then
     if (associated (this%leftmost)) then
        old_leave => this%leftmost
        call this%init_by_leave (old_leave, new_leave)
     else
        this%leftmost => new_leave
     end if
  else
     call fibonacci_root_push_by_leave (this, new_leave)
  end if
end subroutine fibonacci_stub_push_by_leave
```

⟨*Muli fibonacci: fibonacci stub: TBP*⟩+≡

```
procedure :: pop_left => fibonacci_stub_pop_left
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_stub_pop_left (this, leave)
  class(fibonacci_stub_t), intent(inout), target :: this
  class(fibonacci_leave_t), pointer, intent(out) :: leave
```

```
      if (this%depth < 2) then
         if (this%depth == 1) then
            leave => this%leftmost
            this%leftmost => this%rightmost
            nullify (this%rightmost)
            nullify (this%right)
            nullify (this%left)
            this%depth = 0
            this%is_valid_c = .false.
         else
            if (associated (this%leftmost)) then
               leave => this%leftmost
               nullify (this%leftmost)
            end if
         end if
      else
         call fibonacci_root_pop_left (this, leave)
      end if
    end subroutine fibonacci_stub_pop_left
```

⟨*Muli fibonacci: fibonacci stub: TBP*⟩+≡

```
    procedure :: pop_right => fibonacci_stub_pop_right
```

⟨*Muli fibonacci: procedures*⟩+≡

```
    subroutine fibonacci_stub_pop_right (this, leave)
      class(fibonacci_stub_t), intent(inout), target :: this
      class(fibonacci_leave_t), pointer, intent(out) :: leave
      if (this%depth < 2) then
         if (this%depth == 1) then
            this%is_valid_c = .false.
            if (associated (this%rightmost)) then
               leave => this%rightmost
               nullify (this%rightmost)
               nullify (this%right)
            else
               if (associated (this%leftmost)) then
                  leave => this%leftmost
                  nullify (this%leftmost)
                  nullify (this%left)
               else
                  nullify (leave)
               end if
            end if
         end if
      else
         call fibonacci_root_pop_right (this, leave)
      end if
    end subroutine fibonacci_stub_pop_right
```

⟨*Muli fibonacci: public*⟩+≡

```
    public :: fibonacci_leave_list_t
```

⟨*Muli fibonacci: types*⟩+≡

```
    type fibonacci_leave_list_t
```

```
      class(fibonacci_leave_t), pointer :: leave => null()
      class(fibonacci_leave_list_t), pointer :: next => null()
   end type fibonacci_leave_list_t
```

⟨*Muli fibonacci: procedures*⟩+≡

```
!  subroutine fibonacci_node_update_value (this, right_value)
!    class(fibonacci_node_t), target :: this
!    class(fibonacci_node_t), pointer:: node
!    real(default), intent(in) :: right_value
!    if (associated (this%left) .and. associated (this%right)) then
!       node => this
!       ! node%value = node%left%value
!       ! right_value = node%right%value
!       INNER: do while (associated (node%up))
!          if (node%is_right_child ()) then
!             node => node%up
!          else
!             node%up%value = right_value
!             exit
!          end if
!       end do INNER
!    end if
!  end subroutine fibonacci_node_update_value
```

⟨*Muli fibonacci: procedures*⟩+≡

```
!  subroutine fibonacci_root_copy_node (this, primitive)
!    class(fibonacci_root_t), intent(out) :: this
!    type(fibonacci_node_t), intent(in) :: primitive
!    call fibonacci_node_copy_node (this, primitive)
!    call primitive%find_leftmost (this%leftmost)
!    call primitive%find_rightmost (this%rightmost)
!  end subroutine fibonacci_root_copy_node
```

⟨*Muli fibonacci: procedures*⟩+≡

```
!  subroutine fibonacci_root_push_by_node (this, new_leave)
!    class(fibonacci_root_t), target, intent(inout) :: this
!    class(fibonacci_leave_t), pointer, intent(inout) :: new_leave
!    class(fibonacci_leave_t), pointer :: old_leave
!    if (new_leave <= this%leftmost) then
!       old_leave => this%leftmost
!       this%leftmost => new_leave
!    else
!       if (new_leave > this%rightmost) then
!          old_leave => this%rightmost
!          this%rightmost => new_leave
!       else
!          call this%find (new_leave%measure(), old_leave)
!       end if
!    end if
!    ! call old_leave%insert_leave_by_node (new_leave)
!    call fibonacci_leave_insert_leave_by_node (old_leave, new_leave)
!    call new_leave%up%repair ()
```

```
!    ! call new_leave%up%update_value ()
!  end subroutine fibonacci_root_push_by_node
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_leave_write_content (this, unit)
  class(fibonacci_leave_t), intent(in), target :: this
  integer,optional, intent(in) :: unit
  call this%down%print_all (unit)
end subroutine fibonacci_leave_write_content
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_leave_write (this, unit)
  class(fibonacci_leave_t), intent(in), target :: this
  integer,optional, intent(in) :: unit
  call this%print_all (unit)
end subroutine fibonacci_leave_write
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_leave_write_value (this, unit)
  class(fibonacci_leave_t), intent(in), target :: this
  integer, intent(in), optional :: unit
  if (present (unit)) then
     write(unit, fmt=*)  this%measure ()
  else
     print *, this%measure ()
  end if
  ! call this%print_little (unit)
end subroutine fibonacci_leave_write_value
```

⟨*Muli fibonacci: procedures*⟩+≡

```
subroutine fibonacci_node_spawn (new_node, left_leave, right_leave, &
     left_left_leave, right_right_leave)
  class(fibonacci_node_t), pointer, intent(out) :: new_node
  class(fibonacci_leave_t), target, intent(inout) :: left_leave, right_leave
  class(fibonacci_node_t), pointer, intent(inout) :: left_left_leave, &
       right_right_leave
  allocate (new_node)
  new_node%depth = 1
  if (associated (left_left_leave)) then
     left_left_leave%right => left_leave
     left_leave%left => left_left_leave
  else
     nullify (left_leave%left)
  end if
  if (associated (right_right_leave)) then
     right_right_leave%left => right_leave
     right_leave%right => right_right_leave
  else
     nullify (right_leave%right)
  end if
  new_node%left => left_leave
  new_node%right => right_leave
```

```
    new_node%down => left_leave%down
    new_node%depth = 1
    left_leave%up => new_node
    right_leave%up => new_node
    left_leave%right => right_leave
    right_leave%left => left_leave
  end subroutine fibonacci_node_spawn
```

22.7 Adaptive Quadrature

This file contains the module `muli_aq` which is an acronym for adaptive quadrature. The abstract type `aq_class` is extended and the deferred procedure `evaluate` is defined. `muli_aq` calls `evaluate` to evaluate the integrand at any point in the given range. We have tried to use a procedure pointer instead of the deferred type-bound procedure, but no compiler was able to handle procedure pointers plus cuba was not able to handle parameters, to wit dimensions of the integrand that should not get integrated. So we switched to this odd way of using inheritance. Meanwhile these problems got solved and we could go for a more straightforward solution, but it works fine as it is.

`aq_class` uses `muli_trapezium` to approximate the integral. It still has to do the subdivision of segments and has to check whether the precision goal is reached.

Finally, the result is written do disc using the serialization framework defined in `muli_base`. Since QCD is not expected to change frequently, the only reason to regenerate this function is a change of the used PDF set. Then you can read the integral from disc each time you run a simulation with the same PDF set.

⟨`muli_aq.f90`⟩≡

```
  ⟨File header⟩

  module muli_aq

  ⟨Use kinds⟩
    use constants
    use diagnostics
    use muli_base
    use muli_cuba
    use muli_trapezium
    use muli_fibonacci_tree

  ⟨Standard module head⟩

  ⟨Muli aq: public⟩

  ⟨Muli aq: types⟩

  ⟨Muli aq: interfaces⟩

  contains

  ⟨Muli aq: procedures⟩
```

```
end module muli_aq
```

The variables error_goal, err_tree and int_list must be initialised before the main loop can be called. Additionaly, the nodes and segments should be preprocessed by first_run before the main loop is called.

⟨*Muli aq: public*⟩≡

```
public :: aq_class
```

⟨*Muli aq: types*⟩≡

```
type, extends (identified_t), abstract :: aq_class
   logical :: is_deferred_initialised = .false.
   logical :: is_error_tree_initialised = .false.
   logical :: is_goal_set = .false.
   logical :: is_initialised = .false.
   logical :: is_run = .false.
   logical :: is_goal_reached = .false.
   logical :: is_integrated = .false.
   integer(dik) :: n_nodes = 0
   integer(dik) :: max_nodes = 10000
   integer :: dim_integral = 1
   real(default) :: abs_error_goal = 0._default
   real(default) :: rel_error_goal = 0.1_default
   real(default) :: scaled_error_goal = 0._default
   real(default) :: integral = 1._default
   real(default) :: integral_error = 0._default
   real(default), dimension(2) :: region = [0._default, 1._default]
   real(default), dimension(:,:), allocatable :: convergence
   real(default) :: total_time = 0
   real(default) :: loop_time = 0
   real(default) :: int_time = 0
   real(default) :: cuba_time = 0
   real(default) :: init_time = 0
   real(default) :: cpu_time = 0
   real(default) :: error_goal = 0._default
   class(fibonacci_root_t), pointer :: err_tree => null()
   class(muli_trapezium_list_t), pointer :: int_list => null()
 contains
 ⟨Muli aq: aq class: TBP⟩
end type aq_class
```

⟨*Muli aq: aq class: TBP*⟩≡

```
procedure :: basic_write_to_marker => aq_write_to_marker
procedure :: write_to_marker => aq_write_to_marker
```

⟨*Muli aq: procedures*⟩≡

```
subroutine aq_write_to_marker (this, marker, status)
  class(aq_class), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  class(ser_class_t), pointer :: ser
  call marker%mark_begin ("aq_class")
  call this%base_write_to_marker (marker, status)
  call marker%mark ("is_deferred_initialised", &
```

```
        this%is_deferred_initialised)
    call marker%mark ("is_error_tree_initialised", &
        this%is_error_tree_initialised)
    call marker%mark ("is_goal_set", this%is_goal_set)
    call marker%mark ("is_initialised", this%is_initialised)
    call marker%mark ("is_run", this%is_run)
    call marker%mark ("is_goal_reached", this%is_goal_reached)
    call marker%mark ("is_integrated", this%is_integrated)
    call marker%mark ("n_nodes", this%n_nodes)
    call marker%mark ("max_nodes", this%max_nodes)
    call marker%mark ("dim_integral", this%dim_integral)
    call marker%mark ("abs_error_goal", this%abs_error_goal)
    call marker%mark ("rel_error_goal", this%rel_error_goal)
    call marker%mark ("scaled_error_goal", this%scaled_error_goal)
    call marker%mark ("error_goal", this%error_goal)
    call marker%mark ("integral", this%integral)
    call marker%mark ("integral_error", this%integral_error)
    call marker%mark ("region", this%region(1:2))
    ser => this%err_tree
    call marker%mark_pointer ("err_tree", ser)
    ser => this%int_list
    call marker%mark_pointer ("int_list", ser)
    call marker%mark_end ("aq_class")
  end subroutine aq_write_to_marker
```

⟨*Muli aq: aq class: TBP*⟩+≡

```
  procedure :: basic_read_from_marker => aq_read_from_marker
  procedure :: read_from_marker => aq_read_from_marker
```

⟨*Muli aq: procedures*⟩+≡

```
  subroutine aq_read_from_marker (this, marker, status)
    class(aq_class), intent(out) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    class(ser_class_t), pointer :: ser
    call marker%pick_begin ("aq_class", status=status)
    call this%base_read_from_marker (marker, status)
    call marker%pick ("is_deferred_initialised", &
        this%is_deferred_initialised, status)
    call marker%pick ("is_error_tree_initialised", &
        this%is_error_tree_initialised, status)
    call marker%pick ("is_goal_set", this%is_goal_set, status)
    call marker%pick ("is_initialised", this%is_initialised, status)
    call marker%pick ("is_run", this%is_run, status)
    call marker%pick ("is_goal_reached", this%is_goal_reached, status)
    call marker%pick ("is_integrated", this%is_integrated, status)
    call marker%pick ("n_nodes", this%n_nodes, status)
    call marker%pick ("max_nodes", this%max_nodes, status)
    call marker%pick ("dim_integral", this%dim_integral, status)
    call marker%pick ("abs_error_goal", this%abs_error_goal, status)
    call marker%pick ("rel_error_goal", this%rel_error_goal, status)
    call marker%pick ("scaled_error_goal", this%scaled_error_goal, status)
    call marker%pick ("error_goal", this%error_goal, status)
    call marker%pick ("integral", this%integral, status)
```

```
    call marker%pick ("integral_error", this%integral_error, status)
    call marker%pick ("region", this%region(1:2), status)
    call marker%pick_pointer ("err_tree", ser)
    if (associated (ser)) then
       select type (ser)
       class is (fibonacci_root_t)
          this%err_tree => ser
       class default
          nullify (this%err_tree)
       end select
    end if
    call marker%pick_pointer ("int_list", ser)
    if (associated (ser)) then
       select type (ser)
       class is (muli_trapezium_list_t)
          this%int_list => ser
       class default
          nullify (this%int_list)
       end select
    end if
    call marker%pick_end ("aq_class", status)
  end subroutine aq_read_from_marker
```

⟨*Muli aq: aq class: TBP*⟩+≡

```
  procedure :: basic_print_to_unit => aq_print_to_unit
  procedure :: print_to_unit => aq_print_to_unit
```

⟨*Muli aq: procedures*⟩+≡

```
  subroutine aq_print_to_unit (this, unit, parents, components, peers)
    class(aq_class), intent(in) :: this
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    integer :: ite
    class(ser_class_t), pointer :: ser
    if (parents > 0)  call this%base_print_to_unit &
         (unit, parents-1, components, peers)
    write (unit, "(A)")  "Components of aq_class"
    write (unit, "(A,L1)")  "Deferred class initialised: ", &
         this%is_deferred_initialised
    write (unit, "(A,L1)")  "Error tree initialised:     ", &
         this%is_error_tree_initialised
    write (unit, "(A,L1)")  "Accuracy goal set:          ", this%is_goal_set
    write (unit, "(A,L1)")  "Ready for run:              ", this%is_initialised
    write (unit, "(A,L1)")  "Is run:                     ", this%is_run
    write (unit, "(A,L1)")  "Accuracy goal reached:      ", this%is_goal_reached
    write (unit, "(A,L1)")  "Integral calculated:        ", this%is_integrated
    write (unit, "(A,I10)") "Number of nodes:            ", this%n_nodes
    write (unit, "(A,I10)") "Maximal number of nodes:    ", this%max_nodes
    write (unit, "(A,I10)") "Dimension of integral:      ", this%dim_integral
    write (unit, "(A,E20.10)") "Given abs. error goal: ", this%abs_error_goal
    write (unit, "(A,E20.10)") "Given rel. error goal: ", this%rel_error_goal
    write (unit, "(A,E20.10)") "Guessed abs error goal:", this%scaled_error_goal
    write (unit, "(A,E20.10)") "Actual abs error goal: ", this%error_goal
    write (unit, "(A,E20.10)") "Integral               ", this%integral
```

```
    write (unit, "(A,E20.10)") "Estimated abs. error:  ", this%integral_error
    ! if (this%integral == 0) then
    !    write (unit, "(A,E20.10)")  "Estimated rel. error:  ", &
    !       this%integral_error / this%integral
    ! else
    !    write (unit, "(A,E20.10)")  "Estimated rel. error:  INF"
    ! end if
    write (unit, "(A,E10.5,A,E10.5,A)")  "Integration region =  (", &
         this%region(1), " : ", this%region(2), ")"
    ser => this%err_tree
    call serialize_print_comp_pointer &
         (ser, unit, parents, components, peers, "error tree")
    ser => this%int_list
    call serialize_print_comp_pointer &
         (ser, unit, parents, components, peers, "integral list")
  end subroutine aq_print_to_unit
```

⟨*Muli aq: aq class: TBP*⟩+≡

```
  procedure, nopass :: get_type => aq_get_type
```

⟨*Muli aq: procedures*⟩+≡

```
  pure subroutine aq_get_type (type)
    character(:), allocatable, intent(out) :: type
    allocate (type, source="aq_type")
  end subroutine aq_get_type
```

⟨*Muli aq: aq class: TBP*⟩+≡

```
  procedure :: deserialize_from_marker => aq_deserialize_from_marker
```

⟨*Muli aq: procedures*⟩+≡

```
  subroutine aq_deserialize_from_marker (this, name, marker)
    class(aq_class), intent(out) :: this
    character(*), intent(in) :: name
    class(marker_t), intent(inout) :: marker
    class(ser_class_t), pointer :: ser
    allocate (muli_trapezium_t :: ser)
    call marker%push_reference (ser)
    allocate (fibonacci_root_t :: ser)
    call marker%push_reference (ser)
    allocate (fibonacci_leave_t :: ser)
    call marker%push_reference (ser)
    allocate (fibonacci_node_t :: ser)
    call marker%push_reference (ser)
    call serializable_deserialize_from_marker (this, name, marker)
    call marker%pop_reference (ser)
    deallocate (ser)
    call marker%pop_reference (ser)
    deallocate (ser)
    call marker%pop_reference (ser)
    deallocate (ser)
    call marker%pop_reference (ser)
    deallocate (ser)
  end subroutine aq_deserialize_from_marker
```

The relative error goal is supposed to be 10^{-4}.

⟨*Muli aq: aq class: TBP*⟩+≡
```
generic :: initialize => aq_initialize
procedure :: aq_initialize
```

⟨*Muli aq: procedures*⟩+≡
```
subroutine aq_initialize (this, id, name, goal, max_nodes, dim, init)
  class(aq_class), intent(out) :: this
  integer(dik), intent(in) :: id, max_nodes
  integer, intent(in) :: dim
  character, intent(in) :: name
  real(default) :: goal
  real(default), dimension(:), intent(in) :: init
  call this%initialize (id, name)
  this%rel_error_goal = goal
  this%max_nodes = max_nodes
  call this%init_error_tree (dim, init)
end subroutine aq_initialize
```

⟨*Muli aq: aq class: TBP*⟩+≡
```
procedure :: print_times => aq_print_times
```

⟨*Muli aq: procedures*⟩+≡
```
subroutine aq_print_times (this)
  class(aq_class), intent(in) :: this
  write (*, "(A,E20.10)") "Initialization time:  ", this%init_time
  write (*, "(A,E20.10)") "Main loop time:       ", this%loop_time
  write (*, "(A,E20.10)") "Integration time:     ", this%int_time
  write (*, "(A,E20.10)") "Overall run time:     ", this%total_time
  write (*, "(A,E20.10)") "Cuba integration time:", this%cuba_time
end subroutine aq_print_times
```

⟨*Muli aq: aq class: TBP*⟩+≡
```
procedure :: write_convergence => aq_write_convergence
```

⟨*Muli aq: procedures*⟩+≡
```
subroutine aq_write_convergence (this, unit)
  class(aq_class), intent(in) :: this
  integer, intent(in) :: unit
  integer, dimension(2) :: s
  integer :: node
  if (allocated (this%convergence)) then
     s = shape (this%convergence)
     do node = 1, s(2)
        write (unit, *) node, this%convergence (1:2, node)
     end do
  end if
end subroutine aq_write_convergence
```

⟨*Muli aq: aq class: TBP*⟩+≡
```
procedure :: reset => aq_reset
```

⟨*Muli aq: procedures*⟩+≡

```
subroutine aq_reset (this)
  class(aq_class) :: this
  this%is_deferred_initialised = .false.
  this%is_error_tree_initialised = .false.
  this%is_goal_set = .false.
  this%is_initialised = .false.
  this%is_run = .false.
  this%is_goal_reached = .false.
  this%is_integrated = .false.
  this%n_nodes = 0
  this%max_nodes = 10000
  this%dim_integral = 1
  this%abs_error_goal = 1._default
  this%rel_error_goal = 0.1_default
  this%scaled_error_goal = 0.0_default
  this%error_goal = 0.0_default
  this%integral = 0.0_default
  this%integral_error = 0.0_default
  this%region = [ 0.0_default, 1._default ]
  this%total_time = 0
  this%loop_time = 0
  this%int_time = 0
  this%init_time = 0
  call this%dealloc_trees ()
end subroutine aq_reset
```

⟨*Muli aq: aq class: TBP*⟩+≡

```
procedure :: dealloc_trees => aq_dealloc_trees
procedure :: finalize => aq_dealloc_trees
```

⟨*Muli aq: procedures*⟩+≡

```
subroutine aq_dealloc_trees (this)
  class(aq_class) :: this
  if (associated (this%err_tree)) then
     call this%err_tree%deallocate_all ()
     deallocate (this%err_tree)
  end if
  if (associated (this%int_list)) then
     call this%int_list%finalize ()
     deallocate (this%int_list)
  end if
end subroutine aq_dealloc_trees
```

⟨*Muli aq: aq class: TBP*⟩+≡

```
procedure :: init_error_tree => aq_init_error_tree
```

⟨*Muli aq: procedures*⟩+≡

```
subroutine aq_init_error_tree (this, dim_integral, x_array)
  class(aq_class) :: this
  integer, intent(in) :: dim_integral
  real(default), dimension(:), intent(in) :: x_array
  real(default) :: center
  real(default), dimension(:), allocatable :: l_val, c_val, r_val
```

```
class(muli_trapezium_t), pointer :: left_node => null()
class(muli_trapezium_t), pointer :: right_node => null()
integer :: x_size, pos
! print '("Entermarker aq_init_error_tree...")'
call cpu_time (this%init_time)
this%is_initialised = .false.
this%integral = 0._default
this%dim_integral = dim_integral
x_size = size(x_array)
if (x_size < 2) then
     call msg_error ("aq_init_error_tree: I need at least two real values")
else
   allocate (l_val (0:dim_integral-1))
   allocate (c_val (0:dim_integral-1))
   allocate (r_val (0:dim_integral-1))
   this%region = [x_array(1), x_array(x_size)]
   if (x_size < 3) then
      center = (x_array(2) - x_array(1)) / 2._default
      call this%evaluate (x_array(1), l_val)
      call this%evaluate (center, c_val)
      call this%evaluate (x_array(2), r_val)
      allocate (left_node)
      call left_node%initialize (dim=dim_integral, &
           r_position=center, d_position=center-x_array(1))
      call left_node%set_r_value (c_val)
      call left_node%set_d_value (c_val - l_val)
      allocate (right_node)
      call right_node%initialize (dim=dim_integral, &
           r_position=x_array(2), d_position=x_array(2)-center)
      call right_node%set_r_value (r_val)
      call right_node%set_d_value (r_val - c_val)
   else
      call this%evaluate (x_array(1), l_val)
      call this%evaluate (x_array(2), c_val)
      call this%evaluate (x_array(3), r_val)
      allocate (left_node)
      call left_node%initialize (dim=dim_integral, &
           r_position=x_array(2), d_position=x_array(2)-x_array(1))
      call left_node%set_r_value (c_val)
      call left_node%set_d_value (c_val - l_val)
      allocate (right_node)
      call right_node%initialize (dim=dim_integral, &
           r_position=x_array(3), d_position=x_array(3)-x_array(2))
      call right_node%set_r_value (r_val)
      call right_node%set_d_value (r_val - c_val)
   end if
   call left_node%update ()
   call right_node%update ()
   this%integral = sum (left_node%get_d_integral () + &
        right_node%get_d_integral ())
   if (.not. associated (this%err_tree)) then
      allocate(this%err_tree)
   end if
   print *, left_node%measure ()
```

```
      print *, right_node%measure ()
      call this%err_tree%init_by_content (left_node, right_node)
      ! call this%err_tree%write_pstricks (11)
      if (x_size > 3) then
         do pos = 4, x_size
            print *, "aq_init_error_tree", pos, "/", x_size
            l_val = right_node%get_r_value_array ()
            call this%evaluate (x_array(pos), r_val)
            c_val = r_val - l_val
            allocate (right_node)
            call right_node%initialize (dim=dim_integral, &
                 r_position=x_array(pos), &
                 d_position=x_array(pos)-x_array(pos-1))
            call right_node%set_r_value (r_val)
            call right_node%set_d_value (c_val)
            call right_node%update ()
            call this%err_tree%push_by_content (right_node)
            ! call this%err_tree%write_pstricks (11)
            this%integral = this%integral + sum (right_node%get_d_integral())
         end do
         this%n_nodes = x_size
      end if
      this%is_error_tree_initialised = .true.
   end if
   call this%set_goal ()
   this%is_initialised = .true.
   call cpu_time (this%cpu_time)
   this%init_time = this%cpu_time - this%init_time
   this%cuba_time = this%init_time
   allocate (this%convergence (2, this%n_nodes:this%max_nodes))
end subroutine aq_init_error_tree
```

⟨*Muli aq: aq class: TBP*⟩+≡

```
procedure :: set_rel_goal => aq_set_rel_goal
```

⟨*Muli aq: procedures*⟩+≡

```
subroutine aq_set_rel_goal (this, goal)
   class(aq_class) :: this
   real(default) :: goal
   this%rel_error_goal = goal
   call this%set_goal
end subroutine aq_set_rel_goal
```

⟨*Muli aq: aq class: TBP*⟩+≡

```
procedure :: set_abs_goal => aq_set_abs_goal
```

⟨*Muli aq: procedures*⟩+≡

```
subroutine aq_set_abs_goal (this, goal)
   class(aq_class) :: this
   real(default) :: goal
   this%abs_error_goal = goal
   call this%set_goal
end subroutine aq_set_abs_goal
```

⟨*Muli aq: aq class: TBP*⟩+≡
```
procedure :: set_goal => aq_set_goal
```

⟨*Muli aq: procedures*⟩+≡
```
subroutine aq_set_goal (this)
  class(aq_class) :: this
  this%scaled_error_goal = this%rel_error_goal * abs(this%integral)
  if ((this%scaled_error_goal == zero) .and. &
      (this%abs_error_goal == zero)) then
     this%is_goal_set = .false.
     this%error_goal = zero
  else
     if (this%scaled_error_goal == zero) then
        this%error_goal = this%abs_error_goal
     else
        if (this%abs_error_goal == zero) then
           this%error_goal = this%scaled_error_goal
        else
           this%error_goal = max(this%scaled_error_goal, &
                this%abs_error_goal)
        end if
     end if
     if (this%error_goal > zero) then
        this%is_goal_set = .true.
     else
        this%is_goal_set = .false.
     end if
  end if
end subroutine aq_set_goal
```

⟨*Muli aq: aq class: TBP*⟩+≡
```
procedure :: check_init => aq_check_init
```

⟨*Muli aq: procedures*⟩+≡
```
subroutine aq_check_init (this)
  class(aq_class) :: this
  this%is_initialised = this%is_error_tree_initialised .and. &
       this%is_deferred_initialised
end subroutine aq_check_init
```

This routine is unsafe, when `n_nodes` < 4.

⟨*Muli aq: aq class: TBP*⟩+≡
```
procedure :: main_loop => aq_main_loop
```

⟨*Muli aq: procedures*⟩+≡
```
subroutine aq_main_loop (this)
  class(aq_class) :: this
  class(fibonacci_leave_t), pointer :: rightmost
  class(measure_class_t), pointer :: content
  class(muli_trapezium_t), pointer :: new_node !,debug
  logical :: limit = .false.
  real(default) :: center
  real(default), dimension(:), allocatable :: c_val
  allocate (c_val (0:this%dim_integral-1))
```

```
    LOOP: do
       call this%err_tree%pop_right (rightmost)
       if (rightmost < this%error_goal / this%n_nodes) then
          this%is_goal_reached = .true.
          exit LOOP
       else
          call rightmost%get_content (content)
          select type (content)
          class is (muli_trapezium_t)
             write (*, "(A,I5,A,E14.7,A,E14.7,A,E14.7,A,E14.7)")  &
                  "nodes: ", this%n_nodes, " error: ", &
                  rightmost%measure() * this%n_nodes, &
                  " goal: ", this%error_goal, " node at: ", &
                  content%get_l_position(), "-", content%get_r_position()
             this%convergence (1, this%n_nodes) = this%error_goal / this%n_nodes
             this%convergence (2, this%n_nodes) = rightmost%measure ()
             center = content%get_r_position () - &
                  content%get_d_position () / two
             call cpu_time (this%cpu_time)
             this%cuba_time = this%cuba_time - this%cpu_time
             call this%evaluate (center, c_val)
             call cpu_time (this%cpu_time)
             this%cuba_time = this%cuba_time + this%cpu_time
             call content%split (c_val, center, new_node)
             call this%err_tree%push_by_leave (rightmost)
             call this%err_tree%push_by_content (new_node)
          end select
          this%n_nodes = this%n_nodes + 1
          if (this%n_nodes > this%max_nodes) then
             limit = .true.
             exit LOOP
          end if
       end if
    end do LOOP
    call this%err_tree%push_by_leave (rightmost)
  end subroutine aq_main_loop
```

⟨*Muli aq: aq class: TBP*⟩+≡

```
  procedure :: run => aq_run
```

⟨*Muli aq: procedures*⟩+≡

```
  subroutine aq_run (this)
    class(aq_class) :: this
    call cpu_time (this%total_time)
    if (.not. this%is_error_tree_initialised) then
       call this%init_error_tree (this%dim_integral, this%region)
    end if
    this%is_run = .false.
    this%is_goal_reached = .false.
    call this%main_loop ()
    this%is_run = .true.
    call cpu_time (this%cpu_time)
    this%total_time = this%cpu_time - this%total_time
  end subroutine aq_run
```

⟨*Muli aq: aq class: TBP*⟩+≡

```
procedure :: integrate => aq_integrate
```

⟨*Muli aq: procedures*⟩+≡

```
subroutine aq_integrate (this, int_tree)
  class(aq_class) :: this
  class(muli_trapezium_node_class_t), pointer :: node
  type(muli_trapezium_tree_t), intent(out) :: int_tree
  real(default) :: sum
  this%is_integrated = .false.
  this%integral_error = zero
  if (this%is_run) then
  call cpu_time (this%int_time)
     call fibonacci_tree_resort_and_convert_to_trapezium_list &
          (this%err_tree, this%int_list)
     ! call this%int_list%print_all ()
     call this%int_list%integrate (this%integral, this%integral_error)
     call this%int_list%to_tree (int_tree)
     this%is_integrated = .true.
     call cpu_time (this%cpu_time)
     this%int_time = this%cpu_time - this%int_time
  end if
end subroutine aq_integrate
```

⟨*Muli aq: aq class: TBP*⟩+≡

```
procedure(evaluate_if), deferred :: evaluate
! procedure(evaluate_ratios_if), deferred :: evaluate_ratios
```

⟨*Muli aq: procedures*⟩+≡

Usually, the tree is sorted by the sum of errors. Now it shall be sorted by the right position. When at least one branch of the tree is itself a tree, i.e. each branch has got at least two leaves, then process each branch and merge the results. Now we got two sorted lists. Which one's leftmost node has got the lowest value of `r_position`? That one shall be the beginning of the merged list `lin_list`. Everything is prepared for the algorithm: `lin_list` is the beginning of the sorted list, `last_node` is its end. `left_node` and `right_node` are the leftmost nodes of the remainders of `left_list` and `right_list`. The latter will get stripped from left to right, until one of them ends. Then, either `left_list` or `right_list` is completely merged into `lin_list`. The other one gets appended to `lin_list`. In the second part of the big if clause, the tree has got two leaves at most. Is it more than one? There, `fib_tree` is a single leave with an allocated "`content`" component of type `muli_trapezium_t`. If "`content`" is not type-compatible with `muli_trapezium_t`, then this whole conversion cannot succeed. We allocate a new node of type `muli_trapezium_list_t`. This list does not contain the content of `fib_tree`, it *is* a copy of the content, for `muli_trapezium_list_t` is an extension of `muli_trapezium_t`. In the next step, each branch of `fib_tree` is a single leave. We could call this soubroutine for each branch, but we do copy and paste for each branch instead. Finally we append one list to the other, the lowest value of `r_position` comes first.

⟨*Muli aq: procedures*⟩+≡

```
recursive subroutine fibonacci_tree_resort_and_convert_to_trapezium_list &
     (fib_tree, lin_list)
  class(fibonacci_node_t), intent(in) :: fib_tree
  class(fibonacci_node_t),pointer :: leave
  class(muli_trapezium_list_t), pointer, intent(out) :: lin_list
  class(muli_trapezium_list_t), pointer :: left_list, right_list
  class(muli_trapezium_node_class_t), pointer :: &
       left_node, right_node, last_node
  class(measure_class_t), pointer :: content
  if (fib_tree%depth > 1) then
     ! print *,"3A"
     call fibonacci_tree_resort_and_convert_to_trapezium_list &
          (fib_tree%left, left_list)
     call fibonacci_tree_resort_and_convert_to_trapezium_list &
          (fib_tree%right, right_list)
     if (left_list%is_left_of (right_list)) then
        lin_list => left_list
        call left_list%get_right (left_node)
        right_node => right_list
     else
        lin_list => right_list
        left_node => left_list
        call right_list%get_right (right_node)
     end if
     last_node => lin_list
     do while (associated (left_node) .and. associated (right_node))
        if (left_node%is_left_of (right_node)) then
           call last_node%append (left_node)
           call last_node%get_right (last_node)
           call left_node%get_right (left_node)
        else
           call last_node%append (right_node)
           call last_node%get_right (last_node)
           call right_node%get_right (right_node)
        end if
     end do
     if (associated (left_node)) then
        call last_node%append (left_node)
     else
        call last_node%append (right_node)
     end if
     !!! It's done.
     ! print *,"3E"
  else
     if (fib_tree%depth == 0) then
        ! print *,"1A"
        select type (fib_tree)
        class is (fibonacci_leave_t)
           call fib_tree%get_content (content)
           select type (content)
           class is (muli_trapezium_t)
              call content%to_node (content%get_r_position(), list=lin_list)
           class default
              call msg_fatal &
```

```
                  ("fibonacci_tree_resort_and_convert_to_trapezium_list: " // &
                  "Content of fibonacci_tree is not type compatible " // &
                  "to muli_trapezium_t")
            end select
         end select
         ! print *,"1E"
      else
         ! print *,"2A"
         leave => fib_tree%left
         select type (leave)
         class is (fibonacci_leave_t)
            call leave%get_content (content)
            select type (content)
            class is (muli_trapezium_t)
               call content%to_node (content%get_r_position(), list=left_list)
            class default
               call msg_fatal &
                  ("fibonacci_tree_resort_and_convert_to_trapezium_list: " // &
                  "Content of fibonacci_tree is not type compatible " // &
                  "to muli_trapezium_t")
            end select
         end select
         leave => fib_tree%right
         select type (leave)
         class is (fibonacci_leave_t)
            call leave%get_content (content)
            select type (content)
            class is (muli_trapezium_t)
               call content%to_node (content%get_r_position(), list=right_list)
            class default
               call msg_fatal &
                  ("fibonacci_tree_resort_and_convert_to_trapezium_list: " // &
                  "Content of fibonacci_tree is not type compatible " // &
                  "to muli_trapezium_t")
            end select
         end select
         if (left_list%is_left_of (right_list)) then
            call left_list%append (right_list)
            lin_list => left_list
         else
            call right_list%append (left_list)
            lin_list => right_list
         end if
         ! print *,"2E"
      end if
   end if
   ! call lin_list%print_all ()
   ! call lin_list%check ()
end subroutine fibonacci_tree_resort_and_convert_to_trapezium_list
```

⟨*Muli aq: interfaces*⟩≡

```
interface
   subroutine evaluate_if (this, x, y)
     use kinds !NODEP!
```

```
      import aq_class
      class(aq_class), intent(inout) :: this
      real(default), intent(in) :: x
      real(default), intent(out) , dimension(:) :: y
    end subroutine evaluate_if

    ! subroutine evaluate_ratios_if (this, cont)
    !   use kinds
    !   use lin_approx_tree_module, only: muli_trapezium_t
    !   import aq_class
    !   class(aq_class) :: this
    !   class(muli_trapezium_t), intent(inout), pointer :: cont
    ! end subroutine evaluate_ratios_if
  end interface
```

⟨muli_parameters.f90⟩≡

```
! This is a dummy for muli_parameters_module
module muli_parameters_module
end module muli_parameters_module
```

22.8 Integrands for Multiple Interactions

This file contains the module `muli_dsigma`. Its only type `muli_dsigma_t` provides an integrand to `aq_class`. The actual integrand is the normalized differential cross section of a QCD $2 \to 2$ process $1/\sigma_0 \times d^3\sigma/(dp_T^2 dx_1 dx_2)$. We need a root function of this integrand in terms of p_T, so we have to integrate out x_1 and x_2 and have to approximate the root function of the leftover variable p_T. Integration of x_1 and x_2 is done by CUBA, the root function is approximated by `muli_aq`.

⟨muli_dsigma.f90⟩≡

```
⟨File header⟩

module muli_dsigma
⟨Use kinds⟩
  use constants
  use muli_base
  use muli_momentum
  use muli_interactions
  use muli_cuba
  use muli_trapezium
  use muli_aq

⟨Standard module head⟩

⟨Muli dsigma: variables⟩

⟨Muli dsigma: public⟩

⟨Muli dsigma: types⟩
```

```
contains

⟨Muli dsigma: procedures⟩

end module muli_dsigma
```

⟨*Muli dsigma: variables*⟩≡
```
integer, parameter :: dim_f = 17
```

⟨*Muli dsigma: public*⟩≡
```
public :: muli_dsigma_t
```

⟨*Muli dsigma: types*⟩≡
```
type, extends (aq_class) :: muli_dsigma_t
   private
   type(transverse_mom_t) :: pt
   type(cuba_divonne_t) :: cuba_int
 contains
 ⟨Muli dsigma: dsigma: TBP⟩
end type muli_dsigma_t
```

⟨*Muli dsigma: dsigma: TBP*⟩≡
```
procedure :: write_to_marker => muli_dsigma_write_to_marker
```

⟨*Muli dsigma: procedures*⟩≡
```
subroutine muli_dsigma_write_to_marker (this, marker, status)
  class(muli_dsigma_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  class(ser_class_t), pointer :: ser
  call marker%mark_begin ("muli_dsigma_t")
  call this%basic_write_to_marker (marker, status)
  call this%cuba_int%serialize (marker, "cuba_int")
  call marker%mark_end ("muli_dsigma_t")
end subroutine muli_dsigma_write_to_marker
```

⟨*Muli dsigma: dsigma: TBP*⟩+≡
```
procedure :: read_from_marker => muli_dsigma_read_from_marker
```

⟨*Muli dsigma: procedures*⟩+≡
```
subroutine muli_dsigma_read_from_marker (this, marker, status)
  class(muli_dsigma_t), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%pick_begin ("muli_dsigma_t", status=status)
  call this%basic_read_from_marker (marker, status)
  call this%cuba_int%deserialize ("cuba_int", marker)
  call marker%pick_end ("muli_dsigma_t", status)
end subroutine muli_dsigma_read_from_marker
```

⟨*Muli dsigma: dsigma: TBP*⟩+≡
```
procedure :: print_to_unit => muli_dsigma_print_to_unit
```

⟨*Muli dsigma: procedures*⟩+≡

```
subroutine muli_dsigma_print_to_unit &
     (this, unit, parents, components, peers)
  class(muli_dsigma_t), intent(in) :: this
  integer, intent(in) :: unit
  integer(dik), intent(in) :: parents, components, peers
  integer :: ite
  if (parents > 0)  call this%basic_print_to_unit &
       (unit, parents-1, components, peers)
  write (unit, "(A)")  "Components of muli_dsigma_t"
  if (components > 0) then
     write (unit, "(A)")  "Printing components of cuba_int:"
     call this%cuba_int%print_to_unit (unit, parents, components-1, peers)
  else
     write (unit, "(A)")  "Skipping components of cuba_int:"
  end if
end subroutine muli_dsigma_print_to_unit
```

⟨*Muli dsigma: dsigma: TBP*⟩+≡

```
procedure, nopass :: get_type => muli_dsigma_get_type
```

⟨*Muli dsigma: procedures*⟩+≡

```
pure subroutine muli_dsigma_get_type (type)
  character(:), allocatable, intent(out) :: type
  allocate (type, source="muli_dsigma_t")
end subroutine muli_dsigma_get_type
```

⟨*Muli dsigma: dsigma: TBP*⟩+≡

```
procedure :: generate => muli_dsigma_generate
```

⟨*Muli dsigma: procedures*⟩+≡

```
subroutine muli_dsigma_generate (this, gev2_scale_cutoff, gev2_s, int_tree)
  class(muli_dsigma_t), intent(inout) :: this
  real(default), intent(in) :: gev2_scale_cutoff, gev2_s
  type(muli_trapezium_tree_t), intent(out) :: int_tree
  real(default), dimension(ceiling (log (gev2_s/gev2_scale_cutoff)/two)) :: &
       initial_values
  integer :: n
  print *, gev2_s/gev2_scale_cutoff, &
       ceiling (log (gev2_s/gev2_scale_cutoff)/two)
  ! allocate (initial_values (ceiling (-log (gev2_scale_cutoff))/2))
  ! allocate (real(default), &
  !    dimension (ceiling (log(gev2_scale_cutoff))/2) :: initial_values)
  initial_values(1) = sqrt(gev2_scale_cutoff/gev2_s) * two
  do n = 2, size(initial_values) - 1
     initial_values(n) = initial_values(n-1) * euler
  end do
  initial_values(n) = one
  print *, initial_values
  ! stop
  call this%initialize (i_one, "dsigma")
  call this%pt%initialize (gev2_s)
  this%abs_error_goal = zero
  this%rel_error_goal = scale(one, -12) !-12
```

```
    this%max_nodes = 1000
    call this%cuba_int%set_common (dim_f=dim_f, dim_x=2, &
         eps_rel=scale(this%rel_error_goal,-8), flags = 0)
    call this%cuba_int%set_deferred (xgiven_flat = [1.E-2_default, &
         5.E-1_default + epsilon(1._default), 1.E-2_default, &
         5.E-1_default - epsilon(1._default)])
    print *, "muli_dsigma_generate:"
    ! print *, "Cuba Error Goal:    ", this%cuba_int%eps_rel
    print *, "Overall Error Goal: ", this%rel_error_goal
    call this%init_error_tree (dim_f, initial_values)
    call this%run ()
    call this%integrate (int_tree)
    call this%err_tree%deallocate_all ()
    deallocate (this%err_tree)
    nullify (this%int_list)
  end subroutine muli_dsigma_generate
```

⟨*Muli dsigma: dsigma: TBP*⟩+≡

```
  procedure :: evaluate => muli_dsigma_evaluate
```

⟨*Muli dsigma: procedures*⟩+≡

```
  subroutine muli_dsigma_evaluate (this, x, y)
    class(muli_dsigma_t), intent(inout) :: this
    real(default), intent(in) :: x
    real(default), intent(out), dimension(:):: y
    call this%pt%set_unit_scale (x)
    ! print *, "muli_dsigma_evaluate x=", x
    ! call this%cuba_int%integrate_userdata &
    !       (interactions_proton_proton_integrand_param_17_reg, this%pt)
    ! if (this%cuba_int%fail == 0) then
    ! call this%cuba_int%print_all ()
    call this%cuba_int%get_integral_array (y)
    ! else
    !    print *, "muli_dsigma_evaluate: failed."
    !    stop
    ! end if
  end subroutine muli_dsigma_evaluate
```

⟨*Muli dsigma: dsigma: TBP*⟩+≡

```
  generic :: initialize => muli_dsigma_initialize
  procedure :: muli_dsigma_initialize
```

⟨*Muli dsigma: procedures*⟩+≡

```
  subroutine muli_dsigma_initialize &
       (this, id, name, goal, max_nodes, dim, cuba_goal)
    class(muli_dsigma_t), intent(inout) :: this
    integer(dik), intent(in) :: id, max_nodes
    integer, intent(in) :: dim
    character(*), intent(in) :: name
    real(default), intent(in) :: goal, cuba_goal
    call this%initialize (id,name)
    ! 1E-4
    this%rel_error_goal = goal
    this%max_nodes = max_nodes
```

```
    call this%cuba_int%set_common (dim_f=dim, dim_x=2, &
         ! 1E-6
         eps_rel=cuba_goal, flags = 0)
    call this%cuba_int%set_deferred (xgiven_flat = [1.E-2_default, &
         5.E-1_default + epsilon(1._default), &
         1.E-2_default, 5.E-1_default - epsilon(1._default)])
    ! call aq_initialize (this, id, name, d_goal, max_nodes, dim_f, &
    !        [8E-1_default/7E3_default, 2E-3_default, 1E-2_default, &
    !         1E-1_default, one])
    call this%init_error_tree (dim, [8.E-1_default/7.E3_default, &
         2.E-3_default, 1.E-2_default, 1.E-1_default, &
         1._default])
    this%is_deferred_initialised = .true.
  end subroutine muli_dsigma_initialize
```

⟨*Muli dsigma: dsigma: TBP*⟩+≡

```
  ! procedure :: reset => muli_dsigma_reset
```

⟨*Muli dsigma: procedures*⟩+≡

```
  ! subroutine muli_dsigma_reset (this)
  !   class(muli_dsigma_t), intent(inout) :: this
  !   call aq_reset (this)
  !   call this%initialize &
  !      (id, name, d_goal, max_nodes, dim_f, init, cuba_goal)
  ! end subroutine muli_dsigma_reset
```

22.9 MC Integrations for QCD $2 \to 2$ processes

This file contains the module `muli_mcint` which is the Monte Carlo generator for QCD $2 \to 2$ interactions at given evolution parameter and given stratus. While `muli_t` takes care of generating the evolution parameter and the stratus, this module is about bookkeeping the strati and implementing a downstream importance sampling. The evolution parameter is a measure of transferred momentum and a "stratus" is a PDF category or, to be more precise, is whether the incoming partons are gluons or sea quarks or valence quarks.

The importance sampling then subdivides the phase space of variables $\{x_1, x_2, p_T\}$ into n^3 regions such that each region holds approximately n interactions. Thus, we can generate a phase space point very quickly just by randomly picking a region, randomly picking a point within this region and comparing its exact cross section with the mean cross section for this actual evolution parameter and the actual stratus times the area of the picked phase space region.

The mean values must be generated in the module `muli_dsigma` before and are given to the procedure `sample_inclusive_generate_hit`. Finally the generated subregions should be written to a file via `write_to_marker` and then reused for each later `WHIZARD` run.

The type `sample_inclusive_t` holds the 16 strati, while the type `sample_int_kind_t` represents a single stratus, `sample_3d_t` is the whole $\{x_1, x_2, p_T\}$ phase space for each stratus, `sample_2d_t` is the $\{x_1, x_2\}$ plane with a slice of p_T and `sample_region_t` finally is a phase space region.

⟨`muli_mcint.f90`⟩≡

⟨*File header*⟩

```
module muli_mcint
⟨Use kinds⟩
  use constants
  use muli_base
  use tao_random_numbers !NODEP!
  use muli_interactions

⟨Standard module head⟩

⟨Muli MC int: variables⟩

⟨Muli MC int: public⟩

⟨Muli MC int: types⟩

contains

⟨Muli MC int: procedures⟩

end module muli_mcint
```

⟨*Muli MC int: variables*⟩≡

```
integer, parameter :: max_n = 2**30
```

⟨*Muli MC int: variables*⟩+≡

```
real(default), parameter :: max_d = one * max_n
```

⟨*Muli MC int: variables*⟩+≡

```
real(default), parameter, dimension(2,2) :: &
     unit_square = reshape([zero,zero,one,one], [2,2])
```

⟨*Muli MC int: public*⟩≡

```
public :: sample_region_t
```

⟨*Muli MC int: types*⟩≡

```
type, extends (ser_class_t) :: sample_region_t
   integer :: n_hits = 0
   integer :: n_alloc = 0
   real(default), dimension(2,2) :: corners = unit_square
   real(default), dimension(:,:), allocatable :: hyp_hits
 contains
 ⟨Muli MC int: sample region: TBP⟩
end type sample_region_t
```

⟨*Muli MC int: sample region: TBP*⟩≡

```
procedure :: write_to_marker => sample_region_write_to_marker
```

⟨*Muli MC int: procedures*⟩≡

```
subroutine sample_region_write_to_marker (this, marker, status)
  class(sample_region_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
```

```
    integer :: n
    call marker%mark_begin ("sample_region_t")
    call marker%mark ("n_hits", this%n_hits)
    call marker%mark ("n_alloc", this%n_alloc)
    call marker%mark ("lower_corner", this%corners(1:2,1))
    call marker%mark ("upper_corner", this%corners(1:2,2))
    if (allocated (this%hyp_hits)) then
       call marker%mark ("hyp_hits", this%hyp_hits (1:3,:this%n_hits))
    else
       call marker%mark_nothing ("hyp_hits")
    end if
    call marker%mark_end ("sample_region_t")
  end subroutine sample_region_write_to_marker
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
  procedure :: read_from_marker => sample_region_read_from_marker
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_region_read_from_marker (this, marker, status)
    class(sample_region_t), intent(out) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    integer :: n
    call marker%pick_begin ("sample_region_t", status=status)
    call marker%pick ("n_hits", this%n_hits, status)
    call marker%pick ("n_alloc", this%n_alloc, status)
    call marker%pick ("lower_corner", this%corners(1:2,1), status)
    call marker%pick ("upper_corner", this%corners(1:2,2), status)
    if (allocated (this%hyp_hits))  deallocate (this%hyp_hits)
    call marker%verify_nothing ("hyp_hits", status)
    if (.not. status == serialize_nothing) then
       allocate (this%hyp_hits (3,this%n_alloc))
       call marker%pick ("hyp_hits", this%hyp_hits (1:3,:this%n_hits), status)
    end if
    call marker%pick_end ("sample_region_t", status)
  end subroutine sample_region_read_from_marker
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
  procedure :: print_to_unit => sample_region_print_to_unit
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_region_print_to_unit &
       (this, unit, parents, components, peers)
    class(sample_region_t), intent(in) :: this
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    write (unit, "(1x,A)")      "components of sample_region_t"
    write (unit, "(3x,A,I10)")  "n_hits:           ", this%n_hits
    write (unit, "(3x,A,I10)")  "n_alloc:          ", this%n_alloc
    write (unit, "(3x,4(E20.10))") "corners:          ", this%corners
    if (allocated (this%hyp_hits) .and. this%n_hits > 0) then
       if (components > 0) then
          write (unit,"(3x,A)")  "hits:"
          print *, shape (this%hyp_hits)
```

```
         write (unit, "(3(e20.10))")  this%hyp_hits (1:3, this%n_hits)
      else
         write (unit, "(3x,A)")  "skipping hits."
      end if
   else
      write (unit, "(3x,A)")  "hits are not allocated."
   end if
end subroutine sample_region_print_to_unit
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
procedure, nopass :: get_type => sample_region_get_type
```

⟨*Muli MC int: procedures*⟩+≡

```
pure subroutine sample_region_get_type (type)
  character(:), allocatable, intent(out) :: type
  allocate (type, source="sample_region_t")
end subroutine sample_region_get_type
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
procedure :: initialize => sample_region_initialize
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_region_initialize (this, n_alloc)
  class(sample_region_t), intent(out) :: this
  integer, intent(in) :: n_alloc
  if (allocated (this%hyp_hits))  deallocate (this%hyp_hits)
  allocate (this%hyp_hits (3,n_alloc))
  this%n_alloc = n_alloc
end subroutine sample_region_initialize
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
procedure :: generate_hit => sample_region_generate_hit
```

⟨*Muli MC int: procedures*⟩+≡

```
pure subroutine sample_region_generate_hit (this, rnd, area, hit)
  class(sample_region_t), intent(in) :: this
  integer, intent(in), dimension(2) :: rnd
  real(default), dimension(2), intent(out) :: hit
  real(default), intent(out) :: area
  call muli_mcint_generate_hit (rnd, this%corners, hit)
  area = this%area ()
end subroutine sample_region_generate_hit
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
procedure :: confirm_hit => sample_region_confirm_hit
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_region_confirm_hit (this, hit)
  class(sample_region_t), intent(inout) :: this
  real(default), dimension(3), intent(in) :: hit
  ! print *,"sample_region_confirm_hit: ", this%n_hits, this%n_alloc, hit
  this%n_hits = this%n_hits + 1
  if (this%n_hits <= this%n_alloc) then
     this%hyp_hits (1:3, this%n_hits) = hit
```

```
      else
         print *, "sample_region_confirm_hit: Region is already full."
      end if
   end subroutine sample_region_confirm_hit
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
   procedure :: split => sample_region_split
```

⟨*Muli MC int: procedures*⟩+≡

```
   subroutine sample_region_split (this, pos, dimX, n_alloc, lower, upper)
      class(sample_region_t), intent(in) :: this
      type(sample_region_t), intent(out) :: lower, upper
      real(default), dimension(3) :: hit
      real(default), intent(in) :: pos
      integer, intent(in) :: dimX, n_alloc
      integer :: n_hit
      call lower%initialize (n_alloc)
      call upper%initialize (n_alloc)
      do n_hit = 1, this%n_hits
         hit = this%hyp_hits (1:3, n_hit)
         if (hit(dimX) < pos) then
            call lower%confirm_hit (hit)
         else
            call upper%confirm_hit (hit)
         end if
      end do
      lower%corners = this%corners
      upper%corners = this%corners
      if (dimX < 3) then
         lower%corners(dimX,2) = pos
         upper%corners(dimX,1) = pos
      end if
   end subroutine sample_region_split
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
   procedure :: write_hits => sample_region_write_hits
```

⟨*Muli MC int: procedures*⟩+≡

```
   subroutine sample_region_write_hits (this, unit)
      class(sample_region_t), intent(in) :: this
      integer, intent(in) :: unit
      integer :: n
      do n = 1, this%n_hits
         write (unit, *)  this%hyp_hits (1:3,n)
      end do
   end subroutine sample_region_write_hits
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
   procedure :: is_full => sample_region_is_full
```

⟨*Muli MC int: procedures*⟩+≡

```
   elemental logical function sample_region_is_full (this)
      class(sample_region_t), intent(in) :: this
      sample_region_is_full = this%n_alloc == this%n_hits
```

```
end function sample_region_is_full
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
procedure :: move_components => sample_region_move_components
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_region_move_components (this, that)
  class(sample_region_t), intent(inout) :: this
  class(sample_region_t), intent(out) :: that
  that%n_alloc = this%n_alloc
  that%n_hits = this%n_hits
  that%corners = this%corners
  call move_alloc (this%hyp_hits, that%hyp_hits)
  this%n_alloc = 0
  this%n_hits = 0
end subroutine sample_region_move_components
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
procedure :: mean => sample_region_mean
```

⟨*Muli MC int: procedures*⟩+≡

```
elemental function sample_region_mean (this, dim)
  real(default) :: sample_region_mean
  class(sample_region_t), intent(in) :: this
  integer, intent(in) :: dim
  sample_region_mean = sum (this%hyp_hits (dim,1:this%n_hits)) / this%n_hits
end function sample_region_mean
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
procedure :: area => sample_region_area
```

⟨*Muli MC int: procedures*⟩+≡

```
elemental function sample_region_area (this)
  real(default) :: sample_region_area
  class(sample_region_t), intent(in) :: this
  sample_region_area = product (this%corners(1:2,2) - this%corners(1:2,1))
end function sample_region_area
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
procedure :: density => sample_region_density
```

⟨*Muli MC int: procedures*⟩+≡

```
elemental function sample_region_density (this)
  real(default) :: sample_region_density
  class(sample_region_t), intent(in) :: this
  sample_region_density = this%n_hits / this%area ()
end function sample_region_density
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
procedure :: contains => sample_region_contains
```

⟨*Muli MC int: procedures*⟩+≡

```
pure logical function sample_region_contains (this, hit)
  class(sample_region_t), intent(in) :: this
  real(default), intent(in), dimension(3) :: hit
  sample_region_contains = (this%corners(1,1) <= hit(1) .and. &
       hit(1) <= this%corners(1,2) .and. &
       this%corners(2,1) <= hit(2) .and. &
       hit(2) <= this%corners(2,2))
end function sample_region_contains
```

⟨*Muli MC int: sample region: TBP*⟩+≡

```
procedure :: to_generator => sample_region_to_generator
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_region_to_generator (this)
  class(sample_region_t), intent(inout) :: this
  if (allocated (this%hyp_hits))  deallocate (this%hyp_hits)
  this%n_alloc = 0
end subroutine sample_region_to_generator
```

⟨*Muli MC int: public*⟩+≡

```
public :: sample_2d_t
```

⟨*Muli MC int: types*⟩+≡

```
type, extends (ser_class_t) :: sample_2d_t
   integer :: n_regions = 0
   integer :: n_alloc = 0
   integer :: n_hits = 0
   real(default), dimension(2) :: range = [0,1]
   type(sample_region_t), dimension(:), allocatable :: regions
 contains
 ⟨Muli MC int: sample 2D: TBP⟩
end type sample_2d_t
```

⟨*Muli MC int: sample 2D: TBP*⟩≡

```
procedure :: write_to_marker => sample_2d_write_to_marker
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_2d_write_to_marker (this, marker, status)
  class(sample_2d_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  integer :: n
  call marker%mark_begin ("sample_2d_t")
  call marker%mark ("n_regions", this%n_regions)
  call marker%mark ("n_alloc", this%n_alloc)
  call marker%mark ("n_hits", this%n_hits)
  call marker%mark ("range", this%range)
  if (this%n_regions > 0) then
     call marker%mark_instance_begin &
          (this%regions(1), name="sample_2d_t", shape=shape (this%regions))
     do n = 1, this%n_regions
        call sample_region_write_to_marker (this%regions(n), marker, status)
     end do
```

```
      call marker%mark_instance_end ()
    end if
    call marker%mark_end ("sample_2d_t")
  end subroutine sample_2d_write_to_marker
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure :: read_from_marker => sample_2d_read_from_marker
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_2d_read_from_marker (this, marker, status)
    class(sample_2d_t), intent(out) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    integer :: n
    call marker%pick_begin ("sample_2d_t", status=status)
    call marker%pick ("n_regions", this%n_regions, status)
    call marker%pick ("n_alloc", this%n_alloc, status)
    call marker%pick ("n_hits", this%n_hits, status)
    call marker%pick ("range", this%range, status)
    if (this%n_regions > 0) then
       call marker%pick_begin ("regions", status=status)
       allocate (this%regions (this%n_regions))
       do n = 1, this%n_regions
          call sample_region_read_from_marker (this%regions(n), marker, status)
       end do
       call marker%pick_end ("regions", status)
    end if
    call marker%pick_end ("sample_2d_t", status)
  end subroutine sample_2d_read_from_marker
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure :: print_to_unit => sample_2d_print_to_unit
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_2d_print_to_unit (this, unit, parents, components, peers)
    class(sample_2d_t), intent(in) :: this
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    integer :: n
    write (unit, "(1x,A)")           "components of sample_2d_t"
    write (unit, "(3x,A,I10)")       "n_regions:        ", this%n_regions
    write (unit, "(3x,A,I10)")       "n_alloc:          ", this%n_alloc
    write (unit, "(3x,A,2(E20.10))") "range:            ", this%range
    if (allocated (this%regions)) then
       if (components > 0) then
          write (unit, "(3x,A)")  "regions:"
          do n = 1, this%n_regions
             call this%regions(n)%print_to_unit &
                  (unit, parents, components-1, peers)
          end do
       else
          write (unit, "(3x,A)")  "skipping regions."
       end if
    else
```

```
      write (unit, "(3x,A)")  "regions are not allocated."
    end if
  end subroutine sample_2d_print_to_unit
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure, nopass :: get_type => sample_2d_get_type
```

⟨*Muli MC int: procedures*⟩+≡

```
  pure subroutine sample_2d_get_type (type)
    character(:), allocatable, intent(out) :: type
    allocate (type, source="sample_2d_t")
  end subroutine sample_2d_get_type
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure :: initialize => sample_2d_initialize
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_2d_initialize (this, n_alloc)
    class(sample_2d_t), intent(out) :: this
    integer, intent(in) :: n_alloc
    integer :: n
    if (allocated (this%regions))  deallocate (this%regions)
    allocate (this%regions (n_alloc))
    this%n_alloc = n_alloc
    this%n_regions = 1
    call this%regions(1)%initialize (n_alloc)
    ! do n = 1, n_alloc
    !    call this%regions(n)%initialize (n_alloc)
    ! end do
  end subroutine sample_2d_initialize
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure :: contains => sample_2d_contains
```

⟨*Muli MC int: procedures*⟩+≡

```
  pure logical function sample_2d_contains (this, pts2)
    class(sample_2d_t), intent(in) :: this
    real(default), intent(in) :: pts2
    sample_2d_contains = this%range(1) <= pts2 .and. pts2 <= this%range(2)
  end function sample_2d_contains
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure :: generate_hit => sample_2d_generate_hit
```

⟨*Muli MC int: procedures*⟩+≡

```
  pure subroutine sample_2d_generate_hit (this, rnd, boost, hit, region)
    class(sample_2d_t), intent(in) :: this
    integer, dimension(3), intent(in) :: rnd
    integer, intent(out) :: region
    integer :: n, sum
    real(default), dimension(2), intent(out) :: hit
    real(default), intent(out) :: boost
    if (0 < this%n_hits .and. this%n_hits < 10) then
       !!! this should be improved
```

```
      sum = modulo(rnd(1),this%n_hits) + 1
      region = 0
      do while (sum > 0)
         region = region + 1
         sum = sum - this%regions(region)%n_hits
      end do
      call this%regions(region)%generate_hit (rnd(2:3), boost, hit)
      boost = boost * this%n_hits / this%regions(region)%n_hits
   else
      if (this%n_regions > 1) then
         !!! this should be improved
         region = modulo(rnd(1), this%n_regions) + 1
         call this%regions(region)%generate_hit (rnd(2:3), boost, hit)
         boost = boost * this%n_regions
      else
         region = 1
         call this%regions(1)%generate_hit (rnd(2:3), boost, hit)
      end if
   end if
end subroutine sample_2d_generate_hit

!  pure subroutine sample_2d_generate_hit (this, rnd, boost, hit, region)
!    class(sample_2d_t), intent(in) :: this
!    integer, dimension(3), intent(in) :: rnd
!    integer, intent(out) :: region
!    real(double), dimension(2), intent(out) :: hit
!    real(double), intent(out) :: boost
!    region = modulo(rnd(1), this%n_regions) + 1   !!! this should be improved
!    call this%regions(region)%generate_hit (rnd(2:3), boost, hit)
!    boost = boost * this%n_regions
!  end subroutine sample_2d_generate_hit
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
procedure :: confirm_hit => sample_2d_confirm_hit
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_2d_confirm_hit (this, hit, region, full)
  class(sample_2d_t), intent(inout) :: this
  integer, intent(in) :: region
  real(default), dimension(3), intent(in) :: hit
  type(sample_region_t), allocatable :: old_region
  real(default), dimension(2) :: mean, var, diff, cm, cv, c
  integer :: n, n_alloc, dim
  logical, intent(out) :: full
  this%n_hits = this%n_hits + 1
  if (region <= this%n_alloc) then
     full = .false.
     call this%regions(region)%confirm_hit (hit)
     n_alloc = this%regions(region)%n_alloc
     if (this%regions(region)%is_full()) then
        if (this%is_full()) then
           full = .true.
        else
           this%n_regions = this%n_regions + 1
```

```
            allocate (old_region)
            call this%regions(region)%move_components (old_region)
            mean = sum (old_region%hyp_hits(1:2,:), dim=2) / n_alloc
            var = 0
            do n = 1, n_alloc
               var = var + abs(mean-old_region%hyp_hits(1:2,n))
            end do
            var = var / n_alloc
            diff = old_region%corners(1:2,2) - old_region%corners(1:2,1)
            cm = abs ([0.5_default,0.5_default] - &
                 (old_region%corners(1:2,2) - mean) / diff)
            cv = abs(2*([0.25_default,0.25_default] - var / diff))
            c = max(cm,cv)
            if (c(1) < c(2)) then
               dim = 2
            else
               dim = 1
            end if
            call old_region%split (mean(dim), dim, this%n_alloc, &
                 this%regions(region), this%regions(this%n_regions))
         end if
      end if
   else
      write (*,*) "sample_2d_confirm_hit: Region ", region, &
           " not allocated."
   end if
end subroutine sample_2d_confirm_hit
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
procedure :: split => sample_2d_split
```

⟨*Muli MC int: procedures*⟩+≡

```
recursive subroutine sample_2d_split (this, n_alloc, pos, lower, upper)
  class(sample_2d_t), intent(in) :: this
  integer, intent(in) :: n_alloc
  real(default), intent(in) :: pos
  type(sample_2d_t), intent(out) :: lower, upper
  integer :: n_r, n_h
  real(default), dimension(3) :: hit
  ! print *,"sample_2d_split: ", pos, this%range
  call lower%initialize (4*n_alloc)
  call upper%initialize (4*n_alloc)
  do n_r = this%n_regions, 1, -1
     do n_h = 1, this%regions(n_r)%n_hits
        hit = this%regions(n_r)%hyp_hits (1:3,n_h)
        if (hit(3) > pos) then
           call upper%push (hit)
        else
           call lower%push (hit)
        end if
     end do
  end do
  lower%range = [this%range(1), pos]
  upper%range = [pos, this%range(2)]
```

```
  end subroutine sample_2d_split

!  subroutine sample_2d_split (this, n_alloc, pos, lower, upper)
!    class(sample_2d_t), intent(in) :: this
!    integer, intent(in) :: n_alloc
!    real(default), intent(in) :: pos
!    type(sample_2d_t), intent(out) :: lower, upper
!    integer :: n, n_hit
!    real(default), dimension(3) :: hit
!    allocate (lower%regions (n_alloc))
!    allocate (upper%regions (n_alloc))
!    !$OMP PARALLEL DO FIRSTPRIVATE (this, pos, n_alloc) SHARED (lower, upper)
!    do n = 1, this%n_regions
!      call sample_region_split (this%regions(n), pos, 3, n_alloc, &
!         lower%regions(n),upper%regions(n))
!    end do
!    !$OMP END PARALLEL DO
!    lower%n_regions = this%n_regions
!    upper%n_regions = this%n_regions
!    lower%n_alloc = n_alloc
!    upper%n_alloc = n_alloc
!    lower%range = [this%range(1), pos]
!    upper%range = [pos, this%range(2)]
!  end subroutine sample_2d_split
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure :: push => sample_2d_push
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_2d_push (this, hit)
    class(sample_2d_t), intent(inout) :: this
    real(default), dimension(3), intent(in) :: hit
    integer :: region
    logical :: full
    do region = 1, this%n_regions
       if (this%regions(region)%contains (hit)) then
          call this%confirm_hit (hit, region, full)
          ! call this%regions(region)%confirm_hit (hit)
          if (full) print *,"sample_2d_push: region is full now"
          exit
       end if
    end do
    if (region > this%n_regions) &
         print *, "sample_2d_push: no region contains ", hit
  end subroutine sample_2d_push
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure :: write_hits => sample_2d_write_hits
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_2d_write_hits (this, unit)
    class(sample_2d_t), intent(in) :: this
    integer, intent(in) :: unit
    integer :: n
```

```
    do n = 1, this%n_regions
       call this%regions(n)%write_hits (unit)
    end do
  end subroutine sample_2d_write_hits
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure :: is_full => sample_2d_is_full
```

⟨*Muli MC int: procedures*⟩+≡

```
  elemental logical function sample_2d_is_full (this)
    class(sample_2d_t), intent(in) :: this
    sample_2d_is_full = this%n_alloc == this%n_regions
  end function sample_2d_is_full
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure :: move_components => sample_2d_move_components
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_2d_move_components (this, that)
    class(sample_2d_t), intent(inout) :: this
    class(sample_2d_t), intent(out) :: that
    that%n_alloc = this%n_alloc
    that%n_regions = this%n_regions
    that%n_hits = this%n_hits
    that%range = this%range
    call move_alloc (this%regions, that%regions)
    this%n_alloc = 0
    this%n_regions = 0
    this%n_hits = 0
    this%range = [zero,zero]
  end subroutine sample_2d_move_components
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure :: thickness => sample_2d_thickness
```

⟨*Muli MC int: procedures*⟩+≡

```
  elemental function sample_2d_thickness (this)
    class(sample_2d_t), intent(in) :: this
    real(default) :: sample_2d_thickness
    sample_2d_thickness = this%range(2) - this%range(1)
  end function sample_2d_thickness
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure :: analyse => sample_2d_analyse
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_2d_analyse (this, dir, file)
    class(sample_2d_t), intent(in) :: this
    character(*), intent(in) :: dir, file
    integer :: u
    real(default), dimension(1:2,0:100,0:100) :: grid
    integer, dimension(0:100,0:100) :: i_grid
    integer :: r, x, y
    integer, dimension(2,2) :: i
```

```
    call generate_unit (u)
    print *, "sample_2d_analyse: ", dir // "/" // file
    open (u, file=dir//"/"//file)
    do x = 0, 100
       do y = 0, 100
          grid(1:2,x,y) = [-one,-one]
       end do
    end do
    do r = 1, this%n_regions
       i = int(this%regions(r)%corners*1E2_default)
       do x = i(1,1), i(1,2)
          do y = i(2,1), i(2,2)
             i_grid(x,y) = this%regions(r)%n_hits
             grid(1,x,y) = one / this%regions(r)%area ()
             grid(2,x,y) = this%regions(r)%density ()
          end do
       end do
    end do
    do x = 0, 100
       do y = 0, 100
          write (u, *) x, y, i_grid(x,y), grid(1:2,x,y)
       end do
       write (u, *)
    end do
    close (u)
  end subroutine sample_2d_analyse
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure :: to_generator => sample_2d_to_generator
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_2d_to_generator (this)
    class(sample_2d_t), intent(inout) :: this
    integer :: region
    do region = 1, this%n_regions
       call this%regions(region)%to_generator ()
    end do
  end subroutine sample_2d_to_generator
```

⟨*Muli MC int: sample 2D: TBP*⟩+≡

```
  procedure :: mean => sample_2d_mean
```

⟨*Muli MC int: procedures*⟩+≡

```
  elemental function sample_2d_mean (this, dim) result (mean)
    class(sample_2d_t), intent(in) :: this
    integer, intent(in) :: dim
    real(default) :: mean
    integer :: region, hit
    mean = zero
    do region = 1, this%n_regions
       do hit = 1, this%regions(region)%n_hits
          mean = mean + this%regions(region)%hyp_hits (dim, hit)
       end do
    end do
```

```
      mean = mean / this%n_hits
    end function sample_2d_mean
```

⟨*Muli MC int: public*⟩+≡

```
    public :: sample_3d_t
```

⟨*Muli MC int: types*⟩+≡

```
    type, extends (ser_class_t) :: sample_3d_t
       integer::n_slices=0
       integer::n_alloc=0
       type(sample_2d_t), dimension(:),allocatable::slices
     contains
     ⟨Muli MC int: sample 3D: TBP⟩
    end type sample_3d_t
```

⟨*Muli MC int: sample 3D: TBP*⟩≡

```
    procedure :: write_to_marker => sample_3d_write_to_marker
```

⟨*Muli MC int: procedures*⟩+≡

```
    subroutine sample_3d_write_to_marker (this, marker, status)
      class(sample_3d_t), intent(in) :: this
      class(marker_t), intent(inout) :: marker
      integer(dik), intent(out) :: status
      integer :: n
      call marker%mark_begin ("sample_3d_t")
      call marker%mark ("n_slices", this%n_slices)
      call marker%mark ("n_alloc", this%n_alloc)
      if (this%n_slices > 0) then
         call marker%mark_instance_begin &
              (this%slices(1), "slices", shape=shape(this%slices))
         do n = 1, this%n_slices
            call sample_2d_write_to_marker (this%slices(n), marker, status)
         end do
         call marker%mark_instance_end ()
      end if
      call marker%mark_end ("sample_3d_t")
    end subroutine sample_3d_write_to_marker
```

⟨*Muli MC int: sample 3D: TBP*⟩+≡

```
    procedure :: read_from_marker => sample_3d_read_from_marker
```

⟨*Muli MC int: procedures*⟩+≡

```
    subroutine sample_3d_read_from_marker (this, marker, status)
      class(sample_3d_t), intent(out) :: this
      class(marker_t), intent(inout) :: marker
      integer(dik), intent(out) :: status
      integer :: n
      call marker%pick_begin ("sample_3d_t", status=status)
      call marker%pick ("n_slices", this%n_slices, status)
      call marker%pick ("n_alloc", this%n_alloc, status)
      if (this%n_slices > 0) then
         call marker%pick_instance_begin ("slices", status=status)
         allocate(this%slices (this%n_slices))
         do n = 1, this%n_slices
```

```
            call sample_2d_read_from_marker (this%slices(n), marker, status)
         end do
         call marker%pick_instance_end (status)
      end if
      call marker%pick_end ("sample_3d_t", status)
   end subroutine sample_3d_read_from_marker
```

⟨*Muli MC int: sample 3D: TBP*⟩+≡

```
   procedure :: print_to_unit => sample_3d_print_to_unit
```

⟨*Muli MC int: procedures*⟩+≡

```
   subroutine sample_3d_print_to_unit (this, unit, parents, components, peers)
      class(sample_3d_t), intent(in) :: this
      integer, intent(in) :: unit
      integer(dik), intent(in) :: parents, components, peers
      integer :: n
      write (unit, "(1x,A)")     "components of sample_3d_t"
      write (unit, "(3x,A,I10)") "n_slices:         ", this%n_slices
      write (unit, "(3x,A,I10)") "n_alloc:          ", this%n_alloc
      if (allocated (this%slices)) then
         if (components > 0) then
            do n = 1, this%n_slices
               call this%slices(n)%print_to_unit (unit, parents, components-1, peers)
            end do
         else
            write (unit, "(3x,A)")  "skipping slices."
         end if
      else
         write (unit, "(3x,A)")  "slices are not allocated."
      end if
   end subroutine sample_3d_print_to_unit
```

⟨*Muli MC int: sample 3D: TBP*⟩+≡

```
   procedure, nopass :: get_type => sample_3d_get_type
```

⟨*Muli MC int: procedures*⟩+≡

```
   pure subroutine sample_3d_get_type (type)
      character(:), allocatable, intent(out) :: type
      allocate (type, source="sample_3d_t")
   end subroutine sample_3d_get_type
```

⟨*Muli MC int: sample 3D: TBP*⟩+≡

```
   procedure :: measure => sample_3d_measure
```

⟨*Muli MC int: procedures*⟩+≡

```
   elemental function sample_3d_measure (this)
      real(default) :: sample_3d_measure
      class(sample_3d_t), intent(in) :: this
      sample_3d_measure = one
   end function sample_3d_measure
```

⟨*Muli MC int: sample 3D: TBP*⟩+≡

```
   procedure :: to_generator => sample_3d_to_generator
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_3d_to_generator(this)
  class(sample_3d_t), intent(inout)::this
  integer::slice
  do slice=1,this%n_slices
     call this%slices(slice)%to_generator()
  end do
end subroutine sample_3d_to_generator
```

⟨*Muli MC int: sample 3D: TBP*⟩+≡

```
generic :: initialize => sample_3d_initialize
procedure :: sample_3d_initialize
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_3d_initialize (this, n_alloc)
  class(sample_3d_t), intent(out) :: this
  integer, intent(in) :: n_alloc
  if (allocated (this%slices))  deallocate (this%slices)
  if (n_alloc > 0) then
     allocate (this%slices (n_alloc))
     this%n_alloc = n_alloc
     this%n_slices = 1
     call this%slices(1)%initialize (n_alloc)
  else
     this%n_alloc = 0
  end if
end subroutine sample_3d_initialize
```

⟨*Muli MC int: sample 3D: TBP*⟩+≡

```
procedure :: sample_3d_generate_hit
generic :: generate_hit => sample_3d_generate_hit
```

⟨*Muli MC int: procedures*⟩+≡

```
pure subroutine sample_3d_generate_hit &
     (this, rnd, pts2, boost, hit, region, slice)
  class(sample_3d_t), intent(in) :: this
  integer, intent(in), dimension(3) :: rnd
  real(default), intent(in) :: pts2
  integer, intent(out) :: slice, region
  real(default), dimension(3), intent(out) :: hit
  real(default), intent(out) :: boost
  if (this%n_slices == 0) then
     call muli_mcint_generate_hit (rnd, unit_square, hit(1:2))
     boost = 1._default
     slice = 1
     region = 1
  else
     do slice = 1, this%n_slices
        if (this%slices(slice)%contains (pts2))  exit
     end do
     call this%slices(slice)%generate_hit (rnd, boost, hit(1:2), region)
  end if
  hit(3) = pts2
end subroutine sample_3d_generate_hit
```

⟨*Muli MC int: sample 3D: TBP*⟩+≡

```
procedure :: sample_3d_confirm_hit
generic :: confirm_hit => sample_3d_confirm_hit
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_3d_confirm_hit (this, hit, region, slice)
  class(sample_3d_t), intent(inout) :: this
  integer, intent(in) :: slice, region
  real(default), intent(in), dimension(3) :: hit
  type(sample_2d_t), allocatable :: old_slice
  integer :: n
  logical :: full
  if (this%n_alloc < slice) then
     print *, "sample_3d_confirm_hit: Slice ", slice, " not allocated."
  else
  ! if (.not. allocated (this%slices))  call this%initialize (2)
     call this%slices(slice)%confirm_hit (hit, region, full)
     if (full) then
        if (this%n_alloc == this%n_slices)  call this%enlarge ()
        this%n_slices = this%n_slices + 1
        allocate (old_slice)
        call this%slices(slice)%move_components (old_slice)
        call sample_2d_split (old_slice, this%n_alloc, &
             old_slice%mean(3), this%slices(slice), &
             this%slices(this%n_slices))
     end if
  end if
end subroutine sample_3d_confirm_hit
```

⟨*Muli MC int: sample 3D: TBP*⟩+≡

```
procedure :: enlarge => sample_3d_enlarge
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_3d_enlarge (this)
  class(sample_3d_t), intent(inout) :: this
  type(sample_2d_t), allocatable, dimension(:) :: old_slices
  integer :: n
  print *, "sample_3d_enlarge"
  call move_alloc (this%slices, old_slices)
  this%n_alloc = this%n_alloc * 2
  allocate (this%slices (this%n_alloc))
  do n = 1, size(old_slices)
     call old_slices(n)%move_components(this%slices(n))
  end do
end subroutine sample_3d_enlarge
```

⟨*Muli MC int: public*⟩+≡

```
public :: sample_int_kind_t
```

⟨*Muli MC int: types*⟩+≡

```
type, extends (sample_3d_t) :: sample_int_kind_t
   integer :: n_proc = 0
   integer(kind=i64) :: n_tries = 0
   integer :: n_hits = 0
   integer :: n_over = 0
```

```
      integer, dimension(:), allocatable :: hits, weights, processes
      real(default) :: overall_boost = 1E-1_default
   contains
   ⟨Muli MC int: sample int: TBP⟩
end type sample_int_kind_t
```

⟨*Muli MC int: sample int: TBP*⟩≡

```
procedure :: write_to_marker => sample_int_kind_write_to_marker
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_int_kind_write_to_marker (this, marker, status)
  class(sample_int_kind_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%mark_begin ("sample_int_kind_t")
  call sample_3d_write_to_marker (this, marker, status)
  call marker%mark ("n_hits", this%n_hits)
  call marker%mark ("n_proc", this%n_proc)
  call marker%mark ("boost", this%overall_boost)
  if (this%n_hits > 0) then
     call marker%mark ("hits", this%hits)
  end if
  if (this%n_proc > 0) then
     call marker%mark ("processes", this%processes)
     call marker%mark ("weights", this%weights)
  end if
  call marker%mark_end ("sample_int_kind_t")
end subroutine sample_int_kind_write_to_marker
```

⟨*Muli MC int: sample int: TBP*⟩+≡

```
procedure :: read_from_marker => sample_int_kind_read_from_marker
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_int_kind_read_from_marker (this, marker, status)
  class(sample_int_kind_t), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%pick_begin ("sample_int_kind_t", status=status)
  call sample_3d_read_from_marker (this, marker, status)
  call marker%pick ("n_hits", this%n_hits, status)
  call marker%pick ("n_proc", this%n_proc, status)
  call marker%pick ("boost", this%overall_boost, status)
  if (this%n_hits > 0) then
     allocate (this%hits (this%n_hits))
     call marker%pick ("hits", this%hits, status)
  end if
  if (this%n_proc > 0) then
     allocate (this%processes (this%n_proc))
     call marker%pick ("processes", this%processes, status)
     allocate (this%weights (this%n_proc))
     call marker%pick ("weights", this%weights, status)
  end if
  call marker%pick_end ("sample_int_kind_t", status)
end subroutine sample_int_kind_read_from_marker
```

⟨*Muli MC int: sample int: TBP*⟩+≡

```
procedure :: print_to_unit => sample_int_kind_print_to_unit
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_int_kind_print_to_unit (this, unit, parents, components, peers)
  class(sample_int_kind_t), intent(in) :: this
  integer, intent(in) :: unit
  integer(dik), intent(in) :: parents, components, peers
  integer :: n
  if (parents > 0)  call sample_3d_print_to_unit &
       (this, unit, parents, components, peers)
  write (unit, "(1x,A)")       "components of sample_int_kind_t"
  write (unit, "(3x,A,I10)")   "n_hits:          ", this%n_hits
  write (unit, "(3x,A,I10)")   "n_proc:          ", this%n_proc
  write (unit, "(3x,A,E14.7)") "overall_boost:   ", this%overall_boost
  write (unit, "(3x,A)")       "hits:"
  write (unit, "(3x,10(I0,1x))")  this%hits(1:this%n_hits)
  write (unit, "(3x,A)")       "weights:"
  write (unit, "(3x,10(I0,1x))")  this%weights
  write (unit, "(3x,A)")  "processes:"
  write (unit, "(3x,2(I0,1x))") this%processes
end subroutine sample_int_kind_print_to_unit
```

⟨*Muli MC int: sample int: TBP*⟩+≡

```
procedure, nopass :: get_type => sample_int_kind_get_type
```

⟨*Muli MC int: procedures*⟩+≡

```
pure subroutine sample_int_kind_get_type (type)
  character(:),allocatable, intent(out) :: type
  allocate (type, source="sample_int_kind_t")
end subroutine sample_int_kind_get_type
```

⟨*Muli MC int: sample int: TBP*⟩+≡

```
procedure :: to_generator => sample_int_kind_to_generator
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_int_kind_to_generator(this)
  class(sample_int_kind_t), intent(inout)::this
  integer::int_kind
  if (allocated(this%hits))deallocate(this%hits)
  call sample_3d_to_generator(this)
end subroutine sample_int_kind_to_generator
```

⟨*Muli MC int: sample int: TBP*⟩+≡

```
procedure :: process_id => sample_int_kind_process_id
```

⟨*Muli MC int: procedures*⟩+≡

```
elemental integer function sample_int_kind_process_id (this, subprocess)
  class(sample_int_kind_t), intent(in) :: this
  integer, intent(in) :: subprocess
  sample_int_kind_process_id = this%processes(subprocess)
end function sample_int_kind_process_id
```

⟨*Muli MC int: sample int: TBP*⟩+≡

```
procedure :: sample_int_kind_initialize
generic :: initialize => sample_int_kind_initialize
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_int_kind_initialize (this, n_alloc, processes, overall_boost)
  class(sample_int_kind_t), intent(out) :: this
  integer, intent(in) :: n_alloc
  integer, intent(in), dimension(:) :: processes
  real(default), optional, intent(in) :: overall_boost
  integer :: s, n
  s = size(processes)
  call sample_3d_initialize (this, n_alloc)
  if (allocated (this%hits))  deallocate (this%hits)
  allocate (this%hits (n_alloc))
  if (allocated (this%weights))  deallocate (this%weights)
  allocate (this%weights(s))
  if (allocated (this%processes))  deallocate (this%processes)
  allocate (this%processes(s), source=processes)
  do n = 1, s
     this%weights(n) = 0
  end do
  this%n_alloc = n_alloc
  this%n_hits = 0
  this%n_proc = s
  if (present (overall_boost))  this%overall_boost = overall_boost
  this%overall_boost = this%overall_boost * this%n_proc
  ! print *, this%weights
end subroutine sample_int_kind_initialize
```

⟨*Muli MC int: sample int: TBP*⟩+≡

```
procedure :: sample_int_kind_generate_hit
```

⟨*Muli MC int: procedures*⟩+≡

```
pure subroutine sample_int_kind_generate_hit &
     (this, rnd, pts2, boost, hit, region, slice, subprocess)
  class(sample_int_kind_t), intent(in) :: this
  integer, dimension(4), intent(in) :: rnd
  real(default), intent(in) :: pts2
  real(default), dimension(3), intent(out) :: hit
  integer, intent(out) :: region, slice, subprocess
  real(default), intent(out) :: boost
  integer :: n_n
  ! print *, rnd, pts2, boost, hit, region, slice, subprocess
  call sample_3d_generate_hit &
       (this, rnd(2:4), pts2, boost, hit, region, slice)
  n_n = modulo(rnd(1), this%n_hits + size(this%weights)) + 1
  if (n_n > this%n_hits) then
     subprocess = n_n - this%n_hits
  else
     subprocess = this%hits(n_n)
  end if
  boost = boost * this%overall_boost * (this%n_proc + this%n_hits) / &
       (this%n_proc * (this%weights(subprocess) + 1))
end subroutine sample_int_kind_generate_hit
```

⟨*Muli MC int: sample int: TBP*⟩+≡

```
procedure :: mcgenerate_hit => sample_int_kind_mcgenerate_hit
generic :: generate_hit => sample_int_kind_generate_hit
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_int_kind_mcgenerate_hit (this, pts2, mean, &
     integrand_kind, tao_rnd, process_id, cart_hit)
  class(sample_int_kind_t), intent(inout) :: this
  integer, intent(in) :: integrand_kind
  real(default), intent(in) :: pts2, mean
  type(tao_random_state), intent(inout) :: tao_rnd
  real(default), dimension(3), intent(out) :: cart_hit
  integer, intent(out) :: process_id
  real(default) :: boost
  integer :: region, slice, subprocess
  integer, dimension(4) :: i_rnd
  real(default) :: dddsigma, d_rnd
  real(default), dimension(3) :: hyp_hit
  MC: do
     this%n_tries = this%n_tries + 1
     call tao_random_number (tao_rnd, i_rnd)
     call tao_random_number (tao_rnd, d_rnd)
     ! print *, pts2, mean, integrand_kind, process_id, cart_hit
     call this%generate_hit (i_rnd, pts2, boost, hyp_hit, region, &
          slice, subprocess)
     process_id = this%process_id(subprocess)
     call interactions_dddsigma_reg (process_id, integrand_kind, &
          hyp_hit, cart_hit, dddsigma)
     dddsigma = dddsigma * boost
     if (d_rnd*mean < dddsigma) then
        exit MC
     end if
  end do MC
  if (mean < dddsigma) then
     call this%confirm_hit (hyp_hit, region, slice, subprocess, .true.)
  else
     call this%confirm_hit (hyp_hit, region, slice, subprocess, .false.)
  end if
end subroutine sample_int_kind_mcgenerate_hit
```

⟨*Muli MC int: sample int: TBP*⟩+≡

```
procedure :: sample_int_kind_confirm_hit
generic :: confirm_hit => sample_int_kind_confirm_hit
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_int_kind_confirm_hit &
     (this, hit, region, slice, subprocess, over)
  class(sample_int_kind_t), intent(inout) :: this
  real(default), dimension(3), intent(in) :: hit
  integer, intent(in) :: region, slice, subprocess
  integer, dimension(:), allocatable :: tmp_hits
  logical, optional, intent(in) :: over
  this%n_hits = this%n_hits + 1
```

```
    if (present(over)) then
       if (over) then
          this%n_over = this%n_over + 1
          this%overall_boost = this%overall_boost / 1.1_default
       else
          this%overall_boost = this%overall_boost * 1.0001_default
       end if
    end if
    if (0 < size(this%hits)) then
       if (this%n_hits > size(this%hits)) then
          call move_alloc (this%hits, tmp_hits)
          allocate (this%hits (2*size(tmp_hits)))
          this%hits (1:size(tmp_hits)) = tmp_hits
       end if
       this%hits(this%n_hits) = subprocess
    end if
    this%weights(subprocess) = this%weights(subprocess) + 1
    call sample_3d_confirm_hit (this, hit, region, slice)
  end subroutine sample_int_kind_confirm_hit
```

⟨*Muli MC int: sample int: TBP*⟩+≡

```
  procedure :: analyse => sample_int_kind_analyse
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_int_kind_analyse (this, dir, prefix)
    class(sample_int_kind_t), intent(in) :: this
    character(*), intent(in) :: dir, prefix
    integer :: slices_unit, subprocs_unit
    integer :: n, slice
    character(3) :: slice_name
    integer, dimension(:), allocatable :: int_a
    real(default), dimension(:), allocatable :: real_a
    call generate_unit (slices_unit)
    print *, "sample_int_kind_analyse: ", dir // "/" // prefix // &
         "slice_distribution.plot"
    open (slices_unit, file=dir // "/" // prefix // "slice_distribution.plot")
    call generate_unit (subprocs_unit)
    print *, "sample_int_kind_analyse: ", dir // "/" // prefix // &
         "subproc_distribution.plot"
    open (subprocs_unit, file=dir // "/" // prefix // &
         "subproc_distribution.plot")
    allocate (real_a (this%n_slices))
    allocate (int_a (this%n_slices))
    do n = 1, this%n_slices
       real_a(n) = this%slices(n)%range(1)
    end do
    call misc_sort (real_a, int_a)
    do n = 1, size (this%weights)
       if (this%n_hits > 0) then
          write (subprocs_unit, fmt=*) real(this%weights(n)), &
               real(this%weights(n)+1) / this%n_hits
       else
          write (subprocs_unit, fmt=*) 0, 0
       end if
```

```
    end do
    do n = 1, this%n_slices
       slice = int_a (n)
       call integer_with_leading_zeros (n, 3, slice_name)
       call sample_2d_analyse (this%slices(slice), dir, prefix // &
            slice_name // ".plot")
       print *, this%n_hits, this%slices(slice)%range(2) - &
            this%slices(slice)%range(1)
       if (this%n_hits > 0) then
          write (slices_unit, *)  this%slices(slice)%range(1), &
               this%slices(slice)%range(2), this%slices(slice)%n_hits, &
               real (this%slices(slice)%n_hits) / (this%n_hits * &
               (this%slices(slice)%range(2) - this%slices(slice)%range(1)))
       else
          write (slices_unit, *)  this%slices(slice)%range(1), &
               this%slices(slice)%range(2), this%slices(slice)%n_hits, zero
       end if
    end do
    write (slices_unit, *) one, zero, zero, zero
    close (slices_unit)
    close (subprocs_unit)
  end subroutine sample_int_kind_analyse
```

⟨*Muli MC int: public*⟩+≡

```
  public :: sample_inclusive_t
```

⟨*Muli MC int: types*⟩+≡

```
  type, extends (ser_class_t) :: sample_inclusive_t
     integer :: n_alloc = 0
     integer(kind=i64) :: n_tries_sum = i_zero
     integer(kind=i64) :: n_over_sum = i_zero
     integer(kind=i64) :: n_hits_sum = i_zero
     type(sample_int_kind_t), dimension(:), allocatable :: int_kinds
   contains
   ⟨Muli MC int: sample inclusive: TBP⟩
  end type sample_inclusive_t
```

⟨*Muli MC int: sample inclusive: TBP*⟩≡

```
  procedure :: write_to_marker => sample_inclusive_write_to_marker
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_inclusive_write_to_marker (this, marker, status)
    class(sample_inclusive_t), intent(in) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    integer :: n
    call marker%mark_begin ("sample_inclusive_t")
    call marker%mark ("n_alloc", this%n_alloc)
    if (allocated (this%int_kinds)) then
       call marker%mark_begin (tag="int_kinds", shape=shape(this%int_kinds))
       do n = 1, size(this%int_kinds)
          call this%int_kinds(n)%write_to_marker (marker, status)
       end do
       call marker%mark_instance_end ()
```

```
    else
       call marker%mark_empty (tag="int_kinds", shape=[0])
    end if
    call marker%mark_end ("sample_inclusive_t")
  end subroutine sample_inclusive_write_to_marker
```

⟨*Muli MC int: sample inclusive: TBP*⟩+≡

```
  procedure :: read_from_marker => sample_inclusive_read_from_marker
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_inclusive_read_from_marker (this, marker, status)
    class(sample_inclusive_t), intent(out) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    integer :: n
    integer, dimension(:), allocatable :: s
    call marker%pick_begin ("sample_inclusive_t", status=status)
    call marker%pick ("n_alloc", this%n_alloc, status)
    call marker%pick_begin ("int_kinds", shape=s, status=status)
    if (s(1) > 0) then
       do n = 1, size(this%int_kinds)
          call this%int_kinds(n)%read_from_marker (marker, status)
       end do
       call marker%pick_end ("int_kinds",status)
    end if
    call marker%pick_end ("sample_inclusive_t", status)
  end subroutine sample_inclusive_read_from_marker
```

⟨*Muli MC int: sample inclusive: TBP*⟩+≡

```
  procedure :: print_to_unit => sample_inclusive_print_to_unit
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_inclusive_print_to_unit (this, unit, parents, components, peers)
    class(sample_inclusive_t), intent(in) :: this
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    integer :: n
    write (unit, "(1x,A)")  "components of sample_inclusive_t"
    write (unit, "(3x,A,I10)")  "n_alloc:          ", this%n_alloc
    if (allocated (this%int_kinds)) then
       if (components > 0) then
          write (unit, "(3x,A)")  "int_kinds:"
          do n = 1, this%n_alloc
             call this%int_kinds(n)%print_to_unit &
                  (unit, parents, components-1, peers)
          end do
       else
          write (unit, "(3x,A)")  "skipping int_kinds."
       end if
    else
       write (unit, "(3x,A)")  "int_kinds are not allocated."
    end if
  end subroutine sample_inclusive_print_to_unit
```

⟨*Muli MC int: sample inclusive: TBP*⟩+≡

```
procedure, nopass :: get_type => sample_inclusive_get_type
```

⟨*Muli MC int: procedures*⟩+≡

```
pure subroutine sample_inclusive_get_type (type)
  character(:), allocatable, intent(out) :: type
  allocate (type, source = "sample_inclusive_t")
end subroutine sample_inclusive_get_type
```

⟨*Muli MC int: sample inclusive: TBP*⟩+≡

```
procedure :: process_id => sample_inclusive_process_id
```

⟨*Muli MC int: procedures*⟩+≡

```
elemental integer function sample_inclusive_process_id &
     (this, subprocess, int_kind)
  class(sample_inclusive_t), intent(in) :: this
  integer, intent(in) :: subprocess, int_kind
  sample_inclusive_process_id = &
       this%int_kinds(int_kind)%processes (subprocess)
end function sample_inclusive_process_id
```

⟨*Muli MC int: sample inclusive: TBP*⟩+≡

```
procedure :: initialize => sample_inclusive_initialize
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_inclusive_initialize &
     (this, n_alloc, sizes, processes, overall_boost)
  class(sample_inclusive_t), intent(out) :: this
  integer, intent(in) :: n_alloc
  integer, dimension(:), intent(in) :: sizes, processes
  real(default), optional, intent(in) :: overall_boost
  integer :: n, sum
  this%n_tries_sum = i_zero
  this%n_over_sum = 0
  this%n_alloc = size(sizes)
  if (allocated(this%int_kinds))  deallocate (this%int_kinds)
  allocate (this%int_kinds (this%n_alloc))
  sum = 0
  do n = 1, this%n_alloc
     call this%int_kinds(n)%initialize (n_alloc, &
          processes(sum+1:sum+sizes(n)), overall_boost)
     sum = sum + sizes(n)
  end do
end subroutine sample_inclusive_initialize
```

⟨*Muli MC int: sample inclusive: TBP*⟩+≡

```
procedure :: finalize => sample_inclusive_finalize
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_inclusive_finalize (this)
  class(sample_inclusive_t), intent(inout) :: this
  deallocate (this%int_kinds)
  this%n_alloc = 0
end subroutine sample_inclusive_finalize
```

⟨*Muli MC int: sample inclusive: TBP*⟩+≡

```
procedure :: generate_hit => sample_inclusive_generate_hit
```

⟨*Muli MC int: procedures*⟩+≡

```
pure subroutine sample_inclusive_generate_hit &
     (this, rnd, pts2, int_kind, hit, region, boost, slice, process)
  class(sample_inclusive_t), intent(in) :: this
  integer, dimension(4), intent(in) :: rnd
  real(default), intent(in) :: pts2
  integer, intent(in) :: int_kind
  real(default), dimension(3), intent(out) :: hit
  integer, intent(out) :: region, slice, process
  real(default), intent(out) :: boost
  call this%int_kinds(int_kind)%generate_hit &
       (rnd, pts2, boost, hit, region, slice, process)
end subroutine sample_inclusive_generate_hit
```

⟨*Muli MC int: sample inclusive: TBP*⟩+≡

```
procedure :: mcgenerate_hit => sample_inclusive_mcgenerate_hit
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_inclusive_mcgenerate_hit &
     (this, pts2, mean, integrand_kind, tao_rnd, process_id, cart_hit)
  class(sample_inclusive_t), intent(inout) :: this
  real(default), intent(in) :: pts2, mean
  integer, intent(in) :: integrand_kind
  type(tao_random_state), intent(inout) :: tao_rnd
  real(default), dimension(3), intent(out) :: cart_hit
  integer, intent(out) :: process_id
  ! print *, "sample_inclusive_mcgenerate_hit &
  !   (this,",pts2,mean,integrand_kind,process_id,cart_hit,")"
  ! print *, allocated (this%int_kinds)
  call sample_int_kind_mcgenerate_hit (this%int_kinds(integrand_kind), &
       pts2, mean, integrand_kind, tao_rnd, process_id, cart_hit)
end subroutine sample_inclusive_mcgenerate_hit
```

⟨*Muli MC int: sample inclusive: TBP*⟩+≡

```
procedure :: confirm_hit => sample_inclusive_confirm_hit
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_inclusive_confirm_hit &
     (this, hit, int_kind, region, slice, process, over)
  class(sample_inclusive_t), intent(inout) :: this
  real(default), dimension(3), intent(in) :: hit
  integer, intent(in) :: int_kind, region, slice, process
  logical, optional, intent(in) :: over
  call this%int_kinds(int_kind)%confirm_hit &
       (hit, region, slice, process, over)
end subroutine sample_inclusive_confirm_hit
```

⟨*Muli MC int: sample inclusive: TBP*⟩+≡

```
procedure :: sum_up => sample_inclusive_sum_up
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_inclusive_sum_up (this)
  class(sample_inclusive_t), intent(inout) :: this
  integer :: n
  this%n_tries_sum = i_zero
  this%n_hits_sum = i_zero
  this%n_over_sum = i_zero
  do n = 1, this%n_alloc
     this%n_tries_sum = this%n_tries_sum+this%int_kinds(n)%n_tries
     this%n_hits_sum = this%n_hits_sum+this%int_kinds(n)%n_hits
     this%n_over_sum = this%n_over_sum+this%int_kinds(n)%n_over
  end do
end subroutine sample_inclusive_sum_up
```

⟨*Muli MC int: sample inclusive: TBP*⟩+≡

```
procedure :: analyse => sample_inclusive_analyse
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_inclusive_analyse (this, dir, subdirs)
  class(sample_inclusive_t), intent(in) :: this
  character(*), intent(in) :: dir
  logical, intent(in) :: subdirs
  integer :: inclusive_unit
  integer :: n, n_hits
  character(2) :: sample_name
  call generate_unit (inclusive_unit)
  open (inclusive_unit, file = dir // "/int_kinds.plot")
  n_hits = 0
  do n = 1, size(this%int_kinds)
     n_hits = n_hits + this%int_kinds(n)%n_hits
  end do
  do n = 1, size(this%int_kinds)
     write (inclusive_unit, *)  n, real(this%int_kinds(n)%n_hits) / n_hits
     call integer_with_leading_zeros (n, 2, sample_name)
     if (subdirs) then
        call sample_int_kind_analyse (this%int_kinds(n), &
             dir // "/" // sample_name, "")
     else
        call sample_int_kind_analyse (this%int_kinds(n), &
             dir, sample_name // "_")
     end if
  end do
  close (inclusive_unit)
end subroutine sample_inclusive_analyse
```

⟨*Muli MC int: sample inclusive: TBP*⟩+≡

```
procedure :: to_generator => sample_inclusive_to_generator
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine sample_inclusive_to_generator (this)
  class(sample_inclusive_t), intent(inout) :: this
  integer :: int_kind
  do int_kind = 1, size(this%int_kinds)
     call this%int_kinds(int_kind)%to_generator ()
```

```
    end do
  end subroutine sample_inclusive_to_generator
```

⟨*Muli MC int: sample inclusive: TBP*⟩+≡

```
  procedure :: allocate => sample_inclusive_allocate
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine sample_inclusive_allocate (this, n_alloc)
    class(sample_inclusive_t), intent(out) :: this
    integer, intent(in) :: n_alloc
    allocate (this%int_kinds (n_alloc))
    this%n_alloc = n_alloc
  end subroutine sample_inclusive_allocate
```

⟨*Muli MC int: procedures*⟩+≡

```
  pure subroutine muli_mcint_generate_hit (rnd, corners, hit)
    real(default), dimension(2), intent(out) :: hit
    integer, intent(in), dimension(2) :: rnd
    real(default), dimension(2,2), intent(in) :: corners
    ! print *, hit
    ! print *, corners
    ! print *, (corners(1:2,2) - corners(1:2,1))
    hit = (rnd / max_d) * (corners(1:2,2) - corners(1:2,1)) + corners(1:2,1)
  end subroutine muli_mcint_generate_hit
```

⟨*Muli MC int: procedures*⟩+≡

```
  subroutine plot_pstvue3d (unit, corners, density)
    integer, intent(in) :: unit
    real(default), dimension(2,2), intent(in) :: corners
    real(default), intent(in) :: density
    real(default), dimension(2) :: width, mean
    real(default), dimension(3,3) :: plot
    width = (corners(:,2) - corners(:,1)) / two
    mean = (corners(:,1) + corners(:,2)) / two
    plot(1,1) = width(1)
    plot(2,1) = width(2)
    plot(3,1) = density / two
    plot(1,2) = mean(1)
    plot(2,2) = mean(2)
    plot(3,2) = density / two
    call log_color_code (density, plot(1:3,3))
    if (density > one) then
       write (unit, fmt='("\mybigcube{",F14.7,"}{",F14.7,"}{",F14.7,"} &
            & {",F14.7,"}{",F14.7,"}{",F14.7,"}{",F14.7,"}{",F14.7,"} &
            & {",F14.7,"}")') plot
       return
    end if
    write (unit, fmt='("\mycube{",F14.7,"}{",F14.7,"}{",F14.7,"} &
         & {",F14.7,"}{",F14.7,"}{",F14.7,"}{",F14.7,"}{",F14.7,"} &
         & {",F14.7,"}")') plot
  end subroutine plot_pstvue3d
```

⟨*Muli MC int: procedures*⟩+≡

```
subroutine log_color_code (number, rgb)
  real(default), intent(in) :: number
  real(default), dimension(3), intent(out) :: rgb
  if (number < exp(-five)) then
     rgb = [zero, zero, exp(five)*number]
  else
     if (number < exp(-four)) then
        rgb = [zero, (number-exp(-five))/(exp(-four)-exp(-five)), one]
     else
        if (number < exp(-three)) then
           rgb = [zero, one, one-((number-exp(-four))/(exp(-three)-exp(-four)))]
        else
           if (number < exp(-two)) then
              rgb = [(number-exp(-three))/(exp(-two)-exp(-three)), one, zero]
           else
              if (number < exp(-one)) then
                 rgb = [one, one-(number-exp(-two))/(exp(-one)-exp(-two)), zero]
              else
                 if (number < one) then
                    rgb = [one, zero, (number-exp(-three))/(one-exp(-three))]
                 else
                    rgb = [exp(one), one, one] * exp(-number)
                    return
                 end if
              end if
           end if
        end if
     end if
  end if
end subroutine log_color_code
```

⟨*Muli MC int: procedures*⟩+≡

```
recursive subroutine misc_sort (in, out)
  real(default), dimension(:), intent(in) :: in
  integer, dimension(:), intent(out) :: out
  integer, dimension(:), allocatable :: tmp
  integer :: n, k, l, cut
  if (size(in) == 1) then
     out = [1]
  else
     if (size(in) == 2) then
        if (in(1) <= in(2)) then
           out = [1,2]
        else
           out = [2,1]
        end if
     else
        cut = size(in) / 2
        k = 1
        l = cut + 1
        allocate (tmp (size(in)))
        call misc_sort (in(1:cut), tmp(1:cut))
        call misc_sort (in(cut+1:), tmp(cut+1:))
```

```
            do n = cut + 1, size(in)
               tmp(n) = tmp(n) + cut
            end do
            do n = 1, size(in)
               if (k > cut) then
                  out(n) = tmp(l)
                  l = l + 1
               else
                  if (l > size(tmp)) then
                     out(n) = tmp(k)
                     k = k + 1
                  else
                     if (in(tmp(k)) < in(tmp(l))) then
                        out(n) = tmp(k)
                        k = k + 1
                     else
                        out(n) = tmp(l)
                        l = l + 1
                     end if
                  end if
               end if
            end do
         end if
      end if
   end subroutine misc_sort
```

22.10 Proton remnants

This file contains the module `muli_remnant`. All bookkeeping of the proton remnants and twin quarks is done here. Furthermore, reweighting of the PDFs to derive remnant PDFs is done here.

⟨`muli_remnant.f90`⟩≡

```
⟨File header⟩

module muli_remnant
  use, intrinsic :: iso_fortran_env
⟨Use kinds with double⟩
⟨Use strings⟩
  use string_utils
  use constants
  use diagnostics
  use pdf_builtin !NODEP!
  use tao_random_numbers !NODEP!
  use muli_base
  use muli_interactions
  use muli_momentum
!  use sf_lhapdf !NODEP!

⟨Standard module head⟩

⟨Muli remnant: variables⟩
```

```
⟨Muli remnant: public⟩

⟨Muli remnant: types⟩

⟨Muli remnant: interfaces⟩

contains

⟨Muli remnant: procedures⟩

end module muli_remnant
```

⟨*Muli remnant: public*⟩≡

```
public :: pdfnorm_t
```

⟨*Muli remnant: types*⟩≡

```
type, extends (ser_class_t) :: pdfnorm_t
   real(default) :: qmin, qmax, dq
   real(default), dimension(-6:6, 0:nq) :: pdf_int
   real(default), dimension(0:4, 0:nq) :: pdf_norm
 contains
 ⟨Muli remnant: pdfnorm: TBP⟩
end type pdfnorm_t
```

⟨*Muli remnant: pdfnorm: TBP*⟩≡

```
procedure :: write_to_marker => pdfnorm_write_to_marker
```

⟨*Muli remnant: procedures*⟩≡

```
subroutine pdfnorm_write_to_marker (this, marker, status)
  class(pdfnorm_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%mark_begin ("pdfnorm_t")
  call marker%mark ("qmin", this%qmin)
  call marker%mark ("qmax", this%qmax)
  call marker%mark ("dq", this%dq)
  call marker%mark ("pdf_int", this%pdf_int)
  call marker%mark ("pdf_norm", this%pdf_norm)
  call marker%mark_end ("pdfnorm_t")
end subroutine pdfnorm_write_to_marker
```

⟨*Muli remnant: pdfnorm: TBP*⟩+≡

```
procedure :: read_from_marker => pdfnorm_read_from_marker
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine pdfnorm_read_from_marker (this, marker, status)
  class(pdfnorm_t), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  character(:), allocatable :: name
  call marker%pick_begin ("pdfnorm_t", status=status)
  call marker%pick ("qmin", this%qmin, status)
  call marker%pick ("qmax", this%qmax, status)
```

```
    call marker%pick ("dq", this%dq, status)
    call marker%pick ("pdf_int", this%pdf_int, status)
    call marker%pick ("pdf_norm", this%pdf_norm, status)
    call marker%pick_end ("pdfnorm_t", status=status)
  end subroutine pdfnorm_read_from_marker
```

⟨*Muli remnant: pdfnorm: TBP*⟩+≡

```
  procedure :: print_to_unit => pdfnorm_print_to_unit
```

⟨*Muli remnant: procedures*⟩+≡

```
  recursive subroutine pdfnorm_print_to_unit &
      (this, unit, parents, components, peers)
    class(pdfnorm_t), intent(in) :: this
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    write (unit, "(1x,A)")       "Components of pdfnorm_t:"
    write (unit, "(3x,A,F7.6)")  "qmin:    ", this%qmin
    write (unit, "(3x,A,F7.6)")  "qmax:    ", this%qmax
    write (unit, "(3x,A,F7.6)")  "dq:      ", this%dq
    if (components > 0) then
       write(unit, "(3x,A,13(F8.6,1x))") "pdf_int:  ", this%pdf_int
       write(unit, "(3x,A,5(F8.6,1x))")  "pdf_norm: ", this%pdf_norm
    else
       write(unit, "(3x,A)")  "Skipping pdf_int"
       write(unit, "(3x,A)")  "Skipping pdf_norm"
    end if
  end subroutine pdfnorm_print_to_unit
```

⟨*Muli remnant: procedures*⟩+≡

⟨*Muli remnant: pdfnorm: TBP*⟩+≡

```
  procedure, nopass :: get_type => pdfnorm_get_type
```

⟨*Muli remnant: procedures*⟩+≡

```
  pure subroutine pdfnorm_get_type (type)
    character(:), allocatable, intent(out) :: type
    allocate (type, source="pdfnorm_t")
  end subroutine pdfnorm_get_type
```

⟨*Muli remnant: pdfnorm: TBP*⟩+≡

```
  procedure, nopass :: verify_type => pdfnorm_verify_type
```

⟨*Muli remnant: procedures*⟩+≡

```
  elemental logical function pdfnorm_verify_type (type) result (match)
    character(*), intent(in) :: type
    match = type == "pdfnorm_t"
  end function pdfnorm_verify_type
```

⟨*Muli remnant: pdfnorm: TBP*⟩+≡

```
  procedure :: scan => pdfnorm_scan
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine pdfnorm_scan (this)
  class(pdfnorm_t), intent(out) :: this
  integer :: ix, iq
  real(double) :: xmin, xmax, dx
  real(double) :: q, q2min, q2max
  real(double), dimension(-6:6) :: f
  real(double), dimension(0:2) :: x
  call getxmin (0, xmin)
  call getxmax (0, xmax)
  call getq2min (0, q2min)
  call getq2max (0, q2max)
  this%qmin = sqrt(sqrt(q2min))
  this%qmax = sqrt(sqrt(q2max))
  this%dq = (this%qmax - this%qmin) / nq
  xmin = sqrt(xmin)
  xmax = sqrt(xmax)
  dx= (xmax - xmin) / nx
  do iq = 0, nq
     print *, "iq=", iq, "/", nq
     q = (this%qmin + iq * this%dq)**2
     x(0) = xmin**2
     x(1) = (xmin+dx)**2
     call evolvePDF (x(0), q, f)
     f(1) = f(1) - f(-1)
     f(2) = f(2) - f(-2)
     this%pdf_int(:,iq) = (x(1) - x(0)) * f
     do ix = 2, nx
        x(2) = (xmin + ix*dx)**2
        call evolvePDF (x(1), q, f)
        f(1) = f(1) - f(-1)
        f(2) = f(2) - f(-2)
        this%pdf_int(:,iq) = this%pdf_int(:,iq) + f*(x(2) - x(0))
        x(0) = x(1)
        x(1) = x(2)
     end do
     call evolvePDF (x(1), q, f)
     f(1) = f(1) - f(-1)
     f(2) = f(2) - f(-2)
     this%pdf_int(:,iq) = (this%pdf_int(:,iq) + f*(x(1)-x(0))) / two
     this%pdf_norm(4,iq) = this%pdf_int(2,iq)
     this%pdf_norm(3,iq) = this%pdf_int(1,iq)
     this%pdf_int(2,iq) = this%pdf_int(2,iq) + this%pdf_int(-2,iq)
     this%pdf_int(1,iq) = this%pdf_int(1,iq) + this%pdf_int(-1,iq)
     this%pdf_norm(1,iq) = this%pdf_int(0,iq)
     this%pdf_norm(2,iq) = sum (this%pdf_int(-6:-1,iq)) + &
          sum(this%pdf_int(-2:-1,iq)) + sum(this%pdf_int(3:6,iq))
     this%pdf_norm(0,iq) = sum(this%pdf_int(:,iq))
     this%pdf_norm(1,iq) = this%pdf_norm(1,iq) / this%pdf_norm(0,iq)
     this%pdf_norm(2,iq) = this%pdf_norm(2,iq) / this%pdf_norm(0,iq)
     this%pdf_norm(3,iq) = this%pdf_norm(3,iq) / this%pdf_norm(0,iq)
     this%pdf_norm(4,iq) = this%pdf_norm(4,iq) / this%pdf_norm(0,iq)
     ! print *, this%pdf_norm(0,iq) - one
  end do
```

```
end subroutine pdfnorm_scan
```

⟨*Muli remnant: pdfnorm: TBP*⟩+≡

```
procedure :: get_norm => pdfnorm_get_norm
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine pdfnorm_get_norm (this, gev_q, dim, kind, norm)
  class(pdfnorm_t), intent(in) :: this
  real(default), intent(in) :: gev_q
  integer, intent(in) :: dim, kind
  real(default), intent(out)::norm
  integer :: iq
  real(default) :: x,q , z0, z1, z2, z3, z4
  norm = -one
  q = sqrt(gev_q) - this%qmin
  iq = floor(q / this%dq)
  x = q / this%dq - iq
  if (iq < 0) then
     print *, "pdfnorm_getnorm: q < q_min ", gev_q, this%qmin**2
     norm = this%pdf_norm (kind, 0)
  else
     if (iq >= nq) then
        print *, "pdfnorm_getnorm: q >= q_max ", gev_q, this%qmax**2
        norm = this%pdf_norm (kind, nq)
     else
        select case (dim)
        case (0)
           norm = this%pdf_norm (kind, iq)
        case (1)
           norm = this%pdf_norm(kind,iq) * (one - x) + &
                this%pdf_norm(kind,iq+1) * x
        case (2)
           x = x + mod(iq,2)
           iq = iq - mod(iq,2)
           z0 = this%pdf_norm(kind, iq)
           z1 = this%pdf_norm(kind, iq+1)
           z2 = this%pdf_norm(kind, iq+2)
           norm = ((z0 - 2D0*z1 + z2) * x - (three*z0 - four*z1 + z2)) * &
                x / two + z0
        case (3)
           x = x + mod(iq,3)
           iq = iq - mod(iq,3)
           z0 = this%pdf_norm(kind, iq)
           z1 = this%pdf_norm(kind, iq+1)
           z2 = this%pdf_norm(kind, iq+2)
           z3 = this%pdf_norm(kind, iq+3)
           norm = (( - (z0 - 3*z1 + 3*z2 -z3) * x + 3 * (2*z0 - &
                5*z1 + 4*z2 - z3))*x - (11*z0 - 18*z1 + 9*z2 - 2*z3)) * &
                x / 6._default + z0
        case (4)
           x = x + mod(iq,4)
           iq = iq - mod(iq,4)
           z0 = this%pdf_norm(kind, iq)
           z1 = this%pdf_norm(kind, iq+1)
```

```
            z2 = this%pdf_norm(kind, iq+2)
            z3 = this%pdf_norm(kind, iq+3)
            z4 = this%pdf_norm(kind, iq+4)
            norm = (((((z0 - 4*z1 + 6*z2 - 4*z3 + z4) * x &
                 -2 * (5*z0 - 18*z1 + 24*z2 - 14*z3 + 3*z4)) * x &
                 + (35*z0 - 104*z1 + 114*z2 - 56*z3 + 11*z4)) * x &
                 -2 * (25*z0 - 48*z1 + 36*z2 - 16*z3 + 3*z4)) * x) / &
                 24._default + z0
         case default
            norm = this%pdf_norm(kind, iq) * (one - x) + &
                 this%pdf_norm(kind, iq+1) * x
         end select
         ! print *, iq, x, norm
      end if
   end if
end subroutine pdfnorm_get_norm
```

⟨*Muli remnant: variables*⟩≡

```
integer, parameter :: nx = 10000000
```

⟨*Muli remnant: variables*⟩+≡

```
integer, parameter :: nq = 60
```

⟨*Muli remnant: public*⟩+≡

```
public :: remnant_weight_model
```

⟨*Muli remnant: variables*⟩+≡

```
integer :: remnant_weight_model = 2
```

⟨*Muli remnant: variables*⟩+≡

```
integer :: gluon_exp = 4
```

⟨*Muli remnant: public*⟩+≡

```
public :: muli_parton_t
```

⟨*Muli remnant: types*⟩+≡

```
type, extends (ser_class_t) :: muli_parton_t
   private
   integer :: id = -1
   integer :: lha_flavor
   real(default) :: momentum = -1
   class(muli_parton_t), pointer :: twin => null()
   class(muli_parton_t), pointer :: next => null()
 contains
 ⟨Muli remnant: muli parton: TBP⟩
end type muli_parton_t
```

⟨*Muli remnant: muli parton: TBP*⟩≡

```
procedure :: write_to_marker => parton_write_to_marker
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine parton_write_to_marker (this, marker, status)
  class(muli_parton_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%mark_begin ("muli_parton_t")
```

```
    call marker%mark ("id", this%id)
    call marker%mark ("lha", this%lha_flavor)
    call marker%mark ("momentum", this%momentum)
    call marker%mark_end ("muli_parton_t")
  end subroutine parton_write_to_marker
```

⟨*Muli remnant: muli parton: TBP*⟩+≡

```
  procedure :: read_from_marker => parton_read_from_marker
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine parton_read_from_marker (this, marker, status)
    class(muli_parton_t), intent(out) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    character(:), allocatable :: name
    call marker%pick_begin ("muli_parton_t", status=status)
    call marker%pick ("id", this%id, status)
    call marker%pick ("lha", this%lha_flavor, status)
    call marker%pick ("momentum", this%momentum, status)
    call marker%pick_end ("muli_parton_t", status=status)
  end subroutine parton_read_from_marker
```

⟨*Muli remnant: muli parton: TBP*⟩+≡

```
  procedure :: print_to_unit => parton_print_to_unit
```

⟨*Muli remnant: procedures*⟩+≡

```
  recursive subroutine parton_print_to_unit &
      (this, unit, parents, components, peers)
    class(muli_parton_t), intent(in) :: this
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    class(ser_class_t), pointer :: ser
    write (unit, "(1x,A)")      "Components of muli_parton_t:"
    write (unit, "(3x,A,I7)")   "id:         ", this%id
    write (unit, "(3x,A,I7)")   "lha flavor: ", this%lha_flavor
    write (unit, "(3x,A,F7.6)") "momentum:   ", this%momentum
    ser => this%next
    call serialize_print_peer_pointer &
         (ser, unit, parents, components, peers-i_one, "next")
    ser => this%twin
    call serialize_print_comp_pointer &
         (ser, unit, parents, components, peers-i_one, "twin")
  end subroutine parton_print_to_unit
```

⟨*Muli remnant: muli parton: TBP*⟩+≡

```
  procedure, nopass :: get_type => parton_get_type
```

⟨*Muli remnant: procedures*⟩+≡

```
  pure subroutine parton_get_type (type)
    character(:), allocatable, intent(out) :: type
    allocate (type, source="muli_parton_t")
  end subroutine parton_get_type
```

⟨*Muli remnant: muli parton: TBP*⟩+≡
```
procedure :: unweighted_pdf => twin_unweighted_pdf
```

⟨*Muli remnant: procedures*⟩+≡
```
pure function twin_unweighted_pdf (this, momentum_fraction) result (pdf)
  !parton pdf
  class(muli_parton_t), intent(in) :: this
  real(default), intent(in) :: momentum_fraction
  real(default) :: pdf
  if (momentum_fraction + this%twin%momentum < one) then
     pdf = remnant_twin_pdf_p (momentum_fraction, &
          this%twin%momentum, gluon_exp)
  else
     pdf = zero
  end if
end function twin_unweighted_pdf
```

⟨*Muli remnant: muli parton: TBP*⟩+≡
```
procedure :: deallocate => twin_deallocate
```

⟨*Muli remnant: procedures*⟩+≡
```
recursive subroutine twin_deallocate (this)
  class(muli_parton_t) :: this
  if (associated (this%next)) then
     call this%next%deallocate
     deallocate (this%next)
  end if
end subroutine twin_deallocate
```

⟨*Muli remnant: muli parton: TBP*⟩+≡
```
procedure :: push => parton_push
```

⟨*Muli remnant: procedures*⟩+≡
```
subroutine parton_push (this, parton)
  class(muli_parton_t), intent(inout) :: this
  class(muli_parton_t), intent(inout), pointer :: parton
  ! print *, "parton_push ", parton%id
  parton%next => this%next
  this%next => parton
end subroutine parton_push
```

⟨*Muli remnant: muli parton: TBP*⟩+≡
```
generic :: pop => pop_by_id, pop_by_association
procedure :: pop_by_id => parton_pop_by_id
procedure :: pop_by_association => parton_pop_by_association
```

⟨*Muli remnant: procedures*⟩+≡
```
subroutine parton_pop_by_id (this, id, parton)
  class(muli_parton_t), target, intent(inout) :: this
  integer, intent(in) :: id
  class(muli_parton_t), intent(out), pointer :: parton
  class(muli_parton_t), pointer :: tmp_parton
  tmp_parton => this
  do while (associated (tmp_parton%next))
```

```
      if (tmp_parton%next%id == id)  exit
      tmp_parton => tmp_parton%next
   end do
   if (associated (tmp_parton%next)) then
      parton => tmp_parton%next
      tmp_parton%next => parton%next
      nullify (parton%next)
      ! print *,"parton_pop ",id,parton%id
   else
      nullify (parton)
      print *,"parton_pop ", id, "NULL"
   end if
end subroutine parton_pop_by_id
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine parton_pop_by_association (this, parton)
   class(muli_parton_t), target, intent(inout) :: this
   class(muli_parton_t), intent(inout), target :: parton
   class(muli_parton_t), pointer :: tmp_parton
   tmp_parton => this
   do while (associated (tmp_parton%next))
      if (associated (tmp_parton%next, parton))  exit
      tmp_parton=>tmp_parton%next
   end do
   if (associated(tmp_parton%next)) then
      tmp_parton%next => parton%next
      nullify (parton%next)
      ! print *,"parton_pop ", parton%id
   else
      print *, "parton_pop NULL"
   end if
end subroutine parton_pop_by_association
```

⟨*Muli remnant: public*⟩+≡

```
public :: proton_remnant_t
```

⟨*Muli remnant: types*⟩+≡

```
type, extends (ser_class_t) :: proton_remnant_t
   private
   integer, dimension(2) :: valence_content = [1,2]
   integer :: n_twins = 0
   !!! [gluon, sea quark, valence down, valence up, twin]
   real(default), dimension(5) :: pdf_int_weight = [one, one, one, one, one]
   real(default) :: momentum_fraction = one
   real(default) :: twin_norm = one
   type(muli_parton_t) :: twin_partons
   type(muli_parton_t) :: is_partons
   type(muli_parton_t) :: fs_partons
   !!! These pointers shall not be allocated, deallocated,
   !!!    serialized or deserialized explicitly.
   class(pdfnorm_t), pointer :: pdf_norm => null()
 contains
 ⟨Muli remnant: proton remnant: TBP⟩
end type proton_remnant_t
```

⟨*Muli remnant: proton remnant: TBP*⟩≡

```
procedure :: remove_valence_quark => proton_remnant_remove_valence_quark
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_remove_valence_quark &
     (this, id, GeV_scale, momentum_fraction, lha_flavor)
  class(proton_remnant_t), intent(inout) :: this
  integer, intent(in) :: id
  real(default), intent(in) :: GeV_scale, momentum_fraction
  integer, intent(in) :: lha_flavor  !!! d=1, u=2
  if (lha_flavor == 1 .or. lha_flavor == 2) then
     associate (q => this%valence_content (lha_flavor))
       if (q > 0) then
          q = q - 1
          call this%push_is_parton (id, lha_flavor, momentum_fraction)
          this%momentum_fraction = this%momentum_fraction * (one - momentum_fraction)
          call this%calculate_weight (GeV_scale)
       else
          write (*, "(1x,A,I2,A)")  "proton_remnant_remove_valence_quark: " // &
               "Cannot remove parton ", lha_flavor, ": There are no such " // &
               "partons left."
          call this%print_all ()
       end if
     end associate
  else
     write (*, "(1x,A,I2,A)")  "proton_remnant_remove_valence_quark: Cannot " // &
          "remove parton ", lha_flavor, ": There are no such valence partons."
  end if
end subroutine proton_remnant_remove_valence_quark
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: remove_sea_quark => proton_remnant_remove_sea_quark
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_remove_sea_quark &
     (this, id, GeV_scale, momentum_fraction, lha_flavor)
  class(proton_remnant_t), intent(inout) :: this
  integer, intent(in) :: id
  real(default), intent(in) :: GeV_scale, momentum_fraction
  integer, intent(in) :: lha_flavor
  ! print *, "proton_remnant_remove_sea_quark", momentum_fraction
  if (lha_flavor > -6 .and. lha_flavor < 6 .and. lha_flavor .ne. 0) then
     this%momentum_fraction = this%momentum_fraction * &
          (one - momentum_fraction)
     call this%push_twin (id, lha_flavor, momentum_fraction, GeV_scale)
  end if
end subroutine proton_remnant_remove_sea_quark
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: remove_gluon => proton_remnant_remove_gluon
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_remove_gluon &
     (this, id, GeV_scale, momentum_fraction)
  class(proton_remnant_t), intent(inout) :: this
  integer, intent(in) :: id
  real(default), intent(in) :: GeV_scale, momentum_fraction
  this%momentum_fraction = this%momentum_fraction * (one - momentum_fraction)
  call this%push_is_parton (id, LHA_FLAVOR_g, momentum_fraction)
end subroutine proton_remnant_remove_gluon
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: remove_valence_up_quark => proton_remnant_remove_valence_up_quark
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_remove_valence_up_quark &
     (this, id, GeV_scale, momentum_fraction)
  class(proton_remnant_t), intent(inout) :: this
  integer, intent(in) :: id
  real(default), intent(in) :: GeV_scale, momentum_fraction
  associate (q => this%valence_content (LHA_FLAVOR_u))
    if (q > 0) then
       q = q - 1
       call this%push_is_parton (id, LHA_FLAVOR_u, momentum_fraction)
       this%momentum_fraction = this%momentum_fraction * (one - momentum_fraction)
       call this%calculate_weight (GeV_scale)
    else
       write (*, "(1x,A,I2,A)")  "proton_remnant_remove_valence_up_quark: " // &
            "Cannot remove parton ", LHA_FLAVOR_u, ": There are no such " // &
            "partons left."
       call this%print_all
    end if
  end associate
end subroutine proton_remnant_remove_valence_up_quark
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: remove_valence_down_quark => &
     proton_remnant_remove_valence_down_quark
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_remove_valence_down_quark &
     (this, id, GeV_scale, momentum_fraction)
  class(proton_remnant_t), intent(inout) :: this
  integer, intent(in) :: id
  real(default), intent(in) :: GeV_scale, momentum_fraction
  associate (q => this%valence_content(LHA_FLAVOR_d))
    if (q > 0) then
       q = q - 1
       call this%push_is_parton (id, LHA_FLAVOR_d, momentum_fraction)
       this%momentum_fraction = this%momentum_fraction * &
            (one - momentum_fraction)
       call this%calculate_weight (GeV_scale)
    else
       write (*, "(1x,A,I2,A)")  "proton_remnant_remove_valence_down_quark:" // &
            "Cannot remove parton ", LHA_FLAVOR_d, ": There are no " // &
```

```
            "such partons left."
        call this%print_all
      end if
    end associate
  end subroutine proton_remnant_remove_valence_down_quark
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
  procedure :: remove_twin => proton_remnant_remove_twin
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine proton_remnant_remove_twin (this, id, GeV_scale)
    class(proton_remnant_t), intent(inout) :: this
    integer, intent(in) :: id
    real(default), intent(in) :: GeV_scale
    class(muli_parton_t), pointer :: twin
    call this%twin_partons%pop (id, twin)
    call this%fs_partons%push (twin)
    this%twin_norm = this%twin_norm - twin%momentum
    this%n_twins = this%n_twins - 1
    call this%calculate_weight (GeV_scale)
  end subroutine proton_remnant_remove_twin
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
  procedure :: momentum_twin_pdf => proton_remnant_momentum_twin_pdf
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine proton_remnant_momentum_twin_pdf &
      (this, lha_flavor, momentum_fraction,pdf)
    class(proton_remnant_t), intent(in) :: this
    integer, intent(in) :: lha_flavor
    real(default), intent(in) :: momentum_fraction
    real(default), intent(out) :: pdf
    call this%parton_twin_pdf (lha_flavor, momentum_fraction, pdf)
    pdf = pdf * momentum_fraction
  end subroutine proton_remnant_momentum_twin_pdf
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
  procedure :: momentum_twin_pdf_array => proton_remnant_momentum_twin_pdf_array
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine proton_remnant_momentum_twin_pdf_array &
      (this, momentum_fraction, pdf)
    class(proton_remnant_t), intent(in) :: this
    real(default), intent(in) :: momentum_fraction
    real(default), dimension(this%n_twins), intent(out) :: pdf
    call this%parton_twin_pdf_array (momentum_fraction, pdf)
    pdf = pdf * momentum_fraction
  end subroutine proton_remnant_momentum_twin_pdf_array
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
  procedure :: momentum_kind_pdf => proton_remnant_momentum_kind_pdf
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_momentum_kind_pdf (this, GeV_scale, &
     momentum_fraction, lha_flavor, valence_pdf, sea_pdf, twin_pdf)
  class(proton_remnant_t), intent(in) :: this
  real(default), intent(in) :: GeV_scale, momentum_fraction
  integer, intent(in) :: lha_flavor              !!! g, u, d, etc.
  real(default), intent(out) :: valence_pdf, sea_pdf, twin_pdf
  real(double), dimension(-6:6) :: pdf_array
  call evolvePDF (momentum_fraction, GeV_scale, pdf_array)
  select case (lha_flavor)
  case (0) !!! gluon
     valence_pdf = zero
     sea_pdf=pdf_array(0)
  case (1) !!! down
     valence_pdf = this%get_valence_down_weight() * (pdf_array(1) - pdf_array(-1))
     sea_pdf = pdf_array(-1)
  case (2) !!! up
     valence_pdf = this%get_valence_up_weight() * (pdf_array(2) - pdf_array(-2))
     sea_pdf = pdf_array(-2)
  case default
     valence_pdf = zero
     sea_pdf = pdf_array (lha_flavor)
  end select
  sea_pdf = sea_pdf * this%get_sea_weight()
  call this%momentum_twin_pdf (lha_flavor, momentum_fraction, twin_pdf)
end subroutine proton_remnant_momentum_kind_pdf
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: momentum_flavor_pdf => proton_remnant_momentum_flavor_pdf
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_momentum_flavor_pdf (this, GeV_scale, &
     momentum_fraction, lha_flavor, pdf)
  class(proton_remnant_t), intent(in) :: this
  real(default), intent(in) :: GeV_scale, momentum_fraction
  integer, intent(in) :: lha_flavor     !!! g, u, d, etc.
  real(default), intent(out) :: pdf
  real(default) :: valence_pdf, sea_pdf, twin_pdf
  call this%momentum_kind_pdf (GeV_scale, momentum_fraction, &
       lha_flavor, valence_pdf, sea_pdf, twin_pdf)
  pdf = valence_pdf + sea_pdf + twin_pdf
end subroutine proton_remnant_momentum_flavor_pdf
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: momentum_kind_pdf_array => &
     proton_remnant_momentum_kind_pdf_array
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_momentum_kind_pdf_array (this, GeV_scale, &
     momentum_fraction, valence_pdf, sea_pdf)
  class(proton_remnant_t), intent(in) :: this
  real(default), intent(in) :: GeV_scale, momentum_fraction
  real(default), dimension(2), intent(out) :: valence_pdf
```

```
    real(double), dimension(-6:6), intent(out) :: sea_pdf
    call evolvePDF (momentum_fraction, GeV_scale, sea_pdf)
    valence_pdf(1) = (sea_pdf(1) - sea_pdf(-1)) * &
         this%pdf_int_weight (PDF_VALENCE_DOWN)
    valence_pdf(2) = (sea_pdf(2) - sea_pdf(-2)) * &
         this%pdf_int_weight (PDF_VALENCE_UP)
    sea_pdf(1) = sea_pdf(-1)
    sea_pdf(2) = sea_pdf(-2)
    sea_pdf = sea_pdf * this%get_sea_weight()
    !!! no twin yet
  end subroutine proton_remnant_momentum_kind_pdf_array
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
  procedure :: momentum_flavor_pdf_array => proton_remnant_momentum_flavor_pdf_array
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine proton_remnant_momentum_flavor_pdf_array &
       (this, GeV_scale, momentum_fraction, pdf)
    class(proton_remnant_t), intent(in) :: this
    real(default), intent(in) :: GeV_scale, momentum_fraction
    real(double), dimension(-6:6), intent(out) :: pdf
    real(default), dimension(2) :: valence_pdf
    call this%momentum_kind_pdf_array &
         (GeV_scale, momentum_fraction, valence_pdf, pdf)
    pdf(1:2) = pdf(1:2) + valence_pdf
    !!! no twin yet
  end subroutine proton_remnant_momentum_flavor_pdf_array
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
  procedure :: parton_twin_pdf => proton_remnant_parton_twin_pdf
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine proton_remnant_parton_twin_pdf &
       (this, lha_flavor, momentum_fraction, pdf)
    class(proton_remnant_t), intent(in) :: this
    integer, intent(in) :: lha_flavor
    real(default), intent(in) :: momentum_fraction
    real(default) :: pdf
    class(muli_parton_t), pointer :: tmp_twin
    pdf = 0D0
    tmp_twin => this%twin_partons%next
    do while (associated (tmp_twin))
       if (tmp_twin%lha_flavor == lha_flavor) pdf = pdf &
            + tmp_twin%unweighted_pdf (momentum_fraction)
       tmp_twin => tmp_twin%next
    end do
    pdf = pdf * this%get_twin_weight ()
  end subroutine proton_remnant_parton_twin_pdf
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
  procedure :: parton_twin_pdf_array => proton_remnant_parton_twin_pdf_array
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_parton_twin_pdf_array &
     (this, momentum_fraction, pdf)
  class(proton_remnant_t), intent(in) :: this
  real(default), intent(in) :: momentum_fraction
  real(default), dimension(this%n_twins), intent(out) :: pdf
  class(muli_parton_t), pointer :: tmp_twin
  integer :: l
  tmp_twin => this%twin_partons%next
  l = 0
  do while (associated (tmp_twin))
     l = l + 1
     pdf(l) = tmp_twin%unweighted_pdf (momentum_fraction) * this%twin_norm
     tmp_twin => tmp_twin%next
  end do
end subroutine proton_remnant_parton_twin_pdf_array
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: parton_kind_pdf => proton_remnant_parton_kind_pdf
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_parton_kind_pdf (this, GeV_scale, momentum_fraction, &
     lha_flavor, valence_pdf, sea_pdf, twin_pdf)
  class(proton_remnant_t), intent(in) :: this
  real(default), intent(in) :: GeV_scale, momentum_fraction
  integer, intent(in) :: lha_flavor              !!! g, u, d, etc.
  real(default), intent(out) :: valence_pdf, sea_pdf, twin_pdf
  call this%momentum_kind_pdf (GeV_scale, momentum_fraction, &
       lha_flavor, valence_pdf, sea_pdf, twin_pdf)
  valence_pdf = valence_pdf / momentum_fraction
  sea_pdf = sea_pdf / momentum_fraction
  twin_pdf = twin_pdf / momentum_fraction
end subroutine proton_remnant_parton_kind_pdf
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: parton_flavor_pdf => proton_remnant_parton_flavor_pdf
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_parton_flavor_pdf (this, GeV_scale, &
     momentum_fraction, lha_flavor, pdf)
  class(proton_remnant_t), intent(in) :: this
  real(default), intent(in) :: GeV_scale, momentum_fraction
  integer, intent(in) :: lha_flavor     !g,u,d,etc.
  real(default), intent(out) :: pdf
  call this%momentum_flavor_pdf (GeV_scale, momentum_fraction, &
       lha_flavor, pdf)
  pdf = pdf / momentum_fraction
end subroutine proton_remnant_parton_flavor_pdf
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: parton_kind_pdf_array => proton_remnant_parton_kind_pdf_array
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_parton_kind_pdf_array &
    (this, GeV_scale, momentum_fraction, valence_pdf, sea_pdf)
  class(proton_remnant_t), intent(in) :: this
  real(default), intent(in) :: GeV_scale, momentum_fraction
  real(default), dimension(2), intent(out) :: valence_pdf
  real(double), dimension(-6:6), intent(out) :: sea_pdf
  call evolvePDF (momentum_fraction, GeV_scale, sea_pdf)
  sea_pdf = sea_pdf / momentum_fraction
  valence_pdf(1) = (sea_pdf(1) - sea_pdf(-1)) * this%valence_content(1)
  valence_pdf(2) = (sea_pdf(2) - sea_pdf(-2)) * (this%valence_content(2) / two)
  sea_pdf(1) = sea_pdf(-1)
  sea_pdf(2) = sea_pdf(-2)
  valence_pdf = valence_pdf * this%get_valence_weight()
  sea_pdf = sea_pdf * this%get_sea_weight()
  !!! no twin yet
end subroutine proton_remnant_parton_kind_pdf_array
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: parton_flavor_pdf_array => proton_remnant_parton_flavor_pdf_array
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_parton_flavor_pdf_array &
    (this, GeV_scale, momentum_fraction, pdf)
  class(proton_remnant_t), intent(in) :: this
  real(default), intent(in) :: GeV_scale, momentum_fraction
  real(double), dimension(-6:6), intent(out) :: pdf
  real(double), dimension(2) :: valence_pdf
  real(double), dimension(-6:6) :: twin_pdf
  call msg_error ("proton_remnant_flavor_pdf_array: Not yet implemented.")
end subroutine proton_remnant_parton_flavor_pdf_array
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: get_pdf_int_weight => proton_remnant_get_pdf_int_weight
```

⟨*Muli remnant: procedures*⟩+≡

```
pure function proton_remnant_get_pdf_int_weight (this) result(weight)
  class(proton_remnant_t), intent(in) :: this
  real(default), dimension(5) :: weight
  weight = this%pdf_int_weight
end function proton_remnant_get_pdf_int_weight
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: get_valence_down_weight => proton_remnant_get_valence_down_weight
```

⟨*Muli remnant: procedures*⟩+≡

```
elemental function proton_remnant_get_valence_down_weight (this) result (weight)
  class(proton_remnant_t), intent(in) :: this
  real(default) :: weight
  weight = this%pdf_int_weight (PDF_VALENCE_DOWN)
end function proton_remnant_get_valence_down_weight
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: get_valence_up_weight => proton_remnant_get_valence_up_weight
```

⟨*Muli remnant: procedures*⟩+≡

```
elemental function proton_remnant_get_valence_up_weight (this) result (weight)
  class(proton_remnant_t), intent(in) :: this
  real(default) :: weight
  weight = this%pdf_int_weight (PDF_VALENCE_UP)
end function proton_remnant_get_valence_up_weight
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: get_valence_weight => proton_remnant_get_valence_weight
```

⟨*Muli remnant: procedures*⟩+≡

```
pure function proton_remnant_get_valence_weight (this) result (weight)
  class(proton_remnant_t), intent(in) :: this
  real(default), dimension(2) :: weight
  weight = this%pdf_int_weight(3:4)
end function proton_remnant_get_valence_weight
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: get_gluon_weight => proton_remnant_get_gluon_weight
```

⟨*Muli remnant: procedures*⟩+≡

```
elemental function proton_remnant_get_gluon_weight (this) result (weight)
  class(proton_remnant_t), intent(in) :: this
  real(default) :: weight
  weight = this%pdf_int_weight (PDF_GLUON)
end function proton_remnant_get_gluon_weight
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: get_sea_weight => proton_remnant_get_sea_weight
```

⟨*Muli remnant: procedures*⟩+≡

```
elemental function proton_remnant_get_sea_weight (this) result (weight)
  class(proton_remnant_t), intent(in) :: this
  real(default) :: weight
  weight = this%pdf_int_weight (PDF_SEA)
end function proton_remnant_get_sea_weight
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: get_twin_weight => proton_remnant_get_twin_weight
```

⟨*Muli remnant: procedures*⟩+≡

```
elemental function proton_remnant_get_twin_weight (this) result (weight)
  class(proton_remnant_t), intent(in) :: this
  real(default) :: weight
  weight = this%pdf_int_weight (PDF_TWIN)
end function proton_remnant_get_twin_weight
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: get_valence_content => proton_remnant_get_valence_content
```

⟨*Muli remnant: procedures*⟩+≡

```
pure function proton_remnant_get_valence_content (this) result (valence)
  class(proton_remnant_t), intent(in) :: this
  integer, dimension(2) :: valence
  valence = this%valence_content
end function proton_remnant_get_valence_content
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: get_momentum_fraction => proton_remnant_get_momentum_fraction
```

⟨*Muli remnant: procedures*⟩+≡

```
elemental function proton_remnant_get_momentum_fraction (this) result (momentum)
  class(proton_remnant_t), intent(in) :: this
  real(default) :: momentum
  momentum = this%momentum_fraction
end function proton_remnant_get_momentum_fraction
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: deallocate => proton_remnant_deallocate
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_deallocate (this)
  class(proton_remnant_t), intent(inout) :: this
  call this%is_partons%deallocate
  call this%fs_partons%deallocate
  call this%twin_partons%deallocate
  this%twin_norm = zero
  this%n_twins = 0
end subroutine proton_remnant_deallocate
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: initialize => proton_remnant_initialize
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_initialize (this, pdf_norm)
  class(proton_remnant_t), intent(out) :: this
  class(pdfnorm_t),target, intent(in) :: pdf_norm
  this%pdf_norm => pdf_norm
end subroutine proton_remnant_initialize
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: finalize => proton_remnant_finalize
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_finalize (this)
  class(proton_remnant_t), intent(inout) :: this
  call this%deallocate ()
  nullify (this%pdf_norm)
end subroutine proton_remnant_finalize
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: apply_initial_splitting => proton_remnant_apply_initial_splitting
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_apply_initial_splitting &
     (this, id, pdg_flavor, x, gev_scale, rnd)
  class(proton_remnant_t), intent(inout) :: this
  integer, intent(in) :: id, pdg_flavor
  real(default), intent(in) :: x, gev_scale, rnd
  real(default) :: valence_pdf, sea_pdf, twin_pdf
  select case (pdg_flavor)
  case (PDG_FLAVOR_g)
     call this%remove_gluon (id, gev_scale, x)
  case (PDG_FLAVOR_u)
     call this%parton_kind_pdf (gev_scale, x, pdg_flavor, &
          valence_pdf, sea_pdf, twin_pdf)
     if (valence_pdf / (valence_pdf + sea_pdf) < rnd) then
        call this%remove_sea_quark (id, gev_scale, x, pdg_flavor)
     else
        call this%remove_valence_up_quark (id, gev_scale, x)
     end if
  case (PDG_FLAVOR_d)
     call this%parton_kind_pdf (gev_scale, x, &
          pdg_flavor, valence_pdf, sea_pdf, twin_pdf)
     if (valence_pdf / (valence_pdf + sea_pdf) < rnd) then
        call this%remove_sea_quark (id, gev_scale, x, pdg_flavor)
     else
        call this%remove_valence_down_quark (id, gev_scale, x)
     end if
  case default
     call this%remove_sea_quark (id, gev_scale, x, pdg_flavor)
  end select
  this%momentum_fraction = (one - x)
end subroutine proton_remnant_apply_initial_splitting
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: reset => proton_remnant_reset
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_reset (this)
  class(proton_remnant_t), intent(inout) :: this
  call this%deallocate ()
  this%valence_content = [1, 2]
  this%pdf_int_weight = [one, one, one, one, one]
  this%momentum_fraction = one
end subroutine proton_remnant_reset
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: calculate_weight => proton_remnant_calculate_weight
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_calculate_weight (this, GeV_scale)
  class(proton_remnant_t), intent(inout) :: this
  real(default), intent(in) :: GeV_scale
  real(default) :: all, gluon, sea, vu, vd, valence, twin, weight
  call this%pdf_norm%get_norm (GeV_scale, 1, 0, all)
  call this%pdf_norm%get_norm (GeV_scale, 1, PDF_GLUON, gluon)
```

```
    call this%pdf_norm%get_norm (GeV_scale, 1, PDF_SEA, sea)
    call this%pdf_norm%get_norm (GeV_scale, 1, PDF_VALENCE_DOWN, vd)
    call this%pdf_norm%get_norm (GeV_scale, 1, PDF_VALENCE_UP, vu)
    valence = vd * this%valence_content (LHA_FLAVOR_d) + &
              vu * this%valence_content (LHA_FLAVOR_u) / two
    twin = this%twin_norm / all
    ! print *, all, gluon + sea + valence + twin, gluon, sea, valence, twin
    ! pdf_rescale = (one - n_d_valence * mean_d1 - n_u_valence * mean_u2) / &
    !     (1.E-1_default * mean_d1 - two * mean_u2) !!! pythia
    select case (remnant_weight_model)
    case (0) !!! no reweighting
       this%pdf_int_weight = [one, one, one, one, one]
    case (2) !!! pythia-like, only sea
       weight = (one - valence - twin) / (sea + gluon)
       this%pdf_int_weight = [weight, weight, one, one, one]
    case (3) !!! only valence and twin
       weight = (one - sea - gluon) / (valence + twin)
       this%pdf_int_weight = [one, one, weight, weight, weight]
    case (4) !!! only sea and twin
       weight = (one - valence) / (sea + gluon + twin)
       this%pdf_int_weight = [one, weight, one, one, weight]
    case default !!! equal weight
       weight = one / (valence + sea + gluon + twin)
       this%pdf_int_weight = [weight, weight, weight, weight, weight]
    end select
    this%pdf_int_weight(PDF_VALENCE_DOWN) = &
         this%pdf_int_weight(PDF_VALENCE_DOWN) * this%valence_content(1)
    this%pdf_int_weight(PDF_VALENCE_UP) = &
         this%pdf_int_weight(PDF_VALENCE_UP) * this%valence_content(2) * &
         5.E-1_default
    ! print('("New rescale factors are: ",2(I10),7(E14.7))'),&
    !         this%valence_content,&
    !         this%pdf_int_weight,&
    !         sea_norm,&
    !         valence_norm,&
    !         this%twin_norm
  end subroutine proton_remnant_calculate_weight
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
  procedure :: push_is_parton => proton_remnant_push_is_parton
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine proton_remnant_push_is_parton &
      (this, id, lha_flavor, momentum_fraction)
    class(proton_remnant_t), intent(inout) :: this
    integer, intent(in) :: id, lha_flavor
    real(default), intent(in) :: momentum_fraction
    class(muli_parton_t), pointer :: tmp_parton
    allocate (tmp_parton)
    tmp_parton%id = id
    tmp_parton%lha_flavor = lha_flavor
    tmp_parton%momentum = momentum_fraction
    call this%is_partons%push (tmp_parton)
  end subroutine proton_remnant_push_is_parton
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: push_twin => proton_remnant_push_twin
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_push_twin &
     (this, id, lha_flavor, momentum_fraction, gev_scale)
  class(proton_remnant_t), intent(inout) :: this
  integer, intent(in) :: id, lha_flavor !!! of IS parton
  real(default), intent(in) :: momentum_fraction !!! of IS parton
  real(default), intent(in) :: GeV_scale
  class(muli_parton_t), pointer :: new_is, new_twin
  real(default) :: norm
  ! print *, "proton_remnant_push_twin", momentum_fraction
  allocate (new_is)
  allocate (new_twin)
  !!! IS initialization
  new_is%id = id
  new_is%lha_flavor = lha_flavor
  new_is%momentum = momentum_fraction
  new_is%twin => new_twin
  !!! twin initialization
  new_twin%id = -id
  new_twin%lha_flavor = -lha_flavor
  new_twin%momentum = remnant_twin_momentum_4 (momentum_fraction)
  new_twin%twin => new_is
  !!! remnant update
  this%n_twins = this%n_twins + 1
  this%twin_norm = this%twin_norm + new_twin%momentum
  call this%is_partons%push (new_is)
  call this%twin_partons%push (new_twin)
  call this%calculate_weight (GeV_scale)
end subroutine proton_remnant_push_twin
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: calculate_twin_norm => proton_remnant_calculate_twin_norm
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_calculate_twin_norm (this)
  class(proton_remnant_t), intent(inout) :: this
  class(muli_parton_t), pointer :: twin
  integer :: n
  if (associated (this%twin_partons%next)) then
     this%twin_norm = zero
     twin => this%twin_partons%next
     do while (associated (twin))
        this%twin_norm = this%twin_norm + twin%momentum
        twin => twin%next
     end do
  else
     this%twin_norm = zero
  end if
end subroutine proton_remnant_calculate_twin_norm
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: replace_is_parton => proton_remnant_replace_is_parton
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_replace_is_parton &
     (this, old_id, new_id, pdg_f, x_proton, gev_scale)
  class(proton_remnant_t), intent(inout) :: this
  integer, intent(in) :: old_id, new_id, pdg_f
  real(default), intent(in) :: x_proton, gev_scale
  class(muli_parton_t), pointer :: old_is_parton
  integer :: lha_flavor
  real(default) :: momentum_fraction
  momentum_fraction = x_proton / this%momentum_fraction
  !!! convert PDG flavor numbers to LHA flavor numbers
  if (pdg_f == PDG_FLAVOR_g) then
     lha_flavor = LHA_FLAVOR_g
  else
     lha_flavor = pdg_f
  end if
  !!! we remove the old initial state parton from initial state stack.
  call this%is_partons%pop (old_id, old_is_parton)
  !!! this check has no physical meaning, it's just a check for consistency.
  if (associated (old_is_parton)) then
     !!! do we emit a gluon?
     if (lha_flavor == old_is_parton%lha_flavor) then
        !!! has the old initial state parton been a sea quark?
        if (associated (old_is_parton%twin)) then
           !!! the connection of the old IS parton with its twin was
           !!!    provisional. We remove it now
           call this%twin_partons%pop (old_is_parton%twin)
           call this%fs_partons%push (old_is_parton%twin)
           this%n_twins = this%n_twins - 1
           !!! and generate a new initial state parton - twin pair.
           call this%push_twin &
                (new_id, lha_flavor, momentum_fraction, gev_scale)
        else
           !!! there is no twin, so we just insert the new initial state parton.
           call this%push_is_parton (new_id, lha_flavor, momentum_fraction)
        end if
     else
        ! we emit a quark. is this a g->qqbar splitting?
        if (lha_flavor==LHA_FLAVOR_g) then
           !!! we insert the new initial state gloun.
           call this%push_is_parton (new_id, lha_flavor, momentum_fraction)
           !!! has the old initial state quark got a twin?
           if (associated (old_is_parton%twin)) then
              !!! we assume that this twin is the second splitting particle.
              !!! so the twin becomes a final state particle now and
              !!! must be removed from the is stack.
              call this%remove_twin (-old_id, GeV_scale)
           else
              !!! the old initial state quark has been a valence quark.
              !!! what should we do now? is this splitting sensible at all?
              !!! we don't know but allow these splittings. The most trivial
              !!! treatment is to restore the former valence quark.
```

```
            this%valence_content (old_is_parton%lha_flavor) = &
                 this%valence_content (old_is_parton%lha_flavor) + 1
         end if
      else
         !!! this is a q->qg splitting. the new initial state quark emits
         !!! the preceding initial state gluon. yeah, backward evolution is
         !!! confusing! the new initial state quark is not part of the
         !!! proton remnant any longer. how do we remove a quark from
         !!! the remnant? we add a conjugated twin parton and assume,
         !!! that this twin is created in a not yet resolved g->qqbar splitting.
         call this%push_twin (new_id, lha_flavor, momentum_fraction, gev_scale)
      end if
   end if
   !!! everything is done. what shall we do with the old initial state
   !!! parton? we don't need it any more but we store it anyway.
   call this%fs_partons%push (old_is_parton)
   !!! the new initial state parton has taken away momentum, so we update
   !!! the remnant momentum fraction.
   this%momentum_fraction = this%momentum_fraction * (1 - &
        momentum_fraction) / (1 - old_is_parton%momentum)
else
   !!! this is a bug.
   print *, "proton_remnant_replace_is_parton: parton #", old_id, &
        " not found on ISR stack."
   if (associated (this%is_partons%next)) then
      print *, "actual content of isr stack:"
      call this%is_partons%next%print_peers ()
   else
      print *, "ISR stack is not associated."
   end if
   stop
end if
end subroutine proton_remnant_replace_is_parton
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: write_to_marker => proton_remnant_write_to_marker
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_write_to_marker (this, marker, status)
  class(proton_remnant_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%mark_begin ("proton_remnant_t")
  call marker%mark ("valence_content", this%valence_content)
  call marker%mark ("momentum_fraction", this%momentum_fraction)
  call marker%mark ("pdf_int_weight", this%pdf_int_weight)
  call marker%mark_end ("proton_remnant_t")
end subroutine proton_remnant_write_to_marker
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: read_from_marker => proton_remnant_read_from_marker
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_read_from_marker (this, marker, status)
  class(proton_remnant_t), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  character(:), allocatable :: name
  call marker%pick_begin ("proton_remnant_t", status=status)
  call marker%pick ("valence_content", this%valence_content, status)
  call marker%pick ("momentum_fraction", this%momentum_fraction, status)
  call marker%pick ("pdf_int_weight", this%pdf_int_weight, status)
  call marker%pick_end ("proton_remnant_t", status=status)
end subroutine proton_remnant_read_from_marker
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: print_to_unit => proton_remnant_print_to_unit
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_print_to_unit (this, unit, parents, components, peers)
  class(proton_remnant_t), intent(in) :: this
  integer, intent(in) :: unit
  integer(dik), intent(in) :: parents, components, peers
  write (unit, '("Components of proton_remnant_t:")')
  write (unit, '("Valence Content:           ",I1,":",I1)')this&
       &%valence_content
  write (unit, "(1x,A,I1)") "N Twins:                   ", this%n_twins
  write (unit, "(1x,A,5(F7.3))") "INT weights [g,s,d,u,t]    ", &
       this%pdf_int_weight
  write (unit, "(1x,A,F7.3)") "Total Momentum Fraction:   ", &
       this%momentum_fraction
  write (unit, "(1x,A,F7.3)") "Twin Norm:            ", this%twin_norm
end subroutine proton_remnant_print_to_unit
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure, nopass :: get_type => proton_remnant_get_type
```

⟨*Muli remnant: procedures*⟩+≡

```
pure subroutine proton_remnant_get_type (type)
  character(:), allocatable, intent(out) :: type
  allocate (type, source="proton_remnant_t")
end subroutine proton_remnant_get_type
```

⟨*Muli remnant: proton remnant: TBP*⟩+≡

```
procedure :: gnuplot_momentum_kind_pdf_array => &
     proton_remnant_gnuplot_momentum_kind_pdf_array
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine proton_remnant_gnuplot_momentum_kind_pdf_array &
     (this, momentum_unit, parton_unit, GeV_scale)
  class(proton_remnant_t), intent(in) :: this
  integer, intent(in) :: momentum_unit, parton_unit
  real(default), intent(in) :: GeV_scale
  real(default), dimension(2) :: valence_pdf
  real(double), dimension(-6:6) :: sea_pdf
  real(default), dimension(this%n_twins) :: twin_pdf
  integer :: x
```

```
    real(default) :: momentum_fraction
    do x = 1, 100
       momentum_fraction = x * 1E-2_default
       call this%momentum_kind_pdf_array (GeV_scale, momentum_fraction, &
            valence_pdf, sea_pdf)
       call this%momentum_twin_pdf_array (momentum_fraction, twin_pdf)
       write (momentum_unit, fmt=*) momentum_fraction, &
            sum(valence_pdf) + sum(sea_pdf) + sum(twin_pdf), &
            valence_pdf, sea_pdf, twin_pdf
       call this%parton_kind_pdf_array (GeV_scale, momentum_fraction, &
            valence_pdf, sea_pdf)
       call this%parton_twin_pdf_array (momentum_fraction, twin_pdf)
       write(parton_unit,fmt=* )momentum_fraction, &
            sum(valence_pdf) + sum(sea_pdf) + sum(twin_pdf), &
            valence_pdf, sea_pdf, twin_pdf
    end do
  end subroutine proton_remnant_gnuplot_momentum_kind_pdf_array
```

⟨*Muli remnant: public*⟩+≡

```
  public :: pp_remnant_t
```

⟨*Muli remnant: types*⟩+≡

```
  type, extends (ser_class_t) :: pp_remnant_t
     logical :: initialized = .false.
     real(default), private :: gev_initial_cme = gev_cme_tot
     real(default), private :: X = one
     type(proton_remnant_t), dimension(2) :: proton
     class(pdfnorm_t), pointer :: pdf_norm
   contains
   ⟨Muli remnant: PP remnant: TBP⟩
  end type pp_remnant_t
```

⟨*Muli remnant: PP remnant: TBP*⟩≡

```
  procedure :: write_to_marker => pp_remnant_write_to_marker
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine pp_remnant_write_to_marker (this, marker, status)
    class(pp_remnant_t), intent(in) :: this
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    call marker%mark_begin ("pp_remnant_t")
    call marker%mark ("gev_initial_cme", this%gev_initial_cme)
    call marker%mark ("X", this%X)
    call this%proton(1)%write_to_marker (marker, status)
    call this%proton(2)%write_to_marker (marker, status)
    call marker%mark_end ("pp_remnant_t")
  end subroutine pp_remnant_write_to_marker
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: read_from_marker => pp_remnant_read_from_marker
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine pp_remnant_read_from_marker (this, marker, status)
    class(pp_remnant_t), intent(out) :: this
```

```
    class(marker_t), intent(inout) :: marker
    integer(dik), intent(out) :: status
    character(:), allocatable :: name
    call marker%pick_begin ("pp_remnant_t", status=status)
    call marker%pick ("gev_initial_cme", this%gev_initial_cme, status)
    call marker%pick ("X", this%X, status)
    call this%proton(1)%read_from_marker (marker, status)
    call this%proton(2)%read_from_marker (marker, status)
    call marker%pick_end ("pp_remnant_t", status=status)
  end subroutine pp_remnant_read_from_marker
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: print_to_unit => pp_remnant_print_to_unit
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine pp_remnant_print_to_unit (this, unit, parents, components, peers)
    class(pp_remnant_t), intent(in) :: this
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    write (unit, "(1x,A)")  "Components of pp_remnant_t:"
    write (unit, "(1x,A,F10.3)") "Initial center of mass energy: ", &
         this%gev_initial_cme
    write (unit, "(1x,A,F10.3)") "Actual center of mass energy:  ", &
         this%get_gev_actual_cme ()
    write (unit, "(1x,A,F10.3)") "Total Momentum Fraction is:    ", this%X
    if (components>0) then
       write (unit, "(3x,A)") "Proton 1:"
       call this%proton(1)%print_to_unit (unit, parents, components-1, peers)
       write (unit, "(3x,A)") "Proton 2:"
       call this%proton(2)%print_to_unit (unit, parents, components-1, peers)
    end if
    ! write (unit, "(1x,A,F7.2)") "Total Momentum Fraction:   ", &
    !    this%momentum_fraction
  end subroutine pp_remnant_print_to_unit
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure, nopass :: get_type => pp_remnant_get_type
```

⟨*Muli remnant: procedures*⟩+≡

```
  pure subroutine pp_remnant_get_type (type)
    character(:), allocatable, intent(out) :: type
    allocate (type, source="pp_remnant_t")
  end subroutine pp_remnant_get_type
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: initialize => pp_remnant_initialize
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine pp_remnant_initialize (this, muli_dir, &
       lhapdf_dir, lhapdf_file, lhapdf_member)
    class(pp_remnant_t), intent(out) :: this
    character(*), intent(in) :: muli_dir, lhapdf_dir, lhapdf_file
    integer, intent(in) :: lhapdf_member
    logical :: exist
```

```
    allocate (this%pdf_norm)
    ! call InitPDFset (lhapdf_dir // lhapdf_file)
    ! call InitPDF (lhapdf_member)
    print *, "looking for previously generated pdf integrals..."
    inquire (file=muli_dir // "/pdf_norm_" // lhapdf_file // &
         ".xml", exist=exist)
    if (exist) then
       print *, "found. Starting deserialization..."
       call this%pdf_norm%deserialize (name="pdf_norm_" // lhapdf_file, &
            file=muli_dir // "/pdf_norm_" // lhapdf_file // ".xml")
       print *, "done."
    else
       print *, "No integrals found. Starting generation..."
       call this%pdf_norm%scan ()
       print *, "done."
       call this%pdf_norm%serialize (name="pdf_norm_" // lhapdf_file, &
            file=muli_dir // "/pdf_norm_" // lhapdf_file // ".xml")
    end if
    call this%proton(1)%initialize (this%pdf_norm)
    call this%proton(2)%initialize (this%pdf_norm)
    this%initialized = .true.
    ! call this%print_all ()
  end subroutine pp_remnant_initialize
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: finalize => pp_remnant_finalize
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine pp_remnant_finalize (this)
    class(pp_remnant_t), intent(inout) :: this
    call this%proton(1)%finalize ()
    call this%proton(2)%finalize ()
    deallocate (this%pdf_norm)
  end subroutine pp_remnant_finalize
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: apply_initial_interaction => &
       pp_remnant_apply_initial_interaction
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine pp_remnant_apply_initial_interaction (this, gev_cme, x1, &
       x2, pdg_f1, pdg_f2, n1, n2, gev_scale, rnd1, rnd2)
    class(pp_remnant_t), intent(inout) :: this
    real(default), intent(in)::gev_cme,x1,x2,gev_scale,rnd1,rnd2
    integer, intent(in) :: pdg_f1, pdg_f2, n1, n2
    if (this%initialized) then
       call this%proton(1)%apply_initial_splitting &
            (n1, pdg_f1, x1, gev_scale, rnd1)
       call this%proton(2)%apply_initial_splitting &
            (n2, pdg_f2, x2, gev_scale, rnd2)
       this%X = (one - x1) * (one - x2)
       this%gev_initial_cme = gev_cme
       ! call this%print_all ()
    else
```

```
      call msg_error ("pp_remnant_apply_initial_interaction: Not yet " // &
           "initialized, call pp_remnant_initialize first!")
    end if
  end subroutine pp_remnant_apply_initial_interaction
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: reset => pp_remnant_reset
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine pp_remnant_reset (this)
    class(pp_remnant_t), intent(inout) :: this
    call this%proton(1)%reset ()
    call this%proton(2)%reset ()
    this%X = one
  end subroutine pp_remnant_reset
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: replace_parton => pp_remnant_replace_parton
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine pp_remnant_replace_parton (this, proton_id, old_id, &
      new_id, pdg_f, x_proton, gev_scale)
    class(pp_remnant_t), intent(inout) :: this
    integer, intent(in) :: proton_id, old_id, new_id, pdg_f
    real(default), intent(in) :: x_proton, gev_scale
    call this%proton(proton_id)%replace_is_parton &
        (old_id, new_id, pdg_f, x_proton, gev_scale)
  end subroutine pp_remnant_replace_parton
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: momentum_pdf => pp_remnant_momentum_pdf
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine pp_remnant_momentum_pdf &
      (this, x_proton, gev2_scale, n, pdg_f, pdf)
    class(pp_remnant_t), intent(in) :: this
    real(default), intent(in) :: x_proton, gev2_scale
    integer, intent(in) :: n, pdg_f
    real(default), intent(out) :: pdf
    if (n==1 .or. n==2) then
       if (x_proton <= this%proton(n)%momentum_fraction) then
          if (pdg_f == PDG_FLAVOR_g) then
             call this%proton(n)%momentum_flavor_pdf (sqrt(GeV2_scale), &
                  x_proton / this%proton(n)%momentum_fraction, &
                  LHA_FLAVOR_g, pdf)
          else
             call this%proton(n)%momentum_flavor_pdf (sqrt(GeV2_scale), &
                  x_proton / this%proton(n)%momentum_fraction, &
                  pdg_f, pdf)
          end if
          pdf = pdf * this%proton(n)%momentum_fraction
       else
          pdf = zero
       end if
```

```
    else
       call msg_error ("pp_remnant_momentum_pdf: n must be either 1 or 2, " &
            // "but it is " // char (str (n)) // ".")
       stop
    end if
  end subroutine pp_remnant_momentum_pdf
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: parton_pdf => pp_remnant_parton_pdf
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine pp_remnant_parton_pdf (this, x_proton, gev2_scale, n, pdg_f, pdf)
    class(pp_remnant_t), intent(in) :: this
    real(default), intent(in) :: x_proton, gev2_scale
    integer, intent(in) :: n, pdg_f
    real(default), intent(out) :: pdf
    if (n==1 .or. n==2) then
       if (x_proton <= this%proton(n)%momentum_fraction) then
          if (pdg_f == PDG_FLAVOR_g) then
             call this%proton(n)%parton_flavor_pdf (sqrt(GeV2_scale), &
                  x_proton/this%proton(n)%momentum_fraction, LHA_FLAVOR_g, &
                  pdf)
          else
             call this%proton(n)%parton_flavor_pdf (sqrt(GeV2_scale), &
                  x_proton/this%proton(n)%momentum_fraction,pdg_f, pdf)
          end if
          pdf = pdf * this%proton(n)%momentum_fraction
       else
          pdf = zero
       end if
    else
       call msg_error ("pp_remnant_parton_pdf: n must be either 1 or 2, " &
            // "but it is " // char (str (n)) // ".")
       stop
    end if
  end subroutine pp_remnant_parton_pdf
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: apply_interaction => pp_remnant_apply_interaction
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine pp_remnant_apply_interaction (this, qcd_2_2)
    class(pp_remnant_t), intent(inout) :: this
    class(qcd_2_2_class), intent(in) :: qcd_2_2
    integer, dimension(4) :: lha_f
    integer, dimension(2) :: int_k
    real(default) :: gev_pt
    real(default), dimension(2) :: mom_f
    integer :: n
    mom_f = qcd_2_2%get_remnant_momentum_fractions ()
    lha_f = qcd_2_2%get_lha_flavors ()
    int_k = qcd_2_2%get_pdf_int_kinds ()
    gev_pt = qcd_2_2%get_gev_scale ()
    ! print *,"pp_remnant_apply_interaction", mom_f, &
```

```
!    qcd_2_2%get_parton_id(1), qcd_2_2%get_parton_id(2), lha_f
do n = 1, 2
   select case (int_k(n))
   case (PDF_VALENCE_DOWN)
      call this%proton(n)%remove_valence_down_quark &
           (qcd_2_2%get_parton_id(n), gev_pt, mom_f(n))
   case (PDF_VALENCE_UP)
      call this%proton(n)%remove_valence_up_quark &
           (qcd_2_2%get_parton_id(n), gev_pt, mom_f(n))
   case (PDF_SEA)
      call this%proton(n)%remove_sea_quark &
           (qcd_2_2%get_parton_id(n), gev_pt, mom_f(n), lha_f(n))
   case (PDF_GLUON)
      call this%proton(n)%remove_gluon &
           (qcd_2_2%get_parton_id(n), gev_pt, mom_f(n))
   end select
end do
this%X = this%proton(1)%momentum_fraction * &
     this%proton(2)%momentum_fraction
end subroutine pp_remnant_apply_interaction
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
procedure :: get_pdf_int_weights => pp_remnant_get_pdf_int_weights
```

⟨*Muli remnant: procedures*⟩+≡

```
pure function pp_remnant_get_pdf_int_weights &
     (this, pdf_int_kinds) result (weight)
  class(pp_remnant_t), intent(in) :: this
  real(default) :: weight
  integer, dimension(2), intent(in) :: pdf_int_kinds ! pdf_int_kind
  weight = this%proton(1)%pdf_int_weight (pdf_int_kinds(1)) * &
       this%proton(2)%pdf_int_weight (pdf_int_kinds(2)) !*((this%x)**2)
end function pp_remnant_get_pdf_int_weights
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
procedure :: get_pdf_int_weight => pp_remnant_get_pdf_int_weight
```

⟨*Muli remnant: procedures*⟩+≡

```
elemental function pp_remnant_get_pdf_int_weight &
     (this, kind1, kind2) result (weight)
  class(pp_remnant_t), intent(in) :: this
  real(double) :: weight
  integer, intent(in) :: kind1, kind2 ! pdf_int_kind
  weight = this%proton(1)%pdf_int_weight(kind1) * &
       this%proton(2)%pdf_int_weight(kind2)     !*((this%x)**2)
end function pp_remnant_get_pdf_int_weight
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
procedure :: set_pdf_weight => pp_remnant_set_pdf_weight
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine pp_remnant_set_pdf_weight (this, weights)
  class(pp_remnant_t), intent(inout) :: this
  real(default), dimension(10), intent(in) :: weights
```

```
    this%proton(1)%pdf_int_weight = weights(1:5)
    this%proton(2)%pdf_int_weight = weights(6:10)
  end subroutine pp_remnant_set_pdf_weight
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: get_gev_initial_cme => pp_remnant_get_gev_initial_cme
```

⟨*Muli remnant: procedures*⟩+≡

```
  elemental function pp_remnant_get_gev_initial_cme (this) result (cme)
    class(pp_remnant_t), intent(in) :: this
    real(double) :: cme
    cme  =this%gev_initial_cme
  end function pp_remnant_get_gev_initial_cme
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: get_gev_actual_cme => pp_remnant_get_gev_actual_cme
```

⟨*Muli remnant: procedures*⟩+≡

```
  elemental function pp_remnant_get_gev_actual_cme (this) result (cme)
    class(pp_remnant_t), intent(in) :: this
    real(double) :: cme
    cme = this%gev_initial_cme * this%X
  end function pp_remnant_get_gev_actual_cme
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: get_cme_fraction => pp_remnant_get_cme_fraction
```

⟨*Muli remnant: procedures*⟩+≡

```
  elemental function pp_remnant_get_cme_fraction (this) result (cme)
    class(pp_remnant_t), intent(in) :: this
    real(double) :: cme
    cme = this%X
  end function pp_remnant_get_cme_fraction
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: get_proton_remnant_momentum_fractions => &
       pp_remnant_get_proton_remnant_momentum_fractions
```

⟨*Muli remnant: procedures*⟩+≡

```
  pure function pp_remnant_get_proton_remnant_momentum_fractions &
       (this) result (fractions)
    class(pp_remnant_t), intent(in) :: this
    real(double), dimension(2) :: fractions
    fractions = [this%proton(1)%get_momentum_fraction(), &
         this%proton(2)%get_momentum_fraction()]
  end function pp_remnant_get_proton_remnant_momentum_fractions
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
  procedure :: get_proton_remnants => pp_remnant_get_proton_remnants
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine pp_remnant_get_proton_remnants (this, proton1, proton2)
  class(pp_remnant_t), target, intent(in) :: this
  class(proton_remnant_t), intent(out), pointer :: proton1, proton2
  proton1 => this%proton(1)
  proton2 => this%proton(2)
end subroutine pp_remnant_get_proton_remnants
```

⟨*Muli remnant: PP remnant: TBP*⟩+≡

```
procedure :: get_remnant_parton_flavor_pdf_arrays => &
     pp_remnant_get_remnant_parton_flavor_pdf_arrays
```

⟨*Muli remnant: procedures*⟩+≡

```
subroutine pp_remnant_get_remnant_parton_flavor_pdf_arrays &
     (this, GeV_scale, momentum1, momentum2, pdf1, pdf2)
  class(pp_remnant_t), intent(in) :: this
  real(default), intent(in) :: GeV_scale, momentum1, momentum2
  real(double), dimension(-6:6), intent(out) :: pdf1, pdf2
  call this%proton(1)%parton_flavor_pdf_array (GeV_scale, momentum1, pdf1)
  call this%proton(2)%parton_flavor_pdf_array (GeV_scale, momentum2, pdf2)
end subroutine pp_remnant_get_remnant_parton_flavor_pdf_arrays
```

⟨*Muli remnant: interfaces*⟩≡

```
interface
   subroutine getxmin (mem, xmin)
     integer, intent(in) :: mem
     double precision, intent(out) :: xmin
   end subroutine getxmin
end interface
```

⟨*Muli remnant: interfaces*⟩+≡

```
interface
   subroutine getxmax (mem, xmax)
     integer, intent(in) :: mem
     double precision, intent(out) :: xmax
   end subroutine getxmax
end interface
```

⟨*Muli remnant: interfaces*⟩+≡

```
interface
   subroutine getq2min (mem, q2min)
     integer, intent(in) :: mem
     double precision, intent(out) :: q2min
   end subroutine getq2min
end interface
```

⟨*Muli remnant: interfaces*⟩+≡

```
interface
   subroutine getq2max (mem, q2max)
     integer, intent(in) :: mem
     double precision, intent(out) :: q2max
   end subroutine getq2max
end interface
```

⟨*Muli remnant: procedures*⟩+≡

```
pure function remnant_dglap_splitting_gqq (z) result(p)
  real(default) :: p
  real(default), intent(in) :: z
  p = (z**2 + (1-z)**2) / two
end function remnant_dglap_splitting_gqq
```

⟨*Muli remnant: procedures*⟩+≡

```
pure function remnant_gluon_pdf_approx (x, p) result (g)
  real(default) :: g
  integer, intent(in) :: p
  real(default), intent(in) :: x
  g = ((1-x)**p) / x
end function remnant_gluon_pdf_approx
```

⟨*Muli remnant: procedures*⟩+≡

```
pure function remnant_norm_0 (xs) result (c0)
  real(default) :: c0
  real(default), intent(in) :: xs
  c0 = 6*xs / (2 - xs * (3 - 3*xs + 2*xs**2))
end function remnant_norm_0
```

⟨*Muli remnant: procedures*⟩+≡

```
pure function remnant_norm_1 (xs) result (c1)
  real(default) :: c1
  real(default), intent(in)::xs
  c1 = 3*xs / (2 - xs**2 * (3-xs) + 3*xs*log(xs))
end function remnant_norm_1
```

⟨*Muli remnant: procedures*⟩+≡

```
pure function remnant_norm_4 (xs) result (c4)
  real(default) :: c4
  real(default), intent(in) :: xs
  real(default) :: y
  if ((one-xs) > 1E-3_default) then
     c4 = 3*xs / (1 + 11*xs + 6*xs*log(xs) + 12*xs**3*log(xs) + &
          18*xs**2*log(xs) + 9*xs**2 - 19*xs**3 - 2*xs**4)
  else
     y=one / (one-xs)
     c4= 1130._default / 11907._default &
          - 10._default * y**5 &
          - 40._default * y**4 / three &
          -160._default * y**3 / 63._default &
          + 50._default * y**2 / 189._default &
          -565._default * y / 3969._default &
          -186170._default * (one-xs) / 2750517._default
  end if
end function remnant_norm_4
```

⟨*Muli remnant: procedures*⟩+≡

```
pure function remnant_norm (xs, p) result (c)
  real(default) :: c
```

```
    real(default), intent(in) :: xs
    integer, intent(in) :: p
    select case (p)
    case (0)
       c = remnant_norm_0 (xs)
    case (1)
       c = remnant_norm_1 (xs)
    case default
       c = remnant_norm_4 (xs)
    end select
  end function remnant_norm
```

⟨*Muli remnant: procedures*⟩+≡

```
  pure function remnant_twin_pdf_p (x, xs, p) result (qc)
    real(default) :: qc
    real(default), intent(in) :: x, xs
    integer, intent(in) :: p
    qc = remnant_norm (xs, p) * remnant_gluon_pdf_approx (xs + x, p) &
         * remnant_dglap_splitting_gqq (xs / (xs+x)) / (xs+x)
  end function remnant_twin_pdf_p
```

⟨*Muli remnant: procedures*⟩+≡

```
  elemental function remnant_twin_momentum_4 (xs) result (p)
    real(default) :: p
    real(default), intent(in) :: xs
    if (xs < 0.99_default) then
       p = (-9 * (-1+xs) * xs * (1+xs) * (5+xs*(24+xs)) + &
            12*xs*(1+2*xs)*(1+2*xs*(5+2*xs))*Log(xs))/&
            (8*(1+2*xs)*((-1+xs)*(1+xs*(10+xs))-6*xs*(1+xs)*Log(xs)))
    else
       p = (1-xs) / 6 - (5*(-1+xs)**2) / 63 + (5*(-1+xs)**3) / 216
    end if
  end function remnant_twin_momentum_4
```

⟨*Muli remnant: procedures*⟩+≡

```
  subroutine gnuplot_integrated_pdf (this, momentum_unit, parton_unit)
    class(proton_remnant_t), intent(in) :: this
    integer, intent(in) :: momentum_unit, parton_unit
    ! real(double), intent(in) :: gev_scale
    integer, parameter :: x_grid = 1000000
    integer, parameter :: q_grid = 100
    integer :: n, m, mem
    real(default) :: x, q, dx, dq, overall_sum, xmin, xmax, &
         q2min, q2max, qmin, qmax
    real(double) :: q2min_dbl, q2max_dbl, xmin_dbl, xmax_dbl
    real(double), dimension(-6:6) :: sea_pdf, sea_momentum_pdf_sum, &
         sea_parton_pdf_sum
    real(default), dimension(2) :: valence_pdf, valence_momentum_pdf_sum,&
         valence_parton_pdf_sum
    real(default), allocatable, dimension(:) :: twin_momentum_pdf_sum
    class(muli_parton_t), pointer :: tmp_twin
    mem = 1
    call GetXmin (mem, xmin_dbl)
```

```
call GetXmax (mem, xmax_dbl)
call GetQ2max (mem, q2max_dbl)
call GetQ2min (mem, q2min_dbl)
xmin = xmin_dbl
xmax = xmax_dbl
q2min = q2min_dbl
q2max = q2max_dbl
qmin = sqrt(q2min)
qmax = sqrt(q2max)
print *, "qmin=", qmin, "GeV"
print *, "qmax=", qmax, "GeV"
dx = (xmax-xmin) / x_grid
dq = (qmax-qmin) / q_grid
q = qmin + dq / 2D0
tmp_twin => this%twin_partons%next
n = 0
if (this%n_twins > 0) then
   allocate (twin_momentum_pdf_sum (this%n_twins))
   do while (associated (tmp_twin))
      n = n + 1
      twin_momentum_pdf_sum(n) = tmp_twin%momentum
      tmp_twin => tmp_twin%next
   end do
end if
do m = 1, q_grid
   valence_momentum_pdf_sum = [0D0,0D0]
   valence_parton_pdf_sum = [0D0,0D0]
   sea_momentum_pdf_sum = &
        [0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0]
   sea_parton_pdf_sum = &
        [0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0]
   x = xmin + dx / 2D0
   do n = 1, x_grid
      call this%parton_kind_pdf_array (Q, x, valence_pdf, sea_pdf)
      valence_parton_pdf_sum = valence_parton_pdf_sum + valence_pdf
      sea_parton_pdf_sum = sea_parton_pdf_sum + sea_pdf
      call this%momentum_kind_pdf_array (Q, x, valence_pdf, sea_pdf)
      valence_momentum_pdf_sum = valence_momentum_pdf_sum + valence_pdf
      sea_momentum_pdf_sum = sea_momentum_pdf_sum + sea_pdf
      x = x + dx
   end do
   valence_parton_pdf_sum = valence_parton_pdf_sum * dx
   sea_parton_pdf_sum = sea_parton_pdf_sum * dx
   valence_momentum_pdf_sum = valence_momentum_pdf_sum * dx
   sea_momentum_pdf_sum = sea_momentum_pdf_sum * dx
   if (this%n_twins > 0) then
      write (momentum_unit, fmt=*) q, &
           sum(valence_momentum_pdf_sum) + &
           sum(sea_momentum_pdf_sum) + sum(twin_momentum_pdf_sum), &
           valence_momentum_pdf_sum, &
           sea_momentum_pdf_sum, &
           twin_momentum_pdf_sum
   else
      write (momentum_unit, fmt=*) q, &
```

```
            sum(valence_momentum_pdf_sum) + sum(sea_momentum_pdf_sum), &
            valence_momentum_pdf_sum, &
            sea_momentum_pdf_sum
      end if
      write (parton_unit,fmt=*) q, &
           sum(valence_parton_pdf_sum) + sum(sea_parton_pdf_sum), &
           valence_parton_pdf_sum, &
           sea_parton_pdf_sum
      q = q + dq
    end do
  end subroutine gnuplot_integrated_pdf
```

22.11 The Multiple Interactions main module

This file contains the module `muli` which is the multiple parton interactions interface module to `WHIZARD`. `muli` is supposed to run together with initial state radiation. Both share a momentum evolution variable and compete for beam momentum, so the generation of this scale variable must be fully transparent to `WHIZARD`. That is why the corresponding procedures are here, while all other work like phase space integration, flavor generation and treatment of the beam remnant is put into `muli_dsigma`, `muli_mcint` and `muli_remnant`, respectively.

`qcd_2_2_t` is a container class for properties of QCD $2 \to 2$ interactions. It holds a very condensed internal representation and offers a convenient set of TBP to query all aspects without the burden of the generator internals.

`muli_t` then is an extension of `qcd_2_2_t` that adds generator internals like random number generator, integrated cross-sections, the actual Monte Carlo generator for flavor generation and beam remnants in `tao_rnd`, `dsigma`, samples and beam, respectively.

⟨`muli.f90`⟩≡

```
⟨File header⟩

module muli
  use, intrinsic :: iso_fortran_env
⟨Use kinds⟩
  use constants
  use tao_random_numbers !NODEP!
  use muli_base
  use muli_momentum
  use muli_trapezium
  use muli_interactions
  use muli_dsigma
  use muli_mcint
  use muli_remnant

⟨Standard module head⟩

⟨Muli: variables⟩

⟨Muli: public⟩
```

```
⟨Muli: types⟩

contains

⟨Muli: procedures⟩

end module muli
```

⟨*Muli: variables*⟩≡

```
logical, parameter :: muli_default_modify_pdfs = .true.
integer, parameter :: muli_default_lhapdf_member = 0
character(*), parameter :: muli_default_lhapdf_file = "cteq6ll.LHpdf"
```

⟨*Muli: types*⟩≡

```
type, extends(qcd_2_2_class) :: qcd_2_2_t
   private
   integer :: process_id = -1
   integer :: integrand_id = -1
   integer, dimension(2) :: parton_ids = [0,0]
   integer, dimension(4) :: flow = [0,0,0,0]
   real(default), dimension(3) :: momentum_fractions = [-one, -one, -one]
   real(default), dimension(3) :: hyperbolic_fractions = [-one ,- one,- one]
 contains
 ⟨Muli: QCD22: TBP⟩
end type qcd_2_2_t
```

⟨*Muli: QCD22: TBP*⟩≡

```
procedure :: write_to_marker => qcd_2_2_write_to_marker
```

⟨*Muli: procedures*⟩≡

```
subroutine qcd_2_2_write_to_marker (this, marker, status)
  class(qcd_2_2_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%mark_begin ("qcd_2_2_t")
  call this%mom_write_to_marker (marker, status)
  call marker%mark ("process_id", this%process_id)
  call marker%mark ("integrand_id", this%integrand_id)
  call marker%mark ("momentum_fractions", this%momentum_fractions)
  call marker%mark ("hyperbolic_fractions", this%hyperbolic_fractions)
  call marker%mark_end("qcd_2_2_t")
end subroutine qcd_2_2_write_to_marker
```

⟨*Muli: QCD22: TBP*⟩+≡

```
procedure :: read_from_marker => qcd_2_2_read_from_marker
```

⟨*Muli: procedures*⟩+≡

```
subroutine qcd_2_2_read_from_marker (this, marker, status)
  class(qcd_2_2_t), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%pick_begin ("qcd_2_2_t", status=status)
  call this%mom_read_from_marker (marker, status)
  call marker%pick ("process_id", this%process_id, status)
```

```
    call marker%pick ("integrand_id", this%integrand_id, status)
    call marker%pick ("momentum_fractions", this%momentum_fractions, status)
    call marker%pick &
         ("hyperbolic_fractions", this%hyperbolic_fractions, status)
    call marker%pick_end ("qcd_2_2_t", status=status)
  end subroutine qcd_2_2_read_from_marker
```

⟨*Muli: QCD22: TBP*⟩+≡

```
  procedure :: print_to_unit => qcd_2_2_print_to_unit
```

⟨*Muli: procedures*⟩+≡

```
  subroutine qcd_2_2_print_to_unit (this, unit, parents, components, peers)
    class(qcd_2_2_t), intent(in) :: this
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    integer, dimension(2,4) :: flow
    integer :: index
    if (parents > i_zero) &
         call this%mom_print_to_unit (unit, parents-1, components, peers)
    write (unit, "(1x,A)")  "Components of qcd_2_2_t:"
    write (unit, "(3x,A,I3)")  "Process id is:       ", this%get_process_id ()
    write (unit, "(3x,A,I3)")  "Integrand id is:     ", this%get_integrand_id ()
    if (this%get_integrand_id () > 0) then
       write (unit, "(3x,A,4(I3))")  "LHA Flavors are:     ", &
            this%get_lha_flavors ()
       write (unit, "(3x,A,4(I3))")  "PDG Flavors are:     ", &
            this%get_pdg_flavors ()
       write (unit, "(3x,A,2(I3))")  "Parton kinds are:    ", &
            this%get_parton_kinds ()
       write (unit, "(3x,A,2(I3))")  "PDF int kinds are:   ", &
            this%get_pdf_int_kinds ()
       write (unit, "(3x,A,2(I3))")  "Diagram kind is:     ", &
            this%get_diagram_kind ()
    end if
    call this%get_color_correlations (1, index, flow)
    write (unit, "(3x,A,4(I0))")  "Color Permutations:  ", this%flow
    write (unit, "(3x,A)")  "Color Connections:"
    write (unit, &
         '("   (",I0,",",I0,")+(",I0,",",I0,")->(",I0,",",I0,&
         &")+(",I0,",",I0,")")') flow
    write (unit, "(3x,A,E14.7)")  "Evolution scale is:  ", &
         this%get_unit2_scale ()
    write (unit, "(3x,A,E14.7)")  "Momentum boost is:   ", &
         this%get_momentum_boost ()
    write (unit, "(3x,A,2(E14.7))")  "Remant momentum fractions are: ", &
         this%get_remnant_momentum_fractions ()
    write (unit, "(3x,A,2(E14.7))")  "Total momentum fractions are:  ", &
         this%get_total_momentum_fractions ()
  end subroutine qcd_2_2_print_to_unit
```

⟨*Muli: QCD22: TBP*⟩+≡

```
  procedure, nopass :: get_type => qcd_2_2_get_type
```

⟨*Muli: procedures*⟩+≡
```
pure subroutine qcd_2_2_get_type (type)
  character(:), allocatable, intent(out) :: type
  allocate (type, source="qcd_2_2_t")
end subroutine qcd_2_2_get_type
```

⟨*Muli: QCD22: TBP*⟩+≡
```
procedure :: get_process_id => qcd_2_2_get_process_id
```

⟨*Muli: procedures*⟩+≡
```
elemental function qcd_2_2_get_process_id (this) result (id)
  class(qcd_2_2_t), intent(in) :: this
  integer :: id
  id = this%process_id
end function qcd_2_2_get_process_id
```

⟨*Muli: QCD22: TBP*⟩+≡
```
procedure :: get_integrand_id => qcd_2_2_get_integrand_id
```

⟨*Muli: procedures*⟩+≡
```
elemental function qcd_2_2_get_integrand_id (this) result (id)
  class(qcd_2_2_t), intent(in) :: this
  integer :: id
  id = this%integrand_id
end function qcd_2_2_get_integrand_id
```

⟨*Muli: QCD22: TBP*⟩+≡
```
procedure :: get_diagram_kind => qcd_2_2_get_diagram_kind
```

⟨*Muli: procedures*⟩+≡
```
elemental function qcd_2_2_get_diagram_kind (this) result (kind)
  class(qcd_2_2_t), intent(in) :: this
  integer :: kind
  kind = valid_processes (6, this%process_id)
end function qcd_2_2_get_diagram_kind
```

This is one more hack. Before merging into the interleaved algorithm, `muli` has only cared for summed cross sections, but not in specific color flows. So two different diagrams with equal cross sections were summed up to diagram kind 1. Now `muli` also generates color flows, so we must devide diagram kind 1 into diagram color kind 0 and diagram color kind 1.

⟨*Muli: QCD22: TBP*⟩+≡
```
procedure :: get_diagram_color_kind => qcd_2_2_get_diagram_color_kind
```

⟨*Muli: procedures*⟩+≡
```
pure function qcd_2_2_get_diagram_color_kind (this) result (kind)
  class(qcd_2_2_t), intent(in) :: this
  integer :: kind
  kind = valid_processes (6, this%process_id)
  if (kind == 1) then
     if (product (valid_processes (1:2,this%process_id)) > 0) then
        kind = 0
     end if
```

```
    end if
  end function qcd_2_2_get_diagram_color_kind
```

⟨*Muli: QCD22: TBP*⟩+≡

```
  procedure :: get_io_kind => qcd_2_2_get_io_kind
```

⟨*Muli: procedures*⟩+≡

```
  elemental function qcd_2_2_get_io_kind (this) result (kind)
    class(qcd_2_2_t), intent(in) :: this
    integer :: kind
    kind = valid_processes (5, this%process_id)
  end function qcd_2_2_get_io_kind
```

⟨*Muli: QCD22: TBP*⟩+≡

```
  procedure :: get_lha_flavors => qcd_2_2_get_lha_flavors
```

⟨*Muli: procedures*⟩+≡

```
  pure function qcd_2_2_get_lha_flavors (this) result (lha)
    class(qcd_2_2_t), intent(in) :: this
    integer, dimension(4) :: lha
    lha = valid_processes (1:4, this%process_id)
  end function qcd_2_2_get_lha_flavors
```

⟨*Muli: QCD22: TBP*⟩+≡

```
  procedure :: get_pdg_flavors => qcd_2_2_get_pdg_flavors
```

⟨*Muli: procedures*⟩+≡

```
  pure function qcd_2_2_get_pdg_flavors (this) result (pdg)
    class(qcd_2_2_t), intent(in) :: this
    integer, dimension(4) :: pdg
    pdg = this%get_lha_flavors ()
    where (pdg == 0) pdg = 21
  end function qcd_2_2_get_pdg_flavors
```

⟨*Muli: QCD22: TBP*⟩+≡

```
  procedure :: get_parton_id => qcd_2_2_get_parton_id
```

⟨*Muli: procedures*⟩+≡

```
  pure function qcd_2_2_get_parton_id (this, n) result (id)
    class(qcd_2_2_t), intent(in) :: this
    integer, intent(in) :: n
    integer :: id
    id = this%parton_ids (n)
  end function qcd_2_2_get_parton_id
```

⟨*Muli: QCD22: TBP*⟩+≡

```
  procedure :: get_parton_kinds => qcd_2_2_get_parton_kinds
```

⟨*Muli: procedures*⟩+≡

```
  pure function qcd_2_2_get_parton_kinds (this) result (kinds)
    class(qcd_2_2_t), intent(in) :: this
    integer, dimension(2) :: kinds
    kinds = this%get_pdf_int_kinds ()
```

```
    kinds(1) = parton_kind_of_int_kind (kinds(1))
    kinds(2) = parton_kind_of_int_kind (kinds(2))
  end function qcd_2_2_get_parton_kinds
```

⟨*Muli: QCD22: TBP*⟩+≡

```
  procedure :: get_pdf_int_kinds => qcd_2_2_get_pdf_int_kinds
```

⟨*Muli: procedures*⟩+≡

```
  pure function qcd_2_2_get_pdf_int_kinds (this) result (kinds)
    class(qcd_2_2_t), intent(in) :: this
    integer, dimension(2) :: kinds
    kinds = double_pdf_kinds (1:2, this%integrand_id)
  end function qcd_2_2_get_pdf_int_kinds
```

⟨*Muli: QCD22: TBP*⟩+≡

```
  procedure :: get_momentum_boost => qcd_2_2_get_momentum_boost
```

⟨*Muli: procedures*⟩+≡

```
  elemental function qcd_2_2_get_momentum_boost (this) result (boost)
    class(qcd_2_2_t), intent(in) :: this
    real(default) :: boost
    boost = - one
    ! print('("qcd_2_2_get_momentum_boost: not yet implemented.")')
    ! boost = this%momentum_boost
  end function qcd_2_2_get_momentum_boost
```

⟨*Muli: QCD22: TBP*⟩+≡

```
  procedure :: get_hyperbolic_fractions => qcd_2_2_get_hyperbolic_fractions
```

⟨*Muli: procedures*⟩+≡

```
  pure function qcd_2_2_get_hyperbolic_fractions (this) result (fractions)
    class(qcd_2_2_t), intent(in) :: this
    real(double), dimension(3) :: fractions
    fractions = this%hyperbolic_fractions
  end function qcd_2_2_get_hyperbolic_fractions
```

⟨*Muli: QCD22: TBP*⟩+≡

```
  procedure :: get_remnant_momentum_fractions => &
       qcd_2_2_get_remnant_momentum_fractions
```

⟨*Muli: procedures*⟩+≡

```
  pure function qcd_2_2_get_remnant_momentum_fractions &
       (this) result (fractions)
    class(qcd_2_2_t), intent(in) :: this
    real(default), dimension(2) :: fractions
    fractions = this%momentum_fractions(1:2)
  end function qcd_2_2_get_remnant_momentum_fractions
```

⟨*Muli: QCD22: TBP*⟩+≡

```
  procedure :: get_total_momentum_fractions => &
       qcd_2_2_get_total_momentum_fractions
```

⟨*Muli: procedures*⟩+≡

```
pure function qcd_2_2_get_total_momentum_fractions &
     (this) result (fractions)
  class(qcd_2_2_t), intent(in) :: this
  real(default), dimension(2) :: fractions
  fractions = [-one, -one]
  ! fractions = this%hyperbolic_fractions(1:2) * &
  !    this%beam%get_proton_remnant_momentum_fractions()
end function qcd_2_2_get_total_momentum_fractions
```

⟨*Muli: QCD22: TBP*⟩+≡

```
procedure :: get_color_flow => qcd_2_2_get_color_flow
```

⟨*Muli: procedures*⟩+≡

```
pure function qcd_2_2_get_color_flow (this) result (flow)
  class(qcd_2_2_t), intent(in) :: this
  integer, dimension(4) :: flow
  flow = this%flow
end function qcd_2_2_get_color_flow
```

⟨*Muli: QCD22: TBP*⟩+≡

```
procedure :: get_color_correlations => qcd_2_2_get_color_correlations
```

⟨*Muli: procedures*⟩+≡

```
subroutine qcd_2_2_get_color_correlations &
     (this, start_index, final_index, flow)
  class(qcd_2_2_t), intent(in) :: this
  integer, intent(in) :: start_index
  integer, intent(out) :: final_index
  integer, dimension(2,4), intent(out) :: flow
  integer :: pos, f_end, f_beginning
  final_index = start_index
  !!! We set all flows to i_zero. i_zero means no connection.
  flow = reshape([0,0,0,0,0,0,0,0],[2,4])
  !!! look at all four possible ends of color lines
  do f_end = 1, 4
     !!! The beginning of of this potential line is stored in flow.
     !!! i_zero means no line.
     f_beginning = this%flow(f_end)
     !!! Is there a line beginning at f_beginning and ending at f_end?
     if (f_beginning > 0) then
        !!! yes it is. we get a new number for this new line
        final_index = final_index + 1
        !!! Is this line beginning in the initial state?
        if (f_beginning < 3) then
           !!! Yes it is. lets connect the color entry of f_begin.
           flow(1,f_beginning) = final_index
        else
           !!! No, it's the final state.
           !!! lets connect the anticolor entry of f_begin.
           flow(2,f_beginning) = final_index
        end if
        !!! Is this line ending in the final state?
        if (f_end > 2) then
```

```
            !!! Yes it is. lets connect the color entry of f_end.
            flow(1,f_end) = final_index
         else
            !!! No, it's the initial state.
            !!! Lets connect the anticolor entry of f_end.
            flow(2,f_end) = final_index
         end if
      end if
   end do
end subroutine qcd_2_2_get_color_correlations
```

⟨*Muli: QCD22: TBP*⟩+≡

```
generic :: initialize => qcd_2_2_initialize
procedure :: qcd_2_2_initialize
```

⟨*Muli: procedures*⟩+≡

```
subroutine qcd_2_2_initialize (this, gev2_s, process_id, &
     integrand_id, parton_ids, flow, hyp, cart)
  class(qcd_2_2_t), intent(out) :: this
  real(default), intent(in) :: gev2_s
  integer, intent(in) :: process_id, integrand_id
  integer, dimension(2), intent(in) :: parton_ids
  integer, dimension(4), intent(in) :: flow
  real(default), dimension(3), intent(in)::hyp
  real(default), dimension(3), intent(in), optional :: cart
  call this%initialize (gev2_s)
  this%process_id = process_id
  this%integrand_id = integrand_id
  this%parton_ids = parton_ids
  this%flow = flow
  this%hyperbolic_fractions = hyp
  if (present (cart)) then
     this%momentum_fractions = cart
  else
     this%momentum_fractions = h_to_c_param (hyp)
  end if
end subroutine qcd_2_2_initialize
```

22.11.1 The main Multiple Interactions type

⟨*Muli: public*⟩≡

```
public :: muli_t
```

⟨*Muli: types*⟩+≡

```
type, extends(qcd_2_2_t) :: muli_t
   real(default) :: GeV2_scale_cutoff
   logical :: modify_pdfs = muli_default_modify_pdfs
   !!! Pt chain status
   logical :: finished = .false.
   logical :: exceeded = .false.
   !!! Timers
   real(default) :: init_time = zero
   real(default) :: pt_time = zero
```

```
    real(default) :: partons_time = zero
    real(default) :: confirm_time = zero
    !!! Generator internals
    logical :: initialized = .false.
    logical :: initial_interaction_given = .false.
    real(default) :: mean = one
    real(default), dimension(0:16) :: start_integrals = &
         [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
    type(tao_random_state) :: tao_rnd
    type(muli_trapezium_tree_t) :: dsigma
    type(sample_inclusive_t) :: samples
    type(pp_remnant_t) :: beam
    !!! These pointers shall not be allocated, deallocated,
    !!!  serialized or deserialized explicitly.
    class(muli_trapezium_node_class_t), pointer :: node => null()
  contains
  ⟨Muli: muli: TBP⟩
end type muli_t
```

⟨*Muli: muli: TBP*⟩≡

```
procedure :: write_to_marker => muli_write_to_marker
```

⟨*Muli: procedures*⟩+≡

```
subroutine muli_write_to_marker (this, marker, status)
  class(muli_t), intent(in) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%mark_begin ("muli_t")
  call qcd_2_2_write_to_marker (this, marker, status)
  call marker%mark ("modify_pdfs", this%modify_pdfs)
  call marker%mark ("initialized", this%initialized)
  call marker%mark &
       ("initial_interaction_given", this%initial_interaction_given)
  call marker%mark ("finished", this%finished)
  call marker%mark ("init_time", this%init_time)
  call marker%mark ("pt_time", this%pt_time)
  call marker%mark ("partons_time", this%partons_time)
  call marker%mark ("confirm_time", this%confirm_time)
  ! call marker%mark_instance (this%start_values, "start_values")
  call marker%mark_instance (this%dsigma, "dsigma")
  call marker%mark_instance (this%samples, "samples")
  call marker%mark_instance (this%beam, "beam")
  call marker%mark_end ("muli_t")
end subroutine muli_write_to_marker
```

⟨*Muli: muli: TBP*⟩+≡

```
procedure :: read_from_marker => muli_read_from_marker
```

⟨*Muli: procedures*⟩+≡

```
subroutine muli_read_from_marker (this, marker, status)
  class(muli_t), intent(out) :: this
  class(marker_t), intent(inout) :: marker
  integer(dik), intent(out) :: status
  call marker%pick_begin ("muli_t", status=status)
```

```
    call qcd_2_2_read_from_marker (this, marker, status)
    call marker%pick ("modify_pdfs", this%modify_pdfs, status)
    call marker%pick ("initialized", this%initialized, status)
    call marker%pick &
         ("initial_interaction_given", this%initial_interaction_given, status)
    call marker%pick ("finished", this%finished, status)
    call marker%pick ("init_time", this%init_time, status)
    call marker%pick ("pt_time", this%pt_time, status)
    call marker%pick ("partons_time", this%partons_time, status)
    call marker%pick ("confirm_time", this%confirm_time, status)
    ! call marker%pick_instance &
    !    ("start_values", this%start_values, status=status)
    call marker%pick_instance ("dsigma", this%dsigma, status=status)
    call marker%pick_instance ("samples", this%samples, status=status)
    call marker%pick_instance ("beam", this%beam, status=status)
    call marker%pick_end ("muli_t", status)
  end subroutine muli_read_from_marker
```

⟨*Muli: muli: TBP*⟩+≡

```
  procedure :: print_to_unit => muli_print_to_unit
```

⟨*Muli: procedures*⟩+≡

```
  subroutine muli_print_to_unit (this, unit, parents, components, peers)
    class(muli_t), intent(in) :: this
    integer, intent(in) :: unit
    integer(dik), intent(in) :: parents, components, peers
    if (parents>0) &
         call qcd_2_2_print_to_unit (this, unit, parents-1, components, peers)
    write (unit, "(1x,A)")  "Components of muli_t :"
    write (unit, "(3x,A)")  "Model Parameters:"
    write (unit, "(3x,A,E20.10)") "GeV2_scale_cutoff : ", &
         this%GeV2_scale_cutoff
    write (unit, "(3x,A,L1)")  "Modify PDF        : ", this%modify_pdfs
    write (unit, "(3x,A)")  "PT Chain Status:"
    write (unit, "(3x,A,L1)")  "Initialized       : ", this%initialized
    write (unit, "(3x,A,L1)")  "initial_interaction_given: ", &
         this%initial_interaction_given
    write (unit, "(3x,A,L1)")  "Finished          : ", this%finished
    write (unit, "(3x,A,L1)")  "Exceeded          : ", this%exceeded
    write (unit, "(3x,A)")  "Generator Internals:"
    write (unit, "(3x,A,E20.10)")  "Mean Value        : ", this%mean
    if (components > i_zero) then
       write (unit, "(3x,A,16(E20.10))")  "Start Integrals   : ", &
            this%start_integrals(1:16)
       ! write (unit, "(3x,A)")  "start_values Component:"
       ! call this%start_values%print_to_unit &
       !    (unit, parents, components-1, peers)
       write (unit, "(3x,A)")  "dsigma Component:"
       call this%dsigma%print_to_unit (unit, parents, components-1, peers)
       write (unit, "(3x,A)")  "samples Component:"
       call this%samples%print_to_unit (unit, parents, components-1, peers)
       write (unit, "(3x,A)")  "beam Component:"
       call this%beam%print_to_unit (unit, parents, components-1, peers)
    else
```

```
      write (unit, "(3x,A)")  "Skipping Derived-Type Components."
    end if
    ! call print_comp_pointer (this%start_node, unit, i_zero, &
    !    min(components-1,i_one), i_zero, "start_node")
    ! call serialize_print_comp_pointer (this%node, unit, i_zero, &
    !    min(components-1,i_one), i_zero, "node")
  end subroutine muli_print_to_unit
```

⟨*Muli: muli: TBP*⟩+≡

```
  procedure, nopass :: get_type => muli_get_type
```

⟨*Muli: procedures*⟩+≡

```
  pure subroutine muli_get_type(type)
    character(:), allocatable, intent(out) :: type
    allocate (type, source="muli_t")
  end subroutine muli_get_type
```

⟨*Muli: muli: TBP*⟩+≡

```
  generic :: initialize => muli_initialize
  procedure :: muli_initialize
```

⟨*Muli: procedures*⟩+≡

```
  subroutine muli_initialize (this, GeV2_scale_cutoff, gev2_s, &
       muli_dir, random_seed)
    class(muli_t), intent(out) :: this
    real(kind=default), intent(in) :: gev2_s, GeV2_scale_cutoff
    character(*), intent(in) :: muli_dir
    integer, intent(in), optional :: random_seed
    real(double) :: time
    logical :: exist
    type(muli_dsigma_t) :: dsigma_aq
    character(3) :: lhapdf_member_c
    call cpu_time(time)
    this%init_time = this%init_time-time
    print *, "muli_initialize: The MULI modules are still not fully " &
         // "populated, so MULI might generate some dummy values instead " &
         // "of real Monte Carlo generated interactions."
    print *, "Given Parameters:"
    print *, "GeV2_scale_cutoff=", GeV2_scale_cutoff
    print *, "muli_dir=", muli_dir
    print *, "lhapdf_dir=", ""
    print *, "lhapdf_file=", muli_default_lhapdf_file
    print *, "lhapdf_member=", muli_default_lhapdf_member
    print *, ""
    call this%transverse_mom_t%initialize (gev2_s)
    call this%beam%initialize (muli_dir, lhapdf_dir="", &
         lhapdf_file=muli_default_lhapdf_file, &
         lhapdf_member=muli_default_lhapdf_member)
    this%GeV2_scale_cutoff = GeV2_scale_cutoff
    if (present(random_seed)) then
       call tao_random_create (this%tao_rnd, random_seed)
    else
       call tao_random_create (this%tao_rnd, 1)
    end if
```

```
    print *, "looking for previously generated root function..."
    call integer_with_leading_zeros (muli_default_lhapdf_member, 3, &
         lhapdf_member_c)
    inquire (file=muli_dir//"/dsigma_"//muli_default_lhapdf_file//".xml", &
         exist=exist)
    if (exist) then
       print *, "found. Starting deserialization..."
       call this%dsigma%deserialize &
            (name="dsigma_"//muli_default_lhapdf_file//"_"//lhapdf_member_c, &
            file=muli_dir//"/dsigma_"//muli_default_lhapdf_file//".xml")
       ! call this%dsigma%print_all ()
       print *, "done. Starting generation of plots..."
       call this%dsigma%gnuplot (muli_dir)
       print *, "done."
    else
       print *, &
            "No root function found. Starting generation of root function..."
       call dsigma_aq%generate (GeV2_scale_cutoff, gev2_s, this%dsigma)
       print *, "done. Starting serialization of root function..."
       call this%dsigma%serialize &
            (name="dsigma_"//muli_default_lhapdf_file//"_"//lhapdf_member_c, &
            file=muli_dir//"/dsigma_"//muli_default_lhapdf_file//".xml")
       print *, "done. Starting serialization of generator..."
       call dsigma_aq%serialize &
            (name="dsigma_aq_"//muli_default_lhapdf_file//"_" // &
            lhapdf_member_c, file=muli_dir//"/dsigma_aq_" // &
            muli_default_lhapdf_file//".xml")
       print *,"done. Starting generation of plots..."
       call this%dsigma%gnuplot (muli_dir)
       print *, "done."
    end if
    print *, ""
    print *, "looking for previously generated samples..."
    inquire (file=muli_dir//"/samples.xml", exist=exist)
    if (exist) then
       print *, "found. Starting deserialization..."
       call this%samples%deserialize ("samples",muli_dir//"/samples.xml")
    else
       print *,"No samples found. Starting with default initialization."
       call this%samples%initialize (4, int_sizes_all, int_all, 1E-2_default)
    end if
    call this%restart ()
    this%initialized = .true.
    call cpu_time (time)
    this%init_time = this%init_time + time
  end subroutine muli_initialize
```

⟨*Muli: muli: TBP*⟩+≡

```
  procedure :: apply_initial_interaction => muli_apply_initial_interaction
```

⟨*Muli: procedures*⟩+≡

```
  subroutine muli_apply_initial_interaction (this, GeV2_s, &
       x1, x2, pdg_f1, pdg_f2, n1, n2)
    class(muli_t), intent(inout) :: this
```

```
    real(default), intent(in) :: Gev2_s, x1, x2
    integer, intent(in):: pdg_f1, pdg_f2, n1, n2
    real(default) :: rnd1, rnd2, time
    if (this%initialized) then
       call cpu_time (time)
       this%init_time = this%init_time - time
       print *, "muli_apply_initial_interaction:"
       print *, "gev2_s=", gev2_s
       print *, "x1=", x1
       print *, "x2=", x2
       print *, "pdg_f1=", pdg_f1
       print *, "pdg_f2=", pdg_f2
       print *, "n1=", n1
       print *, "n2=", n2
       call tao_random_number (this%tao_rnd, rnd1)
       call tao_random_number (this%tao_rnd, rnd2)
       call cpu_time (time)
       this%init_time = this%init_time + time
       call this%beam%apply_initial_interaction &
                 (sqrt (gev2_s), x1, x2, pdg_f1, pdg_f2, n1, n2,&
                 !!! This is a hack: We should give the pt scale of the initial
                 !!! interaction. Unfortunately, we only know the invariant
                 !!! mass shat. shat/2 is the upper bound of pt, so we
                 !!! use it for now.
            sqrt(gev2_s) * x1 *x2 / 2D0, &
            rnd1, rnd2)
       this%initial_interaction_given = .true.
    else
       print *, &
            "muli_apply_initial_interaction: call muli_initialize first. STOP"
       stop
    end if
  end subroutine muli_apply_initial_interaction
```

⟨*Muli: muli: TBP*⟩+≡

```
  procedure :: finalize => muli_finalize
```

⟨*Muli: procedures*⟩+≡

```
  subroutine muli_finalize (this)
    class(muli_t), intent(inout) :: this
    print *, "muli_finalize"
    nullify (this%node)
    call this%dsigma%finalize ()
    call this%samples%finalize ()
    call this%beam%finalize ()
  end subroutine muli_finalize
```

⟨*Muli: muli: TBP*⟩+≡

```
  procedure :: stop_trainer => muli_stop_trainer
```

⟨*Muli: procedures*⟩+≡

```
  subroutine muli_stop_trainer (this)
    class(muli_t), intent(inout) :: this
    print *, "muli_stop_trainer: DUMMY!"
```

```
    end subroutine muli_stop_trainer
```

⟨*Muli: muli: TBP*⟩+≡

```
    procedure :: reset_timer => muli_reset_timer
```

⟨*Muli: procedures*⟩+≡

```
    subroutine muli_reset_timer (this)
      class(muli_t), intent(inout) :: this
      this%init_time = 0D0
      this%pt_time = 0D0
      this%partons_time = 0D0
      this%confirm_time = 0D0
    end subroutine muli_reset_timer
```

⟨*Muli: muli: TBP*⟩+≡

```
    procedure :: restart => muli_restart
```

⟨*Muli: procedures*⟩+≡

```
    subroutine muli_restart (this)
      class(muli_t), intent(inout) :: this
      call this%dsigma%get_rightmost (this%node)
      call this%beam%reset ()
      ! print *, associated(this%node)
      ! nullify (this%node)
      this%finished = .false.
      this%process_id = -1
      this%integrand_id = -1
      this%momentum_fractions = [-1D0,-1D0,1D0]
      this%hyperbolic_fractions = [-1D0,-1D0,1D0]
      ! this%start_values%process_id = -1
      ! this%start_values%integrand_id = -1
      ! this%start_values%momentum_fractions = [-1D0,-1D0,1D0]
      ! this%start_values%hyperbolic_fractions = [-1D0,-1D0,1D0]
      this%start_integrals = &
           [0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0,0D0]
    end subroutine muli_restart
```

⟨*Muli: muli: TBP*⟩+≡

```
    procedure :: is_initialized => muli_is_initialized
```

⟨*Muli: procedures*⟩+≡

```
    elemental function muli_is_initialized (this) result (res)
      logical :: res
      class(muli_t), intent(in) :: this
      res = this%initialized
    end function muli_is_initialized
```

⟨*Muli: muli: TBP*⟩+≡

```
    procedure :: is_initial_interaction_given => &
         muli_is_initial_interaction_given
```

⟨*Muli: procedures*⟩+≡

```
elemental function muli_is_initial_interaction_given (this) result (res)
  logical :: res
  class(muli_t), intent(in) :: this
  res = this%initial_interaction_given
end function muli_is_initial_interaction_given
```

⟨*Muli: muli: TBP*⟩+≡

```
procedure :: is_finished => muli_is_finished
```

⟨*Muli: procedures*⟩+≡

```
elemental function muli_is_finished (this) result (res)
  logical :: res
  class(muli_t), intent(in) :: this
  res = this%finished
end function muli_is_finished
```

⟨*Muli: muli: TBP*⟩+≡

```
procedure :: enable_remnant_pdf => muli_enable_remnant_pdf
```

⟨*Muli: procedures*⟩+≡

```
subroutine muli_enable_remnant_pdf (this)
  class(muli_t), intent(inout) :: this
  this%modify_pdfs = .true.
end subroutine muli_enable_remnant_pdf
```

⟨*Muli: muli: TBP*⟩+≡

```
procedure :: disable_remnant_pdf => muli_disable_remnant_pdf
```

⟨*Muli: procedures*⟩+≡

```
subroutine muli_disable_remnant_pdf (this)
  class(muli_t), intent(inout) :: this
  this%modify_pdfs = .false.
end subroutine muli_disable_remnant_pdf
```

⟨*Muli: muli: TBP*⟩+≡

```
procedure :: generate_gev2_pt2 => muli_generate_gev2_pt2
```

⟨*Muli: procedures*⟩+≡

```
subroutine muli_generate_gev2_pt2 (this, gev2_start_scale, gev2_new_scale)
  class(muli_t), intent(inout) :: this
  real(kind=default), intent(in) :: gev2_start_scale
  real(kind=default), intent(out) :: gev2_new_scale
  real(double) :: time
  call cpu_time (time)
  this%pt_time = this%pt_time - time
  call this%set_gev2_scale (gev2_start_scale)
  this%start_integrals = this%node%approx_integral (this%get_unit_scale ())
  call this%generate_next_scale ()
  gev2_new_scale = this%get_gev2_scale ()
  call cpu_time (time)
  this%pt_time = this%pt_time + time
end subroutine muli_generate_gev2_pt2
```

⟨*Muli: muli: TBP*⟩+≡

```
procedure :: generate_partons => muli_generate_partons
```

⟨*Muli: procedures*⟩+≡

```
subroutine muli_generate_partons (this, n1, n2, x_proton_1, x_proton_2, &
    pdg_f1, pdg_f2, pdg_f3, pdg_f4)
  class(muli_t), intent(inout) :: this
  integer, intent(in) :: n1, n2
  real(kind=default), intent(out) :: x_proton_1, x_proton_2
  integer, intent(out) :: pdg_f1, pdg_f2, pdg_f3, pdg_f4
  integer, dimension(4) :: pdg_f
  real(double) :: time
  ! print *, "muli_generate_partons: n1=", n1, " n2=", n2
  this%parton_ids(1) = n1
  this%parton_ids(2) = n2
  call cpu_time (time)
  this%partons_time = this%partons_time - time
  this%mean = this%node%approx_value_n (this%get_unit_scale(), &
       this%integrand_id)
  call this%samples%mcgenerate_hit (this%get_unit2_scale(), &
       this%mean, this%integrand_id, this%tao_rnd, this%process_id, &
       this%momentum_fractions)
  ! print *,"muli_generate_partons", this%momentum_fractions
  call this%generate_flow ()
  if (this%modify_pdfs) then
     call cpu_time (time)
     this%partons_time = this%partons_time + time
     this%confirm_time = this%confirm_time - time
     call this%beam%apply_interaction (this)
     call cpu_time (time)
     this%confirm_time = this%confirm_time + time
     this%partons_time = this%partons_time - time
  end if
  x_proton_1 = this%momentum_fractions(1)
  x_proton_2 = this%momentum_fractions(2)
  pdg_f = this%get_pdg_flavors ()
  pdg_f1 = pdg_f(1)
  pdg_f2 = pdg_f(2)
  pdg_f3 = pdg_f(3)
  pdg_f4 = pdg_f(4)
  call cpu_time (time)
  this%partons_time = this%partons_time - time
  call qcd_2_2_print_to_unit (this, output_unit, 100_dik, 100_dik, 100_dik)
end subroutine muli_generate_partons
```

⟨*Muli: muli: TBP*⟩+≡

```
procedure :: generate_flow => muli_generate_flow
```

⟨*Muli: procedures*⟩+≡

```
subroutine muli_generate_flow(this)
  class(muli_t), intent(inout)::this
  integer::rnd
  integer::m,n
  logical, dimension(3)::t
  integer, dimension(4)::tmp_flow, tmp_array
```

```
    ! we initialize with zeros. a i_zero means no line ends here.
    this%flow=[0,0,0,0]
    ! we randomly pick a color flow
    call tao_random_number(this%tao_rnd,rnd)
    ! the third position of muli_flow_stats is the sum of all flow wheights of stratum diagram_kin
    ! so we generate a random number 0 <= m < sum(weights)
    m=modulo(rnd,muli_flow_stats(3,this%get_diagram_color_kind()))
    ! lets visit all color flows of stratum diagram_kind. the first and second position of muli_fl
    ! tells us the index of the first and the last valid color flow.
    do n=muli_flow_stats(1,this%get_diagram_color_kind()),muli_flow_stats(2,this%get_diagram_color
       ! now we remove the weight of flow n from our random number.
       m=m-muli_flows(0,n)
       ! this is how we pick a flow.
       if (m<0) then
          ! the actual flow
          this%flow=muli_flows(1:4,n)
          exit
       end if
    end do
    ! the diagram kind contains a primitive diagram and all diagramms which can be deriven by
    ! (1) global charge conjugation
    ! (2) permutation of the initial state particles
    ! (3) permutation of the final state particles
    ! lets see, what transformations we have got in our actual interaction.
    tmp_array = this%get_lha_flavors ()
    t = muli_get_state_transformations (this%get_diagram_color_kind (), &
         tmp_array)
    !     this%get_lha_flavors ())
    ! now we have to apply these transformations to our flow.
    ! (1) means: swap beginning and end of a line. flow is a permutation that maps
    ! ends to their beginnings, so we apply flow to itself:
!!$    print *,"(0)",this%flow
    if (t(1)) then
       tmp_flow=this%flow
       this%flow=[0,0,0,0]
       do n=1,4
          if (tmp_flow(n)>0)this%flow(tmp_flow(n))=n
       end do
!!$       print *,"(1)",this%flow
    end if
    if (t(2)) then
       ! we swap the particles 1 and 2
       tmp_flow(1)=this%flow(2)
       tmp_flow(2)=this%flow(1)
       tmp_flow(3:4)=this%flow(3:4)
!!$       print *,"(2)",tmp_flow
       ! we swap the beginnings assigned to particle 1 and 2
       where(tmp_flow==1)
          this%flow=2
       elsewhere(tmp_flow==2)
          this%flow=1
       elsewhere
          this%flow=tmp_flow
       end where
```

```
!!$       print *,"(2)",this%flow
    end if
    if (t(3)) then
       ! we swap the particles 3 and 4
       tmp_flow(1:2)=this%flow(1:2)
       tmp_flow(3)=this%flow(4)
       tmp_flow(4)=this%flow(3)
!!$       print *,"(3)",tmp_flow
       ! we swap the beginnings assigned to particle 3 and 4
       where(tmp_flow==3)
          this%flow=4
       elsewhere(tmp_flow==4)
          this%flow=3
       elsewhere
          this%flow=tmp_flow
       end where
!!$       print *,"(3)",this%flow
    end if
  end subroutine muli_generate_flow
```

⟨*Muli: muli: TBP*⟩+≡

```
  procedure :: replace_parton => muli_replace_parton
```

⟨*Muli: procedures*⟩+≡

```
  subroutine muli_replace_parton &
       (this, proton_id, old_id, new_id, pdg_f, x_proton, gev_scale)
    class(muli_t), intent(inout) :: this
    integer, intent(in) :: proton_id, old_id, new_id, pdg_f
    real(kind=default), intent(in) :: x_proton, gev_scale
    ! print *, "muli_replace_parton(", proton_id, old_id, new_id, &
    !    pdg_f, x_proton, gev_scale, ")"
    if (proton_id==1 .or. proton_id==2) then
       call this%beam%replace_parton &
            (proton_id, old_id, new_id, pdg_f, x_proton, gev_scale)
    else
       print *, "muli_replace_parton: proton_id must be 1 or 2, but ", &
            proton_id, " was given."
       stop
    end if
  end subroutine muli_replace_parton
```

⟨*Muli: muli: TBP*⟩+≡

```
  procedure :: get_parton_pdf => muli_get_parton_pdf
```

⟨*Muli: procedures*⟩+≡

```
  function muli_get_parton_pdf &
       (this, x_proton, gev2_scale, n, pdg_f) result (pdf)
    real(default) :: pdf
    class(muli_t), intent(in) :: this
    real(default), intent(in) :: x_proton, gev2_scale
    integer, intent(in) :: n, pdg_f
    call this%beam%parton_pdf (x_proton, gev2_scale, n, pdg_f, pdf)
  end function muli_get_parton_pdf
```

⟨*Muli: muli: TBP*⟩+≡

```
procedure :: get_momentum_pdf => muli_get_momentum_pdf
```

⟨*Muli: procedures*⟩+≡

```
function muli_get_momentum_pdf &
     (this, x_proton, gev2_scale, n, pdg_f) result (pdf)
  real(default) :: pdf
  class(muli_t), intent(in) :: this
  real(default), intent(in) :: x_proton, gev2_scale
  integer, intent(in) :: n, pdg_f
  call this%beam%momentum_pdf (x_proton, gev2_scale, n, pdg_f, pdf)
end function muli_get_momentum_pdf
```

⟨*Muli: muli: TBP*⟩+≡

```
procedure :: print_timer => muli_print_timer
```

⟨*Muli: procedures*⟩+≡

```
subroutine muli_print_timer(this)
  class(muli_t), intent(in) :: this
  print ("(1x,A,E20.10)"), "Init time:    ", this%init_time
  print ("(1x,A,E20.10)"), "PT gen time:  ", this%pt_time
  print ("(1x,A,E20.10)"), "Partons time: ", this%partons_time
  print ("(1x,A,E20.10)"), "Confirm time: ", this%confirm_time
  print ("(1x,A,E20.10)"), "Overall time: ", &
       this%init_time + this%pt_time + this%partons_time + this%confirm_time
end subroutine muli_print_timer
```

⟨*Muli: muli: TBP*⟩+≡

```
procedure :: generate_samples => muli_generate_samples
```

⟨*Muli: procedures*⟩+≡

```
subroutine muli_generate_samples &
     (this, n_total, n_print, integrand_kind, muli_dir, analyse)
  class(muli_t), intent(inout) :: this
  integer(dik), intent(in) :: n_total, n_print
  integer, intent(in) :: integrand_kind
  character(*), intent(in) :: muli_dir
  logical, intent(in) :: analyse
  integer(dik) :: n_inner

  class(muli_trapezium_node_class_t), pointer :: start_node => null()
  class(muli_trapezium_node_class_t), pointer, save :: s_node => null()
  class(muli_trapezium_node_class_t), pointer, save :: node => null()

  character(2) :: prefix
  integer, save :: t_slice, t_region, t_proc, t_subproc, t_max_n = 0
  integer(dik) :: n_t, n_p, n_m
  integer :: n, m, u, unit = 0
  integer(dik) :: n_tries = 0
  integer(dik) :: n_hits = 0
  integer(dik) :: n_over = 0
  integer(dik) :: n_miss = 0
  real(default), save, dimension(3) :: cart_hit
  integer, save, dimension(4) :: t_i_rnd
  ! integer, save, dimension(5) :: r_n_proc
```

```
real(default), dimension(16) :: d_rnd
real(default), save :: t_area, t_dddsigma, t_rnd, t_weight, t_arg
real(default) :: mean = 0D0
real(default) :: time = 0D0
real(default) :: timepa = 0D0
real(default) :: timept = 0D0
real(default) :: timet = 0D0
real(default) :: pts, s_pts = 1D0
real(default) :: pts2 = 1D0
real(default) :: rnd
logical :: running
character(3) :: num
integer :: success = -1
integer :: chain_length = 0
integer :: int_kind
integer :: process_id
real(double), dimension(0:16) :: integral
call this%print_parents ()
n_tries = one
n_inner = n_total / n_print
n_t = i_zero
PRINT: do while (n_t < n_total)
   call cpu_time (time)
   timet = - time
   n_p = i_zero
   INNER: do while (n_p < n_print)
      chain_length = 0
      ! print *,"new chain"
      call this%restart ()
      this%integrand_id = integrand_kind
      call cpu_time (time)
      timept = timept - time
      call this%generate_next_scale (integrand_kind)
      call cpu_time (time)
      timept = timept + time
      CHAIN: do while (.not. this%is_finished ())
         chain_length = chain_length + 1
         n_p = n_p + 1
         call this%confirm ()
         call cpu_time (time)
         timepa = timepa - time
         ! print *, this%get_unit2_scale ()
         call this%samples%mcgenerate_hit (this%get_unit2_scale(), &
              this%mean, this%integrand_id, this%tao_rnd, this%process_id, &
              this%momentum_fractions)
         call cpu_time (time)
         timepa = timepa + time
         timept = timept - time
         call this%generate_next_scale (integrand_kind)
         call cpu_time (time)
         timept = timept + time
      end do CHAIN
      ! print *, "chain length = ", chain_length
   end do INNER
```

```
      n_t = n_t + n_p
      call this%samples%sum_up ()
      call cpu_time (time)
      timet = timet + time
      print *, n_t, "/", n_total
      print *, "time: ", timet
      print *, "pt time: ", timept
      print *, "pa time: ", timepa
      print *, this%samples%n_tries_sum, this%samples%n_hits_sum, &
           this%samples%n_over_sum
      if (this%samples%n_hits_sum > 0) then
         print *, (this%samples%n_hits_sum * 10000) / &
              this%samples%n_tries_sum, (this%samples%n_over_sum * 10000) / &
              this%samples%n_hits_sum
      else
         print *, "no hits"
      end if
      ! print ('(7(I11," "),5(E14.7," "))'), n_p, n_print, n_tries, &
      !   n_hits,n_over, int((n_hits*1D3)/n_tries), &
      !   int((n_over*1D6)/n_tries), n_hits/real(n_over), time1, time2, &
      !   time3, this%samples%int_kinds(integrand_kind)%overall_boost
   end do print
   call integer_with_leading_zeros (integrand_kind, 2, prefix)
   if (analyse) then
      call this%samples%int_kinds(integrand_kind)%analyse &
           (muli_dir, prefix//"_")
      call this%samples%int_kinds(integrand_kind)%serialize &
           ("sample_int_kind_"//prefix, &
           muli_dir//"/sample_int_kind/"//prefix//".xml")
   end if
   call this%samples%int_kinds(integrand_kind)%serialize &
        ("sample_int_kind_"//prefix, &
        muli_dir//"/sample_int_kind/"//prefix//".xml")
end subroutine muli_generate_samples
```

⟨*Muli: muli: TBP*⟩+≡

```
procedure :: fake_interaction => muli_fake_interaction
```

⟨*Muli: procedures*⟩+≡

```
subroutine muli_fake_interaction (this, GeV2_scale, x1, x2, &
     process_id, integrand_id, n1, n2, flow)
   class(muli_t), intent(inout) :: this
   real(default), intent(in) :: Gev2_scale, x1, x2
   integer, intent(in) :: process_id, integrand_id, n1, n2
   integer, dimension(4), intent(in), optional :: flow
   call this%set_gev2_scale (Gev2_scale)
   this%process_id = process_id
   this%integrand_id = integrand_id
   this%parton_ids = [n1, n2]
   if (present (flow)) then
      this%flow = flow
   else
      this%flow = [0,0,0,0]
   end if
```

```
    this%momentum_fractions = [x1, x2, this%get_unit2_scale()]
    call this%beam%apply_interaction (this)
    call this%beam%print_all ()
  end subroutine muli_fake_interaction
```

⟨*Muli: muli: TBP*⟩+≡

```
  procedure :: generate_next_scale => muli_generate_next_scale
```

⟨*Muli: procedures*⟩+≡

```
  subroutine muli_generate_next_scale (this, integrand_kind)
    class(muli_t), intent(inout) :: this
    integer, intent(in), optional :: integrand_kind
    real(default) :: pts, tmp_pts, rnd
    integer :: tmp_int_kind
    class(muli_trapezium_node_class_t), pointer :: tmp_node
      pts = - one
      if (present (integrand_kind)) then
         call tao_random_number (this%tao_rnd, rnd)
         call generate_single_pts (integrand_kind, &
              this%start_integrals(integrand_kind), &
              this%beam%get_pdf_int_weights &
              (double_pdf_kinds (1:2,integrand_kind)), rnd, this%dsigma, &
              pts, this%node)
      else
         do tmp_int_kind = 1, 16
            call tao_random_number (this%tao_rnd, rnd)
            call generate_single_pts (tmp_int_kind, &
                 this%start_integrals(tmp_int_kind), &
                 this%beam%get_pdf_int_weights &
                 (double_pdf_kinds(1:2,tmp_int_kind)), rnd, &
                 this%dsigma, tmp_pts, tmp_node)
            if (tmp_pts > pts) then
               pts = tmp_pts
               this%integrand_id = tmp_int_kind
               this%node => tmp_node
            end if
         end do
      end if
      if (pts > 0) then
         call this%set_unit_scale (pts)
      else
         this%finished = .true.
      end if
      ! print *, this%finished, this%integrand_id
  contains
    subroutine generate_single_pts &
         (int_kind, start_int, weight, rnd, int_tree, pts, node)
      integer, intent(in) :: int_kind
      real(default), intent(in) :: start_int, weight, rnd
      type(muli_trapezium_tree_t), intent(in) :: int_tree
      real(default), intent(out) :: pts
      class(muli_trapezium_node_class_t),pointer, intent(out) :: node
      real(default) :: arg
      ! print *, int_kind, start_int, weight, rnd
```

```
      if (weight > 0D0) then
         arg = start_int - log(rnd) / weight
         call int_tree%find_decreasing (arg, int_kind, node)
         if (node%get_l_integral(int_kind) > arg) then
            pts = node%approx_position_by_integral (int_kind, arg)
         else
            pts = -1D0
         end if
      else
         pts = -1D0
      end if
    end subroutine generate_single_pts
  end subroutine muli_generate_next_scale
```

⟨*Muli: muli: TBP*⟩+≡

```
  procedure :: confirm => muli_confirm
```

⟨*Muli: procedures*⟩+≡

```
  subroutine muli_confirm (this)
    class(muli_t), intent(inout) :: this
    this%mean = this%node%approx_value_n(this%get_unit_scale (), &
         this%integrand_id)
    this%start_integrals = this%node%approx_integral (this%get_unit_scale ())
  end subroutine muli_confirm
```

⟨muli_cross_sections.f90⟩≡

```
! This is a dummy for muli_cross_sections_module
module muli_cross_sections_module
end module muli_cross_sections_module
```

⟨muli_sampling.f90⟩≡

```
! This is a dummy for muli_sampling_module
module muli_sampling_module
end module muli_sampling_module
```

Chapter 23

User Plugin Support

Here we collect interface code that enables the user to inject his own code into the WHIZARD workflow. The code uses data types defined elsewhere, and is referenced in the `eval_trees` module.

These are the modules:

user_code_interface Generic support and specific additions.

sf_user Handle user-defined structure functions.

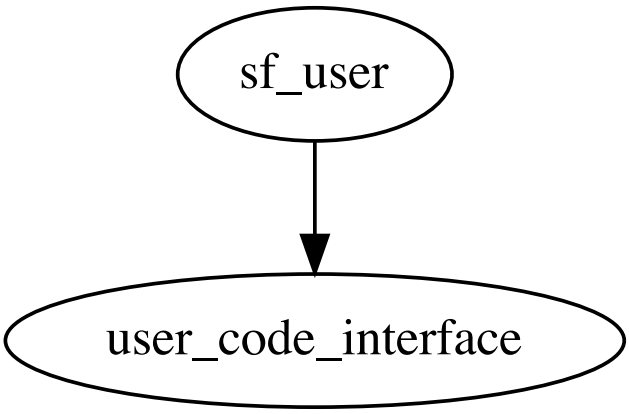

Figure 23.1: Module dependencies in `src/user`.

23.1 User Code Interface

⟨user_code_interface.f90⟩≡

```
⟨File header⟩

module user_code_interface

  use iso_c_binding !NODEP!
⟨Use kinds⟩
⟨Use strings⟩
  use diagnostics
  use c_particles
  use os_interface

⟨Standard module head⟩

⟨User Code: public⟩

⟨User Code: variables⟩

⟨User Code: interfaces⟩

contains

⟨User Code: procedures⟩

end module user_code_interface
```

23.1.1 User Code Management

This data structure globally holds the filehandle of the user-code library:

⟨*User Code: public*⟩≡

```
  public :: has_user_lib
```

⟨*User Code: variables*⟩≡

```
  type(dlaccess_t), save :: user_lib_handle
  logical, save :: has_user_lib = .false.
  type(string_t), save :: user
```

Compile, link and load user code files. Dlopen all user-provided libraries, included the one just compiled (if any).

By default, we are looking for a library `user.so/dylib`. If this is not present, try `user.f90` and compile it. This can be overridden.

In detail: First, compile all sources explicitly specified on the command line. Then collect all libraries specified on the command line, including `user.so` if it was generated. If there is still no code, check for an existing `user.f90` and compile this. Link everything into a `user.la` libtool library. When done, dlopen all libraries that we have so far.

⟨*User Code: public*⟩+≡

```
  public :: user_code_init
```

⟨*User Code: procedures*⟩≡

```
subroutine user_code_init (user_src, user_lib, user_target, rebuild, os_data)
  type(string_t), dimension(:), intent(in) :: user_src, user_lib
  type(string_t), intent(in) :: user_target
  logical, intent(in) :: rebuild
  type(os_data_t), intent(in) :: os_data
  type(string_t) :: user_src_file, user_obj_files, user_lib_file
  logical :: exist
  type(c_funptr) :: fptr
  integer :: i
  call msg_message ("Initializing user code")
  user = user_target;  if (user == "")  user = "user"
  user_obj_files = ""
  inquire (file = char (user) // ".la", exist = exist)
  if (rebuild .or. .not. exist) then
     do i = 1, size (user_src)
        user_src_file = user_src(i) // os_data%fc_src_ext
        inquire (file = char (user_src_file), exist = exist)
        if (exist) then
           call msg_message ("Found user-code source '" &
                // char (user_src_file) // "'.")
           call compile_user_src (user_src_file, user_obj_files)
        else
           call msg_fatal ("User-code source '" // char (user_src_file) &
                // "' not found")
        end if
     end do
     do i = 1, size (user_lib)
        user_lib_file = user_lib(i) // ".la"
        inquire (file = char (user_lib_file), exist = exist)
        if (exist) then
           call msg_message ("Found user-code library '" &
                // char (user_lib_file) // "'.")
        else
           user_lib_file = user_lib(i) // os_data%fc_shrlib_ext
           inquire (file = char (user_lib_file), exist = exist)
           if (exist) then
              call msg_message ("Found user-code library '" &
                   // char (user_lib_file) // "'.")
           else
              call msg_fatal ("User-code library '" // char (user_lib(i)) &
                   // "' not found")
           end if
        end if
        user_obj_files = user_obj_files // " " // user_lib_file
     end do
     if (user_obj_files == "") then
        user_src_file = user // os_data%fc_src_ext
        inquire (file = char (user_src_file), exist = exist)
        if (exist) then
           call msg_message ("Found user-code source '" &
                // char (user_src_file) // "'.")
           call compile_user_src (user_src_file, user_obj_files)
        else
```

```
            call msg_fatal ("User-code source '" // char (user_src_file) &
                 // "' not found")
         end if
      end if
      if (user_obj_files /= "") then
         call link_user (char (user), user_obj_files)
      end if
   end if
   call dlaccess_init &
        (user_lib_handle, var_str (."), &
         user // os_data%fc_shrlib_ext, os_data)
   if (dlaccess_has_error (user_lib_handle)) then
      call msg_error (char (dlaccess_get_error (user_lib_handle)))
      call msg_fatal ("Loading user code library '" // char (user) &
           // ".la' failed")
   else
      call msg_message ("User code library '" // char (user) &
           // ".la' successfully loaded")
      has_user_lib = .true.
   end if
contains
   subroutine compile_user_src (user_src_file, user_obj_files)
     type(string_t), intent(in) :: user_src_file
     type(string_t), intent(inout) :: user_obj_files
     type(string_t) :: basename, ext
     logical :: exist
     basename = user_src_file
     call split (basename, ext, ".", back=.true.)
     if ("." // ext == os_data%fc_src_ext) then
        inquire (file = char (user_src_file), exist = exist)
        if (exist) then
           call msg_message ("Compiling user code file '" &
                // char (user_src_file) // "'")
           call os_compile_shared (basename, os_data)
           user_obj_files = user_obj_files // " " // basename // ".lo"
        else
           call msg_error ("User code file '" // char (user_src_file) &
                // "' not found.")
        end if
     else
        call msg_error ("User code file '" // char (user_src_file) &
             // "' should have file extension '" &
             // char (os_data%fc_src_ext) // "'")
     end if
   end subroutine compile_user_src
   subroutine link_user (user_lib, user_obj_files)
     character(*), intent(in) :: user_lib
     type(string_t), intent(in) :: user_obj_files
     call msg_message ("Linking user code library '" &
          // user_lib // char (os_data%fc_shrlib_ext) // "'")
     call os_link_shared (user_obj_files, var_str (user_lib), os_data)
   end subroutine link_user
end subroutine user_code_init
```

Unload all user-code libraries.

⟨*User Code: public*⟩+≡
```
public :: user_code_final
```

⟨*User Code: procedures*⟩+≡
```
subroutine user_code_final ()
  if (has_user_lib) then
     call dlaccess_final (user_lib_handle)
     has_user_lib = .false.
  end if
end subroutine user_code_final
```

Try to load the possible user-defined procedures from the dlopened libraries. If a procedure is not found, do nothing.

⟨*User Code: public*⟩+≡
```
public :: user_code_find_proc
```

⟨*User Code: procedures*⟩+≡
```
function user_code_find_proc (name) result (fptr)
  type(string_t), intent(in) :: name
  type(c_funptr) :: fptr
  integer :: i
  fptr = c_null_funptr
  !!! Ticket #529
  ! fptr = libmanager_get_c_funptr (char (user), char (name))
  if (.not. c_associated (fptr)) then
     if (has_user_lib) then
        fptr = dlaccess_get_c_funptr (user_lib_handle, name)
        if (.not. c_associated (fptr)) then
           call msg_fatal ("User procedure '" // char (name) // "' not found")
        end if
     else
        call msg_fatal ("User procedure '" // char (name) &
             // "' called without user library (missing -u flag?)")
     end if
  end if
end function user_code_find_proc
```

23.1.2 Interfaces for user-defined functions

The following functions represent user-defined real observables. There may be one or two particles as argument, the result is a real value.

⟨*User Code: public*⟩+≡
```
public :: user_obs_int_unary
public :: user_obs_int_binary
public :: user_obs_real_unary
public :: user_obs_real_binary
```

⟨*User Code: interfaces*⟩≡
```
abstract interface
   function user_obs_int_unary (prt1) result (ival) bind(C)
     use iso_c_binding !NODEP!
```

```
      use c_particles !NODEP!
      type(c_prt_t), intent(in) :: prt1
      integer(c_int) :: ival
   end function user_obs_int_unary
end interface

abstract interface
   function user_obs_int_binary (prt1, prt2) result (ival) bind(C)
      use iso_c_binding !NODEP!
      use c_particles !NODEP!
      type(c_prt_t), intent(in) :: prt1, prt2
      integer(c_int) :: ival
   end function user_obs_int_binary
end interface

abstract interface
   function user_obs_real_unary (prt1) result (rval) bind(C)
      use iso_c_binding !NODEP!
      use c_particles !NODEP!
      type(c_prt_t), intent(in) :: prt1
      real(c_double) :: rval
   end function user_obs_real_unary
end interface

abstract interface
   function user_obs_real_binary (prt1, prt2) result (rval) bind(C)
      use iso_c_binding !NODEP!
      use c_particles !NODEP!
      type(c_prt_t), intent(in) :: prt1, prt2
      real(c_double) :: rval
   end function user_obs_real_binary
end interface
```

The following function takes an array of C-compatible particles and return a single value. The particle array represents a subevent. For C interoperability, we have to use an assumed-size array, hence the array size has to be transferred explicitly.

The cut function returns an `int`, which we should interpret as a logical value (nonzero=true).

⟨*User Code: public*⟩+≡

```
public :: user_cut_fun
```

⟨*User Code: interfaces*⟩+≡

```
abstract interface
   function user_cut_fun (prt, n_prt) result (iflag) bind(C)
      use iso_c_binding !NODEP!
      use c_particles !NODEP!
      type(c_prt_t), dimension(*), intent(in) :: prt
      integer(c_int), intent(in) :: n_prt
      integer(c_int) :: iflag
   end function user_cut_fun
end interface
```

The event-shape function returns a real value.

⟨*User Code: public*⟩+≡
```
public :: user_event_shape_fun
```

⟨*User Code: interfaces*⟩+≡
```
abstract interface
   function user_event_shape_fun (prt, n_prt) result (rval) bind(C)
     use iso_c_binding !NODEP!
     use c_particles !NODEP!
     type(c_prt_t), dimension(*), intent(in) :: prt
     integer(c_int), intent(in) :: n_prt
     real(c_double) :: rval
   end function user_event_shape_fun
end interface
```

23.1.3 Interfaces for user-defined interactions

The following procedure interfaces pertain to user-defined interactions, e.g., spectra or structure functions.

This subroutine retrieves the basic information for setting up the interaction and event generation. All parameters are `intent(inout)`, so we can provide default values. `n_in` and `n_out` are the number of incoming and outgoing particles, respectively. `n_states` is the total number of distinct states that should be generated (counting all states of the incoming particles). `n_col` is the maximal number of color entries a particle can have. `n_dim` is the number of input parameters, i.e., integration dimensions, that the structure function call requires for computing kinematics and matrix elements. `n_var` is the number of variables (e.g., momentum fractions) that the structure function call has to transfer from the kinematics to the dynamics evaluation.

⟨*User Code: public*⟩+≡
```
public :: user_int_info
```

⟨*User Code: interfaces*⟩+≡
```
abstract interface
   subroutine user_int_info (n_in, n_out, n_states, n_col, n_dim, n_var) &
        bind(C)
     use iso_c_binding !NODEP!
     integer(c_int), intent(inout) :: n_in, n_out, n_states, n_col
     integer(c_int), intent(inout) :: n_dim, n_var
   end subroutine user_int_info
end interface
```

This subroutine retrieves the settings for the quantum number mask of a given particle index in the interaction. A nonzero value indicates that the corresponding quantum number is to be ignored. The lock index is the index of a particle that the current particle is related to. The relation applies if quantum numbers of one of the locked particles are summed over. (This is intended for helicity.)

⟨*User Code: public*⟩+≡
```
public :: user_int_mask
```

⟨*User Code: interfaces*⟩+≡

```
abstract interface
   subroutine user_int_mask (i_prt, m_flv, m_hel, m_col, i_lock) bind(C)
     use iso_c_binding !NODEP!
     integer(c_int), intent(in) :: i_prt
     integer(c_int), intent(inout) :: m_flv, m_hel, m_col, i_lock
   end subroutine user_int_mask
end interface
```

This subroutine retrieves the quantum numbers for the particle index `i_prt` in state `i_state`. The `flv` value is a PDG code. The `hel` value is an integer helicity (twice the helicity for fermions). The `col` array is an array which has at most `n_col` entries (see above). All parameters are `intent(inout)` since default values exist. In particular, if a mask entry is set by the previous procedure, the corresponding quantum number is ignored anyway.

⟨*User Code: public*⟩+≡

```
public :: user_int_state
```

⟨*User Code: interfaces*⟩+≡

```
abstract interface
   subroutine user_int_state (i_state, i_prt, flv, hel, col) bind(C)
     use iso_c_binding !NODEP!
     integer(c_int), intent(in) :: i_state, i_prt
     integer(c_int), intent(inout) :: flv, hel
     integer(c_int), dimension(*), intent(inout) :: col
   end subroutine user_int_state
end interface
```

This subroutine takes an array of particle objects with array length `n_in` and an array of input parameters between 0 and 1 with array length `n_dim`. It returns an array of particle objects with array length `n_out`. In addition, it returns an array of internal variables (e.g., momentum fractions, Jacobian) with array length `n_var` that is used by the following subroutine for evaluating the dynamics, i.e., the matrix elements.

⟨*User Code: public*⟩+≡

```
public :: user_int_kinematics
```

⟨*User Code: interfaces*⟩+≡

```
abstract interface
   subroutine user_int_kinematics (prt_in, rval, prt_out, xval) bind(C)
     use iso_c_binding !NODEP!
     use c_particles !NODEP!
     type(c_prt_t), dimension(*), intent(in) :: prt_in
     real(c_double), dimension(*), intent(in) :: rval
     type(c_prt_t), dimension(*), intent(inout) :: prt_out
     real(c_double), dimension(*), intent(out) :: xval
   end subroutine user_int_kinematics
end interface
```

This subroutine takes the array of variables (e.g., momentum fractions) with length `n_var` which has been generated by the previous subroutine and a real variable, the energy scale of the event. It returns an array of matrix-element

values, one entry for each quantum state n_states. The ordering of matrix elements must correspond to the ordering of states.

⟨*User Code: public*⟩+≡
```
public :: user_int_evaluate
```

⟨*User Code: interfaces*⟩+≡
```
abstract interface
   subroutine user_int_evaluate (xval, scale, fval) bind(C)
     use iso_c_binding !NODEP!
     real(c_double), dimension(*), intent(in) :: xval
     real(c_double), intent(in) :: scale
     real(c_double), dimension(*), intent(out) :: fval
   end subroutine user_int_evaluate
end interface
```

23.2 User Plugin for Structure Functions

This variant gives access to user-defined structure functions or spectra.

23.2.1 The module

⟨sf_user.f90⟩≡
```
⟨File header⟩

module sf_user

  use, intrinsic :: iso_c_binding !NODEP!
⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_defs, only: FMT_17
  use diagnostics
  use c_particles
  use lorentz
  use subevents
  use user_code_interface
  use pdg_arrays
  use model_data
  use flavors
  use helicities
  use colors
  use quantum_numbers
  use state_matrices
  use polarizations
  use interactions
  use sf_aux
  use sf_base

⟨Standard module head⟩

⟨SF user: public⟩
```

⟨*SF user: types*⟩

```
contains
```

⟨*SF user: procedures*⟩

```
end module sf_user
```

23.2.2 The user structure function data block

The data block holds the procedure pointers that are used for retrieving static information, as well as the actual evaluation.

⟨*SF user: public*⟩≡

```
public :: user_data_t
```

⟨*SF user: types*⟩≡

```
type, extends(sf_data_t) :: user_data_t
   private
   type(string_t) :: name
   integer :: n_in
   integer :: n_out
   integer :: n_tot
   integer :: n_states
   integer :: n_col
   integer :: n_dim
   integer :: n_var
   integer, dimension(2) :: pdg_in
   class(model_data_t), pointer :: model => null ()
   procedure(user_int_info), nopass, pointer :: info => null ()
   procedure(user_int_mask), nopass, pointer :: mask => null ()
   procedure(user_int_state), nopass, pointer :: state => null ()
   procedure(user_int_kinematics), nopass, pointer :: kinematics => null ()
   procedure(user_int_evaluate), nopass, pointer :: evaluate => null ()
 contains
 ⟨SF user: user data: TBP⟩
end type user_data_t
```

Assign procedure pointers from a dynamically loaded library, given the specified `name`.

We have to distinguish three cases: (1) Both beams are affected, and the user spectrum implements both beams. There is a single data object. (2) Both beams are affected, and the user spectrum applies to single beams. Fill two different objects. (3) A single beam is affected.

⟨*SF User: public*⟩≡

```
public :: sf_user_data_init
```

⟨*SF User: procedures*⟩≡

```
subroutine sf_user_data_init (data, name, flv, model)
  type(sf_user_data_t), intent(out) :: data
  type(string_t), intent(in) :: name
  type(flavor_t), dimension(2), intent(in) :: flv
  class(model_data_t), intent(in), target :: model
  integer(c_int) :: n_in
```

```
    integer(c_int) :: n_out
    integer(c_int) :: n_states
    integer(c_int) :: n_col
    integer(c_int) :: n_dim
    integer(c_int) :: n_var
    data%name = name
    data%pdg_in = flavor_get_pdg (flv)
    data%model => model
    call c_f_procpointer (user_code_find_proc (name // "_info"), data%info)
    call c_f_procpointer (user_code_find_proc (name // "_mask"), data%mask)
    call c_f_procpointer (user_code_find_proc (name // "_state"), data%state)
    call c_f_procpointer &
         (user_code_find_proc (name // "_kinematics"), data%kinematics)
    call c_f_procpointer &
         (user_code_find_proc (name // "_evaluate"), data%evaluate)
    n_in = 1
    n_out = 2
    n_states = 1
    n_col = 2
    n_dim = 1
    n_var = 1
    call data%info (n_in, n_out, n_states, n_col, n_dim, n_var)
    data%n_in = n_in
    data%n_out = n_out
    data%n_tot = n_in + n_out
    data%n_states = n_states
    data%n_col = n_col
    data%n_dim = n_dim
    data%n_var = n_var
  end subroutine sf_user_data_init
```

Output

⟨*SF user: user data: TBP*⟩≡

```
  procedure :: write => user_data_write
```

⟨*SF user: procedures*⟩≡

```
  subroutine user_data_write (data, unit, verbose)
    class(user_data_t), intent(in) :: data
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(1x,A,A)") "User structure function: ", char (data%name)
  end subroutine user_data_write
```

Retrieving contents

⟨*SF User: public*⟩+≡

```
  public :: sf_user_data_get_name
```

⟨*SF User: procedures*⟩+≡

```
  function sf_user_data_get_name (data) result (name)
    type(string_t) :: name
    type(sf_user_data_t), intent(in) :: data
    name = data%name
```

```
end function sf_user_data_get_name
```

⟨*SF User: public*⟩+≡

```
public :: sf_user_data_get_n_in
public :: sf_user_data_get_n_out
public :: sf_user_data_get_n_tot
public :: sf_user_data_get_n_dim
public :: sf_user_data_get_n_var
```

⟨*SF User: procedures*⟩+≡

```
function sf_user_data_get_n_in (data) result (n_in)
  integer :: n_in
  type(sf_user_data_t), intent(in) :: data
  n_in = data%n_in
end function sf_user_data_get_n_in

 function sf_user_data_get_n_out (data) result (n_out)
  integer :: n_out
  type(sf_user_data_t), intent(in) :: data
  n_out = data%n_out
end function sf_user_data_get_n_out

function sf_user_data_get_n_tot (data) result (n_tot)
  integer :: n_tot
  type(sf_user_data_t), intent(in) :: data
  n_tot = data%n_tot
end function sf_user_data_get_n_tot

function sf_user_data_get_n_dim (data) result (n_dim)
  integer :: n_dim
  type(sf_user_data_t), intent(in) :: data
  n_dim = data%n_dim
end function sf_user_data_get_n_dim

function sf_user_data_get_n_var (data) result (n_var)
  integer :: n_var
  type(sf_user_data_t), intent(in) :: data
  n_var = data%n_var
end function sf_user_data_get_n_var
```

23.2.3 The interaction

We fill the interaction by looking up the table of states using the interface functions.

For particles which have a known flavor (as indicated by the mask), we compute the mass squared, so we can use it for the invariant mass of the particle objects.

⟨*SF user: user: TBP*⟩≡

```
procedure :: init => user_init
```

⟨*SF user: procedures*⟩+≡

```
subroutine user_init (sf_int, data)
  !!! JRR: WK please check (#529)
```

```
class(user_t), intent(out) :: sf_int
class(sf_data_t), intent(in), target :: data
type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
integer, dimension(:), allocatable :: hel_lock
integer(c_int) :: m_flv, m_hel, m_col, i_lock
type(quantum_numbers_t), dimension(:), allocatable :: qn
integer(c_int) :: f, h
integer(c_int), dimension(:), allocatable :: c
type(flavor_t) :: flv
type(helicity_t) :: hel
type(color_t) :: col
integer :: i, s
integer(c_int) :: i_prt, i_state
select type (data)
type is (user_data_t)
   allocate (mask (data%n_tot))
   allocate (hel_lock (data%n_tot))
   allocate (qn (data%n_tot))
   allocate (c (data%n_col))
   do i = 1, size (mask)
      i_prt = i
      m_flv = 0;  m_col = 0;  m_hel = 0;  i_lock = 0
      call data%mask (i_prt, m_flv, m_col, m_hel, i_lock)
      mask(i) = &
           quantum_numbers_mask (m_flv /= 0, m_col /= 0, m_hel /= 0)
      hel_lock(i) = i_lock
   end do
   !!! JRR: WK please check (#529)
   !!! Will have to be filled in later.
   ! call sf_int%base_init (mask, &
   !      hel_lock = hel_lock)
   call sf_int%basic_init &
        (data%n_in, 0, data%n_out, mask=mask, &
        hel_lock=hel_lock, set_relations=.true.)
   do s = 1, data%n_states
      i_state = s
      do i = 1, data%n_tot
         i_prt = i
         f = 0;  h = 0;  c = 0
         call data%state (i_state, i_prt, f, h, c)
         if (m_flv == 0) then
            call flv%init (int (f), data%model)
         else
            call flv%init ()
         end if
         if (m_hel == 0) then
            call hel%init (int (h))
         else
            call hel%init ()
         end if
         if (m_col == 0) then
            call color_init_from_array (col, int (c))
         else
            call col%init ()
```

```
            end if
            call qn(i)%init (flv, col, hel)
         end do
         call sf_int%add_state (qn)
      end do
      call sf_int%freeze ()
      !!! JRR: WK please check (#529)
      !!! What has to be inserted here?
      ! call sf_int%set_incoming (??)
      ! call sf_int%set_radiated (??)
      ! call sf_int%set_outgoing (??)
      sf_int%status = SF_INITIAL
    end select
  end subroutine user_init
```

Allocate the interaction record.

⟨*SF user: user data: TBP*⟩+≡

```
  procedure :: allocate_sf_int => user_data_allocate_sf_int
```

⟨*SF user: procedures*⟩+≡

```
  subroutine user_data_allocate_sf_int (data, sf_int)
    class(user_data_t), intent(in) :: data
    class(sf_int_t), intent(inout), allocatable :: sf_int
    allocate (user_t :: sf_int)
  end subroutine user_data_allocate_sf_int
```

The number of parameters is one. We do not generate transverse momentum.

⟨*SF user: user data: TBP*⟩+≡

```
  procedure :: get_n_par => user_data_get_n_par
```

⟨*SF user: procedures*⟩+≡

```
  function user_data_get_n_par (data) result (n)
    class(user_data_t), intent(in) :: data
    integer :: n
    n = data%n_var
  end function user_data_get_n_par
```

Return the outgoing particle PDG codes. This has to be inferred from the states (right?). JRR: WK please check.

⟨*SF user: user data: TBP*⟩+≡

```
  procedure :: get_pdg_out => user_data_get_pdg_out
```

⟨*SF user: procedures*⟩+≡

```
  subroutine user_data_get_pdg_out (data, pdg_out)
    class(user_data_t), intent(in) :: data
    type(pdg_array_t), dimension(:), intent(inout) :: pdg_out
    !!! JRR: WK please check (#529)
    !!! integer :: n, np, i
    !!! n = count (data%mask)
    !!! np = 0;  if (data%has_photon .and. data%mask_photon)  np = 1
    !!! allocate (pdg_out (n + np))
    !!! pdg_out(1:n) = pack ([(i, i = -6, 6)], data%mask)
    !!! if (np == 1)  pdg_out(n+np) = PHOTON
```

```
end subroutine user_data_get_pdg_out
```

23.2.4 The user structure function

For maximal flexibility, user structure functions separate kinematics from dynamics just as the PDF interface does. (JRR: Ok, I guess this now done for all structure functions, right?) We create `c_prt_t` particle objects from the incoming momenta (all other quantum numbers are irrelevant) and call the user-supplied kinematics function to compute the outgoing momenta, along with other variables that will be needed for matrix element evaluation. If known, we use the mass squared computed above. !!! JRR: WK please check (#529) I don't know actually whether this really fits into the setup done by WK.

⟨*SF user: types*⟩+≡

```
!!! JRR: WK please check (#529)
type, extends (sf_int_t) :: user_t
   type(user_data_t), pointer :: data => null ()
   real(default) :: x = 0
   real(default) :: q = 0
 contains
 ⟨SF user: user: TBP⟩
end type user_t
```

Type string: display the name of the user structure function.

⟨*SF user: user: TBP*⟩+≡

```
procedure :: type_string => user_type_string
```

⟨*SF user: procedures*⟩+≡

```
function user_type_string (object) result (string)
  class(user_t), intent(in) :: object
  type(string_t) :: string
  if (associated (object%data)) then
     string = "User structure function: " // object%data%name
  else
     string = "User structure function: [undefined]"
  end if
end function user_type_string
```

Output. Call the interaction routine after displaying the configuration.

⟨*SF user: user: TBP*⟩+≡

```
procedure :: write => user_write
```

⟨*SF user: procedures*⟩+≡

```
subroutine user_write (object, unit, testflag)
  !!! JRR: WK please check (#529)
  !!! Guess these variables do not exist for user strfun (?)
  class(user_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  if (associated (object%data)) then
```

```
      call object%data%write (u)
      if (object%status >= SF_DONE_KINEMATICS) then
         write (u, "(1x,A)")  "SF parameters:"
         write (u, "(3x,A," // FMT_17 // ")")  "x =", object%x
         if (object%status >= SF_FAILED_EVALUATION) then
            write (u, "(3x,A," // FMT_17 // ")")  "Q =", object%q
         end if
      end if
      call object%base_write (u, testflag)
   else
      write (u, "(1x,A)")  "User structure function data: [undefined]"
   end if
end subroutine user_write
```

23.2.5 Kinematics

Set kinematics. If `map` is unset, the r and x values coincide, and the Jacobian $f(r)$ is trivial.

If `map` is set, we are asked to provide an efficient mapping. For the test case, we set $x = r^2$ and consequently $f(r) = 2r$.

⟨*SF user: user: TBP*⟩+≡

```
procedure :: complete_kinematics => user_complete_kinematics
```

⟨*SF user: procedures*⟩+≡

```
subroutine user_complete_kinematics (sf_int, x, f, r, rb, map)
  !!! JRR: WK please check (#529)
  !!! This cannot be correct, as the CIRCE1 structure function has
  !!! twice the variables (2->4 instead of 1->2 splitting)
  class(user_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(out) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(in) :: r
  real(default), dimension(:), intent(in) :: rb
  logical, intent(in) :: map
  real(default) :: xb1
  if (map) then
     call msg_fatal ("User structure function: map flag not supported")
  else
     x(1) = r(1)
     f = 1
  end if
  xb1 = 1 - x(1)
  call sf_int%split_momentum (x, xb1)
  select case (sf_int%status)
  case (SF_DONE_KINEMATICS)
     sf_int%x = x(1)
  case (SF_FAILED_KINEMATICS)
     sf_int%x = 0
     f = 0
  end select
end subroutine user_complete_kinematics
```

Compute inverse kinematics. Here, we start with the x array and compute the "input" r values and the Jacobian f. After this, we can set momenta by the same formula as for normal kinematics.

⟨*SF user: user: TBP*⟩+≡
```
procedure :: inverse_kinematics => user_inverse_kinematics
```

⟨*SF user: procedures*⟩+≡
```
subroutine user_inverse_kinematics (sf_int, x, f, r, rb, map, set_momenta)
  !!! JRR: WK please check (#529)
  !!! This cannot be correct, as the CIRCE1 structure function has
  !!! twice the variables (2->4 instead of 1->2 splitting)
  class(user_t), intent(inout) :: sf_int
  real(default), dimension(:), intent(in) :: x
  real(default), intent(out) :: f
  real(default), dimension(:), intent(out) :: r
  real(default), dimension(:), intent(out) :: rb
  logical, intent(in) :: map
  logical, intent(in), optional :: set_momenta
  real(default) :: xb1
  logical :: set_mom
  set_mom = .false.;  if (present (set_momenta))  set_mom = set_momenta
  if (map) then
     call msg_fatal ("User structure function: map flag not supported")
  else
     r(1) = x(1)
     f = 1
  end if
  xb1 = 1 - x(1)
  rb = 1 - r
  if (set_mom) then
     call sf_int%split_momentum (x, xb1)
     select case (sf_int%status)
     case (SF_DONE_KINEMATICS)
        sf_int%x = x(1)
     case (SF_FAILED_KINEMATICS)
        sf_int%x = 0
        f = 0
     end select
  end if
end subroutine user_inverse_kinematics
```

⟨*SF User: public*⟩+≡
```
public :: interaction_set_kinematics_sf_user
```

⟨*SF User: procedures*⟩+≡
```
subroutine interaction_set_kinematics_sf_user (int, x, r, data)
  type(interaction_t), intent(inout) :: int
  real(default), dimension(:), intent(out) :: x
  real(default), dimension(:), intent(in) :: r
  type(sf_user_data_t), intent(in) :: data
  type(vector4_t), dimension(data%n_in) :: p_in
  type(vector4_t), dimension(data%n_out) :: p_out
  type(c_prt_t), dimension(data%n_in) :: prt_in
  type(c_prt_t), dimension(data%n_out) :: prt_out
```

```
    real(c_double), dimension(data%n_var) :: xval
    call int%get_momenta_sub (p_in, outgoing=.false.)
    prt_in = vector4_to_c_prt (p_in)
    prt_in%type = PRT_INCOMING
    call data%kinematics (prt_in, real (r, c_double), prt_out, xval)
    x = xval
    p_out = vector4_from_c_prt (prt_out)
    call int%set_momenta (p_out, outgoing=.true.)
  end subroutine interaction_set_kinematics_sf_user
```

The matrix-element evaluation may require a scale parameter, therefore this routine is separate. We take the variables computed above together with the event energy scale and call the user function that computes the matrix elements.

⟨*SF user: user: TBP*⟩+≡

```
  procedure :: apply => user_apply
```

⟨*SF user: procedures*⟩+≡

```
  subroutine user_apply (sf_int, scale) !, x, data)
    !!! JRR: WK please check (#529)
    class(user_t), intent(inout) :: sf_int
    real(default), intent(in) :: scale
    real(default), dimension(:), allocatable :: x
    real(c_double), dimension(sf_int%data%n_states) :: fval
    complex(default), dimension(sf_int%data%n_states) :: fc
    associate (data => sf_int%data)
      !!! This is wrong, has to be replaced
      ! allocate (x, size (sf_int%x)))
      x = sf_int%x
      call data%evaluate (real (x, c_double), real (scale, c_double), fval)
      fc = fval
      call sf_int%set_matrix_element (fc)
    end associate
    sf_int%status = SF_EVALUATED
  end subroutine user_apply
```

Chapter 24

BLHA Interface

The code in this chapter implements support for the BLHA record that communicates data for NLO processes.

These are the modules:

blha_config

blha_interface

blha_driver

24.1 Module definition

These modules implement the communication with one loop matrix element providers according to the Binoth LesHouches Accord Interface. The actual matrix element(s) are loaded as a dynamic library.

This module defines the common OLP-interfaces defined through the Binoth Les-Houches accord.

⟨blha_olp_interfaces.f90⟩≡
  ⟨*File header*⟩

```
module blha_olp_interfaces

  use, intrinsic :: iso_c_binding !NODEP!
  use, intrinsic :: iso_fortran_env

  use kinds
```
⟨*Use strings*⟩
```
  use constants
  use numeric_utils, only: vanishes
  use numeric_utils, only: extend_integer_array, crop_integer_array
  use io_units
  use string_utils
  use physics_defs
  use diagnostics
  use os_interface
  use lorentz
```

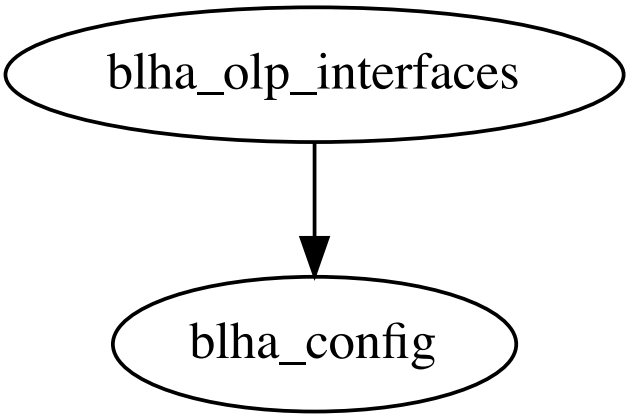

Figure 24.1: Module dependencies in `src/blha`.

```
use sm_qcd
use interactions
use flavors
use model_data
use pdg_arrays, only: is_gluon, is_quark

use prclib_interfaces
use process_libraries
use prc_core_def
use prc_core

use prc_user_defined

use blha_config
```

⟨*Standard module head*⟩

⟨*BLHA OLP interfaces: public*⟩

⟨*BLHA OLP interfaces: public parameters*⟩

⟨*BLHA OLP interfaces: parameters*⟩

⟨*BLHA OLP interfaces: types*⟩

⟨*BLHA OLP interfaces: interfaces*⟩

```
contains

⟨BLHA OLP interfaces: procedures⟩

end module blha_olp_interfaces
```

⟨*BLHA OLP interfaces: public*⟩≡

```
public :: blha_template_t
```

⟨*BLHA OLP interfaces: types*⟩≡

```
type :: blha_template_t
  integer :: I_BORN = 0
  integer :: I_REAL = 1
  integer :: I_LOOP = 2
  integer :: I_SUB = 3
  logical, dimension(0:3) :: compute_component
  logical :: include_polarizations = .false.
  logical :: switch_off_muon_yukawas = .false.
  logical :: use_internal_color_correlations = .true.
  real(default) :: external_top_yukawa = -1._default
  integer :: ew_scheme
contains
⟨BLHA OLP interfaces: blha template: TBP⟩
end type blha_template_t
```

⟨*BLHA OLP interfaces: blha template: TBP*⟩≡

```
procedure :: write => blha_template_write
```

⟨*BLHA OLP interfaces: procedures*⟩≡

```
subroutine blha_template_write (blha_template, unit)
  class(blha_template_t), intent(in) :: blha_template
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u,"(A,4(L1))") "Compute components: ", &
     blha_template%compute_component
  write (u,"(A,L1)") "Include polarizations: ", &
     blha_template%include_polarizations
  write (u,"(A,L1)") "Switch off muon yukawas: ", &
     blha_template%switch_off_muon_yukawas
  write (u,"(A,L1)") "Use internal color correlations: ", &
     blha_template%use_internal_color_correlations
end subroutine blha_template_write
```

⟨*BLHA OLP interfaces: parameters*⟩≡

```
integer, parameter :: I_ALPHA = 1
integer, parameter :: I_GF = 2
integer, parameter :: I_SW2 = 3
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
public :: prc_blha_t
```

⟨*BLHA OLP interfaces: types*⟩+≡

```
type, abstract, extends (prc_user_defined_base_t) :: prc_blha_t
  integer :: n_particles
  integer :: n_hel
  integer :: n_proc
  integer, dimension(:), allocatable :: i_tree, i_sc, i_cc
  integer, dimension(:), allocatable :: i_virt
  integer, dimension(:,:), allocatable :: i_hel
  integer, dimension(:), allocatable :: i_whizard_to_i_olc
  logical, dimension(3) :: ew_parameter_mask
  integer :: sqme_tree_pos
contains
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩

```
end type prc_blha_t
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
public :: blha_driver_t
```

⟨*BLHA OLP interfaces: types*⟩+≡

```
type, abstract, extends (user_defined_driver_t) :: blha_driver_t
  type(string_t) :: contract_file
  type(string_t) :: nlo_suffix
  logical :: include_polarizations = .false.
  logical :: switch_off_muon_yukawas = .false.
  real(default) :: external_top_yukawa = -1.0
  procedure(olp_start),nopass,  pointer :: &
            blha_olp_start => null ()
  procedure(olp_eval), nopass, pointer :: &
            blha_olp_eval => null()
  procedure(olp_info), nopass, pointer :: &
            blha_olp_info => null ()
  procedure(olp_set_parameter), nopass, pointer :: &
            blha_olp_set_parameter => null ()
  procedure(olp_eval2), nopass, pointer :: &
            blha_olp_eval2 => null ()
  procedure(olp_option), nopass, pointer :: &
            blha_olp_option => null ()
  procedure(olp_polvec), nopass, pointer :: &
            blha_olp_polvec => null ()
  procedure(olp_finalize), nopass, pointer :: &
            blha_olp_finalize => null ()
  procedure(olp_print_parameter), nopass, pointer :: &
            blha_olp_print_parameter => null ()
contains
```

⟨*BLHA OLP interfaces: blha driver: TBP*⟩

```
end type blha_driver_t
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
public :: prc_blha_writer_t
```

⟨*BLHA OLP interfaces: types*⟩+≡

```
type, abstract, extends (prc_user_defined_writer_t) :: prc_blha_writer_t
  type(blha_configuration_t) :: blha_cfg
contains
```

```
⟨BLHA OLP interfaces: blha writer: TBP⟩
end type prc_blha_writer_t
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
public :: blha_def_t
```

⟨*BLHA OLP interfaces: types*⟩+≡

```
type, abstract, extends (user_defined_def_t) :: blha_def_t
  type(string_t) :: suffix
contains
⟨BLHA OLP interfaces: blha def: TBP⟩
end type blha_def_t
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
public :: blha_state_t
```

⟨*BLHA OLP interfaces: types*⟩+≡

```
type, abstract, extends (user_defined_state_t) :: blha_state_t
contains
⟨BLHA OLP interfaces: blha state: TBP⟩
end type blha_state_t
```

⟨*BLHA OLP interfaces: blha state: TBP*⟩≡

```
procedure :: reset_new_kinematics => blha_state_reset_new_kinematics
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine blha_state_reset_new_kinematics (object)
  class(blha_state_t), intent(inout) :: object
  object%new_kinematics = .true.
end subroutine blha_state_reset_new_kinematics
```

⟨*BLHA OLP interfaces: public parameters*⟩≡

```
integer, parameter, public :: OLP_PARAMETER_LIMIT = 10
integer, parameter, public :: OLP_MOMENTUM_LIMIT = 50
integer, parameter, public :: OLP_RESULTS_LIMIT = 60
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
public :: olp_start
```

⟨*BLHA OLP interfaces: interfaces*⟩≡

```
interface
  subroutine olp_start (contract_file_name, ierr) bind (C,name = "OLP_Start")
    import
    character(kind = c_char, len = 1), intent(in) :: contract_file_name
    integer(kind = c_int), intent(out) :: ierr
  end subroutine olp_start
end interface
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
public :: olp_eval
```

⟨*BLHA OLP interfaces: interfaces*⟩+≡

```
interface
  subroutine olp_eval (label, momenta, mu, parameters, res) &
       bind (C, name = "OLP_EvalSubProcess")
    import
    integer(kind = c_int), value, intent(in) :: label
    real(kind = c_double), value, intent(in) :: mu
    real(kind = c_double), dimension(OLP_MOMENTUM_LIMIT), intent(in) :: &
         momenta
    real(kind = c_double), dimension(OLP_PARAMETER_LIMIT), intent(in) :: &
         parameters
    real(kind = c_double), dimension(OLP_RESULTS_LIMIT), intent(out) :: res
  end subroutine olp_eval
end interface
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
public :: olp_info
```

⟨*BLHA OLP interfaces: interfaces*⟩+≡

```
interface
  subroutine olp_info (olp_file, olp_version, message) bind(C)
    import
    character(kind = c_char), intent(inout), dimension(15) :: olp_file
    character(kind = c_char), intent(inout), dimension(15) :: olp_version
    character(kind = c_char), intent(inout), dimension(255) :: message
  end subroutine olp_info
end interface
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
public :: olp_set_parameter
```

⟨*BLHA OLP interfaces: interfaces*⟩+≡

```
interface
  subroutine olp_set_parameter &
       (variable_name, real_part, complex_part, success) bind(C)
    import
    character(kind = c_char,len = 1), intent(in) :: variable_name
    real(kind = c_double), intent(in) :: real_part, complex_part
    integer(kind = c_int), intent(out) :: success
  end subroutine olp_set_parameter
end interface
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
public :: olp_eval2
```

⟨*BLHA OLP interfaces: interfaces*⟩+≡

```
interface
  subroutine olp_eval2 (label, momenta, mu, res, acc) bind(C)
    import
    integer(kind = c_int), intent(in) :: label
    real(kind = c_double), intent(in) :: mu
    real(kind = c_double), dimension(OLP_MOMENTUM_LIMIT), intent(in) :: momenta
    real(kind = c_double), dimension(OLP_RESULTS_LIMIT), intent(out) :: res
    real(kind = c_double), intent(out) :: acc
```

```
    end subroutine olp_eval2
  end interface
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
  public :: olp_option
```

⟨*BLHA OLP interfaces: interfaces*⟩+≡

```
  interface
    subroutine olp_option (line, stat) bind(C)
      import
      character(kind = c_char, len=1), intent(in) :: line
      integer(kind = c_int), intent(out) :: stat
    end subroutine
  end interface
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
  public :: olp_polvec
```

⟨*BLHA OLP interfaces: interfaces*⟩+≡

```
  interface
    subroutine olp_polvec (p, q, eps) bind(C)
      import
      real(kind = c_double), dimension(0:3), intent(in) :: p, q
      real(kind = c_double), dimension(0:7), intent(out) :: eps
    end subroutine
  end interface
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
  public :: olp_finalize
```

⟨*BLHA OLP interfaces: interfaces*⟩+≡

```
  interface
    subroutine olp_finalize () bind(C)
      import
    end subroutine olp_finalize
  end interface
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
  public :: olp_print_parameter
```

⟨*BLHA OLP interfaces: interfaces*⟩+≡

```
  interface
    subroutine olp_print_parameter (filename) bind(C)
      import
      character(kind = c_char, len = 1), intent(in) :: filename
    end subroutine olp_print_parameter
  end interface
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
  public :: blha_result_array_size
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
pure function blha_result_array_size (n_part, amp_type) result (rsize)
  integer, intent(in) :: n_part, amp_type
  integer :: rsize
  select case (amp_type)
     case (BLHA_AMP_TREE)
        rsize = 1
     case (BLHA_AMP_LOOP)
        rsize = 4
     case (BLHA_AMP_CC)
        rsize = n_part * (n_part - 1) / 2
     case (BLHA_AMP_SC)
        rsize = 2 * n_part**2
     case default
        rsize = 0
   end select
end function blha_result_array_size
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩≡

```
procedure :: create_momentum_array => prc_blha_create_momentum_array
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
function prc_blha_create_momentum_array (object, p) result (mom)
  class(prc_blha_t), intent(in) :: object
  type(vector4_t), intent(in), dimension(:) :: p
  real(double), dimension(5*object%n_particles) :: mom
  integer :: n, i, k

  n = size (p)
  if (n > 10) call msg_fatal ("Number of external particles exceeds" &
         // "size of BLHA-internal momentum array")

  mom = zero
  k = 1
  do i = 1, n
     mom(k : k + 3) = vector4_get_components (p(i))
     mom(k + 4) = invariant_mass (p(i))
     k = k + 5
  end do
end function prc_blha_create_momentum_array
```

⟨*BLHA OLP interfaces: blha template: TBP*⟩+≡

```
procedure :: init => blha_template_init
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine blha_template_init (template, requires_polarizations, &
     switch_off_muon_yukawas, external_top_yukawa, ew_scheme)
  class(blha_template_t), intent(inout) :: template
  logical, intent(in) :: requires_polarizations, switch_off_muon_yukawas
  real(default), intent(in) :: external_top_yukawa
  type(string_t), intent(in) :: ew_scheme
  template%compute_component = .false.
  template%include_polarizations = requires_polarizations
  template%switch_off_muon_yukawas = switch_off_muon_yukawas
```

```
    template%external_top_yukawa = external_top_yukawa
    template%ew_scheme = ew_scheme_string_to_int (ew_scheme)
  end subroutine blha_template_init
```

⟨*BLHA OLP interfaces: blha template: TBP*⟩+≡

```
  procedure :: set_born => blha_template_set_born
  procedure :: set_real_trees => blha_template_set_real_trees
  procedure :: set_loop => blha_template_set_loop
  procedure :: set_subtraction => blha_template_set_subtraction
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
  subroutine blha_template_set_born (template)
    class(blha_template_t), intent(inout) :: template
    template%compute_component (template%I_BORN) = .true.
  end subroutine blha_template_set_born

  subroutine blha_template_set_real_trees (template)
    class(blha_template_t), intent(inout) :: template
    template%compute_component (template%I_REAL) = .true.
  end subroutine blha_template_set_real_trees

  subroutine blha_template_set_loop (template)
    class(blha_template_t), intent(inout) :: template
    template%compute_component(template%I_LOOP) = .true.
  end subroutine blha_template_set_loop

  subroutine blha_template_set_subtraction (template)
    class(blha_template_t), intent(inout) :: template
    template%compute_component (template%I_SUB) = .true.
  end subroutine blha_template_set_subtraction
```

⟨*BLHA OLP interfaces: blha template: TBP*⟩+≡

```
  procedure :: set_internal_color_correlations &
     => blha_template_set_internal_color_correlations
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
  subroutine blha_template_set_internal_color_correlations (template)
    class(blha_template_t), intent(inout) :: template
    template%use_internal_color_correlations = .true.
  end subroutine blha_template_set_internal_color_correlations
```

⟨*BLHA OLP interfaces: blha template: TBP*⟩+≡

```
  procedure :: get_internal_color_correlations &
     => blha_template_get_internal_color_correlations
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
  pure function blha_template_get_internal_color_correlations (template) &
     result (val)
    logical :: val
    class(blha_template_t), intent(in) :: template
    val = template%use_internal_color_correlations
  end function blha_template_get_internal_color_correlations
```

⟨*BLHA OLP interfaces: blha template: TBP*⟩+≡
```
procedure :: compute_born => blha_template_compute_born
procedure :: compute_real_trees => blha_template_compute_real_trees
procedure :: compute_loop => blha_template_compute_loop
procedure :: compute_subtraction => blha_template_compute_subtraction
```

⟨*BLHA OLP interfaces: procedures*⟩+≡
```
pure function blha_template_compute_born (template) result (val)
  class(blha_template_t), intent(in) :: template
  logical :: val
  val = template%compute_component (template%I_BORN)
end function blha_template_compute_born

pure function blha_template_compute_real_trees (template) result (val)
  class(blha_template_t), intent(in) :: template
  logical :: val
  val = template%compute_component (template%I_REAL)
end function blha_template_compute_real_trees

pure function blha_template_compute_loop (template) result (val)
  class(blha_template_t), intent(in) :: template
  logical :: val
  val = template%compute_component (template%I_LOOP)
end function blha_template_compute_loop

pure function blha_template_compute_subtraction (template) result (val)
  class(blha_template_t), intent(in) :: template
  logical :: val
  val = template%compute_component (template%I_SUB)
end function blha_template_compute_subtraction
```

⟨*BLHA OLP interfaces: blha template: TBP*⟩+≡
```
procedure :: check => blha_template_check
```

⟨*BLHA OLP interfaces: procedures*⟩+≡
```
function blha_template_check (template) result (val)
  class(blha_template_t), intent(in) :: template
  logical :: val
  val = count (template%compute_component) == 1
end function blha_template_check
```

⟨*BLHA OLP interfaces: blha template: TBP*⟩+≡
```
procedure :: reset => blha_template_reset
```

⟨*BLHA OLP interfaces: procedures*⟩+≡
```
subroutine blha_template_reset (template)
  class(blha_template_t), intent(inout) :: template
  template%compute_component = .false.
end subroutine blha_template_reset
```

⟨*BLHA OLP interfaces: blha writer: TBP*⟩≡
```
procedure :: write => prc_blha_writer_write
```

⟨*BLHA OLP interfaces: procedures*⟩+≡
```
subroutine prc_blha_writer_write (writer, unit)
  class(prc_blha_writer_t), intent(in) :: writer
  integer, intent(in) :: unit
  write (unit, "(1x,A)")  char (writer%get_process_string ())
end subroutine prc_blha_writer_write
```

⟨*BLHA OLP interfaces: blha writer: TBP*⟩+≡
```
procedure :: get_process_string => prc_blha_writer_get_process_string
```

⟨*BLHA OLP interfaces: procedures*⟩+≡
```
function prc_blha_writer_get_process_string (writer) result (s_proc)
  class(prc_blha_writer_t), intent(in) :: writer
  type(string_t) :: s_proc
  s_proc = var_str ("")
end function prc_blha_writer_get_process_string
```

⟨*BLHA OLP interfaces: blha writer: TBP*⟩+≡
```
procedure :: get_n_proc => prc_blha_writer_get_n_proc
```

⟨*BLHA OLP interfaces: procedures*⟩+≡
```
function prc_blha_writer_get_n_proc (writer) result (n_proc)
  class(prc_blha_writer_t), intent(in) :: writer
  integer :: n_proc
  n_proc = blha_configuration_get_n_proc (writer%blha_cfg)
end function prc_blha_writer_get_n_proc
```

⟨*BLHA OLP interfaces: blha driver: TBP*⟩≡
```
procedure(blha_driver_set_GF), deferred :: &
   set_GF
```

⟨*BLHA OLP interfaces: interfaces*⟩+≡
```
abstract interface
  subroutine blha_driver_set_GF (driver, GF)
    import
    class(blha_driver_t), intent(inout) :: driver
    real(default), intent(in) :: GF
  end subroutine blha_driver_set_GF
end interface
```

⟨*BLHA OLP interfaces: blha driver: TBP*⟩+≡
```
procedure(blha_driver_set_alpha_s), deferred :: &
   set_alpha_s
```

⟨*BLHA OLP interfaces: interfaces*⟩+≡
```
abstract interface
  subroutine blha_driver_set_alpha_s (driver, alpha_s)
     import
     class(blha_driver_t), intent(in) :: driver
     real(default), intent(in) :: alpha_s
  end subroutine blha_driver_set_alpha_s
end interface
```

⟨*BLHA OLP interfaces: blha driver: TBP*⟩+≡
```
procedure(blha_driver_set_weinberg_angle), deferred :: &
   set_weinberg_angle
```
⟨*BLHA OLP interfaces: interfaces*⟩+≡
```
abstract interface
  subroutine blha_driver_set_weinberg_angle (driver, sw2)
    import
    class(blha_driver_t), intent(inout) :: driver
    real(default), intent(in) :: sw2
  end subroutine blha_driver_set_weinberg_angle
end interface
```

⟨*BLHA OLP interfaces: blha driver: TBP*⟩+≡
```
procedure(blha_driver_set_alpha_qed), deferred :: set_alpha_qed
```
⟨*BLHA OLP interfaces: interfaces*⟩+≡
```
abstract interface
  subroutine blha_driver_set_alpha_qed (driver, alpha)
    import
    class(blha_driver_t), intent(inout) :: driver
    real(default), intent(in) :: alpha
  end subroutine blha_driver_set_alpha_qed
end interface
```

⟨*BLHA OLP interfaces: blha driver: TBP*⟩+≡
```
procedure(blha_driver_print_alpha_s), deferred :: &
   print_alpha_s
```
⟨*BLHA OLP interfaces: interfaces*⟩+≡
```
abstract interface
  subroutine blha_driver_print_alpha_s (object)
    import
    class(blha_driver_t), intent(in) :: object
  end subroutine blha_driver_print_alpha_s
end interface
```

⟨*BLHA OLP interfaces: public*⟩+≡
```
public :: parameter_error_message
```
⟨*BLHA OLP interfaces: procedures*⟩+≡
```
subroutine parameter_error_message (par)
   type(string_t), intent(in) :: par
   type(string_t) :: message
   message = "Setting of parameter " // par &
      // "failed. This happens because the chosen " &
      // "EWScheme in the BLHA file does not fit " &
      // "your parameter choice"
   call msg_fatal (char (message))
end subroutine parameter_error_message
```

⟨*BLHA OLP interfaces: blha driver: TBP*⟩+≡
```
procedure :: set_mass_and_width => blha_driver_set_mass_and_width
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine blha_driver_set_mass_and_width &
       (driver, i_pdg, mass, width)
  class(blha_driver_t), intent(inout) :: driver
  integer, intent(in) :: i_pdg
  real(default), intent(in), optional :: mass
  real(default), intent(in), optional :: width
  type(string_t) :: buf
  character(kind=c_char,len=20) :: c_string
  integer :: ierr
  if (present (mass)) then
     buf = 'mass(' // str (abs(i_pdg)) // ')'
     c_string = char(buf) // c_null_char
     call driver%blha_olp_set_parameter &
              (c_string, dble(mass), 0._double, ierr)
     if (ierr == 0) then
        buf = "BLHA driver: Attempt to set mass of particle " // &
              str (abs(i_pdg)) // "failed"
        call msg_fatal (char(buf))
     end if
  end if
  if (present (width)) then
     buf = 'width(' // str (abs(i_pdg)) // ')'
     c_string = char(buf)//c_null_char
     call driver%blha_olp_set_parameter &
              (c_string, dble(width), 0._double, ierr)
     if (ierr == 0) then
        buf = "BLHA driver: Attempt to set width of particle " // &
              str (abs(i_pdg)) // "failed"
        call msg_fatal (char(buf))
     end if
  end if
end subroutine blha_driver_set_mass_and_width
```

⟨*BLHA OLP interfaces: blha driver: TBP*⟩+≡

```
procedure(blha_driver_init_dlaccess_to_library), deferred :: &
  init_dlaccess_to_library
```

⟨*BLHA OLP interfaces: interfaces*⟩+≡

```
abstract interface
  subroutine blha_driver_init_dlaccess_to_library &
     (object, os_data, dlaccess, success)
    import
    class(blha_driver_t), intent(in) :: object
    type(os_data_t), intent(in) :: os_data
    type(dlaccess_t), intent(out) :: dlaccess
    logical, intent(out) :: success
  end subroutine blha_driver_init_dlaccess_to_library
end interface
```

⟨*BLHA OLP interfaces: blha driver: TBP*⟩+≡

```
procedure :: load => blha_driver_load
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine blha_driver_load (object, os_data, success)
  class(blha_driver_t), intent(inout) :: object
  type(os_data_t), intent(in) :: os_data
  logical, intent(out) :: success
  type(dlaccess_t) :: dlaccess
  type(c_funptr) :: c_fptr
  logical :: init_success

  call object%init_dlaccess_to_library (os_data, dlaccess, init_success)

  c_fptr = dlaccess_get_c_funptr (dlaccess, var_str ("OLP_Start"))
  call c_f_procpointer (c_fptr, object%blha_olp_start)
  call check_for_error (var_str ("OLP_Start"))

  c_fptr = dlaccess_get_c_funptr (dlaccess, var_str ("OLP_EvalSubProcess"))
  call c_f_procpointer (c_fptr, object%blha_olp_eval)
  call check_for_error (var_str ("OLP_EvalSubProcess"))

  c_fptr = dlaccess_get_c_funptr (dlaccess, var_str ("OLP_Info"))
  call c_f_procpointer (c_fptr, object%blha_olp_info)
  call check_for_error (var_str ("OLP_Info"))

  c_fptr = dlaccess_get_c_funptr (dlaccess, var_str ("OLP_SetParameter"))
  call c_f_procpointer (c_fptr, object%blha_olp_set_parameter)
  call check_for_error (var_str ("OLP_SetParameter"))

  c_fptr = dlaccess_get_c_funptr (dlaccess, var_str ("OLP_EvalSubProcess2"))
  call c_f_procpointer (c_fptr, object%blha_olp_eval2)
  call check_for_error (var_str ("OLP_EvalSubProcess2"))

  !!! The following three functions are not implemented in OpenLoops.
  !!! In another BLHA provider, they need to be implemented separately.

  !!! c_fptr = dlaccess_get_c_funptr (dlaccess, var_str ("OLP_Option"))
  !!! call c_f_procpointer (c_fptr, object%blha_olp_option)
  !!! call check_for_error (var_str ("OLP_Option"))

  !!! c_fptr = dlaccess_get_c_funptr (dlaccess, var_str ("OLP_Polvec"))
  !!! call c_f_procpointer (c_fptr, object%blha_olp_polvec)
  !!! call check_for_error (var_str ("OLP_Polvec"))

  !!! c_fptr = dlaccess_get_c_funptr (dlaccess, var_str ("OLP_Finalize"))
  !!! call c_f_procpointer (c_fptr, object%blha_olp_finalize)
  !!! call check_for_error (var_str ("OLP_Finalize"))

  c_fptr = dlaccess_get_c_funptr (dlaccess, var_str ("OLP_PrintParameter"))
  call c_f_procpointer (c_fptr, object%blha_olp_print_parameter)
  call check_for_error (var_str ("OLP_PrintParameter"))

  success = .true.
  contains
    subroutine check_for_error (function_name)
      type(string_t), intent(in) :: function_name
```

```
      if (dlaccess_has_error (dlaccess)) &
         call msg_fatal (char ("Loading of " // function_name // " failed!"))
   end subroutine check_for_error
end subroutine blha_driver_load
```

⟨*BLHA OLP interfaces: parameters*⟩+≡

```
integer, parameter :: LEN_MAX_FLAVOR_STRING = 100
integer, parameter :: N_MAX_FLAVORS = 100
```

⟨*BLHA OLP interfaces: blha driver: TBP*⟩+≡

```
procedure :: read_contract_file => blha_driver_read_contract_file
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine blha_driver_read_contract_file (driver, flavors, &
     amp_type, flv_index, label, helicities)
  class(blha_driver_t), intent(inout) :: driver
  integer, intent(in), dimension(:,:) :: flavors
  integer, intent(out), dimension(:), allocatable :: amp_type, flv_index, label
  integer, intent(out), dimension(:,:) :: helicities
  integer :: unit, filestat
  character(len=LEN_MAX_FLAVOR_STRING) :: rd_line
  logical :: read_flavor, give_warning
  integer :: label_count, i_flv
  integer :: i_hel, n_in
  integer :: i_next, n_entries
  integer, dimension(size(flavors, 1) + 2) :: i_array
  integer, dimension(size(flavors, 1) + 2) :: hel_array
  integer, parameter :: NO_NUMBER = -1000
  integer, parameter :: PROC_NOT_FOUND = -1001
  integer, parameter :: list_incr = 50
  integer :: n_found
  allocate (amp_type (N_MAX_FLAVORS), flv_index (N_MAX_FLAVORS), &
         label (N_MAX_FLAVORS))
  amp_type = -1; flv_index = -1; label = -1
  helicities = 0
  n_in = size (helicities, dim = 2)
  n_entries = size (flavors, 1) + 2
  unit = free_unit ()
  open (unit, file = char (driver%contract_file), status="old")
  read_flavor = .false.
  label_count = 1
  i_hel = 0
  n_found = 0
  give_warning = .false.
  do
    read (unit, "(A)", iostat = filestat) rd_line
    if (filestat == iostat_end) then
       exit
    else
       if (rd_line(1:13) == 'AmplitudeType') then
          if (i_hel > 2 * n_in - 1) i_hel = 0
          i_next = find_next_word_index (rd_line, 13)
          if (label_count > size (amp_type)) &
               call extend_integer_array (amp_type, list_incr)
          if (rd_line(i_next : i_next + 4) == 'Loop') then
```

```
            amp_type(label_count) = BLHA_AMP_LOOP
         else if (rd_line(i_next : i_next + 4) == 'Tree') then
            amp_type(label_count) = BLHA_AMP_TREE
         else if (rd_line(i_next : i_next + 6) == 'ccTree') then
            amp_type(label_count) = BLHA_AMP_CC
         else if (rd_line(i_next : i_next + 6) == 'scTree' .or. &
              rd_line(i_next : i_next + 14) == 'sctree_polvect') then
            amp_type(label_count) = BLHA_AMP_SC
         else
            call msg_fatal ("AmplitudeType present but AmpType not known!")
         end if
         read_flavor = .true.
      else if (read_flavor) then
         i_array = create_flavor_string (rd_line, n_entries)
         if (driver%include_polarizations) then
            hel_array = create_helicity_string (rd_line, n_entries)
            call check_helicity_array (hel_array, n_entries, n_in)
         else
            hel_array = 0
         end if
         if (.not. all (i_array == PROC_NOT_FOUND)) then
            do i_flv = 1, size (flavors, 2)
                if (all (i_array (1 : n_entries - 2) == flavors (:,i_flv))) then
                   if (label_count > size (label)) &
                        call extend_integer_array (label, list_incr)
                   label(label_count) = i_array (n_entries)
                   if (label_count > size (flv_index)) &
                        call extend_integer_array (flv_index, list_incr)
                   flv_index (label_count) = i_flv + i_hel
                   if (driver%include_polarizations) then
                      helicities (label(label_count), :) = hel_array (1:n_in)
                      i_hel = i_hel + 1
                   end if
                   n_found = n_found + 1
                   label_count = label_count + 1
                   exit
                end if
            end do
            give_warning = .false.
         else
            give_warning = .true.
         end if
         read_flavor = .false.
      end if
   end if
end do
call crop_integer_array (amp_type, label_count-1)
if (n_found == 0) then
   call msg_fatal ("The desired process has not been found ",  &
         [var_str ("by the OLP-Provider. Maybe the value of alpha_power "), &
          var_str ("or alphas_power does not correspond to the process. "), &
          var_str ("If you are using OpenLoops, you can set the option "), &
          var_str ("openloops_verbosity to a value larger than 1 to obtain "),  &
          var_str ("more information")])
```

```
  else if (give_warning) then
     call msg_warning ("Some processes have not been found in the OLC file.", &
          [var_str ("This is because these processes do not fit the required "), &
           var_str ("coupling alpha_power and alphas_power. Be aware that the "), &
           var_str ("results of this calculation are not necessarily an accurate "), &
           var_str ("description of the physics of interest.")])
  end if
  close(unit)

contains

  function create_flavor_string (s, n_entries) result (i_array)
    character(len=LEN_MAX_FLAVOR_STRING), intent(in) :: s
    integer, intent(in) :: n_entries
    integer, dimension(n_entries) :: i_array
    integer :: k, current_position
    integer :: i_entry
    k = 1; current_position = 1
    do
       if (current_position > LEN_MAX_FLAVOR_STRING) &
          call msg_fatal ("Read OLC File: Current position exceeds maximum value")
       if (s(current_position:current_position) /= " ") then
          call create_flavor (s, i_entry, current_position)
          if (i_entry /= NO_NUMBER .and. i_entry /= PROC_NOT_FOUND) then
             i_array(k) = i_entry
             k = k + 1
             if (k > n_entries) then
                return
             else
                call increment_current_position (s, current_position)
             end if
          else if (i_entry == PROC_NOT_FOUND) then
             i_array = PROC_NOT_FOUND
             return
          else
             call increment_current_position (s, current_position)
          end if
       else
          call increment_current_position (s, current_position)
       end if
    end do
  end function create_flavor_string

  function create_helicity_string (s, n_entries) result (hel_array)
    character(len = LEN_MAX_FLAVOR_STRING), intent(in) :: s
    integer, intent(in) :: n_entries
    integer, dimension(n_entries) :: hel_array
    integer :: k, current_position
    integer :: hel
    k = 1; current_position = 1
    do
       if (current_position > LEN_MAX_FLAVOR_STRING) &
          call msg_fatal ("Read OLC File: Current position exceeds maximum value")
       if (s(current_position:current_position) /= " ") then
```

```
        call create_helicity (s, hel, current_position)
        if (hel >= -1 .and. hel <= 1) then
           hel_array(k) = hel
           k = k + 1
           if (k > n_entries) then
              return
           else
              call increment_current_position (s, current_position)
           end if
        else
           call increment_current_position (s, current_position)
        end if
     else
        call increment_current_position (s, current_position)
     end if
  end do
end function create_helicity_string

subroutine increment_current_position (s, current_position)
  character(len = LEN_MAX_FLAVOR_STRING), intent(in) :: s
  integer, intent(inout) :: current_position
  current_position = find_next_word_index (s, current_position)
end subroutine increment_current_position

subroutine get_next_buffer (s, current_position, buf, last_buffer_index)
  character(len = LEN_MAX_FLAVOR_STRING), intent(in) :: s
  integer, intent(inout) :: current_position
  character(len = 10), intent(out) :: buf
  integer, intent(out) :: last_buffer_index
  integer :: i
  i = 1; buf = ""
  do
     if (s(current_position:current_position) /= " ") then
        buf(i:i) = s(current_position:current_position)
        i = i + 1; current_position = current_position + 1
     else
        exit
     end if
  end do
  last_buffer_index = i
end subroutine get_next_buffer

function is_particle_buffer (buf, i) result (valid)
  logical :: valid
  character(len = 10), intent(in) :: buf
  integer, intent(in) :: i
  valid = (buf(1 : i - 1) /= "->" .and. buf(1 : i - 1) /= "|" &
     .and. buf(1 : i - 1) /= "Process")
end function is_particle_buffer

subroutine create_flavor (s, i_particle, current_position)
  character(len=LEN_MAX_FLAVOR_STRING), intent(in) :: s
  integer, intent(out) :: i_particle
  integer, intent(inout) :: current_position
```

```
  character(len=10) :: buf
  integer :: i, last_buffer_index
  call get_next_buffer (s, current_position, buf, last_buffer_index)
  i = last_buffer_index
  if (is_particle_buffer (buf, i)) then
     call strip_helicity (buf, i)
     i_particle = read_ival (var_str (buf(1 : i - 1)))
  else if (buf(1 : i - 1) == "Process") then
     i_particle = PROC_NOT_FOUND
  else
     i_particle = NO_NUMBER
  end if
end subroutine create_flavor

subroutine create_helicity (s, helicity, current_position)
  character(len = LEN_MAX_FLAVOR_STRING), intent(in) :: s
  integer, intent(out) :: helicity
  integer, intent(inout) :: current_position
  character(len = 10) :: buf
  integer :: i, last_buffer_index
  logical :: success
  call get_next_buffer (s, current_position, buf, last_buffer_index)
  i = last_buffer_index
  if (is_particle_buffer (buf, i)) then
     call strip_flavor (buf, i, helicity, success)
  else
     helicity = 0
  end if
end subroutine create_helicity

subroutine strip_helicity (buf, i)
  character(len = 10), intent(in) :: buf
  integer, intent(inout) :: i
  integer :: i_last
  i_last = i - 1
  if (i_last < 4) return
  if (buf(i_last - 2 : i_last) == "(1)") then
     i = i - 3
  else if (buf(i_last - 3 : i_last) == "(-1)") then
     i = i - 4
  end if
end subroutine strip_helicity

subroutine strip_flavor (buf, i, helicity, success)
  character(len = 10), intent(in) :: buf
  integer, intent(in) :: i
  integer, intent(out) :: helicity
  logical, intent(out) :: success
  integer :: i_last
  i_last = i - 1
  helicity = 0
  if (i_last < 4) return
  if (buf(i_last - 2 : i_last) == "(1)") then
     helicity = 1
```

```
      success = .true.
    else if (buf(i_last - 3 : i_last) == "(-1)") then
      helicity = -1
      success = .true.
    else
      success = .false.
    end if
  end subroutine strip_flavor

  function find_next_word_index (word, i_start) result (i_next)
    character(len = LEN_MAX_FLAVOR_STRING), intent(in) :: word
    integer, intent(in) :: i_start
    integer :: i_next
    i_next = i_start + 1
    do
      if (word(i_next : i_next) /= " ") then
        exit
      else
        i_next = i_next + 1
      end if
      if (i_next > LEN_MAX_FLAVOR_STRING) &
          call msg_fatal ("Find next word: line limit exceeded")
    end do
  end function find_next_word_index

  subroutine check_helicity_array (hel_array, n_entries, n_in)
    integer, intent(in), dimension(:) :: hel_array
    integer, intent(in) :: n_entries, n_in
    integer :: n_particles, i
    logical :: valid
    n_particles = n_entries - 2
    !!! only allow polarisations for incoming fermions for now
    valid = all (hel_array (n_in + 1 : n_particles) == 0)
    do i = 1, n_in
      valid = valid .and. (hel_array(i) == 1 .or. hel_array(i) == -1)
    end do
    if (.not. valid) &
      call msg_fatal ("Invalid helicities encountered!")
  end subroutine check_helicity_array

end subroutine blha_driver_read_contract_file
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
procedure :: set_alpha_qed => prc_blha_set_alpha_qed
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine prc_blha_set_alpha_qed (object, model)
  class(prc_blha_t), intent(inout) :: object
  type(model_data_t), intent(in), target :: model
  real(default) :: alpha

  alpha = one / model%get_real (var_str ('alpha_em_i'))

  select type (driver => object%driver)
```

```
      class is (blha_driver_t)
         call driver%set_alpha_qed (alpha)
      end select
    end subroutine prc_blha_set_alpha_qed
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
    procedure :: set_GF => prc_blha_set_GF
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
    subroutine prc_blha_set_GF (object, model)
      class(prc_blha_t), intent(inout) :: object
      type(model_data_t), intent(in), target :: model
      real(default) :: GF

      GF = model%get_real (var_str ('GF'))
      select type (driver => object%driver)
      class is (blha_driver_t)
         call driver%set_GF (GF)
      end select
    end subroutine prc_blha_set_GF
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
    procedure :: set_weinberg_angle => prc_blha_set_weinberg_angle
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
    subroutine prc_blha_set_weinberg_angle (object, model)
      class(prc_blha_t), intent(inout) :: object
      type(model_data_t), intent(in), target :: model
      real(default) :: sw2

      sw2 = model%get_real (var_str ('sw2'))
      select type (driver => object%driver)
      class is (blha_driver_t)
        call driver%set_weinberg_angle (sw2)
      end select
    end subroutine prc_blha_set_weinberg_angle
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
    procedure :: set_electroweak_parameters => &
       prc_blha_set_electroweak_parameters
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
    subroutine prc_blha_set_electroweak_parameters (object, model)
       class(prc_blha_t), intent(inout) :: object
       type(model_data_t), intent(in), target :: model
       if (count (object%ew_parameter_mask) == 0) then
          call msg_fatal ("Cannot decide EW parameter setting: No scheme set!")
       else if (count (object%ew_parameter_mask) > 1) then
          call msg_fatal ("Cannot decide EW parameter setting: More than one scheme set!")
       end if
       if (object%ew_parameter_mask (I_ALPHA)) call object%set_alpha_qed (model)
       if (object%ew_parameter_mask (I_GF)) call object%set_GF (model)
       if (object%ew_parameter_mask (I_SW2)) call object%set_weinberg_angle (model)
    end subroutine prc_blha_set_electroweak_parameters
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
procedure :: read_contract_file => prc_blha_read_contract_file
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine prc_blha_read_contract_file (object, flavors)
  class(prc_blha_t), intent(inout) :: object
  integer, intent(in), dimension(:,:) :: flavors
  integer, dimension(:), allocatable :: amp_type, flv_index, label
  integer, dimension(:,:), allocatable :: helicities
  integer :: i_proc
  allocate (helicities (N_MAX_FLAVORS, object%data%n_in))
  select type (driver => object%driver)
  class is (blha_driver_t)
     call driver%read_contract_file (flavors, amp_type, flv_index, &
          label, helicities)
  end select
  object%n_proc = count (amp_type >= 0)
  do i_proc = 1, object%n_proc
     if (amp_type (i_proc) < 0) exit
     select case (amp_type (i_proc))
     case (BLHA_AMP_TREE)
        if (allocated (object%i_tree)) then
           object%i_tree(flv_index(i_proc)) = label(i_proc)
        else
           call msg_fatal ("Tree matrix element present, &
                &but neither Born nor real indices are allocated!")
        end if
     case (BLHA_AMP_CC)
        if (allocated (object%i_cc)) then
           object%i_cc(flv_index(i_proc)) = label(i_proc)
        else
           call msg_fatal ("Color-correlated matrix element present, &
                &but cc-indices are not allocated!")
        end if
     case (BLHA_AMP_SC)
        if (allocated (object%i_sc)) then
           object%i_sc(flv_index(i_proc)) = label(i_proc)
        else
           call msg_fatal ("Spin-correlated matrix element present, &
                &but sc-indices are not allocated!")
        end if
     case (BLHA_AMP_LOOP)
        if (allocated (object%i_virt)) then
           object%i_virt(flv_index(i_proc)) = label(i_proc)
        else
           call msg_fatal ("Loop matrix element present, &
                &but virt-indices are not allocated!")
        end if
     case default
        call msg_fatal ("Undefined amplitude type")
     end select
     if (allocated (object%i_hel)) &
        object%i_hel (i_proc, :) = helicities (label(i_proc), :)
  end do
end subroutine prc_blha_read_contract_file
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
procedure :: print_parameter_file => prc_blha_print_parameter_file
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine prc_blha_print_parameter_file (object, i_component)
  class(prc_blha_t), intent(in) :: object
  integer, intent(in) :: i_component
  type(string_t) :: filename

  select type (def => object%def)
  class is (blha_def_t)
     filename = def%basename // '_' // str (i_component) // '.olp_parameters'
  end select
  select type (driver => object%driver)
  class is (blha_driver_t)
     call driver%blha_olp_print_parameter (char(filename)//c_null_char)
  end select
end subroutine prc_blha_print_parameter_file
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
procedure :: compute_amplitude => prc_blha_compute_amplitude
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
function prc_blha_compute_amplitude &
     (object, j, p, f, h, c, fac_scale, ren_scale, alpha_qcd_forced, &
     core_state)  result (amp)
  class(prc_blha_t), intent(in) :: object
  integer, intent(in) :: j
  type(vector4_t), dimension(:), intent(in) :: p
  integer, intent(in) :: f, h, c
  real(default), intent(in) :: fac_scale, ren_scale
  real(default), intent(in), allocatable :: alpha_qcd_forced
  class(prc_core_state_t), intent(inout), allocatable, optional :: core_state
  complex(default) :: amp
  select type (core_state)
  class is (blha_state_t)
    core_state%alpha_qcd = object%qcd%alpha%get (fac_scale)
  end select
  amp = zero
end function prc_blha_compute_amplitude
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
procedure :: init_blha => prc_blha_init_blha
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine prc_blha_init_blha (object, blha_template, n_in, &
       n_particles, n_flv, n_hel)
  class(prc_blha_t), intent(inout) :: object
  type(blha_template_t), intent(in) :: blha_template
  integer, intent(in) :: n_in, n_particles, n_flv, n_hel
  object%n_particles = n_particles
  object%n_flv = n_flv
  object%n_hel = n_hel
```

```
    if (blha_template%compute_loop ()) then
       if (blha_template%include_polarizations) then
          allocate (object%i_virt (n_flv * n_hel), &
               object%i_cc (n_flv * n_hel))
          if (blha_template%use_internal_color_correlations) then
             allocate (object%i_hel (n_flv * n_in * n_hel * 2, n_in))
          else
             allocate (object%i_hel (n_flv * n_in * n_hel, n_in))
          end if
       else
          allocate (object%i_virt (n_flv), object%i_cc (n_flv))
       end if
       object%i_virt = 0
       object%i_cc = 0
    else if (blha_template%compute_subtraction ()) then
       if (blha_template%include_polarizations) then
          allocate (object%i_tree (n_flv * n_hel), &
               object%i_cc (n_flv * n_hel), &
               object%i_sc (n_flv * n_hel))
       else
          allocate (object%i_tree (n_flv), object%i_cc (n_flv) , &
               object%i_sc (n_flv))
       end if
       object%i_tree = 0
       object%i_cc = 0
       object%i_sc = 0
    else if (blha_template%compute_real_trees () .or. blha_template%compute_born ()) then
       if (blha_template%include_polarizations) allocate (object%i_hel (n_hel, n_in))
       allocate (object%i_tree (n_flv * n_hel))
       object%i_tree = 0
    end if

    call object%init_ew_parameters (blha_template%ew_scheme)

    select type (driver => object%driver)
    class is (blha_driver_t)
       driver%include_polarizations = blha_template%include_polarizations
       driver%switch_off_muon_yukawas = blha_template%switch_off_muon_yukawas
       driver%external_top_yukawa = blha_template%external_top_yukawa
    end select
  end subroutine prc_blha_init_blha
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
  procedure :: set_mass_and_width => prc_blha_set_mass_and_width
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
  subroutine prc_blha_set_mass_and_width (object, i_pdg, mass, width)
    class(prc_blha_t), intent(inout) :: object
    integer, intent(in) :: i_pdg
    real(default), intent(in) :: mass, width
    select type (driver => object%driver)
    class is (blha_driver_t)
       call driver%set_mass_and_width (i_pdg, mass, width)
    end select
```

```
end subroutine prc_blha_set_mass_and_width
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
procedure :: set_particle_properties => prc_blha_set_particle_properties
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine prc_blha_set_particle_properties (object, model)
  class(prc_blha_t), intent(inout) :: object
  class(model_data_t), intent(in), target :: model
  integer :: i, i_pdg
  type(flavor_t) :: flv
  real(default) :: mass, width
  integer :: ierr
  real(default) :: top_yukawa
  do i = 1, OLP_N_MASSIVE_PARTICLES
     i_pdg = OLP_MASSIVE_PARTICLES(i)
     if (i_pdg < 0) cycle
     call flv%init (i_pdg, model)
     mass = flv%get_mass (); width = flv%get_width ()
     select type (driver => object%driver)
     class is (blha_driver_t)
        call driver%set_mass_and_width (i_pdg, mass = mass, width = width)
        if (i_pdg == 5) call driver%blha_olp_set_parameter &
           ('yuk(5)'//c_null_char, dble(mass), 0._double, ierr)
        if (i_pdg == 6) then
           if (driver%external_top_yukawa > 0._default) then
              top_yukawa = driver%external_top_yukawa
           else
              top_yukawa = mass
           end if
           call driver%blha_olp_set_parameter &
              ('yuk(6)'//c_null_char, dble(top_yukawa), 0._double, ierr)
        end if
        if (driver%switch_off_muon_yukawas) then
           if (i_pdg == 13) call driver%blha_olp_set_parameter &
              ('yuk(13)' //c_null_char, 0._double, 0._double, ierr)
        end if
     end select
  end do
end subroutine prc_blha_set_particle_properties
```

This mask adapts which electroweak parameters are supposed to set according to the chosen BLHA EWScheme. This is only implemented for the default OLP method so far.

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
procedure :: init_ew_parameters => prc_blha_init_ew_parameters
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine prc_blha_init_ew_parameters (object, ew_scheme)
  class(prc_blha_t), intent(inout) :: object
  integer, intent(in) :: ew_scheme
  object%ew_parameter_mask = .false.
  select case (ew_scheme)
  case (BLHA_EW_QED)
```

```
      object%ew_parameter_mask (I_ALPHA) = .true.
    case (BLHA_EW_GF)
      object%ew_parameter_mask (I_GF) = .true.
    end select
  end subroutine prc_blha_init_ew_parameters
```

Computes a virtual matrix element from an interface to an external one-loop provider. The output of `blha_olp_eval2` is an array of `dimension(4)`, corresponding to the ϵ^2-, ϵ^1- and ϵ^0-poles of the virtual matrix element at position `r(1:3)` and the Born matrix element at position `r(4)`. The matrix element is rejected if its accuracy is larger than the maximal allowed accuracy. OpenLoops includes a factor of 1 / `n_hel` in the amplitudes, which we have to undo if polarized matrix elements are requested (GoSam does not support polarized matrix elements).

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
  procedure :: compute_sqme_virt => prc_blha_compute_sqme_virt
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
  subroutine prc_blha_compute_sqme_virt (object, &
      i_flv, p, ren_scale, sqme, bad_point)
    class(prc_blha_t), intent(in) :: object
    integer, intent(in) :: i_flv
    type(vector4_t), dimension(:), intent(in) :: p
    real(default), intent(in) :: ren_scale
    real(default), dimension(4), intent(out) :: sqme
    logical, intent(out) :: bad_point
    real(double), dimension(5 * object%n_particles) :: mom
    real(double), dimension(:), allocatable :: r
    real(double) :: mu_dble
    real(double) :: acc_dble
    real(default) :: acc
    real(default) :: alpha_s
    if (object%i_virt(i_flv) > 0) then
       allocate (r (blha_result_array_size (object%n_particles, BLHA_AMP_LOOP)))
       call msg_debug2 (D_VIRTUAL, "prc_blha_compute_sqme_virt")
       call msg_debug2 (D_VIRTUAL, "i_flv", i_flv)
       call msg_debug2 (D_VIRTUAL, "object%i_virt(i_flv)", object%i_virt(i_flv))
       if (debug2_active (D_VIRTUAL)) then
           call msg_debug2 (D_VIRTUAL, "use momenta: ")
           call vector4_write_set (p, show_mass = .true., &
                check_conservation = .true.)
       end if
       mom = object%create_momentum_array (p)
       if (vanishes (ren_scale)) &
            call msg_fatal ("prc_blha_compute_sqme_virt: ren_scale vanishes")
       mu_dble = dble(ren_scale)
       alpha_s = object%qcd%alpha%get (ren_scale)
       select type (driver => object%driver)
       class is (blha_driver_t)
          call driver%set_alpha_s (alpha_s)
          call driver%blha_olp_eval2 (object%i_virt(i_flv), mom, mu_dble, r, acc_dble)
       end select
       acc = acc_dble
```

```
      sqme = r(1:4)
      bad_point = acc > object%maximum_accuracy
      if (object%includes_polarization ()) sqme = object%n_hel * sqme
   else
      sqme = zero
   end if
end subroutine prc_blha_compute_sqme_virt
```

Computes a tree-level matrix element from an interface to an external one-loop provider. The matrix element is rejected if its accuracy is larger than the maximal allowed accuracy. OpenLoops includes a factor of 1 / `n_hel` in the amplitudes, which we have to undo if polarized matrix elements are requested (GoSam does not support polarized matrix elements).

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
procedure :: compute_sqme => prc_blha_compute_sqme
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine prc_blha_compute_sqme (object, i_flv, p, &
    ren_scale, sqme, bad_point)
  class(prc_blha_t), intent(in) :: object
  integer, intent(in) :: i_flv
  type(vector4_t), intent(in), dimension(:) :: p
  real(default), intent(in) :: ren_scale
  real(default), intent(out) :: sqme
  logical, intent(out) :: bad_point
  real(double), dimension(5*object%n_particles) :: mom
  real(double), dimension(OLP_RESULTS_LIMIT) :: r
  real(double) :: mu_dble, acc_dble
  real(default) :: acc, alpha_s
  if (object%i_tree(i_flv) > 0) then
     mom = object%create_momentum_array (p)
     if (vanishes (ren_scale)) &
          call msg_fatal ("prc_blha_compute_sqme: ren_scale vanishes")
     mu_dble = dble(ren_scale)
     alpha_s = object%qcd%alpha%get (ren_scale)
     select type (driver => object%driver)
     class is (blha_driver_t)
        call driver%set_alpha_s (alpha_s)
        call driver%blha_olp_eval2 (object%i_tree(i_flv), mom, &
             mu_dble, r, acc_dble)
        sqme = r(object%sqme_tree_pos)
     end select
     acc = acc_dble
     bad_point = acc > object%maximum_accuracy
     if (object%includes_polarization ()) sqme = object%n_hel * sqme
  else
     sqme = zero
  end if
end subroutine prc_blha_compute_sqme
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
public :: blha_cc_fill_diag
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine blha_cc_fill_diag (sqme_born, flavors, sqme_cc)
   real(default), intent(in) :: sqme_born
   integer, intent(in), dimension(:) :: flavors
   real(default), intent(inout), dimension(:,:) :: sqme_cc
   integer :: i
   do i = 1, size (flavors)
      if (is_quark (flavors(i))) then
         sqme_cc (i, i) = -cf * sqme_born
      else if (is_gluon (flavors(i))) then
         sqme_cc (i, i) = -ca * sqme_born
      else
         sqme_cc (i, i) = zero
      end if
   end do
end subroutine blha_cc_fill_diag
```

⟨*BLHA OLP interfaces: public*⟩+≡

```
public :: blha_cc_fill_offdiag
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine blha_cc_fill_offdiag (n, r, sqme_cc)
  integer, intent(in) :: n
  real(default), intent(in), dimension(:) :: r
  real(default), intent(inout), dimension(:,:) :: sqme_cc
  integer :: i, j, im1, jm1, pos
  do j = 1, n
     do i = 1, j
        if (i /= j) then
           im1 = i - 1; jm1 = j - 1
           pos = im1 + jm1 * (jm1 - 1) / 2 + 1
           sqme_cc (i, j) = -r (pos)
        end if
        sqme_cc (j, i) = sqme_cc (i, j)
     end do
  end do
end subroutine blha_cc_fill_offdiag
```

Computes a color-correlated matrix element from an interface to an external one-loop provider. The output of `blha_olp_eval2` is an array of `dimension(n * (n - 1) / 2)`. The matrix element is rejected if its accuracy is larger than the maximal allowed accuracy. OpenLoops includes a factor of 1 / `n_hel` in the amplitudes, which we have to undo if polarized matrix elements are requested (GoSam does not support polarized matrix elements).

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
procedure :: compute_sqme_cc_raw => prc_blha_compute_sqme_cc_raw
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
subroutine prc_blha_compute_sqme_cc_raw &
   (object, i_flv, p, ren_scale, rr, bad_point)
  class(prc_blha_t), intent(in) :: object
  integer, intent(in) :: i_flv
  type(vector4_t), intent(in), dimension(:) :: p
  real(default), intent(in) :: ren_scale
```

```
    real(default), intent(out), dimension(:) :: rr
    logical, intent(out) :: bad_point
    real(double), dimension(5 * object%n_particles) :: mom
    real(double), dimension(size(rr)) :: r
    real(default) :: alpha_s, acc
    real(double) :: mu_dble, acc_dble
    if (object%i_cc(i_flv) > 0) then
       mom = object%create_momentum_array (p)
       if (vanishes (ren_scale)) &
          call msg_fatal ("prc_blha_compute_sqme_cc: ren_scale vanishes")
       mu_dble = dble(ren_scale)
       alpha_s = object%qcd%alpha%get (ren_scale)

       select type (driver => object%driver)
       class is (blha_driver_t)
          call driver%set_alpha_s (alpha_s)
          call driver%blha_olp_eval2 (object%i_cc(i_flv), &
               mom, mu_dble, r, acc_dble)
       end select
       rr = r
       acc = acc_dble
       bad_point = acc > object%maximum_accuracy
       if (object%includes_polarization ()) rr = object%n_hel * rr
    else
       rr = zero
    end if
  end subroutine prc_blha_compute_sqme_cc_raw
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
  procedure :: compute_sqme_cc => prc_blha_compute_sqme_cc
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
  subroutine prc_blha_compute_sqme_cc &
         (object, i_flv, p, ren_scale, born_cc, bad_point, born_out)
    class(prc_blha_t), intent(inout) :: object
    integer, intent(in) :: i_flv
    type(vector4_t), intent(in), dimension(:) :: p
    real(default), intent(in) :: ren_scale
    real(default), intent(inout), dimension(:,:) :: born_cc
    real(default), intent(out), optional :: born_out
    logical, intent(out) :: bad_point
    real(default), dimension(:), allocatable :: r
    logical :: bad_point2
    real(default) :: born
    integer, dimension(:), allocatable :: flavors
    allocate (r (blha_result_array_size &
         (size(born_cc, dim=1), BLHA_AMP_CC)))
    call object%compute_sqme_cc_raw (i_flv, p, ren_scale, r, bad_point)

    select type (driver => object%driver)
    class is (blha_driver_t)
       if (allocated (object%i_tree)) then
          call object%compute_sqme (i_flv, p, ren_scale, born, bad_point2)
       else
```

```
          born = zero
       end if
       if (present (born_out)) born_out = born
    end select
    call blha_cc_fill_offdiag (object%n_particles, r, born_cc)
    !!! !!! !!! Workaround for ifort 16.0 missing default allocate-on-assignment
    allocate (flavors (object%get_n_flvs (i_flv)))
    !!! !!! !!! End of workaround
    flavors = object%get_flv_state (i_flv)
    call blha_cc_fill_diag (born, flavors, born_cc)

    bad_point = bad_point .or. bad_point2
  end subroutine prc_blha_compute_sqme_cc
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
  generic :: get_beam_helicities => get_beam_helicities_single
  generic :: get_beam_helicities => get_beam_helicities_array
  procedure :: get_beam_helicities_single => prc_blha_get_beam_helicities_single
  procedure :: get_beam_helicities_array => prc_blha_get_beam_helicities_array
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
  function prc_blha_get_beam_helicities_single (object, i, invert_second) result (hel)
    integer, dimension(:), allocatable :: hel
    class(prc_blha_t), intent(in) :: object
    logical, intent(in), optional :: invert_second
    integer, intent(in) :: i
    logical :: inv
    inv = .false.; if (present (invert_second)) inv = invert_second
    allocate (hel (object%data%n_in))
    hel = object%i_hel (i, :)
    if (inv .and. object%data%n_in == 2) hel(2) = -hel(2)
  end function prc_blha_get_beam_helicities_single
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
  procedure :: includes_polarization => prc_blha_includes_polarization
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
  function prc_blha_includes_polarization (object) result (polarized)
    logical :: polarized
    class(prc_blha_t), intent(in) :: object
    select type (driver => object%driver)
    class is (blha_driver_t)
       polarized = driver%include_polarizations
    end select
  end function prc_blha_includes_polarization
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
  function prc_blha_get_beam_helicities_array (object, invert_second) result (hel)
    integer, dimension(:,:), allocatable :: hel
    class(prc_blha_t), intent(in) :: object
    logical, intent(in), optional :: invert_second
    integer :: i
    allocate (hel (object%n_proc, object%data%n_in))
```

```
    do i = 1, object%n_proc
       hel(i,:) = object%get_beam_helicities (i, invert_second)
    end do
  end function prc_blha_get_beam_helicities_array
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
  procedure(prc_blha_init_driver), deferred :: &
      init_driver
```

⟨*BLHA OLP interfaces: interfaces*⟩+≡

```
  abstract interface
    subroutine prc_blha_init_driver (object, os_data)
      import
      class(prc_blha_t), intent(inout) :: object
      type(os_data_t), intent(in) :: os_data
    end subroutine prc_blha_init_driver
  end interface
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
  procedure :: reset_helicity_list => prc_blha_reset_helicity_list
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
  subroutine prc_blha_reset_helicity_list (object)
    class(prc_blha_t), intent(inout) :: object
    if (allocated (object%i_whizard_to_i_olc)) &
         deallocate (object%i_whizard_to_i_olc)
  end subroutine prc_blha_reset_helicity_list
```

In general, the BLHA consits of a virtual matrix element and $n_{\rm sub}$ subtraction terms. The subtractions terms can be pure Born matrix elements (to be used in collinear subtraction or in internal color-correlation), color-correlated matrix elements or spin-correlated matrix elements. The numbers should be ordered in such a way that $\mathcal{V}_{\rm fin}$ is first, followed by the pure Born, the color-correlated and the spin-correlated matrix elements. This repeats $n_{\rm flv}$ times. Let ν_i be the position of the *ith* virtual matrix element. The next $\mathcal{V}_{\rm fin}$ is at position $\nu_i = \nu_{i-1} + n_{\rm sub} + 1$. Obviously, $\nu_1 = 1$. This allows us to determine the virtual matrix element positions using the recursive function implemented below.

⟨*BLHA OLP interfaces: public*⟩+≡

```
  public :: blha_loop_positions
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
  recursive function blha_loop_positions (i_flv, n_sub) result (index)
    integer :: index
    integer, intent(in) :: i_flv, n_sub
    index = 0
    if (i_flv == 1) then
       index = 1
    else
       index = blha_loop_positions (i_flv - 1, n_sub) + n_sub + 1
    end if
  end function blha_loop_positions
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡
```
procedure :: set_helicity_list_trivial => prc_blha_set_helicity_list_trivial
```
⟨*BLHA OLP interfaces: procedures*⟩+≡
```
subroutine prc_blha_set_helicity_list_trivial (object, n_flv, n_sub)
  class(prc_blha_t), intent(inout) :: object
  integer, intent(in) :: n_flv, n_sub
  integer :: i_flv
  if (allocated (object%i_whizard_to_i_olc)) &
       deallocate (object%i_whizard_to_i_olc)
  allocate (object%i_whizard_to_i_olc (n_flv))
  do i_flv = 1, n_flv
     object%i_whizard_to_i_olc (i_flv) = blha_loop_positions (i_flv, n_sub)
  end do
end subroutine prc_blha_set_helicity_list_trivial
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡
```
procedure :: set_helicity_list => prc_blha_set_helicity_list
```
⟨*BLHA OLP interfaces: procedures*⟩+≡
```
subroutine prc_blha_set_helicity_list (object, &
   helicities)
  class(prc_blha_t), intent(inout) :: object
  integer, intent(in), dimension(:,:) :: helicities
  integer, dimension(:, :), allocatable :: hel_olc
  integer :: n1, n2
  integer :: i, j
  n1 = size (helicities, dim=1)
  n2 = size (object%get_beam_helicities (), dim=1)
  if (allocated (object%i_whizard_to_i_olc)) &
       deallocate (object%i_whizard_to_i_olc)
  allocate (object%i_whizard_to_i_olc (n2))
  allocate (hel_olc (n2, 2))
  hel_olc = object%get_beam_helicities (invert_second = .true.)
  do i = 1, n1
     do j = 1, n2
        if (all (helicities (i, :) == hel_olc (j, :))) &
           object%i_whizard_to_i_olc (j) = i
     end do
  end do
  deallocate (hel_olc)
end subroutine prc_blha_set_helicity_list
```

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡
```
generic :: get_helicity_list => get_helicity_list_all
generic :: get_helicity_list => get_helicity_list_single
procedure :: get_helicity_list_all => prc_blha_get_helicity_list_all
procedure :: get_helicity_list_single => prc_blha_get_helicity_list_single
```
⟨*BLHA OLP interfaces: procedures*⟩+≡
```
function prc_blha_get_helicity_list_all (object) result (i_out)
  integer, dimension(:), allocatable :: i_out
  class(prc_blha_t), intent(in) :: object
  allocate (i_out (size (object%i_whizard_to_i_olc)))
  i_out = object%i_whizard_to_i_olc
```

```
  end function prc_blha_get_helicity_list_all

  function prc_blha_get_helicity_list_single (object, i) result (i_out)
    integer :: i_out
    class(prc_blha_t), intent(in) :: object
    integer, intent(in) :: i
    i_out = object%i_whizard_to_i_olc (i)
  end function prc_blha_get_helicity_list_single
```

OpenLoops has problems initializing the first helicity-correlated matrix element. Therefore, we compute one dummy value during the initialization phase.

⟨*BLHA OLP interfaces: prc blha: TBP*⟩+≡

```
  procedure :: warmup_helicities => prc_blha_warmup_helicities
```

⟨*BLHA OLP interfaces: procedures*⟩+≡

```
  subroutine prc_blha_warmup_helicities (object, p)
    class(prc_blha_t), intent(inout) :: object
    type(vector4_t), intent(in), dimension(:) :: p
    real(double), dimension(5 * object%n_particles) :: mom
    real(double), dimension(:), allocatable :: r
    real(default) :: alpha_s
    real(double) :: ren_scale, acc
    integer :: i_hel
    allocate (r (blha_result_array_size (object%n_particles, BLHA_AMP_LOOP)))
    alpha_s = 0.1178_double
    ren_scale = 200._double
    mom = object%create_momentum_array (p)
    select type (driver => object%driver)
    class is (blha_driver_t)
       call driver%set_alpha_s (alpha_s)
       do i_hel = 1, object%n_hel
          call driver%blha_olp_eval2 (i_hel, mom, ren_scale, r, acc)
       end do
    end select
  end subroutine prc_blha_warmup_helicities
```

The module is split into a configuration interface which manages configuration and handles the request and contract files, a module which interfaces the OLP matrix elements and a driver.

⟨blha_config.f90⟩≡

```
  ⟨File header⟩

  module blha_config

    use kinds
  ⟨Use strings⟩
    use io_units
    use constants
    use string_utils
    use variables, only: var_list_t
    use diagnostics
    use md5
    use model_data
```

```
  use flavors
  use quantum_numbers
  use pdg_arrays
  use sorting
  use lexers
  use parser
  use syntax_rules
  use ifiles

  use beam_structures, only: beam_structure_t
```

⟨*Standard module head*⟩

⟨*BLHA config: public*⟩

⟨*BLHA config: parameters*⟩

⟨*BLHA config: types*⟩

⟨*BLHA config: variables*⟩

⟨*BLHA config: interfaces*⟩

```
contains
```

⟨*BLHA config: procedures*⟩

```
end module blha_config
```

24.2 Configuration

Parameters to enumerate the different options in the order.

⟨*BLHA config: parameters*⟩≡

```
  integer, public, parameter :: &
       BLHA_CT_QCD = 1, BLHA_CT_EW = 2, BLHA_CT_QED = 3, BLHA_CT_OTHER = 4
  integer, public, parameter :: &
       BLHA_IRREG_CDR = 1, BLHA_IRREG_DRED = 2, BLHA_IRREG_THV = 3, &
       BLHA_IRREG_MREG = 4, BLHA_IRREG_OTHER = 5
  integer, public, parameter :: &
       BLHA_MPS_ONSHELL = 1, BLHA_MPS_OTHER = 2
  integer, public, parameter :: &
       BLHA_MODE_GOSAM = 1, BLHA_MODE_FEYNARTS = 2, BLHA_MODE_GENERIC = 3, &
       BLHA_MODE_OPENLOOPS = 4
  integer, public, parameter :: &
       BLHA_VERSION_1 = 1, BLHA_VERSION_2 = 2
  integer, public, parameter :: &
       BLHA_AMP_LOOP = 1, BLHA_AMP_CC = 2, BLHA_AMP_SC = 3, &
       BLHA_AMP_TREE = 4, BLHA_AMP_LOOPINDUCED = 5
  integer, public, parameter :: &
       BLHA_EW_QED = 0, &
       BLHA_EW_GF = 1, BLHA_EW_MZ = 2, BLHA_EW_MSBAR = 3, &
```

```
     BLHA_EW_0 = 4, BLHA_EW_RUN = 5
integer, public, parameter :: &
     BLHA_WIDTH_COMPLEX = 1, BLHA_WIDTH_FIXED = 2, &
     BLHA_WIDTH_RUNNING = 3, BLHA_WIDTH_POLE = 4, &
     BLHA_WIDTH_DEFAULT = 5
```

Those are the default pdg codes for massive particles in BLHA programs

⟨*BLHA config: parameters*⟩+≡

```
integer, parameter, public :: OLP_N_MASSIVE_PARTICLES = 12
integer, dimension(OLP_N_MASSIVE_PARTICLES), public :: &
  OLP_MASSIVE_PARTICLES = [5, -5, 6, -6, 13, -13, 15, -15, 23, 24, -24, 25]
integer, parameter :: OLP_HEL_UNPOLARIZED = 0
```

The user might provide an extra command string for OpenLoops to apply special libraries instead of the default ones, such as signal-only amplitudes for off-shell top production. We check in this subroutine that the provided string is valid and print out the possible options to ease the user's memory.

⟨*BLHA config: parameters*⟩+≡

```
integer, parameter :: N_KNOWN_SPECIAL_OL_METHODS = 3
```

⟨*BLHA config: procedures*⟩≡

```
subroutine check_extra_cmd (extra_cmd)
  type(string_t), intent(in) :: extra_cmd
  type(string_t), dimension(N_KNOWN_SPECIAL_OL_METHODS) :: known_methods
  integer :: i
  logical :: found
  known_methods(1) = 'top'
  known_methods(2) = 'not'
  known_methods(3) = 'stop'
  if (extra_cmd == var_str ("")) return
  found = .false.
  do i = 1, N_KNOWN_SPECIAL_OL_METHODS
     found = found .or. (extra_cmd == var_str ('extra approx ') // known_methods(i))
  end do
  if (.not. found) &
    call msg_fatal ("The given extra OpenLoops method is not kown ", &
       [var_str ("Available commands are: "), &
        var_str ("extra approx top (only WbWb signal),"), &
        var_str ("extra approx stop (only WbWb singletop),"), &
        var_str ("extra approx not (no top in WbWb).")])
end subroutine check_extra_cmd
```

This type contains the pdg code of the particle to be written in the process specification string and an optional additional information about the polarization of the particles. Note that the output can only be processed by OpenLoops.

⟨*BLHA config: types*⟩≡

```
type :: blha_particle_string_element_t
   integer :: pdg = 0
   integer :: hel = OLP_HEL_UNPOLARIZED
   logical :: polarized = .false.
contains
```

⟨*BLHA config: blha particle string element: TBP*⟩

```
end type blha_particle_string_element_t
```

⟨*BLHA config: blha particle string element: TBP*⟩≡

```
generic :: init => init_default
generic :: init => init_polarized
procedure :: init_default => blha_particle_string_element_init_default
procedure :: init_polarized => blha_particle_string_element_init_polarized
```

⟨*BLHA config: procedures*⟩+≡

```
subroutine blha_particle_string_element_init_default (blha_p, id)
  class(blha_particle_string_element_t), intent(out) :: blha_p
  integer, intent(in) :: id
  blha_p%pdg = id
end subroutine blha_particle_string_element_init_default
```

⟨*BLHA config: procedures*⟩+≡

```
subroutine blha_particle_string_element_init_polarized (blha_p, id, hel)
  class(blha_particle_string_element_t), intent(out) :: blha_p
  integer, intent(in) :: id, hel
  blha_p%polarized = .true.
  blha_p%pdg = id
  blha_p%hel = hel
end subroutine blha_particle_string_element_init_polarized
```

⟨*BLHA config: blha particle string element: TBP*⟩+≡

```
generic :: write_pdg => write_pdg_unit
generic :: write_pdg => write_pdg_character
procedure :: write_pdg_unit => blha_particle_string_element_write_pdg_unit
procedure :: write_pdg_character &
   => blha_particle_string_element_write_pdg_character
```

⟨*BLHA config: procedures*⟩+≡

```
subroutine blha_particle_string_element_write_pdg_unit (blha_p, unit)
  class(blha_particle_string_element_t), intent(in) :: blha_p
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, '(I3)') blha_p%pdg
end subroutine blha_particle_string_element_write_pdg_unit
```

⟨*BLHA config: procedures*⟩+≡

```
subroutine blha_particle_string_element_write_pdg_character (blha_p, c)
  class(blha_particle_string_element_t), intent(in) :: blha_p
  character(3), intent(inout) :: c
  write (c, '(I3)') blha_p%pdg
end subroutine blha_particle_string_element_write_pdg_character
```

⟨*BLHA config: blha particle string element: TBP*⟩+≡

```
generic :: write_helicity => write_helicity_unit
generic :: write_helicity => write_helicity_character
procedure :: write_helicity_unit &
   => blha_particle_string_element_write_helicity_unit
```

```
procedure :: write_helicity_character &
   => blha_particle_string_element_write_helicity_character
```

⟨*BLHA config: procedures*⟩+≡

```
subroutine blha_particle_string_element_write_helicity_unit (blha_p, unit)
  class(blha_particle_string_element_t), intent(in) :: blha_p
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, '(A1,I0,A1)') '(', blha_p%hel, ')'
end subroutine blha_particle_string_element_write_helicity_unit
```

⟨*BLHA config: procedures*⟩+≡

```
subroutine blha_particle_string_element_write_helicity_character (blha_p, c)
  class(blha_particle_string_element_t), intent(in) :: blha_p
  character(4), intent(inout) :: c
  write (c, '(A1,I0,A1)') '(', blha_p%hel, ')'
end subroutine blha_particle_string_element_write_helicity_character
```

This type encapsulates a BLHA request.

⟨*BLHA config: public*⟩≡

```
public :: blha_configuration_t
public :: blha_cfg_process_node_t
```

⟨*BLHA config: types*⟩+≡

```
type :: blha_cfg_process_node_t
   type(blha_particle_string_element_t), dimension(:), allocatable :: pdg_in, pdg_out
   integer, dimension(:), allocatable :: fingerprint
   integer :: nsub
   integer, dimension(:), allocatable :: ids
   integer :: amplitude_type
   type(blha_cfg_process_node_t), pointer :: next => null ()
end type blha_cfg_process_node_t

type :: blha_configuration_t
   type(string_t) :: name
   class(model_data_t), pointer :: model => null ()
   type(string_t) :: md5
   integer :: version = 2
   logical :: dirty = .false.
   integer :: n_proc = 0
   real(default) :: accuracy_target
   logical :: debug_unstable = .false.
   integer :: mode = BLHA_MODE_GENERIC
   logical :: polarized = .false.
   type(blha_cfg_process_node_t), pointer :: processes => null ()
   !integer, dimension(2) :: matrix_element_square_type = BLHA_MEST_SUM
   integer :: correction_type = BLHA_CT_QCD
   type(string_t) :: correction_type_other
   integer :: irreg = BLHA_IRREG_THV
   type(string_t) :: irreg_other
   integer :: massive_particle_scheme = BLHA_MPS_ONSHELL
   type(string_t) :: massive_particle_scheme_other
   type(string_t) :: model_file
```

```
    logical :: subdivide_subprocesses = .false.
    integer :: alphas_power = -1, alpha_power = -1
    integer :: ew_scheme = BLHA_EW_GF
    integer :: width_scheme = BLHA_WIDTH_DEFAULT
    logical :: openloops_use_cms = .false.
    integer :: openloops_phs_tolerance = 0
    type(string_t) :: openloops_extra_cmd
    integer :: openloops_stability_log = 0
    logical :: openloops_use_collier = .false.
end type blha_configuration_t
```

Translate the SINDARIN input string to the corresponding named integer.

⟨*BLHA config: public*⟩+≡

```
public :: ew_scheme_string_to_int
```

⟨*BLHA config: procedures*⟩+≡

```
function ew_scheme_string_to_int (ew_scheme_str) result (ew_scheme_int)
  integer :: ew_scheme_int
  type(string_t), intent(in) :: ew_scheme_str
  select case (char (ew_scheme_str))
  case ('GF', 'Gmu')
     ew_scheme_int = BLHA_EW_GF
  case ('alpha_qed')
     ew_scheme_int = BLHA_EW_QED
  case ('alpha_mz')
     ew_scheme_int = BLHA_EW_MZ
  case ('alpha_0', 'alpha_thompson')
     ew_scheme_int = BLHA_EW_0
  case default
     call msg_fatal ("ew_scheme: " // char (ew_scheme_str) // &
          " not supported. Try 'Gmu', 'alpha_qed', 'alpha_mz' or 'alpha_0'.")
  end select
end function ew_scheme_string_to_int
```

This types control the creation of BLHA-interface files

⟨*BLHA config: public*⟩+≡

```
public :: blha_flv_state_t
public :: blha_master_t
```

⟨*BLHA config: types*⟩+≡

```
type:: blha_flv_state_t
  integer, dimension(:), allocatable :: flavors
  integer :: flv_mult
  logical :: flv_real = .false.
end type blha_flv_state_t

type :: blha_master_t
  integer, dimension(4) :: blha_mode
  logical :: compute_borns = .false.
  logical :: compute_real_trees = .false.
  logical :: compute_loops = .true.
  logical :: compute_correlations = .false.
  integer :: ew_scheme
  type(string_t), dimension(:), allocatable :: suffix
```

```
    type(blha_configuration_t), dimension(:), allocatable :: blha_cfg
    integer :: n_files = 0
  contains
  ⟨BLHA config: blha master: TBP⟩
  end type blha_master_t
```

Master-Routines

⟨*BLHA config: blha master: TBP*⟩≡

```
  procedure :: set_methods => blha_master_set_methods
```

⟨*BLHA config: procedures*⟩+≡

```
  subroutine blha_master_set_methods (master, is_nlo, var_list)
    class(blha_master_t), intent(inout) :: master
    logical, intent(in) :: is_nlo
    type(var_list_t), intent(in) :: var_list
    type(string_t) :: born_me_method, real_tree_me_method
    type(string_t) :: loop_me_method, correlation_me_method
    logical :: cmp_born, cmp_real
    logical :: cmp_loop, cmp_corr
    born_me_method = var_list%get_sval (var_str ("$born_me_method"))
    real_tree_me_method = var_list%get_sval (var_str ("$real_tree_me_method"))
    loop_me_method = var_list%get_sval (var_str ("$loop_me_method"))
    correlation_me_method = var_list%get_sval (var_str ("$correlation_me_method"))
    cmp_born = born_me_method /= 'omega'
    cmp_real = is_nlo .and. (real_tree_me_method /= 'omega')
    cmp_loop = is_nlo .and. (loop_me_method /= 'omega')
    cmp_corr = is_nlo .and. (correlation_me_method /= 'omega')
    call set_me_method (1, loop_me_method)
    call set_me_method (2, correlation_me_method)
    call set_me_method (3, real_tree_me_method)
    call set_me_method (4, born_me_method)
    master%n_files = count ([cmp_born, cmp_real, cmp_loop, cmp_corr])
    master%compute_borns = cmp_born
    master%compute_real_trees = cmp_real
    master%compute_loops = cmp_loop
    master%compute_correlations = cmp_corr
  contains
    subroutine set_me_method (i, me_method)
      integer, intent(in) :: i
      type(string_t) :: me_method
      select case (char (me_method))
      case ('gosam')
         call master%set_gosam (i)
      case ('openloops')
         call master%set_openloops (i)
      end select
    end subroutine set_me_method
  end subroutine blha_master_set_methods
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
  procedure :: allocate_config_files => blha_master_allocate_config_files
```

⟨*BLHA config: procedures*⟩+≡

```
  subroutine blha_master_allocate_config_files (master)
```

```
    class(blha_master_t), intent(inout) :: master
    allocate (master%blha_cfg (master%n_files))
    allocate (master%suffix (master%n_files))
  end subroutine blha_master_allocate_config_files
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
  procedure :: set_ew_scheme => blha_master_set_ew_scheme
```

⟨*BLHA config: procedures*⟩+≡

```
  subroutine blha_master_set_ew_scheme (master, ew_scheme)
    class(blha_master_t), intent(inout) :: master
    type(string_t), intent(in) :: ew_scheme
    master%ew_scheme = ew_scheme_string_to_int (ew_scheme)
  end subroutine blha_master_set_ew_scheme
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
  procedure :: generate => blha_master_generate
```

⟨*BLHA config: procedures*⟩+≡

```
  subroutine blha_master_generate (master, basename, model, &
      n_in, alpha_power, alphas_power, flv_born, flv_real)
    class(blha_master_t), intent(inout) :: master
    type(string_t), intent(in) :: basename
    class(model_data_t), intent(in), target :: model
    integer, intent(in) :: n_in
    integer, intent(in) :: alpha_power, alphas_power
    integer, intent(in), dimension(:,:), allocatable :: flv_born, flv_real
    integer :: i_file
    if (master%n_files < 1) &
        call msg_fatal ("Attempting to generate OLP-files, but none are specified!")
    i_file = 1
    call master%generate_loop (basename, model, n_in, alpha_power, &
        alphas_power, flv_born, i_file)
    call master%generate_correlation (basename, model, n_in, alpha_power, &
        alphas_power, flv_born, i_file)
    call master%generate_real_tree (basename, model, n_in, alpha_power, &
        alphas_power, flv_real, i_file)
    call master%generate_born (basename, model, n_in, alpha_power, &
        alphas_power, flv_born, i_file)
  end subroutine blha_master_generate
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
  procedure :: generate_loop => blha_master_generate_loop
```

⟨*BLHA config: procedures*⟩+≡

```
  subroutine blha_master_generate_loop (master, basename, model, n_in, &
        alpha_power, alphas_power, flv_born, i_file)
    class(blha_master_t), intent(inout) :: master
    type(string_t), intent(in) :: basename
    class(model_data_t), intent(in), target :: model
    integer, intent(in) :: n_in
    integer, intent(in) :: alpha_power, alphas_power
    integer, dimension(:,:), allocatable, intent(in) :: flv_born
    integer, intent(inout) :: i_file
    type(blha_flv_state_t), dimension(:), allocatable :: blha_flavor
```

```
    integer :: i_flv
    if (master%compute_loops) then
       if (allocated (flv_born)) then
          allocate (blha_flavor (size (flv_born, 2)))
          do i_flv = 1, size (flv_born, 2)
             allocate (blha_flavor(i_flv)%flavors (size (flv_born(:,i_flv))))
             blha_flavor(i_flv)%flavors = flv_born(:,i_flv)
             blha_flavor(i_flv)%flv_mult = 2
          end do
          master%suffix(i_file) = "_LOOP"
          call blha_init_virtual (master%blha_cfg(i_file), blha_flavor, &
               n_in, alpha_power, alphas_power, master%ew_scheme, &
               basename, model, master%blha_mode(1))
          i_file = i_file + 1
       else
          call msg_fatal ("BLHA Loops requested but " &
                          // "Born flavor not existing")
       end if
    end if
  end subroutine blha_master_generate_loop
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
  procedure :: generate_correlation => blha_master_generate_correlation
```

⟨*BLHA config: procedures*⟩+≡

```
  subroutine blha_master_generate_correlation (master, basename, model, n_in, &
        alpha_power, alphas_power, flv_born, i_file)
    class(blha_master_t), intent(inout) :: master
    type(string_t), intent(in) :: basename
    class(model_data_t), intent(in), target :: model
    integer, intent(in) :: n_in
    integer, intent(in) :: alpha_power, alphas_power
    integer, dimension(:,:), allocatable, intent(in) :: flv_born
    integer, intent(inout) :: i_file
    type(blha_flv_state_t), dimension(:), allocatable :: blha_flavor
    integer :: i_flv
    if (master%compute_correlations) then
       if (allocated (flv_born)) then
          allocate (blha_flavor (size (flv_born, 2)))
          do i_flv = 1, size (flv_born, 2)
             allocate (blha_flavor(i_flv)%flavors (size (flv_born(:,i_flv))))
             blha_flavor(i_flv)%flavors = flv_born(:,i_flv)
             blha_flavor(i_flv)%flv_mult = 3
          end do
          master%suffix(i_file) = "_SUB"
          call blha_init_subtraction (master%blha_cfg(i_file), blha_flavor, &
               n_in, alpha_power, alphas_power, master%ew_scheme, &
               basename, model, master%blha_mode(2))
          i_file = i_file + 1
       else
          call msg_fatal ("BLHA Correlations requested but "&
                          // "Born flavor not existing")
       end if
    end if
```

```
    end subroutine blha_master_generate_correlation
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
    procedure :: generate_real_tree => blha_master_generate_real_tree
```

⟨*BLHA config: procedures*⟩+≡

```
    subroutine blha_master_generate_real_tree (master, basename, model, n_in, &
           alpha_power, alphas_power, flv_real, i_file)
      class(blha_master_t), intent(inout) :: master
      type(string_t), intent(in) :: basename
      class(model_data_t), intent(in), target :: model
      integer, intent(in) :: n_in
      integer, intent(in) :: alpha_power, alphas_power
      integer, dimension(:,:), allocatable, intent(in) :: flv_real
      integer, intent(inout) :: i_file
      type(blha_flv_state_t), dimension(:), allocatable :: blha_flavor
      integer :: i_flv
      if (master%compute_real_trees) then
         if (allocated (flv_real)) then
            allocate (blha_flavor (size (flv_real, 2)))
            do i_flv = 1, size (flv_real, 2)
               allocate (blha_flavor(i_flv)%flavors (size (flv_real(:,i_flv))))
               blha_flavor(i_flv)%flavors = flv_real(:,i_flv)
               blha_flavor(i_flv)%flv_mult = 1
            end do
            master%suffix(i_file) = "_REAL"
            call blha_init_real (master%blha_cfg(i_file), blha_flavor, &
                 n_in, alpha_power, alphas_power, master%ew_scheme, &
                 basename, model, master%blha_mode(3))
            i_file = i_file + 1
         else
            call msg_fatal ("BLHA Trees requested but "&
                            // "Real flavor not existing")
         end if
      end if
    end subroutine blha_master_generate_real_tree
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
    procedure :: generate_born => blha_master_generate_born
```

⟨*BLHA config: procedures*⟩+≡

```
  subroutine blha_master_generate_born (master, basename, model, n_in, &
           alpha_power, alphas_power, flv_born, i_file)
      class(blha_master_t), intent(inout) :: master
      type(string_t), intent(in) :: basename
      class(model_data_t), intent(in), target :: model
      integer, intent(in) :: n_in
      integer, intent(in) :: alpha_power, alphas_power
      integer, dimension(:,:), allocatable, intent(in) :: flv_born
      integer, intent(inout) :: i_file
      type(blha_flv_state_t), dimension(:), allocatable :: blha_flavor
      integer :: i_flv
      if (master%compute_borns) then
         if (allocated (flv_born)) then
```

```
            allocate (blha_flavor (size (flv_born, 2)))
            do i_flv = 1, size (flv_born, 2)
               allocate (blha_flavor(i_flv)%flavors (size (flv_born(:,i_flv))))
               blha_flavor(i_flv)%flavors = flv_born(:,i_flv)
               blha_flavor(i_flv)%flv_mult = 1
            end do
            master%suffix(i_file) = "_BORN"
            call blha_init_born (master%blha_cfg(i_file), blha_flavor, &
                 n_in, alpha_power, alphas_power, master%ew_scheme, &
                 basename, model, master%blha_mode(4))
         end if
      end if
   end subroutine blha_master_generate_born
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
   procedure :: setup_additional_features => blha_master_setup_additional_features
```

⟨*BLHA config: procedures*⟩+≡

```
   subroutine blha_master_setup_additional_features (master, &
      phs_tolerance, use_cms, stability_log, use_collier, extra_cmd, beam_structure)
      class(blha_master_t), intent(inout) :: master
      integer, intent(in) :: phs_tolerance
      logical, intent(in) :: use_cms
      type(string_t), intent(in), optional :: extra_cmd
      integer, intent(in) :: stability_log
      logical, intent(in) :: use_collier
      type(beam_structure_t), intent(in), optional :: beam_structure
      integer :: i_file
      logical :: polarized

      polarized = .false.
      if (present (beam_structure)) polarized = beam_structure%has_polarized_beams ()

      if (use_cms .and. any (master%blha_mode /= BLHA_MODE_OPENLOOPS)) &
         call cms_warning ()
      do i_file = 1, master%n_files
         if (phs_tolerance > 0) then
            select case (master%blha_mode(i_file))
            case (BLHA_MODE_GOSAM)
               if (polarized) &
                  call gosam_error_message ()
            case (BLHA_MODE_OPENLOOPS)
               master%blha_cfg(i_file)%openloops_use_cms = use_cms
               master%blha_cfg(i_file)%openloops_phs_tolerance = phs_tolerance
               master%blha_cfg(i_file)%polarized = polarized
               if (present (extra_cmd)) then
                 master%blha_cfg(i_file)%openloops_extra_cmd = extra_cmd
               else
                 master%blha_cfg(i_file)%openloops_extra_cmd = var_str ('')
               end if
               master%blha_cfg(i_file)%openloops_stability_log = stability_log
               master%blha_cfg(i_file)%openloops_use_collier = use_collier
            end select
         end if
```

```
      end do
   contains
      subroutine cms_warning ()
         call msg_warning ("You have set ?openloops_use_cms = true, but not all matrix ", &
              [var_str ("element methods are set to OpenLoops. Note that other "), &
               var_str ("methods might not necessarily support the complex mass "), &
               var_str ("scheme. This can yield inconsistencies in your NLO results!")])
      end subroutine cms_warning

      subroutine gosam_error_message ()
         call msg_fatal ("You are trying to evaluate a process at NLO ", &
              [var_str ("which involves polarized beams using GoSam. "), &
               var_str ("This feature is not supported yet. "), &
               var_str ("Please use OpenLoops instead")])
      end subroutine gosam_error_message
   end subroutine blha_master_setup_additional_features
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
   procedure :: set_gosam => blha_master_set_gosam
```

⟨*BLHA config: procedures*⟩+≡

```
   subroutine blha_master_set_gosam (master, i)
     class(blha_master_t), intent(inout) :: master
     integer, intent(in) :: i
     master%blha_mode(i) = BLHA_MODE_GOSAM
   end subroutine blha_master_set_gosam
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
   procedure :: set_openloops => blha_master_set_openloops
```

⟨*BLHA config: procedures*⟩+≡

```
   subroutine blha_master_set_openloops (master, i)
     class(blha_master_t), intent(inout) :: master
     integer, intent(in) :: i
     master%blha_mode(i) = BLHA_MODE_OPENLOOPS
   end subroutine blha_master_set_openloops
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
   procedure :: reset_olp_modes => blha_master_reset_olp_modes
```

⟨*BLHA config: procedures*⟩+≡

```
   subroutine blha_master_reset_olp_modes (master, i)
     class(blha_master_t), intent(inout) :: master
     integer, intent(in) :: i
     master%blha_cfg(i)%mode = master%blha_mode(i)
   end subroutine blha_master_reset_olp_modes
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
   procedure :: set_polarization => blha_master_set_polarization
```

⟨*BLHA config: procedures*⟩+≡

```
   subroutine blha_master_set_polarization (master, i)
     class(blha_master_t), intent(inout) :: master
     integer, intent(in) :: i
```

```
    master%blha_cfg(i)%polarized = .true.
  end subroutine blha_master_set_polarization
```

⟨*BLHA config: procedures*⟩+≡

```
  subroutine blha_init_born (blha_cfg, blha_flavor, n_in, &
       ap, asp, ew_scheme, basename, model, blha_mode)
    type(blha_configuration_t), intent(inout) :: blha_cfg
    type(blha_flv_state_t), intent(in), dimension(:) :: blha_flavor
    integer, intent(in) :: n_in
    integer, intent(in) :: ap, asp
    integer, intent(in) :: ew_scheme
    type(string_t), intent(in) :: basename
    type(model_data_t), intent(in), target :: model
    integer, intent(in) :: blha_mode
    integer, dimension(:), allocatable :: amp_type
    integer :: i

    allocate (amp_type (size (blha_flavor)))
    do i = 1, size (blha_flavor)
       amp_type(i) = BLHA_AMP_TREE
    end do
    call blha_configuration_init (blha_cfg, basename // "_BORN" , &
         model, blha_mode)
    call blha_configuration_append_processes (blha_cfg, n_in, &
         blha_flavor, amp_type)
    call blha_configuration_set (blha_cfg, BLHA_VERSION_2, &
         correction_type = BLHA_CT_QCD, &
         irreg = BLHA_IRREG_CDR, alphas_power = asp, &
         alpha_power = ap, ew_scheme = ew_scheme, &
         debug = blha_mode == BLHA_MODE_GOSAM)
  end subroutine blha_init_born

  subroutine blha_init_virtual (blha_cfg, blha_flavor, n_in, &
     ap, asp, ew_scheme, basename, model, blha_mode)
    type(blha_configuration_t), intent(inout) :: blha_cfg
    type(blha_flv_state_t), intent(in), dimension(:) :: blha_flavor
    integer, intent(in) :: n_in
    integer, intent(in) :: ap, asp
    integer, intent(in) :: ew_scheme
    type(string_t), intent(in) :: basename
    type(model_data_t), intent(in), target :: model
    integer, intent(in) :: blha_mode
    integer, dimension(:), allocatable :: amp_type
    integer :: i

    allocate (amp_type (size (blha_flavor) * 2))
    do i = 1, size (blha_flavor)
       amp_type(2 * i - 1) = BLHA_AMP_LOOP
       amp_type(2 * i) = BLHA_AMP_CC
    end do
    call blha_configuration_init (blha_cfg, basename // "_LOOP" , &
         model, blha_mode)
    call blha_configuration_append_processes (blha_cfg, n_in, &
         blha_flavor, amp_type)
```

```
  call blha_configuration_set (blha_cfg, BLHA_VERSION_2, &
       correction_type = BLHA_CT_QCD, &
       irreg = BLHA_IRREG_CDR, &
       alphas_power = asp, &
       alpha_power = ap, &
       ew_scheme = ew_scheme, &
       debug = blha_mode == BLHA_MODE_GOSAM)
end subroutine blha_init_virtual

subroutine blha_init_subtraction (blha_cfg, blha_flavor, n_in, &
   ap, asp, ew_scheme, basename, model, blha_mode)
  type(blha_configuration_t), intent(inout) :: blha_cfg
  type(blha_flv_state_t), intent(in), dimension(:) :: blha_flavor
  integer, intent(in) :: n_in
  integer, intent(in) :: ap, asp
  integer, intent(in) :: ew_scheme
  type(string_t), intent(in) :: basename
  type(model_data_t), intent(in), target :: model
  integer, intent(in) :: blha_mode
  integer, dimension(:), allocatable :: amp_type
  integer :: i

  allocate (amp_type (size (blha_flavor) * 3))
  do i = 1, size (blha_flavor)
     amp_type(3 * i - 2) = BLHA_AMP_TREE
     amp_type(3 * i - 1) = BLHA_AMP_CC
     amp_type(3 * i) = BLHA_AMP_SC
  end do
  call blha_configuration_init (blha_cfg, basename // "_SUB" , &
       model, blha_mode)
  call blha_configuration_append_processes (blha_cfg, n_in, &
       blha_flavor, amp_type)
  call blha_configuration_set (blha_cfg, BLHA_VERSION_2, &
       correction_type = BLHA_CT_QCD, &
       irreg = BLHA_IRREG_CDR, &
       alphas_power = asp, &
       alpha_power = ap, &
       ew_scheme = ew_scheme, &
       debug = blha_mode == BLHA_MODE_GOSAM)
end subroutine blha_init_subtraction

subroutine blha_init_real (blha_cfg, blha_flavor, n_in, &
   ap, asp, ew_scheme, basename, model, blha_mode)
  type(blha_configuration_t), intent(inout) :: blha_cfg
  type(blha_flv_state_t), intent(in), dimension(:) :: blha_flavor
  integer, intent(in) :: n_in
  integer, intent(in) :: ap, asp
  integer, intent(in) :: ew_scheme
  type(string_t), intent(in) :: basename
  type(model_data_t), intent(in), target :: model
  integer, intent(in) :: blha_mode
  integer, dimension(:), allocatable :: amp_type
  integer :: i
```

```
    allocate (amp_type (size (blha_flavor)))
    do i = 1, size (blha_flavor)
       amp_type(i) = BLHA_AMP_TREE
    end do
    call blha_configuration_init (blha_cfg, basename // "_REAL" , &
         model, blha_mode)
    call blha_configuration_append_processes (blha_cfg, n_in, &
         blha_flavor, amp_type)
    call blha_configuration_set (blha_cfg, BLHA_VERSION_2, &
         correction_type = BLHA_CT_QCD, &
         irreg = BLHA_IRREG_CDR, &
         alphas_power = asp + 1, &
         alpha_power = ap, &
         ew_scheme = ew_scheme, &
         debug = blha_mode == BLHA_MODE_GOSAM)
  end subroutine blha_init_real
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
  procedure :: write_olp => blha_master_write_olp
```

⟨*BLHA config: procedures*⟩+≡

```
  subroutine blha_master_write_olp (master, basename)
    class(blha_master_t), intent(in) :: master
    type(string_t), intent(in) :: basename
    integer :: unit
    type(string_t) :: filename
    integer :: i_file
    do i_file = 1, master%n_files
       filename = basename // master%suffix(i_file) // ".olp"
       unit = free_unit ()
       open (unit, file = char (filename), status = 'replace', action = 'write')
       call blha_configuration_write (master%blha_cfg(i_file), unit)
       close (unit)
    end do
  end subroutine blha_master_write_olp
```

⟨*BLHA config: blha master: TBP*⟩+≡

```
  procedure :: final => blha_master_final
```

⟨*BLHA config: procedures*⟩+≡

```
  subroutine blha_master_final (master)
    class(blha_master_t), intent(inout) :: master
    master%n_files = 0
    deallocate (master%suffix)
    deallocate (master%blha_cfg)
  end subroutine blha_master_final
```

⟨*BLHA config: public*⟩+≡

```
  public :: blha_configuration_init
```

⟨*BLHA config: procedures*⟩+≡

```
  subroutine blha_configuration_init (cfg, name, model, mode)
    type(blha_configuration_t), intent(inout) :: cfg
    type(string_t), intent(in) :: name
```

```
    class(model_data_t), target, intent(in) :: model
    integer, intent(in), optional :: mode
    if (.not. associated (cfg%model)) then
       cfg%name = name
       cfg%model => model
    end if
    if (present (mode)) cfg%mode = mode
  end subroutine blha_configuration_init
```

Create an array of massive particle indices, to be used by the "MassiveParticle"-statement of the order file.

⟨*BLHA config: procedures*⟩+≡

```
  subroutine blha_configuration_get_massive_particles &
             (cfg, massive, i_massive)
    type(blha_configuration_t), intent(in) :: cfg
    logical, intent(out) :: massive
    integer, intent(out), dimension(:), allocatable :: i_massive
    integer, parameter :: max_particles = 10
    integer, dimension(max_particles) :: i_massive_tmp
    integer, dimension(max_particles) :: checked
    type(blha_cfg_process_node_t), pointer :: current_process
    integer :: k
    integer :: n_massive
    n_massive = 0; k = 1
    checked = 0
    if (associated (cfg%processes)) then
       current_process => cfg%processes
    else
       call msg_fatal ("BLHA, massive particles: " // &
                       "No processes allocated!")
    end if
    do
       call check_pdg_list (current_process%pdg_in%pdg)
       call check_pdg_list (current_process%pdg_out%pdg)
       if (k > max_particles) &
          call msg_fatal ("BLHA, massive particles: " // &
                          "Max. number of particles exceeded!")
       if (associated (current_process%next)) then
          current_process => current_process%next
       else
          exit
       end if
    end do
    if (n_massive > 0) then
       allocate (i_massive (n_massive))
       i_massive = i_massive_tmp (1:n_massive)
       massive = .true.
    else
       massive = .false.
    end if
  contains
    subroutine check_pdg_list (pdg_list)
       integer, dimension(:), intent(in) :: pdg_list
```

```
      integer :: i, i_pdg
      type(flavor_t) :: flv
      do i = 1, size (pdg_list)
         i_pdg = abs (pdg_list(i))
         call flv%init (i_pdg, cfg%model)
         if (flv%get_mass () > 0._default) then
            !!! Avoid duplicates in output
            if (.not. any (checked == i_pdg)) then
               i_massive_tmp(k) = i_pdg
               checked(k) = i_pdg
               k = k + 1
               n_massive = n_massive + 1
            end if
         end if
      end do
    end subroutine check_pdg_list
  end subroutine blha_configuration_get_massive_particles
```

⟨*BLHA config: public*⟩+≡

```
  public :: blha_configuration_append_processes
```

⟨*BLHA config: procedures*⟩+≡

```
  subroutine blha_configuration_append_processes (cfg, n_in, flavor, amp_type)
    type(blha_configuration_t), intent(inout) :: cfg
    integer, intent(in) :: n_in
    type(blha_flv_state_t), dimension(:), intent(in) :: flavor
    integer, dimension(:), intent(in), optional :: amp_type
    integer :: n_tot
    type(blha_cfg_process_node_t), pointer :: current_node
    integer :: i_process, i_flv
    integer, dimension(:), allocatable :: pdg_in, pdg_out
    integer, dimension(:), allocatable :: flavor_state
    integer :: proc_offset, n_proc_tot
    proc_offset = 0; n_proc_tot = 0
    do i_flv = 1, size (flavor)
       n_proc_tot = n_proc_tot + flavor(i_flv)%flv_mult
    end do
    if (.not. associated (cfg%processes)) &
      allocate (cfg%processes)
    current_node => cfg%processes
    do i_flv = 1, size (flavor)
       n_tot = size (flavor(i_flv)%flavors)
       allocate (pdg_in (n_in), pdg_out (n_tot - n_in))
       allocate (flavor_state (n_tot))
       flavor_state = flavor(i_flv)%flavors
       do i_process = 1, flavor(i_flv)%flv_mult
          pdg_in = flavor_state (1 : n_in)
          pdg_out = flavor_state (n_in + 1 : )
          if (cfg%polarized) then
             select case (cfg%mode)
             case (BLHA_MODE_OPENLOOPS)
                call allocate_and_init_pdg_and_helicities (current_node, &
                   pdg_in, pdg_out, amp_type (proc_offset + i_process))
             case (BLHA_MODE_GOSAM)
```

```
            !!! Nothing special for GoSam yet. This exception is already caught
            !!! in blha_master_setup_additional_features
         end select
      else
         call allocate_and_init_pdg (current_node, pdg_in, pdg_out, &
            amp_type (proc_offset + i_process))
      end if
      if (proc_offset + i_process /= n_proc_tot) then
        allocate (current_node%next)
        current_node => current_node%next
      end if
      if (i_process == flavor(i_flv)%flv_mult) &
         proc_offset = proc_offset + flavor(i_flv)%flv_mult
    end do
    deallocate (pdg_in, pdg_out)
    deallocate (flavor_state)
  end do

contains

  subroutine allocate_and_init_pdg (node, pdg_in, pdg_out, amp_type)
    type(blha_cfg_process_node_t), intent(inout), pointer :: node
    integer, intent(in), dimension(:), allocatable :: pdg_in, pdg_out
    integer, intent(in) :: amp_type
    allocate (node%pdg_in (size (pdg_in)))
    allocate (node%pdg_out (size (pdg_out)))
    node%pdg_in%pdg = pdg_in
    node%pdg_out%pdg = pdg_out
    node%amplitude_type = amp_type
  end subroutine allocate_and_init_pdg

  subroutine allocate_and_init_pdg_and_helicities (node, pdg_in, pdg_out, amp_type)
    type(blha_cfg_process_node_t), intent(inout), pointer :: node
    integer, intent(in), dimension(:), allocatable :: pdg_in, pdg_out
    integer, intent(in) :: amp_type
    integer :: h1, h2
    if (size (pdg_in) == 2) then
       do h1 = -1, 1, 2
          do h2 = -1, 1, 2
             call allocate_and_init_pdg (current_node, pdg_in, pdg_out, amp_type)
             current_node%pdg_in(1)%polarized = .true.
             current_node%pdg_in(2)%polarized = .true.
             current_node%pdg_in(1)%hel = h1
             current_node%pdg_in(2)%hel = h2
             if (h1 + h2 /= 2) then !!! not end of loop
                allocate (current_node%next)
                current_node => current_node%next
             end if
          end do
       end do
    else
       do h1 = -1, 1, 2
          call allocate_and_init_pdg (current_node, pdg_in, pdg_out, amp_type)
          current_node%pdg_in(1)%polarized = .true.
```

```
            current_node%pdg_in(1)%hel = h1
            if (h1 /= 1) then !!! not end of loop
               allocate (current_node%next)
               current_node => current_node%next
            end if
         end do
      end if
   end subroutine allocate_and_init_pdg_and_helicities

end subroutine blha_configuration_append_processes
```

Change parameter(s).

⟨*BLHA config: public*⟩+≡

```
public :: blha_configuration_set
```

⟨*BLHA config: procedures*⟩+≡

```
subroutine blha_configuration_set (cfg, &
     version, correction_type, irreg, massive_particle_scheme, &
     model_file, alphas_power, alpha_power, ew_scheme, width_scheme, &
     accuracy, debug)
  type(blha_configuration_t), intent(inout) :: cfg
  integer, optional, intent(in) :: version
  integer, optional, intent(in) :: correction_type
  integer, optional, intent(in) :: irreg
  integer, optional, intent(in) :: massive_particle_scheme
  type(string_t), optional, intent(in) :: model_file
  integer, optional, intent(in) :: alphas_power, alpha_power
  integer, optional, intent(in) :: ew_scheme
  integer, optional, intent(in) :: width_scheme
  real(default), optional, intent(in) :: accuracy
  logical, optional, intent(in) :: debug
  if (present (version)) &
     cfg%version = version
  if (present (correction_type)) &
     cfg%correction_type = correction_type
  if (present (irreg)) &
     cfg%irreg = irreg
  if (present (massive_particle_scheme)) &
     cfg%massive_particle_scheme = massive_particle_scheme
  if (present (model_file)) &
     cfg%model_file = model_file
  if (present (alphas_power)) &
     cfg%alphas_power = alphas_power
  if (present (alpha_power)) &
     cfg%alpha_power = alpha_power
  if (present (ew_scheme)) &
     cfg%ew_scheme = ew_scheme
  if (present (width_scheme)) &
     cfg%width_scheme = width_scheme
  if (present (accuracy)) &
     cfg%accuracy_target = accuracy
  if (present (debug)) &
     cfg%debug_unstable = debug
  cfg%dirty = .false.
```

```
end subroutine blha_configuration_set
```

⟨*BLHA config: public*⟩+≡

```
public :: blha_configuration_get_n_proc
```

⟨*BLHA config: procedures*⟩+≡

```
function blha_configuration_get_n_proc (cfg) result (n_proc)
  type(blha_configuration_t), intent(in) :: cfg
  integer :: n_proc
  n_proc = cfg%n_proc
end function blha_configuration_get_n_proc
```

Write the BLHA file. Internal mode is intented for md5summing only.

⟨*BLHA config: public*⟩+≡

```
public :: blha_configuration_write
```

⟨*BLHA config: procedures*⟩+≡

```
subroutine blha_configuration_write (cfg, unit, internal, no_version)
  type(blha_configuration_t), intent(in) :: cfg
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: internal, no_version
  integer :: u
  logical :: full
  type(string_t) :: buf
  type(blha_cfg_process_node_t), pointer :: node
  integer :: i
  character(3) :: pdg_char
  character(4) :: hel_char
  character(len=25), parameter :: pad = ""
  logical :: write_process, no_v
  no_v = .false. ; if (present (no_version))  no_v = no_version
  u = given_output_unit (unit); if (u < 0) return
  full = .true.; if (present (internal)) full = .not. internal
  if (full .and. cfg%dirty) call msg_bug ( &
     "BUG: attempted to write out a dirty BLHA configuration")
  if (full) then
     if (no_v) then
        write (u, "(A)") "# BLHA order written by WHIZARD [version]"
     else
        write (u, "(A)") "# BLHA order written by WHIZARD ⟨Version⟩"
     end if
     write (u, "(A)")
  end if
  select case (cfg%mode)
     case (BLHA_MODE_GOSAM); buf = "GoSam"
     case (BLHA_MODE_OPENLOOPS); buf = "OpenLoops"
     case default; buf = "vanilla"
  end select
  write (u, "(A)") "# BLHA interface mode: " // char (buf)
  write (u, "(A)") "# process: " // char (cfg%name)
  write (u, "(A)") "# model: " // char (cfg%model%get_name ())
  select case (cfg%version)
     case (1); buf = "BLHA1"
     case (2); buf = "BLHA2"
```

```
end select
write (u, '(A25,A)') "InterfaceVersion " // pad, char (buf)
select case (cfg%correction_type)
   case (BLHA_CT_QCD); buf = "QCD"
   case (BLHA_CT_EW); buf = "EW"
   case (BLHA_CT_QED); buf = "QED"
   case default; buf = cfg%correction_type_other
end select
write (u,'(A25,A)') "CorrectionType" // pad, char (buf)

select case (cfg%mode)
case (BLHA_MODE_OPENLOOPS)
   buf = cfg%name // '.olc'
   write (u, '(A25,A)') "Extra AnswerFile" // pad, char (buf)
end select

select case (cfg%irreg)
   case (BLHA_IRREG_CDR); buf = "CDR"
   case (BLHA_IRREG_DRED); buf = "DRED"
   case (BLHA_IRREG_THV); buf = "tHV"
   case (BLHA_IRREG_MREG); buf = "MassReg"
   case default; buf = cfg%irreg_other
end select
write (u,'(A25,A)') "IRregularisation" // pad, char (buf)
select case (cfg%massive_particle_scheme)
   case (BLHA_MPS_ONSHELL); buf = "OnShell"
   case default; buf = cfg%massive_particle_scheme_other
end select
if (cfg%mode == BLHA_MODE_GOSAM) &
   write (u,'(A25,A)') "MassiveParticleScheme" // pad, char (buf)
select case (cfg%version)
case (1)
  if (cfg%alphas_power >= 0) write (u,'(A25,A)') &
     "AlphasPower" // pad, int2char (cfg%alphas_power)
  if (cfg%alpha_power >= 0) write (u,'(A25,A)') &
     "AlphaPower " // pad, int2char (cfg%alpha_power)
case (2)
  if (cfg%alphas_power >= 0) write (u,'(A25,A)') &
     "CouplingPower QCD " // pad, int2char (cfg%alphas_power)
  if (cfg%alpha_power >= 0) write (u, '(A25,A)') &
     "CouplingPower QED " // pad, int2char (cfg%alpha_power)
end select
if (cfg%ew_scheme > BLHA_EW_QED) then
   select case (cfg%mode)
   case (BLHA_MODE_GOSAM)
      select case (cfg%ew_scheme)
         case (BLHA_EW_GF); buf = "alphaGF"
         case (BLHA_EW_MZ); buf = "alphaMZ"
         case (BLHA_EW_MSBAR); buf = "alphaMSbar"
         case (BLHA_EW_0); buf = "alpha0"
         case (BLHA_EW_RUN); buf = "alphaRUN"
      end select
      write (u, '(A25, A)') "EWScheme " // pad, char (buf)
   case (BLHA_MODE_OPENLOOPS)
```

```
      select case (cfg%ew_scheme)
         case (BLHA_EW_0); buf = "alpha0"
         case (BLHA_EW_GF); buf = "Gmu"
         case default
            call msg_fatal ("OpenLoops input: Only supported EW schemes are 'alpha0' and 'Gmu'
      end select
      write (u, '(A25, A)') "ewscheme " // pad, char (buf)
   end select
end if
select case (cfg%mode)
case (BLHA_MODE_GOSAM)
   write (u, '(A25)', advance='no') "MassiveParticles " // pad
   do i = 1, size (OLP_MASSIVE_PARTICLES)
      if (OLP_MASSIVE_PARTICLES(i) > 0) &
         write (u, '(I2,1X)', advance='no') OLP_MASSIVE_PARTICLES(i)
   end do
   write (u,*)
case (BLHA_MODE_OPENLOOPS)
   if (cfg%openloops_use_cms) then
      write (u, '(A25,I1)') "extra use_cms " // pad, 1
   else
      write (u, '(A25,I1)') "extra use_cms " // pad, 0
   end if
   write (u, '(A25,I1)') "extra me_cache " // pad, 0
   if (cfg%openloops_phs_tolerance > 0) then
      write (u, '(A25,A4,I0)') "extra psp_tolerance " // pad, "10e-", &
         cfg%openloops_phs_tolerance
   end if
   call check_extra_cmd (cfg%openloops_extra_cmd)
   write (u, '(A)') char (cfg%openloops_extra_cmd)
   if (cfg%openloops_stability_log > 0) &
      write (u, '(A25,I1)') "extra stability_log " // pad, cfg%openloops_stability_log
   if (cfg%openloops_use_collier) &
      write (u, '(A25,I1)') "extra preset " // pad, 2
end select
if (full) then
   write (u, "(A)")
   write (u, "(A)") "# Process definitions"
   write (u, "(A)")
end if
if (cfg%debug_unstable) &
  write (u, '(A25,A)') "DebugUnstable " // pad, "True"
write (u, *)
node => cfg%processes
do while (associated (node))
   write_process = .true.
   select case (node%amplitude_type)
     case (BLHA_AMP_LOOP); buf = "Loop"
     case (BLHA_AMP_CC); buf = "ccTree"
     case (BLHA_AMP_SC)
        if (cfg%mode == BLHA_MODE_OPENLOOPS) then
           buf = "sctree_polvect"
        else
           buf = "scTree"
```

```
            end if
          case (BLHA_AMP_TREE); buf = "Tree"
          case (BLHA_AMP_LOOPINDUCED); buf = "LoopInduced"
        end select
        if (write_process) then
           write (u, '(A25, A)') "AmplitudeType " // pad, char (buf)
           buf = ""
           do i = 1, size (node%pdg_in)
              call node%pdg_in(i)%write_pdg (pdg_char)
              if (node%pdg_in(i)%polarized) then
                 call node%pdg_in(i)%write_helicity (hel_char)
                 buf = (buf // pdg_char // hel_char) // " "
              else
                 buf = (buf // pdg_char) // " "
              end if
           end do
           buf = buf // "-> "
           do i = 1, size (node%pdg_out)
              call node%pdg_out(i)%write_pdg (pdg_char)
              buf = (buf // pdg_char) // " "
           end do
           write (u, "(A)") char (trim (buf))
           write (u, *)
        end if
        node => node%next
      end do

    end subroutine blha_configuration_write
```

24.2.1 Unit tests

Test module, followed by the corresponding implementation module.

⟨blha_ut.f90⟩≡
  ⟨*File header*⟩

```
module blha_ut
  use unit_tests
  use blha_uti
```

⟨*Standard module head*⟩

⟨*BLHA: public tests*⟩

```
contains
```

⟨*BLHA: test driver*⟩

```
end module blha_ut
```

⟨blha_uti.f90⟩≡
  ⟨*File header*⟩

```
module blha_uti

⟨Use strings⟩
  use format_utils, only: write_separator
  use variables, only: var_list_t
  use os_interface
  use models
  use blha_config


⟨Standard module head⟩

⟨BLHA: test declarations⟩

contains

⟨BLHA: test procedures⟩

⟨BLHA: tests⟩

end module blha_uti
```

API: driver for the unit tests below.

⟨*BLHA: public tests*⟩≡

```
public :: blha_test
```

⟨*BLHA: test driver*⟩≡

```
subroutine blha_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
  call test(blha_1, "blha_1", "Test the creation of BLHA-OLP files", u, results)
  call test(blha_2, "blha_2", "Test the creation of BLHA-OLP files for "&
     &"multiple flavor structures", u, results)
  call test(blha_3, "blha_3", "Test helicity-information in OpenLoops OLP files", &
     u, results)
end subroutine blha_test
```

⟨*BLHA: test procedures*⟩≡

```
subroutine setup_and_write_blha_configuration (u, single, polarized)
  integer, intent(in) :: u
  logical, intent(in), optional :: single
  logical, intent(in), optional :: polarized
  logical :: polrzd, singl
  type(blha_master_t) :: blha_master
  integer :: i
  integer :: n_in, n_out
  integer :: alpha_power, alphas_power
  integer, dimension(:,:), allocatable :: flv_born, flv_real
  type(string_t) :: proc_id, method
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(var_list_t) :: var_list
  type(model_t), pointer :: model => null ()
```

```
integer :: openloops_phs_tolerance

polrzd = .false.; if (present (polarized)) polrzd = polarized
singl = .true.; if (present (single)) singl = single

if (singl) then
   write (u, "(A)") "* Process: e+ e- -> W+ W- b b~"
   n_in = 2; n_out = 4
   alpha_power = 4; alphas_power = 0
   allocate (flv_born (n_in + n_out, 1))
   allocate (flv_real (n_in + n_out + 1, 1))
   flv_born(1,1) = 11; flv_born(2,1) = -11
   flv_born(3,1) = 24; flv_born(4,1) = -24
   flv_born(5,1) = 5; flv_born(6,1) = -5
   flv_real(1:6,1) = flv_born(:,1)
   flv_real(7,1) = 21
else
   write (u, "(A)") "* Process: e+ e- -> u:d:s U:D:S"
   n_in = 2; n_out = 2
   alpha_power = 2; alphas_power = 0
   allocate (flv_born (n_in + n_out, 3))
   allocate (flv_real (n_in + n_out + 1, 3))
   flv_born(1,:) = 11; flv_born(2,:) = -11
   flv_born(3,1) = 1; flv_born(4,1) = -1
   flv_born(3,2) = 2; flv_born(4,2) = -2
   flv_born(3,3) = 3; flv_born(4,3) = -3
   flv_real(1:4,:) = flv_born
   flv_real(5,:) = 21
end if
proc_id = var_str ("BLHA_Test")

call syntax_model_file_init ()
call os_data_init (os_data)
call model_list%read_model &
   (var_str ("SM"), var_str ("SM.mdl"), os_data, model)

write (u, "(A)") "* BLHA matrix elements assumed for all process components"
write (u, "(A)") "* Mode: GoSam"

method = var_str ("gosam")
call var_list%append_string (var_str ("$born_me_method"), method)
call var_list%append_string (var_str ("$real_tree_me_method"), method)
call var_list%append_string (var_str ("$loop_me_method"), method)
call var_list%append_string (var_str ("$correlation_me_method"), method)
call blha_master%set_ew_scheme (var_str ("GF"))
call blha_master%set_methods (.true., var_list)
call blha_master%allocate_config_files ()
call blha_master%generate (proc_id, model, n_in, &
     alpha_power, alphas_power, flv_born, flv_real)

call test_output (u)

call blha_master%final ()
call var_list%final ()
```

```
  write (u, "(A)") "* Switch to OpenLoops"
  openloops_phs_tolerance = 7

  method = var_str ("openloops")
  call var_list%append_string (var_str ("$born_me_method"), method)
  call var_list%append_string (var_str ("$real_tree_me_method"), method)
  call var_list%append_string (var_str ("$loop_me_method"), method)
  call var_list%append_string (var_str ("$correlation_me_method"), method)
  call blha_master%set_methods (.true., var_list)
  call blha_master%allocate_config_files ()
  call blha_master%generate (proc_id, model, n_in, &
       alpha_power, alphas_power, flv_born, flv_real)

  if (polrzd) then
     do i = 1, 4
        call blha_master%set_polarization (i)
     end do
  end if
  call blha_master%setup_additional_features &
     (openloops_phs_tolerance, .false., 0, .false.)

  call test_output (u)

contains

  subroutine test_output (u)
    integer, intent(in) :: u
    do i = 1, 4
       call write_separator (u)
       call write_component_type (i, u)
       call write_separator (u)
       call blha_configuration_write &
            (blha_master%blha_cfg(i), u, no_version = .true.)
    end do
  end subroutine test_output

  subroutine write_component_type (i, u)
    integer, intent(in) :: i, u
    type(string_t) :: message, component_type
    message = var_str ("OLP-File content for ")
    select case (i)
    case (1)
       component_type = var_str ("loop")
    case (2)
       component_type = var_str ("subtraction")
    case (3)
       component_type = var_str ("real")
    case (4)
       component_type = var_str ("born")
    end select
    message = message // component_type // " matrix elements"
    write (u, "(A)") char (message)
  end subroutine write_component_type
```

```
end subroutine setup_and_write_blha_configuration
```

⟨*BLHA: test declarations*⟩≡

```
public :: blha_1
```

⟨*BLHA: tests*⟩≡

```
subroutine blha_1 (u)
  integer, intent(in) :: u
  write (u, "(A)") "* Test output: blha_1"
  write (u, "(A)") "* Purpose: Test the creation of olp-files for single "&
     &"and unpolarized flavor structures"
  write (u, "(A)")
  call setup_and_write_blha_configuration (u, single = .true., polarized = .false.)
end subroutine blha_1
```

⟨*BLHA: test declarations*⟩+≡

```
public :: blha_2
```

⟨*BLHA: tests*⟩+≡

```
subroutine blha_2 (u)
  integer, intent(in) :: u
  write (u, "(A)") "* Test output: blha_2"
  write (u, "(A)") "* Purpose: Test the creation of olp-files for multiple "&
    &"and unpolarized flavor structures"
  write (u, "(A)")
  call setup_and_write_blha_configuration (u, single = .false., polarized = .false.)
end subroutine blha_2
```

⟨*BLHA: test declarations*⟩+≡

```
public :: blha_3
```

⟨*BLHA: tests*⟩+≡

```
subroutine blha_3 (u)
  integer, intent(in) :: u
  write (u, "(A)") "* Test output: blha_3"
  write (u, "(A)") "* Purpose: Test the creation of olp-files for single "&
    &"and polarized flavor structures"
  write (u, "(A)")
  call setup_and_write_blha_configuration (u, single = .true., polarized = .true.)
end subroutine blha_3
```

Chapter 25

GoSam Interface

The code in this chapter makes amplitudes accessible to `WHIZARD` that are generated and computed by the GoSam package.

These are the modules:

loop_archive Provide some useful extra functionality.

prc_gosam The actual interface, following the `WHIZARD` conventions for matrix-element generator methods.

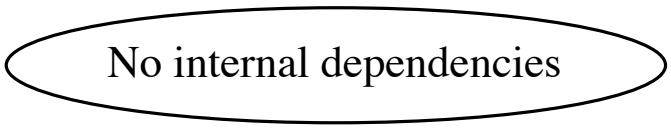

Figure 25.1: Module dependencies in `src/gosam`.

25.1 Gosam Interface

⟨prc_gosam.f90⟩≡
⟨*File header*⟩

```
module prc_gosam

  use, intrinsic :: iso_c_binding !NODEP!
  use, intrinsic :: iso_fortran_env

  use kinds
```
⟨*Use strings*⟩
```
  use io_units
  use constants
  use numeric_utils
  use system_defs, only: TAB
  use system_dependencies
  use file_utils
  use string_utils
  use physics_defs
  use diagnostics
  use os_interface
  use lorentz
  use interactions
  use pdg_arrays
  use sm_qcd
  use flavors
  use model_data
  use variables

  use process_constants
  use prclib_interfaces
  use process_libraries
  use prc_core_def
  use prc_core

  use blha_config
  use blha_olp_interfaces
```

⟨*Standard module head*⟩

⟨*prc gosam: constants*⟩

⟨*prc gosam: public*⟩

⟨*prc gosam: types*⟩

⟨*prc gosam: interfaces*⟩

```
contains
```

⟨*prc gosam: procedures*⟩

```
end module prc_gosam
```

⟨*prc gosam: types*⟩≡

```
type, extends (prc_blha_writer_t) :: gosam_writer_t
  type(string_t) :: gosam_dir
  type(string_t) :: golem_dir
  type(string_t) :: samurai_dir
  type(string_t) :: ninja_dir
  type(string_t) :: form_dir
  type(string_t) :: qgraf_dir
  type(string_t) :: filter_lo, filter_nlo
  type(string_t) :: symmetries
  integer :: form_threads
  integer :: form_workspace
  type(string_t) :: fc
contains
⟨prc gosam: gosam writer: TBP⟩
end type gosam_writer_t
```

⟨*prc gosam: public*⟩≡

```
public :: gosam_def_t
```

⟨*prc gosam: types*⟩+≡

```
type, extends (blha_def_t) :: gosam_def_t
  logical :: execute_olp = .true.
contains
⟨prc gosam: gosam def: TBP⟩
end type gosam_def_t
```

⟨*prc gosam: types*⟩+≡

```
type, extends (blha_driver_t) :: gosam_driver_t
  type(string_t) :: gosam_dir
  type(string_t) :: olp_file
  type(string_t) :: olc_file
  type(string_t) :: olp_dir
  type(string_t) :: olp_lib
contains
⟨prc gosam: gosam driver: TBP⟩
end type gosam_driver_t
```

⟨*prc gosam: public*⟩+≡

```
public :: prc_gosam_t
```

⟨*prc gosam: types*⟩+≡

```
type, extends (prc_blha_t) :: prc_gosam_t
  logical :: initialized = .false.
contains
⟨prc gosam: prc gosam: TBP⟩
end type prc_gosam_t
```

⟨*prc gosam: types*⟩+≡

```
type, extends (blha_state_t) :: gosam_state_t
contains
```

```
⟨prc gosam: gosam state: TBP⟩
end type gosam_state_t
```

⟨*prc gosam: gosam def: TBP*⟩≡

```
procedure :: init => gosam_def_init
```

⟨*prc gosam: procedures*⟩≡

```
subroutine gosam_def_init (object, basename, model_name, &
   prt_in, prt_out, nlo_type, var_list)
  class(gosam_def_t), intent(inout) :: object
  type(string_t), intent(in) :: basename
  type(string_t), intent(in) :: model_name
  type(string_t), dimension(:), intent(in) :: prt_in, prt_out
  integer, intent(in) :: nlo_type
  type(var_list_t), intent(in) :: var_list
  object%basename = basename
  allocate (gosam_writer_t :: object%writer)
  select case (nlo_type)
  case (BORN)
     object%suffix = '_BORN'
  case (NLO_REAL)
     object%suffix = '_REAL'
  case (NLO_VIRTUAL)
     object%suffix = '_LOOP'
  case (NLO_SUBTRACTION)
     object%suffix = '_SUB'
  end select
  select type (writer => object%writer)
  type is (gosam_writer_t)
    call writer%init (model_name, prt_in, prt_out)
    writer%filter_lo = var_list%get_sval (var_str ("$gosam_filter_lo"))
    writer%filter_nlo = var_list%get_sval (var_str ("$gosam_filter_nlo"))
    writer%symmetries = &
         var_list%get_sval (var_str ("$gosam_symmetries"))
    writer%form_threads = &
         var_list%get_ival (var_str ("form_threads"))
    writer%form_workspace = &
         var_list%get_ival (var_str ("form_workspace"))
    writer%fc = &
         var_list%get_sval (var_str ("$gosam_fc"))
  end select
end subroutine gosam_def_init
```

⟨*prc gosam: gosam writer: TBP*⟩≡

```
procedure :: write_config => gosam_writer_write_config
```

⟨*prc gosam: procedures*⟩+≡

```
subroutine gosam_writer_write_config (gosam_writer)
  class(gosam_writer_t), intent(in) :: gosam_writer
  integer :: unit
  unit = free_unit ()
  open (unit, file = "golem.in", status = "replace", action = "write")
  call gosam_writer%generate_configuration_file (unit)
  close(unit)
```

```
    end subroutine gosam_writer_write_config
```

⟨*prc gosam: gosam def: TBP*⟩+≡

```
    procedure, nopass :: type_string => gosam_def_type_string
```

⟨*prc gosam: procedures*⟩+≡

```
    function gosam_def_type_string () result (string)
      type(string_t) :: string
      string = "gosam"
    end function gosam_def_type_string
```

⟨*prc gosam: gosam def: TBP*⟩+≡

```
    procedure :: write => gosam_def_write
```

⟨*prc gosam: procedures*⟩+≡

```
    subroutine gosam_def_write (object, unit)
      class(gosam_def_t), intent(in) :: object
      integer, intent(in) :: unit
      select type (writer => object%writer)
      type is (gosam_writer_t)
        call writer%write (unit)
      end select
    end subroutine gosam_def_write
```

⟨*prc gosam: gosam def: TBP*⟩+≡

```
    procedure :: read => gosam_def_read
```

⟨*prc gosam: procedures*⟩+≡

```
    subroutine gosam_def_read (object, unit)
      class(gosam_def_t), intent(out) :: object
      integer, intent(in) :: unit
    end subroutine gosam_def_read
```

⟨*prc gosam: gosam def: TBP*⟩+≡

```
    procedure :: allocate_driver => gosam_def_allocate_driver
```

⟨*prc gosam: procedures*⟩+≡

```
    subroutine gosam_def_allocate_driver (object, driver, basename)
      class(gosam_def_t), intent(in) :: object
      class(prc_core_driver_t), intent(out), allocatable :: driver
      type(string_t), intent(in) :: basename
      if (.not. allocated (driver)) allocate (gosam_driver_t :: driver)
    end subroutine gosam_def_allocate_driver
```

⟨*prc gosam: gosam writer: TBP*⟩+≡

```
    procedure, nopass :: type_name => gosam_writer_type_name
```

⟨*prc gosam: procedures*⟩+≡

```
    function gosam_writer_type_name () result (string)
      type(string_t) :: string
      string = "gosam"
    end function gosam_writer_type_name
```

⟨*prc gosam: gosam writer: TBP*⟩+≡
```
procedure :: init => gosam_writer_init
```

⟨*prc gosam: procedures*⟩+≡
```
pure subroutine gosam_writer_init (writer, model_name, prt_in, prt_out, restrictions)
  class(gosam_writer_t), intent(inout) :: writer
  type(string_t), intent(in) :: model_name
  type(string_t), dimension(:), intent(in) :: prt_in, prt_out
  type(string_t), intent(in), optional :: restrictions
  writer%gosam_dir = GOSAM_DIR
  writer%golem_dir = GOLEM_DIR
  writer%samurai_dir = SAMURAI_DIR
  writer%ninja_dir = NINJA_DIR
  writer%form_dir = FORM_DIR
  writer%qgraf_dir = QGRAF_DIR
  call writer%base_init (model_name, prt_in, prt_out)
end subroutine gosam_writer_init
```

⟨*prc gosam: gosam driver: TBP*⟩≡
```
procedure, nopass :: type_name => gosam_driver_type_name
```

⟨*prc gosam: procedures*⟩+≡
```
function gosam_driver_type_name () result (string)
  type(string_t) :: string
  string = "gosam"
end function gosam_driver_type_name
```

⟨*prc gosam: gosam driver: TBP*⟩+≡
```
procedure :: init_gosam => gosam_driver_init_gosam
```

⟨*prc gosam: procedures*⟩+≡
```
subroutine gosam_driver_init_gosam (object, os_data, olp_file, &
                                    olc_file, olp_dir, olp_lib)
  class(gosam_driver_t), intent(inout) :: object
  type(os_data_t), intent(in) :: os_data
  type(string_t), intent(in) :: olp_file, olc_file, olp_dir, olp_lib
  object%gosam_dir = GOSAM_DIR
  object%olp_file = olp_file
  object%contract_file = olc_file
  object%olp_dir = olp_dir
  object%olp_lib = olp_lib
end subroutine gosam_driver_init_gosam
```

⟨*prc gosam: gosam driver: TBP*⟩+≡
```
procedure :: init_dlaccess_to_library => gosam_driver_init_dlaccess_to_library
```

⟨*prc gosam: procedures*⟩+≡
```
subroutine gosam_driver_init_dlaccess_to_library &
   (object, os_data, dlaccess, success)
  class(gosam_driver_t), intent(in) :: object
  type(os_data_t), intent(in) :: os_data
  type(dlaccess_t), intent(out) :: dlaccess
  logical, intent(out) :: success
  type(string_t) :: libname, msg_buffer
```

```
    libname = object%olp_dir // '/.libs/libgolem_olp.' // &
      os_data%shrlib_ext
    msg_buffer = "One-Loop-Provider: Using Gosam"
    call msg_message (char(msg_buffer))
    msg_buffer = "Loading library: " // libname
    call msg_message (char(msg_buffer))
    call dlaccess_init (dlaccess, var_str ("."), libname, os_data)
    success = .not. dlaccess_has_error (dlaccess)
  end subroutine gosam_driver_init_dlaccess_to_library
```

⟨*prc gosam: gosam writer: TBP*⟩+≡

```
  procedure :: generate_configuration_file => &
            gosam_writer_generate_configuration_file
```

⟨*prc gosam: procedures*⟩+≡

```
  subroutine gosam_writer_generate_configuration_file &
          (object, unit)
      class(gosam_writer_t), intent(in) :: object
      integer, intent(in) :: unit
      type(string_t) :: fc_bin
      type(string_t) :: form_bin, qgraf_bin, haggies_bin
      type(string_t) :: fcflags_golem, ldflags_golem
      type(string_t) :: fcflags_samurai, ldflags_samurai
      type(string_t) :: fcflags_ninja, ldflags_ninja
      type(string_t) :: ldflags_avh_olo, ldflags_qcdloop
      fc_bin = DEFAULT_FC
      form_bin = object%form_dir // '/bin/tform'
      qgraf_bin = object%qgraf_dir // '/bin/qgraf'
      if (object%gosam_dir /= "") then
        haggies_bin = '/usr/bin/java -jar ' // object%gosam_dir // &
                        '/share/golem/haggies/haggies.jar'
      else
        call msg_fatal ("generate_configuration_file: At least " // &
              "the GoSam Directory has to be specified!")
      end if
      if (object%golem_dir /= "") then
        fcflags_golem = "-I" // object%golem_dir // "/include/golem95"
        ldflags_golem = "-L" // object%golem_dir // "/lib -lgolem"
      end if
      if (object%samurai_dir /= "") then
        fcflags_samurai = "-I" // object%samurai_dir // "/include/samurai"
        ldflags_samurai = "-L" // object%samurai_dir // "/lib -lsamurai"
        ldflags_avh_olo = "-L" // object%samurai_dir // "/lib -lavh_olo"
        ldflags_qcdloop = "-L" // object%samurai_dir // "/lib -lqcdloop"
      end if
      if (object%ninja_dir /= "") then
        fcflags_ninja = "-I" // object%ninja_dir // "/include/ninja " &
                         // "-I" // object%ninja_dir // "/include"
        ldflags_ninja = "-L" // object%ninja_dir // "/lib -lninja"
      end if
      write (unit, "(A)") "#+avh_olo.ldflags=" &
            // char (ldflags_avh_olo)
      write (unit, "(A)") "reduction_programs=golem95, samurai, ninja"
      write (unit, "(A)") "extensions=autotools"
```

```
      write (unit, "(A)") "#+qcdloop.ldflags=" &
            // char (ldflags_qcdloop)
      write (unit, "(A)") "#+zzz.extensions=qcdloop, avh_olo"
      write (unit, "(A)") "#fc.bin=" // char (fc_bin)
      write (unit, "(A)") "form.bin=" // char (form_bin)
      write (unit, "(A)") "qgraf.bin=" // char (qgraf_bin)
      write (unit, "(A)") "#golem95.fcflags=" // char (fcflags_golem)
      write (unit, "(A)") "#golem95.ldflags=" // char (ldflags_golem)
      write (unit, "(A)") "haggies.bin=" // char (haggies_bin)
      write (unit, "(A)") "#samurai.fcflags=" // char (fcflags_samurai)
      write (unit, "(A)") "#samurai.ldflags=" // char (ldflags_samurai)
      write (unit, "(A)") "#ninja.fcflags=" // char (fcflags_ninja)
      write (unit, "(A)") "#ninja.ldflags=" // char (ldflags_ninja)
      !!! This might collide with the mass-setup in the order-file
      !!! write (unit, "(A)") "zero=mU,mD,mC,mS,mB"
      !!! This is covered by the BLHA2 interface
      write (unit, "(A)") "PSP_check=False"
      if (char (object%filter_lo) /= "") &
         write (unit, "(A)") "filter.lo=" // char (object%filter_lo)
      if (char (object%filter_nlo) /= "") &
         write (unit, "(A)") "filter.nlo=" // char (object%filter_nlo)
      if (char (object%symmetries) /= "") &
         write (unit, "(A)") "symmetries=" // char(object%symmetries)
      write (unit, "(A,I0)") "form.threads=", object%form_threads
      write (unit, "(A,I0)") "form.workspace=", object%form_workspace
      if (char (object%fc) /= "") &
         write (unit, "(A)") "fc.bin=" // char(object%fc)
   end subroutine gosam_writer_generate_configuration_file
```

We have to assure that all files necessary for the configure process in the GoSam code are ready. This is done with a stamp mechanism.

⟨*prc gosam: gosam driver: TBP*⟩+≡

```
   procedure :: write_makefile => gosam_driver_write_makefile
```

⟨*prc gosam: procedures*⟩+≡

```
   subroutine gosam_driver_write_makefile (object, unit, libname)
     class(gosam_driver_t), intent(in) :: object
     integer, intent(in) :: unit
     type(string_t), intent(in) :: libname
     write (unit, "(2A)")  "OLP_FILE = ", char (object%olp_file)
     write (unit, "(2A)")  "OLP_DIR = ", char (object%olp_dir)
     write (unit, "(A)")
     write (unit, "(A)")   "all: $(OLP_DIR)/config.log"
     write (unit, "(2A)")  TAB, "make -C $(OLP_DIR) install"
     write (unit, "(A)")
     write (unit, "(3A)")  "$(OLP_DIR)/config.log: "
     write (unit, "(4A)")  TAB, char (object%gosam_dir // "/bin/gosam.py "), &
                              "--olp $(OLP_FILE) --destination=$(OLP_DIR)", &
                              " -f -z"
     write (unit, "(3A)")  TAB, "cd $(OLP_DIR); ./autogen.sh --prefix=", &
          "$(dir $(abspath $(lastword $(MAKEFILE_LIST))))"
   end subroutine gosam_driver_write_makefile
```

⟨*prc gosam: gosam driver: TBP*⟩+≡

```
   procedure :: set_alpha_s => gosam_driver_set_alpha_s
```

⟨*prc gosam: procedures*⟩+≡

```
subroutine gosam_driver_set_alpha_s (driver, alpha_s)
   class(gosam_driver_t), intent(in) :: driver
   real(default), intent(in) :: alpha_s
   integer :: ierr
   call driver%blha_olp_set_parameter &
            (c_char_'alphaS'//c_null_char, &
             dble (alpha_s), 0._double, ierr)
end subroutine gosam_driver_set_alpha_s
```

⟨*prc gosam: gosam driver: TBP*⟩+≡

```
procedure :: set_alpha_qed => gosam_driver_set_alpha_qed
```

⟨*prc gosam: procedures*⟩+≡

```
subroutine gosam_driver_set_alpha_qed (driver, alpha)
  class(gosam_driver_t), intent(inout) :: driver
  real(default), intent(in) :: alpha
  integer :: ierr
  call driver%blha_olp_set_parameter &
     (c_char_'alpha'//c_null_char, &
      dble (alpha), 0._double, ierr)
  if (ierr == 0) call parameter_error_message (var_str ('alpha'))
end subroutine gosam_driver_set_alpha_qed
```

⟨*prc gosam: gosam driver: TBP*⟩+≡

```
procedure :: set_GF => gosam_driver_set_GF
```

⟨*prc gosam: procedures*⟩+≡

```
subroutine gosam_driver_set_GF (driver, GF)
  class(gosam_driver_t), intent(inout) :: driver
  real(default), intent(in) :: GF
  integer :: ierr
  call driver%blha_olp_set_parameter &
     (c_char_'GF'//c_null_char, &
      dble(GF), 0._double, ierr)
  if (ierr == 0) call parameter_error_message (var_str ('GF'))
end subroutine gosam_driver_set_GF
```

⟨*prc gosam: gosam driver: TBP*⟩+≡

```
procedure :: set_weinberg_angle => gosam_driver_set_weinberg_angle
```

⟨*prc gosam: procedures*⟩+≡

```
subroutine gosam_driver_set_weinberg_angle (driver, sw2)
  class(gosam_driver_t), intent(inout) :: driver
  real(default), intent(in) :: sw2
  integer :: ierr
  call driver%blha_olp_set_parameter &
     (c_char_'sw2'//c_null_char, &
      dble(sw2), 0._double, ierr)
  if (ierr == 0) call parameter_error_message (var_str ('sw2'))
end subroutine gosam_driver_set_weinberg_angle
```

⟨*prc gosam: gosam driver: TBP*⟩+≡

```
procedure :: print_alpha_s => gosam_driver_print_alpha_s
```

⟨*prc gosam: procedures*⟩+≡
```
subroutine gosam_driver_print_alpha_s (object)
  class(gosam_driver_t), intent(in) :: object
  call object%blha_olp_print_parameter (c_char_'alphaS'//c_null_char)
end subroutine gosam_driver_print_alpha_s
```

⟨*prc gosam: prc gosam: TBP*⟩≡
```
procedure :: prepare_library => prc_gosam_prepare_library
```

⟨*prc gosam: procedures*⟩+≡
```
subroutine prc_gosam_prepare_library (object, os_data, libname)
  class(prc_gosam_t), intent(inout) :: object
  type(os_data_t), intent(in) :: os_data
  type(string_t), intent(in) :: libname
  select type (writer => object%def%writer)
  type is (gosam_writer_t)
     call writer%write_config ()
  end select
  call object%create_olp_library (libname)
  call object%load_driver (os_data)
end subroutine prc_gosam_prepare_library
```

⟨*prc gosam: prc gosam: TBP*⟩+≡
```
procedure :: create_and_load_extra_libraries => &
     prc_gosam_create_and_load_extra_libraries
```

⟨*prc gosam: procedures*⟩+≡
```
subroutine prc_gosam_create_and_load_extra_libraries ( &
       core, flv_states, var_list, os_data, libname, model, i_core)
  class(prc_gosam_t), intent(inout) :: core
  integer, intent(in), dimension(:,:), allocatable :: flv_states
  type(var_list_t), intent(in) :: var_list
  type(os_data_t), intent(in) :: os_data
  type(string_t), intent(in) :: libname
  type(model_data_t), intent(in), target :: model
  integer, intent(in) :: i_core
  core%sqme_tree_pos = 4
  call core%prepare_library (os_data, libname)
  call core%start ()
  call core%read_contract_file (flv_states)
  call core%set_particle_properties (model)
  call core%set_electroweak_parameters (model)
  call core%print_parameter_file (i_core)
end subroutine prc_gosam_create_and_load_extra_libraries
```

⟨*prc gosam: prc gosam: TBP*⟩+≡
```
procedure :: write_makefile => prc_gosam_write_makefile
```

⟨*prc gosam: procedures*⟩+≡
```
subroutine prc_gosam_write_makefile (object, unit, libname)
  class(prc_gosam_t), intent(in) :: object
  integer, intent(in) :: unit
  type(string_t), intent(in) :: libname
  select type (driver => object%driver)
```

```
      type is (gosam_driver_t)
         call driver%write_makefile (unit, libname)
      end select
    end subroutine prc_gosam_write_makefile
```

⟨*prc gosam: prc gosam: TBP*⟩+≡

```
    procedure :: execute_makefile => prc_gosam_execute_makefile
```

⟨*prc gosam: procedures*⟩+≡

```
    subroutine prc_gosam_execute_makefile (object, libname)
      class(prc_gosam_t), intent(in) :: object
      type(string_t), intent(in) :: libname
      select type (driver => object%driver)
      type is (gosam_driver_t)
         call os_system_call ("make -f " // &
            libname // "_gosam.makefile")
      end select
    end subroutine prc_gosam_execute_makefile
```

⟨*prc gosam: prc gosam: TBP*⟩+≡

```
    procedure :: create_olp_library => prc_gosam_create_olp_library
```

⟨*prc gosam: procedures*⟩+≡

```
    subroutine prc_gosam_create_olp_library (object, libname)
      class(prc_gosam_t), intent(inout) :: object
      type(string_t), intent(in) :: libname
      integer :: unit
      select type (driver => object%driver)
      type is (gosam_driver_t)
         unit = free_unit ()
         open (unit, file = char (libname // "_gosam.makefile"), &
              status = "replace", action= "write")
         call object%write_makefile (unit, libname)
         close (unit)
         call object%execute_makefile (libname)
      end select
    end subroutine prc_gosam_create_olp_library
```

⟨*prc gosam: prc gosam: TBP*⟩+≡

```
    procedure :: load_driver => prc_gosam_load_driver
```

⟨*prc gosam: procedures*⟩+≡

```
    subroutine prc_gosam_load_driver (object, os_data)
      class(prc_gosam_t), intent(inout) :: object
      type(os_data_t), intent(in) :: os_data
      logical :: dl_success

      select type (driver => object%driver)
      type is (gosam_driver_t)
         call driver%load (os_data, dl_success)
         if (.not. dl_success) &
            call msg_fatal ("GoSam Libraries could not be loaded")
      end select
    end subroutine prc_gosam_load_driver
```

⟨*prc gosam: prc gosam: TBP*⟩+≡
```
procedure :: start => prc_gosam_start
```

⟨*prc gosam: procedures*⟩+≡
```
subroutine prc_gosam_start (object)
  class(prc_gosam_t), intent(inout) :: object
  integer :: ierr
  if (object%includes_polarization())  &
       call msg_fatal ('GoSam does not support polarized beams!')
  select type (driver => object%driver)
  type is (gosam_driver_t)
     call driver%blha_olp_start (string_f2c (driver%contract_file), ierr)
  end select
end subroutine prc_gosam_start
```

⟨*prc gosam: prc gosam: TBP*⟩+≡
```
procedure :: write => prc_gosam_write
```

⟨*prc gosam: procedures*⟩+≡
```
subroutine prc_gosam_write (object, unit)
  class(prc_gosam_t), intent(in) :: object
  integer, intent(in), optional :: unit
  call msg_message (unit = unit, string = "GOSAM")
end subroutine prc_gosam_write
```

⟨*prc gosam: prc gosam: TBP*⟩+≡
```
procedure :: write_name => prc_gosam_write_name
```

⟨*prc gosam: procedures*⟩+≡
```
subroutine prc_gosam_write_name (object, unit)
  class(prc_gosam_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u,"(1x,A)") "Core: GoSam"
end subroutine prc_gosam_write_name
```

⟨*prc gosam: prc gosam: TBP*⟩+≡
```
procedure :: init_driver => prc_gosam_init_driver
```

⟨*prc gosam: procedures*⟩+≡
```
subroutine prc_gosam_init_driver (object, os_data)
  class(prc_gosam_t), intent(inout) :: object
  type(os_data_t), intent(in) :: os_data
  type(string_t) :: olp_file, olc_file, olp_dir
  type(string_t) :: suffix

  select type (def => object%def)
  type is (gosam_def_t)
     suffix = def%suffix
     olp_file = def%basename // suffix // '.olp'
     olc_file = def%basename // suffix // '.olc'
     olp_dir = def%basename // suffix // '_olp_modules'
  class default
```

```
      call msg_bug ("prc_gosam_init_driver: core_def should be of gosam-type")
    end select

    select type(driver => object%driver)
    type is (gosam_driver_t)
      driver%nlo_suffix = suffix
      call driver%init_gosam (os_data, olp_file, olc_file, olp_dir, &
           var_str ("libgolem_olp"))
    end select
  end subroutine prc_gosam_init_driver
```

⟨*prc gosam: prc gosam: TBP*⟩+≡

```
  procedure :: set_initialized => prc_gosam_set_initialized
```

⟨*prc gosam: procedures*⟩+≡

```
  subroutine prc_gosam_set_initialized (prc_gosam)
    class(prc_gosam_t), intent(inout) :: prc_gosam
    prc_gosam%initialized = .true.
  end subroutine prc_gosam_set_initialized
```

The BLHA-interface conventions require the quantity $S_{ij} = \langle M_{i,+}|T_iT_j|M_{i,-}\rangle$ to be produced, where i is the position of the splitting gluon. However, $\tilde{M} = \langle M_{i,-}|M_{i,+}\rangle$ is needed. This can be obtained using color conservation, $\sum_j T_j|M\rangle = 0$, so that

$$\sum_{j\neq i} S_{ij} = -\langle M_{i,+}|T_i^2|M_{i,-}\rangle = -C_A\langle M_{i,+}|M_{i,-}\rangle = -C_A\tilde{M}^*$$

According to BLHA conventions, the real part of S_{ij} is located at positions $2i + 2nj$ in the output array, where n denotes the number of external particles and the enumeration of particles starts at zero. The subsequent position, i.e. $2i + 2nj + 1$ is designated to the imaginary part of S_{ij}. Note that, since the first array position is 1, the implemented position association deviates from the above one in the addition of 1.

⟨*prc gosam: procedures*⟩+≡

⟨*prc gosam: prc gosam: TBP*⟩+≡

```
  procedure :: compute_sqme_sc => prc_gosam_compute_sqme_sc
```

⟨*prc gosam: procedures*⟩+≡

```
  subroutine prc_gosam_compute_sqme_sc (object, &
       i_flv, em, p, ren_scale, me_sc, bad_point)
    class(prc_gosam_t), intent(inout) :: object
    integer, intent(in) :: i_flv
    integer, intent(in) :: em
    type(vector4_t), intent(in), dimension(:) :: p
    real(default), intent(in) :: ren_scale
    complex(default), intent(out) :: me_sc
    logical, intent(out) :: bad_point
    real(double), dimension(5*object%n_particles) :: mom
    real(double), dimension(OLP_RESULTS_LIMIT) :: r
    real(double) :: ren_scale_dble
    integer :: i, igm1, n
    integer :: pos_real, pos_imag
```

```
    real(double) :: acc_dble
    real(default) :: acc, alpha_s
    if (object%i_sc(i_flv) > 0) then
       me_sc = cmplx (zero ,zero, kind=default)
       mom = object%create_momentum_array (p)
       if (vanishes (ren_scale)) &
          call msg_fatal ("prc_gosam_compute_sqme_sc: ren_scale vanishes")
       alpha_s = object%qcd%alpha%get (ren_scale)
       ren_scale_dble = dble (ren_scale)
       select type (driver => object%driver)
       type is (gosam_driver_t)
          call driver%set_alpha_s (alpha_s)
          call driver%blha_olp_eval2 (object%i_sc(i_flv), &
               mom, ren_scale_dble, r, acc_dble)
       end select
       igm1 = em - 1
       n = size(p)
       do i = 0, n - 1
          pos_real = 2 * igm1 + 2 * n * i + 1
          pos_imag = pos_real + 1
          me_sc = me_sc + cmplx (r(pos_real), r(pos_imag), default)
       end do

       me_sc = - conjg(me_sc) / CA

       acc = acc_dble
       if (acc > object%maximum_accuracy) bad_point = .true.
    else
       r = 0._double
    end if
  end subroutine prc_gosam_compute_sqme_sc
```

⟨*prc gosam: prc gosam: TBP*⟩+≡

```
  procedure :: allocate_workspace => prc_gosam_allocate_workspace
```

⟨*prc gosam: procedures*⟩+≡

```
  subroutine prc_gosam_allocate_workspace (object, core_state)
    class(prc_gosam_t), intent(in) :: object
    class(prc_core_state_t), intent(inout), allocatable :: core_state
    allocate (gosam_state_t :: core_state)
  end subroutine prc_gosam_allocate_workspace
```

⟨*prc gosam: gosam state: TBP*⟩≡

```
  procedure :: write => gosam_state_write
```

⟨*prc gosam: procedures*⟩+≡

```
  subroutine gosam_state_write (object, unit)
    class(gosam_state_t), intent(in) :: object
    integer, intent(in), optional :: unit
    call msg_warning (unit = unit, string = "gosam_state_write: What to write?")
  end subroutine gosam_state_write
```

Chapter 26

OpenLoops Interface

The interface to OpenLoops.

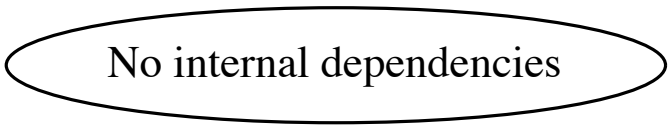

Figure 26.1: Module dependencies in `src/openloops`.

⟨prc_openloops.f90⟩≡
⟨*File header*⟩

```
module prc_openloops

  use, intrinsic :: iso_c_binding !NODEP!

  use kinds
  use io_units
```
⟨*Use strings*⟩
```
  use constants
  use numeric_utils
  use diagnostics
  use system_dependencies
  use physics_defs
  use variables
  use os_interface
  use lorentz
  use interactions
  use sm_qcd
  use sm_physics, only: top_width_sm_lo, top_width_sm_qcd_nlo_jk
  use model_data

  use prclib_interfaces
  use prc_core_def
  use prc_core

  use blha_config
  use blha_olp_interfaces
```

⟨*Standard module head*⟩

⟨*prc openloops: public*⟩

⟨*prc openloops: parameters*⟩

⟨*prc openloops: types*⟩

⟨*prc openloops: interfaces*⟩

```
contains
```

⟨*prc openloops: procedures*⟩

```
end module prc_openloops
```

⟨*prc openloops: parameters*⟩≡
```
  real(default), parameter :: openloops_default_bmass = 0._default
  real(default), parameter :: openloops_default_topmass = 172._default
  real(default), parameter :: openloops_default_topwidth = 0._default
  real(default), parameter :: openloops_default_wmass = 80.399_default
  real(default), parameter :: openloops_default_wwidth = 0._default
  real(default), parameter :: openloops_default_zmass = 91.1876_default
  real(default), parameter :: openloops_default_zwidth = 0._default
  real(default), parameter :: openloops_default_higgsmass = 125._default
```

```
real(default), parameter :: openloops_default_higgswidth = 0._default

integer :: N_EXTERNAL = 0
```

⟨*prc openloops: interfaces*⟩≡

```
abstract interface
   subroutine ol_evaluate_scpowheg (id, pp, emitter, res, resmunu) bind(C)
     import
     integer(kind = c_int), value :: id, emitter
     real(kind = c_double), intent(in) :: pp(5 * N_EXTERNAL)
     real(kind = c_double), intent(out) :: res(N_EXTERNAL), resmunu(16)
   end subroutine ol_evaluate_scpowheg
end interface
```

⟨*prc openloops: types*⟩≡

```
type, extends (prc_blha_writer_t) :: openloops_writer_t
contains
⟨prc openloops: openloops writer: TBP⟩
end type openloops_writer_t
```

⟨*prc openloops: public*⟩≡

```
public :: openloops_def_t
```

⟨*prc openloops: types*⟩+≡

```
type, extends (blha_def_t) :: openloops_def_t
   integer :: verbosity
contains
⟨prc openloops: openloops def: TBP⟩
end type openloops_def_t
```

⟨*prc openloops: types*⟩+≡

```
type, extends (blha_driver_t) :: openloops_driver_t
  integer :: n_external = 0
  type(string_t) :: olp_file
  procedure(ol_evaluate_scpowheg), nopass, pointer :: &
       evaluate_spin_correlations_powheg => null ()
contains
⟨prc openloops: openloops driver: TBP⟩
end type openloops_driver_t
```

⟨*prc openloops: types*⟩+≡

```
type :: openloops_threshold_data_t
  logical :: nlo = .true.
  real(default) :: alpha_ew
  real(default) :: sinthw
  real(default) :: m_b, m_W
  real(default) :: vtb
contains
⟨prc openloops: openloops threshold data: TBP⟩
end type openloops_threshold_data_t
```

⟨*prc openloops: openloops threshold data: TBP*⟩≡

```
procedure :: compute_top_width => openloops_threshold_data_compute_top_width
```

⟨*prc openloops: procedures*⟩≡

```
function openloops_threshold_data_compute_top_width (data, mtop, alpha_s) result (wtop)
  real(default) :: wtop
  class(openloops_threshold_data_t), intent(in) :: data
  real(default), intent(in) :: mtop, alpha_s
  if (data%nlo) then
     wtop = top_width_sm_qcd_nlo_jk (data%alpha_ew, data%sinthw, mtop, &
            data%m_W, data%m_b, alpha_s)
  else
     wtop = top_width_sm_lo (data%alpha_ew, data%sinthw, data%vtb, &
            mtop, data%m_W, data%m_b)
  end if
end function openloops_threshold_data_compute_top_width
```

⟨*prc openloops: public*⟩+≡

```
public :: openloops_state_t
```

⟨*prc openloops: types*⟩+≡

```
type, extends (blha_state_t) :: openloops_state_t
  type(openloops_threshold_data_t), allocatable :: threshold_data
contains
⟨prc openloops: openloops state: TBP⟩
end type openloops_state_t
```

⟨*prc openloops: openloops state: TBP*⟩≡

```
procedure :: init_threshold => openloops_state_init_threshold
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine openloops_state_init_threshold (object, model)
  class(openloops_state_t), intent(inout) :: object
  type(model_data_t), intent(in) :: model
  if (model%get_name () == "SM_tt_threshold") then
     allocate (object%threshold_data)
     associate (data => object%threshold_data)
        data%nlo = btest (int (model%get_real (var_str ('offshell_strategy'))), 0)
        data%alpha_ew = one / model%get_real (var_str ('alpha_em_i'))
        data%sinthw = model%get_real (var_str ('sw'))
        data%m_b = model%get_real (var_str ('mb'))
        data%m_W = model%get_real (var_str ('mW'))
        data%vtb = model%get_real (var_str ('Vtb'))
     end associate
  end if
end subroutine openloops_state_init_threshold
```

⟨*prc openloops: public*⟩+≡

```
public :: prc_openloops_t
```

⟨*prc openloops: types*⟩+≡

```
type, extends (prc_blha_t) :: prc_openloops_t
contains
⟨prc openloops: prc openloops: TBP⟩
end type prc_openloops_t
```

⟨*prc openloops: openloops writer: TBP*⟩≡
```
procedure, nopass :: type_name => openloops_writer_type_name
```

⟨*prc openloops: procedures*⟩+≡
```
function openloops_writer_type_name () result (string)
  type(string_t) :: string
  string = "openloops"
end function openloops_writer_type_name
```

⟨*prc openloops: openloops def: TBP*⟩≡
```
procedure :: init => openloops_def_init
```

⟨*prc openloops: procedures*⟩+≡
```
subroutine openloops_def_init (object, basename, model_name, &
   prt_in, prt_out, nlo_type, var_list)
  class(openloops_def_t), intent(inout) :: object
  type(string_t), intent(in) :: basename, model_name
  type(string_t), dimension(:), intent(in) :: prt_in, prt_out
  integer, intent(in) :: nlo_type
  type(var_list_t), intent(in) :: var_list
  object%basename = basename
  allocate (openloops_writer_t :: object%writer)
  select case (nlo_type)
  case (BORN)
     object%suffix = '_BORN'
  case (NLO_REAL)
     object%suffix = '_REAL'
  case (NLO_VIRTUAL)
     object%suffix = '_LOOP'
  case (NLO_SUBTRACTION)
     object%suffix = '_SUB'
  case (NLO_MISMATCH)
     object%suffix = '_MISMATCH'
  case (NLO_DGLAP)
     object%suffix = '_DGLAP'
  end select
  select type (writer => object%writer)
  class is (prc_blha_writer_t)
     call writer%init (model_name, prt_in, prt_out)
  end select
  object%verbosity = var_list%get_ival (var_str ("openloops_verbosity"))
end subroutine openloops_def_init
```

⟨*prc openloops: openloops def: TBP*⟩+≡
```
procedure, nopass :: type_string => openloops_def_type_string
```

⟨*prc openloops: procedures*⟩+≡
```
function openloops_def_type_string () result (string)
  type(string_t) :: string
  string = "openloops"
end function openloops_def_type_string
```

⟨*prc openloops: openloops def: TBP*⟩+≡
```
procedure :: write => openloops_def_write
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine openloops_def_write (object, unit)
  class(openloops_def_t), intent(in) :: object
  integer, intent(in) :: unit
  select type (writer => object%writer)
  type is (openloops_writer_t)
     call writer%write (unit)
  end select
end subroutine openloops_def_write
```

⟨*prc openloops: openloops driver: TBP*⟩≡

```
procedure :: init_dlaccess_to_library => openloops_driver_init_dlaccess_to_library
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine openloops_driver_init_dlaccess_to_library &
   (object, os_data, dlaccess, success)
  class(openloops_driver_t), intent(in) :: object
  type(os_data_t), intent(in) :: os_data
  type(dlaccess_t), intent(out) :: dlaccess
  logical, intent(out) :: success
  type(string_t) :: ol_library, msg_buffer
  ol_library = OPENLOOPS_DIR // '/lib/libopenloops.' // &
       os_data%shrlib_ext
  msg_buffer = "One-Loop-Provider: Using OpenLoops"
  call msg_message (char(msg_buffer))
  msg_buffer = "Loading library: " // ol_library
  call msg_message (char(msg_buffer))
  if (os_file_exist (ol_library)) then
     call dlaccess_init (dlaccess, var_str (""), ol_library, os_data)
  else
     call msg_fatal ("Link OpenLoops: library not found")
  end if
  success = .not. dlaccess_has_error (dlaccess)
end subroutine openloops_driver_init_dlaccess_to_library
```

⟨*prc openloops: openloops driver: TBP*⟩+≡

```
procedure :: set_alpha_s => openloops_driver_set_alpha_s
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine openloops_driver_set_alpha_s (driver, alpha_s)
  class(openloops_driver_t), intent(in) :: driver
  real(default), intent(in) :: alpha_s
  integer :: ierr
  if (associated (driver%blha_olp_set_parameter)) then
     call driver%blha_olp_set_parameter &
          (c_char_'alphas'//c_null_char, &
           dble (alpha_s), 0._double, ierr)
  else
     call msg_fatal ("blha_olp_set_parameter not associated!")
  end if
  if (ierr == 0) call parameter_error_message (var_str ('alphas'))
end subroutine openloops_driver_set_alpha_s
```

⟨*prc openloops: openloops driver: TBP*⟩+≡
```
procedure :: set_alpha_qed => openloops_driver_set_alpha_qed
```

⟨*prc openloops: procedures*⟩+≡
```
subroutine openloops_driver_set_alpha_qed (driver, alpha)
  class(openloops_driver_t), intent(inout) :: driver
  real(default), intent(in) :: alpha
  integer :: ierr
  call driver%blha_olp_set_parameter &
     (c_char_'alpha_qed'//c_null_char, &
      dble (alpha), 0._double, ierr)
  if (ierr == 0) call parameter_error_message (var_str ('alpha_qed'))
end subroutine openloops_driver_set_alpha_qed
```

⟨*prc openloops: openloops driver: TBP*⟩+≡
```
procedure :: set_GF => openloops_driver_set_GF
```

⟨*prc openloops: procedures*⟩+≡
```
subroutine openloops_driver_set_GF (driver, GF)
  class(openloops_driver_t), intent(inout) :: driver
  real(default), intent(in) :: GF
  integer :: ierr
  call driver%blha_olp_set_parameter &
     (c_char_'Gmu'//c_null_char, &
      dble(GF), 0._double, ierr)
  if (ierr == 0) call parameter_error_message (var_str ('Gmu'))
end subroutine openloops_driver_set_GF
```

⟨*prc openloops: openloops driver: TBP*⟩+≡
```
procedure :: set_weinberg_angle => openloops_driver_set_weinberg_angle
```

⟨*prc openloops: procedures*⟩+≡
```
subroutine openloops_driver_set_weinberg_angle (driver, sw2)
  class(openloops_driver_t), intent(inout) :: driver
  real(default), intent(in) :: sw2
  integer :: ierr
  call driver%blha_olp_set_parameter &
     (c_char_'sw2'//c_null_char, &
      dble(sw2), 0._double, ierr)
  if (ierr == 0) call parameter_error_message (var_str ('sw2'))
end subroutine openloops_driver_set_weinberg_angle
```

⟨*prc openloops: openloops driver: TBP*⟩+≡
```
procedure :: print_alpha_s => openloops_driver_print_alpha_s
```

⟨*prc openloops: procedures*⟩+≡
```
subroutine openloops_driver_print_alpha_s (object)
  class(openloops_driver_t), intent(in) :: object
  call object%blha_olp_print_parameter (c_char_'alphas'//c_null_char)
end subroutine openloops_driver_print_alpha_s
```

⟨*prc openloops: openloops driver: TBP*⟩+≡
```
procedure, nopass :: type_name => openloops_driver_type_name
```

⟨*prc openloops: procedures*⟩+≡

```
function openloops_driver_type_name () result (type)
  type(string_t) :: type
  type = "OpenLoops"
end function openloops_driver_type_name
```

⟨*prc openloops: openloops driver: TBP*⟩+≡

```
procedure :: load_sc_procedure => openloops_driver_load_sc_procedure
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine openloops_driver_load_sc_procedure (object, os_data, success)
  class(openloops_driver_t), intent(inout) :: object
  type(os_data_t), intent(in) :: os_data
  logical, intent(out) :: success
  type(dlaccess_t) :: dlaccess
  type(c_funptr) :: c_fptr
  logical :: init_success

  call object%init_dlaccess_to_library (os_data, dlaccess, init_success)

  c_fptr = dlaccess_get_c_funptr (dlaccess, var_str ("ol_evaluate_scpowheg"))
  call c_f_procpointer (c_fptr, object%evaluate_spin_correlations_powheg)
  if (dlaccess_has_error (dlaccess)) then
     call msg_fatal ("Could not load Openloops-powheg spin correlations!")
  else
     success = .true.
  end if

end subroutine openloops_driver_load_sc_procedure
```

⟨*prc openloops: openloops def: TBP*⟩+≡

```
procedure :: read => openloops_def_read
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine openloops_def_read (object, unit)
  class(openloops_def_t), intent(out) :: object
  integer, intent(in) :: unit
end subroutine openloops_def_read
```

⟨*prc openloops: openloops def: TBP*⟩+≡

```
procedure :: allocate_driver => openloops_def_allocate_driver
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine openloops_def_allocate_driver (object, driver, basename)
  class(openloops_def_t), intent(in) :: object
  class(prc_core_driver_t), intent(out), allocatable :: driver
  type(string_t), intent(in) :: basename
  if (.not. allocated (driver)) allocate (openloops_driver_t :: driver)
end subroutine openloops_def_allocate_driver
```

⟨*prc openloops: openloops state: TBP*⟩+≡

```
procedure :: write => openloops_state_write
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine openloops_state_write (object, unit)
  class(openloops_state_t), intent(in) :: object
  integer, intent(in), optional :: unit
end subroutine openloops_state_write
```

⟨*prc openloops: prc openloops: TBP*⟩≡

```
procedure :: allocate_workspace => prc_openloops_allocate_workspace
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine prc_openloops_allocate_workspace (object, core_state)
  class(prc_openloops_t), intent(in) :: object
  class(prc_core_state_t), intent(inout), allocatable :: core_state
  allocate (openloops_state_t :: core_state)
end subroutine prc_openloops_allocate_workspace
```

⟨*prc openloops: prc openloops: TBP*⟩+≡

```
procedure :: init_driver => prc_openloops_init_driver
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine prc_openloops_init_driver (object, os_data)
  class(prc_openloops_t), intent(inout) :: object
  type(os_data_t), intent(in) :: os_data
  type(string_t) :: olp_file, olc_file
  type(string_t) :: suffix

  select type (def => object%def)
  type is (openloops_def_t)
     suffix = def%suffix
     olp_file = def%basename // suffix // '.olp'
     olc_file = def%basename // suffix // '.olc'
  class default
     call msg_bug ("prc_openloops_init_driver: core_def should be openloops-type")
  end select

  select type (driver => object%driver)
  type is (openloops_driver_t)
     driver%olp_file = olp_file
     driver%contract_file = olc_file
     driver%nlo_suffix = suffix
  end select
end subroutine prc_openloops_init_driver
```

⟨*prc openloops: prc openloops: TBP*⟩+≡

```
procedure :: write => prc_openloops_write
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine prc_openloops_write (object, unit)
  class(prc_openloops_t), intent(in) :: object
  integer, intent(in), optional :: unit
  call msg_message (unit = unit, string = "OpenLoops")
end subroutine prc_openloops_write
```

⟨*prc openloops: prc openloops: TBP*⟩+≡

```
procedure :: write_name => prc_openloops_write_name
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine prc_openloops_write_name (object, unit)
  class(prc_openloops_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u,"(1x,A)") "Core: OpenLoops"
end subroutine prc_openloops_write_name
```

⟨*prc openloops: prc openloops: TBP*⟩+≡

```
procedure :: prepare_library => prc_openloops_prepare_library
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine prc_openloops_prepare_library (object, os_data, model)
  class(prc_openloops_t), intent(inout) :: object
  type(os_data_t), intent(in) :: os_data
  type(model_data_t), intent(in), target :: model
  call object%load_driver (os_data)
  call object%reset_parameters ()
  call object%set_particle_properties (model)
  call object%set_electroweak_parameters (model)
  select type(def => object%def)
  type is (openloops_def_t)
     call object%set_verbosity (def%verbosity)
  end select
end subroutine prc_openloops_prepare_library
```

⟨*prc openloops: prc openloops: TBP*⟩+≡

```
procedure :: load_driver => prc_openloops_load_driver
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine prc_openloops_load_driver (object, os_data)
  class(prc_openloops_t), intent(inout) :: object
  type(os_data_t), intent(in) :: os_data
  logical :: success
  select type (driver => object%driver)
  type is (openloops_driver_t)
     call driver%load (os_data, success)
     call driver%load_sc_procedure (os_data, success)
  end select
end subroutine prc_openloops_load_driver
```

⟨*prc openloops: prc openloops: TBP*⟩+≡

```
procedure :: start => prc_openloops_start
```

⟨*prc openloops: procedures*⟩+≡

```
subroutine prc_openloops_start (object)
  class(prc_openloops_t), intent(inout) :: object
  integer :: ierr
  select type (driver => object%driver)
  type is (openloops_driver_t)
```

```
      call driver%blha_olp_start (char (driver%olp_file)//c_null_char, ierr)
    end select
  end subroutine prc_openloops_start
```

⟨*prc openloops: prc openloops: TBP*⟩+≡

```
  procedure :: set_n_external => prc_openloops_set_n_external
```

⟨*prc openloops: procedures*⟩+≡

```
  subroutine prc_openloops_set_n_external (object, n)
    class(prc_openloops_t), intent(inout) :: object
    integer, intent(in) :: n
    N_EXTERNAL = n
  end subroutine prc_openloops_set_n_external
```

⟨*prc openloops: prc openloops: TBP*⟩+≡

```
  procedure :: reset_parameters => prc_openloops_reset_parameters
```

⟨*prc openloops: procedures*⟩+≡

```
  subroutine prc_openloops_reset_parameters (object)
    class(prc_openloops_t), intent(inout) :: object
    integer :: ierr
    select type (driver => object%driver)
    type is (openloops_driver_t)
       call driver%blha_olp_set_parameter ('mass(5)'//c_null_char, &
          dble(openloops_default_bmass), 0._double, ierr)
       call driver%blha_olp_set_parameter ('mass(6)'//c_null_char, &
          dble(openloops_default_topmass), 0._double, ierr)
       call driver%blha_olp_set_parameter ('width(6)'//c_null_char, &
          dble(openloops_default_topwidth), 0._double, ierr)
       call driver%blha_olp_set_parameter ('mass(23)'//c_null_char, &
          dble(openloops_default_zmass), 0._double, ierr)
       call driver%blha_olp_set_parameter ('width(23)'//c_null_char, &
          dble(openloops_default_zwidth), 0._double, ierr)
       call driver%blha_olp_set_parameter ('mass(24)'//c_null_char, &
          dble(openloops_default_wmass), 0._double, ierr)
       call driver%blha_olp_set_parameter ('width(24)'//c_null_char, &
          dble(openloops_default_wwidth), 0._double, ierr)
       call driver%blha_olp_set_parameter ('mass(25)'//c_null_char, &
          dble(openloops_default_higgsmass), 0._double, ierr)
       call driver%blha_olp_set_parameter ('width(25)'//c_null_char, &
          dble(openloops_default_higgswidth), 0._double, ierr)
    end select
  end subroutine prc_openloops_reset_parameters
```

Set the verbosity level for openloops. The different levels are as follows:

0 minimal output (startup message et.al.)

1 show which libraries are loaded

2 show debug information of the library loader, but not during run time

3 show debug information during run time

4 output for each call of `set_parameters`.

⟨*prc openloops: prc openloops: TBP*⟩+≡
```
procedure :: set_verbosity => prc_openloops_set_verbosity
```

⟨*prc openloops: procedures*⟩+≡
```
subroutine prc_openloops_set_verbosity (object, verbose)
  class(prc_openloops_t), intent(inout) :: object
  integer, intent(in) :: verbose
  integer :: ierr
  select type (driver => object%driver)
  type is (openloops_driver_t)
     call driver%blha_olp_set_parameter ('verbose'//c_null_char, &
        dble(verbose), 0._double, ierr)
  end select
end subroutine prc_openloops_set_verbosity
```

⟨*prc openloops: prc openloops: TBP*⟩+≡
```
procedure :: create_and_load_extra_libraries => &
     prc_openloops_create_and_load_extra_libraries
```

⟨*prc openloops: procedures*⟩+≡
```
subroutine prc_openloops_create_and_load_extra_libraries ( &
       core, flv_states, var_list, os_data, libname, model, i_core)
  class(prc_openloops_t), intent(inout) :: core
  integer, intent(in), dimension(:,:), allocatable :: flv_states
  type(var_list_t), intent(in) :: var_list
  type(os_data_t), intent(in) :: os_data
  type(string_t), intent(in) :: libname
  type(model_data_t), intent(in), target :: model
  integer, intent(in) :: i_core
  core%sqme_tree_pos = 1
  call core%set_n_external (core%data%get_n_tot ())
  call core%prepare_library (os_data, model)
  call core%start ()
  call core%read_contract_file (flv_states)
  call core%print_parameter_file (i_core)
  call core%reset_helicity_list ()
end subroutine prc_openloops_create_and_load_extra_libraries
```

Computes a spin-correlated matrix element from an interface to an external one-loop provider. The output of `blha_olp_eval2` is an array of `dimension(16)`. The current interface does not give out an accuracy, so that `bad_point` is always `.false.`. OpenLoops includes a factor of 1 / `n_hel` in the amplitudes, which we have to undo if polarized matrix elements are requested.

⟨*prc openloops: prc openloops: TBP*⟩+≡
```
procedure :: compute_sqme_sc => prc_openloops_compute_sqme_sc
```

⟨*prc openloops: procedures*⟩+≡
```
subroutine prc_openloops_compute_sqme_sc (object, &
     i_flv, em, p, ren_scale, sqme_sc, bad_point)
  class(prc_openloops_t), intent(inout) :: object
  integer, intent(in) :: i_flv
  integer, intent(in) :: em
  type(vector4_t), intent(in), dimension(:) :: p
  real(default), intent(in) :: ren_scale
```

```
  real(default), intent(out), dimension(0:3, 0:3) :: sqme_sc
  logical, intent(out) :: bad_point
  real(double), dimension(5*N_EXTERNAL) :: mom
  real(double), dimension(N_EXTERNAL) :: res
  real(double), dimension(16) :: res_munu
  real(default) :: alpha_s
  if (object%i_sc(i_flv) > 0) then
     mom = object%create_momentum_array (p)
     sqme_sc = zero
     if (vanishes (ren_scale)) &
          call msg_fatal ("prc_openloops_compute_sqme_sc: ren_scale vanishes")
     alpha_s = object%qcd%alpha%get (ren_scale)

     select type (driver => object%driver)
     type is (openloops_driver_t)
        call driver%set_alpha_s (alpha_s)
        call driver%evaluate_spin_correlations_powheg &
             (object%i_sc(i_flv), mom, em, res, res_munu)
     end select
     sqme_sc = reshape (res_munu, (/4,4/))
     bad_point = .false.
     if (object%includes_polarization ()) sqme_sc = object%n_hel * sqme_sc
  else
     sqme_sc = zero
  end if
end subroutine prc_openloops_compute_sqme_sc
```

Chapter 27

FKS Subtraction Scheme

The code in this chapter implements the FKS subtraction scheme for use with `WHIZARD`.

These are the modules:

fks_regions Given a process definition, identify singular regions in the associated phase space.

virtual Handle the virtual correction matrix element.

real_subtraction Handle the real-subtraction matrix element.

nlo_data Manage the subtraction objects.

This chapter deals with next-to-leading order contributions to cross sections. Basically, there are three major issues to be adressed: The creation of the $N+1$-particle flavor structure, the construction of the $N+1$-particle phase space and the actual calculation of the real- and virtual-subtracted matrix elements. The first is dealt with using the `auto_components` class, and it will be shown that the second and third issue are connected in FKS subtraction.

27.1 Brief outline of FKS subtraction

In the current state, this discussion is only concerned with lepton collisions. For hadron collisions, renormalization of parton distributions has to be taken into account. Further, for QCD corrections, initial-state radiation is necessarily present. However, most quantities have so far been only constructed for final-state emissions

The aim is to calculate the next-to-leading order cross section according to

$$d\sigma_{\mathrm{NLO}} = \mathcal{B} + \mathcal{V} + \mathcal{R} d\Phi_{\mathrm{rad}}.$$

Analytically, the divergences, in terms of poles in the complex quantity $\varepsilon = 2 - d/2$, cancel. However, this is in general only valid in an arbitrary, comlex number of dimensions. This is, roughly, the content of the KLN-theorem. `WHIZARD`, as any other numerical program, is confined to four dimensions. We will assume

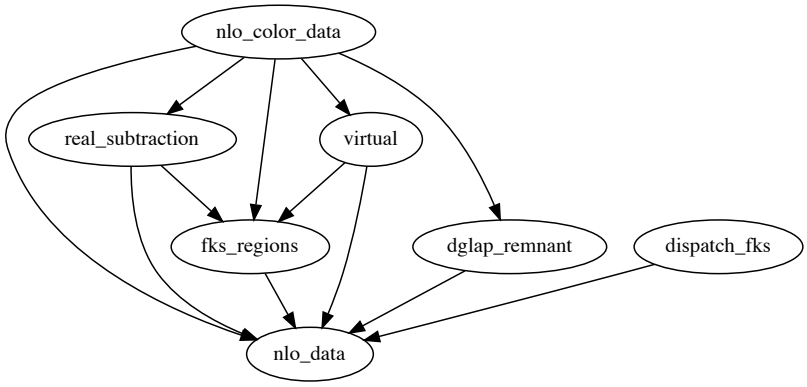

Figure 27.1: Module dependencies in `src/fks`.

that the KLN-theorem is valid and that there exist subtraction terms $\mathcal{C}$ such that

$$d\sigma_{\mathrm{NLO}} = \mathcal{B} + \underbrace{\mathcal{V} + \mathcal{C}}_{\text{finite}} + \underbrace{\mathcal{R} - \mathcal{C}}_{\text{finite}},$$

i.e. the subtraction terms correspond to the divergent limits of the real and virtual matrix element.

Because $\mathcal{C}$ subtracts the divergences of $\mathcal{R}$ as well as those of $\mathcal{V}$, it suffices to consider one of them, so we focus on $\mathcal{R}$. For this purpose, $\mathcal{R}$ is rewritten,

$$\mathcal{R} = \frac{1}{\xi^2}\frac{1}{1-y}\left(\xi^2(1-y)\mathcal{R}\right) = \frac{1}{\xi^2}\frac{1}{1-y}\tilde{\mathcal{R}},$$

with $\xi = \left(2k^0_{\mathrm{rad}}\right)/\sqrt{s}$ and $y = \cos\theta$, where k^0_{rad} denotes the energy of the radiated parton and θ is the angle between emitter and radiated parton. $\tilde{\mathcal{R}}$ is finite, therefore the whole singularity structure is contained in the prefactor $\xi^{-2}(1-y)^{-1}$. Combined with the d-dimensional phase space element,

$$\frac{d^{d-1}k}{2k^0(2\pi)^{d-1}} = \frac{s^{1-\varepsilon}}{(4\pi)^{d-1}}\xi^{1-2\varepsilon}\left(1-y^2\right)^{-\varepsilon} d\xi dy d\Omega^{d-2},$$

this yields

$$d\Phi_{\mathrm{rad}}\mathcal{R} = dy(1-y)^{-1-\varepsilon}d\xi\xi^{-1-2\varepsilon}\tilde{R}.$$

This can further be rewritten in terms of plus-distributions,

$$\xi^{-1-2\varepsilon} = -\frac{1}{2\varepsilon}\delta(\xi) + \left(\frac{1}{\xi}\right)_+ - 2\varepsilon\left(\frac{\log\xi}{\xi}\right)_+ + \mathcal{O}(\varepsilon^2),$$

$$(1-y)^{-1-\varepsilon} = -\frac{2^{-\varepsilon}}{\varepsilon}\delta(1-y) + \left(\frac{1}{1-y}\right)_+ - \varepsilon\left(\frac{1}{1-y}\right)_+ \log(1-y) + \mathcal{O}(\varepsilon^2),$$

(imagine that all this is written inside of integrals, which are spared for ease of notation) such that

$$d\Phi_{\text{rad}}\mathcal{R} = -\frac{1}{2\varepsilon}dy(1-y)^{-1-\varepsilon}\tilde{R}(0,y) - d\xi\left[\frac{2^{-\varepsilon}}{\varepsilon}\left(\frac{1}{\xi}\right)_+ - 2\left(\frac{\log\xi}{\xi}\right)_+\right]\tilde{R}(\xi,1)$$
$$+ dyd\xi\left(\frac{1}{\xi}\right)_+\left(\frac{1}{1-y}\right)_+\tilde{R}(\xi,y) + \mathcal{O}(\varepsilon).$$

The summand in the second line is of order $\mathcal{O}(1)$ and is the only one to reproduce $\mathcal{R}\xi, y$. It thus constitutes the sum of the real matrix element and the corresponding counterterms. The first summand consequently consists of the subtraction terms to the virtual matrix elements. Above formula thus allows to calculate all quantities to render the matrix elements finite.

alr	flst_alr	emi	ftuple_list
1	[-11,11,2,-2,21,21]	3	(3,5), (3,6), (4,5), (4,6), (5,6)
2	[-11,11,2,-2,21,21]	4	(3,5), (3,6), (4,5), (4,6), (5,6)
3	[-11,11,2,-2,21,21]	5	(3,5), (3,6), (4,5), (4,6), (5,6)
4	[-11,11,2,-2,2,-2]	5	(5,6)

Table 27.1: List of singular regions. The particles are represented by their PDG codes. The third column contains the emitter for the specific singular region. For the process involving an additional gluon, the gluon can either be emitted from one of the quarks or from the first gluon. Each emitter yields the same list of fundamental tuples, five in total. The last singular region corresponds to the process where the gluon splits up into two quarks. Here, there is only one fundamental tuple, corresponding to a singular configuration of the momenta of the additional quarks.

27.2 Identifying singular regions

In the FKS subtraction scheme, the phase space is decomposed into disjoint singular regions, such that

$$\sum_i \mathcal{S}_i + \sum_{ij} \mathcal{S}_{ij} = 1. \tag{27.1}$$

The quantities $\mathcal{S}_i$ and $\mathcal{S}_{ij}$ are functions of phase space corresponding to a pair of particles indices which can make up a divergent phase space region. We call such an index pair a fundamental tuple. For example, the process $e^+ e^- \to u\,\bar{u}\,g$ has two singular regions, $(3,5)$ and $(4,5)$, indicating that the gluon can be soft or collinear with respect to either the quark or the anti-quark. Therefore, the functions S_{ij} have to be chosen in such a way that their contribution makes up most of (27.1) in phase-space configurations where (final-state) particle j is collinear to particle i or/and particle j is soft. The functions S_i is the corresponding quantity for initial-state divergences.

As a singular region we understand the collection of real flavor structures associated with an emitter and a list of all possible fundamental tuples. As an example, consider the process $e^+ e^- \to u\,\bar{u}\,g$. At next-to-leading order, processes with an additionally radiated particle have to be considered. In this case, these are $e^+ e^- \to u\,\bar{u}$, $g\,g$, and $e^+ e^- \to u\,\bar{u}\,u\,\bar{u}$ (or the same process with any other quark). Table 27.2 sums up all possible singular regions for this problem.

Thus, during the preparation of a NLO-calculation, the possible singular regions have to be identified. `fks_regions.f90` deals with this issue.

alr	ftuple	emitter	flst_alr
1	$(3,5)$	5	[-11,11,-2,21,2,21]
2	$(4,5)$	5	[-11,11,2,21,-2,21]
3	$(3,6)$	5	[-11,11,-2,21,2,21]
4	$(4,6)$	5	[-11,11,2,21,-2,21]
5	$(5,6)$	5	[-11,11,2,-2,21,21]
6	$(5,6)$	5	[-11,11,2,-2,2,-2]

Table 27.2: Initial list of singular regions

27.3 FKS Regions

⟨fks_regions.f90⟩≡
⟨*File header*⟩

```
module fks_regions
```

⟨*Use kinds*⟩

```
  use format_utils, only: write_separator
  use numeric_utils, only: remove_duplicates_from_list, extend_integer_array
  use numeric_utils, only: remove_duplicates_from_list
  use string_utils, only: str
  use io_units
  use os_interface
```

⟨*Use strings*⟩

```
  use constants
  use permutations
  use diagnostics
  use flavors
  use process_constants
  use lorentz
  use pdg_arrays
  use models
  use physics_defs
  use resonances, only: resonance_contributors_t, resonance_history_t
  use phs_fks, only: phs_identifier_t, check_for_phs_identifier

  use nlo_data
```

⟨*Standard module head*⟩

⟨*fks regions: public*⟩

⟨*fks regions: parameters*⟩

⟨*fks regions: types*⟩

⟨*fks regions: interfaces*⟩

```
contains
```

⟨*fks regions: procedures*⟩

```
end module fks_regions
```

We need these parameters for various temporary buffers

⟨*fks regions: parameters*⟩≡

```
  integer, parameter, public :: N_MAX_ALR = 200
  integer, parameter, public :: N_MAX_FLV = 50
```

There are three fundamental splitting types: $q \to qg$, $g \to gg$ and $g \to qq$.

⟨*fks regions: parameters*⟩+≡

```
  integer, parameter :: UNDEFINED_SPLITTING = 0
  integer, parameter :: Q_TO_QG = 1
```

```
integer, parameter :: G_TO_GG = 2
integer, parameter :: G_TO_QQ = 3
```

We group the indices of the emitting and the radiated particle in the `ftuple`-object.

⟨*fks regions: public*⟩≡
```
public :: ftuple_t
```

⟨*fks regions: types*⟩≡
```
type :: ftuple_t
  integer, dimension(2) :: ireg = [-1,-1]
  integer :: i_res = 0
  integer :: splitting_type
  logical :: pseudo_isr = .false.
contains
⟨fks regions: ftuple: TBP⟩
end type ftuple_t
```

⟨*fks regions: interfaces*⟩≡
```
interface assignment(=)
   module procedure ftuple_assign
end interface

interface operator(==)
   module procedure ftuple_equal
end interface
```

⟨*fks regions: procedures*⟩≡
```
pure subroutine ftuple_assign (ftuple_out, ftuple_in)
  type(ftuple_t), intent(out) :: ftuple_out
  type(ftuple_t), intent(in) :: ftuple_in
  ftuple_out%ireg = ftuple_in%ireg
  ftuple_out%i_res = ftuple_in%i_res
  ftuple_out%splitting_type = ftuple_in%splitting_type
  ftuple_out%pseudo_isr = ftuple_in%pseudo_isr
end subroutine ftuple_assign
```

⟨*fks regions: procedures*⟩+≡
```
elemental function ftuple_equal (f1, f2) result (value)
  logical :: value
  type(ftuple_t), intent(in) :: f1, f2
  value = all (f1%ireg == f2%ireg) .and. f1%i_res == f2%i_res &
       .and. f1%splitting_type == f2%splitting_type &
       .and. (f1%pseudo_isr .eqv. f2%pseudo_isr)
end function ftuple_equal
```

⟨*fks regions: procedures*⟩+≡
```
elemental function ftuple_compare (f1, f2) result (greater)
  logical :: greater
  type(ftuple_t), intent(in) :: f1, f2
  if (f1%ireg(1) == f2%ireg(1)) then
     greater = f1%ireg(2) > f2%ireg(2)
```

```
    else
       greater = f1%ireg(1) > f2%ireg(2)
    end if
  end function ftuple_compare
```

⟨*fks regions: ftuple: TBP*⟩≡

```
  procedure :: write => ftuple_write
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine ftuple_write (ftuple, unit, newline)
    class(ftuple_t), intent(in) :: ftuple
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: newline
    integer :: u
    logical :: nl
    u = given_output_unit (unit); if (u < 0) return
    nl = .true.; if (present(newline)) nl = newline
    if (all (ftuple%ireg > -1)) then
       if (ftuple%i_res > 0) then
          if (nl) then
             write (u, "(A1,I1,A1,I1,A1,I1,A1)") &
                 '(', ftuple%ireg(1), ',', ftuple%ireg(2), ';', ftuple%i_res, ')'
          else
             write (u, "(A1,I1,A1,I1,A1,I1,A1)", advance = "no") &
                 '(', ftuple%ireg(1), ',', ftuple%ireg(2), ';', ftuple%i_res, ')'
          end if
       else
          if (nl) then
             write (u, "(A1,I1,A1,I1,A1)") &
                  '(', ftuple%ireg(1), ',', ftuple%ireg(2), ')'
          else
             write (u, "(A1,I1,A1,I1,A1)", advance = "no") &
                  '(', ftuple%ireg(1), ',', ftuple%ireg(2), ')'
          end if
       end if
    else
       write (u, "(A)") "(Empty)"
    end if
  end subroutine ftuple_write
```

⟨*fks regions: procedures*⟩+≡

```
  function ftuple_string (ftuples, latex)
    type(string_t) :: ftuple_string
    type(ftuple_t), intent(in), dimension(:) :: ftuples
    logical, intent(in) :: latex
    integer :: i, nreg
    integer :: i1, i2
    if (latex) then
       ftuple_string = var_str ("$\left\{")
    else
       ftuple_string = var_str ("{")
    end if
    nreg = size(ftuples)
    do i = 1, nreg
```

```
      if (ftuples(i)%i_res == 0) then
         ftuple_string = ftuple_string // var_str ("(") // &
              str (ftuples(i)%ireg(1)) // var_str (",") // &
              str (ftuples(i)%ireg(2)) // var_str (")")
      else
         ftuple_string = ftuple_string // var_str ("(") // &
              str (ftuples(i)%ireg(1)) // var_str (",") // &
              str (ftuples(i)%ireg(2)) // var_str (";") // &
              str (ftuples(i)%i_res) // var_str (")")
      end if
      if (ftuples(i)%pseudo_isr) ftuple_string = ftuple_string // var_str ("*")
      if (i < nreg) ftuple_string = ftuple_string // var_str (",")
   end do
   if (latex) then
      ftuple_string = ftuple_string // var_str ("\right\}$")
   else
      ftuple_string = ftuple_string // var_str ("}")
   end if
end function ftuple_string
```

⟨*fks regions: ftuple: TBP*⟩+≡

```
procedure :: get => ftuple_get
```

⟨*fks regions: procedures*⟩+≡

```
subroutine ftuple_get (ftuple, pos1, pos2)
  class(ftuple_t), intent(in) :: ftuple
  integer, intent(out) :: pos1, pos2
  pos1 = ftuple%ireg(1)
  pos2 = ftuple%ireg(2)
end subroutine ftuple_get
```

⟨*fks regions: ftuple: TBP*⟩+≡

```
procedure :: set => ftuple_set
```

⟨*fks regions: procedures*⟩+≡

```
subroutine ftuple_set (ftuple, pos1, pos2)
  class(ftuple_t), intent(inout) :: ftuple
  integer, intent(in) ::  pos1, pos2
  ftuple%ireg(1) = pos1
  ftuple%ireg(2) = pos2
end subroutine ftuple_set
```

⟨*fks regions: ftuple: TBP*⟩+≡

```
procedure :: determine_splitting_type_fsr => ftuple_determine_splitting_type_fsr
```

⟨*fks regions: procedures*⟩+≡

```
subroutine ftuple_determine_splitting_type_fsr (ftuple, flv, i, j)
  class(ftuple_t), intent(inout) :: ftuple
  type(flv_structure_t), intent(in) :: flv
  integer, intent(in) :: i, j
  associate (flst => flv%flst)
     if (flst(i) == GLUON .and. flst(j) == GLUON) then
        ftuple%splitting_type = G_TO_GG
     else if (flst(i)+flst(j) == 0 &
```

```
            .and. is_quark (abs(flst(i)))) then
         ftuple%splitting_type = G_TO_QQ
      else if (is_quark(abs(flst(i))) .and. flst(j) == GLUON &
            .or. is_quark(abs(flst(j))) .and. flst(i) == GLUON) then
         ftuple%splitting_type = Q_TO_QG
      else
         ftuple%splitting_type = UNDEFINED_SPLITTING
      end if
   end associate
end subroutine ftuple_determine_splitting_type_fsr
```

⟨*fks regions: ftuple: TBP*⟩+≡

```
procedure :: determine_splitting_type_isr => ftuple_determine_splitting_type_isr
```

⟨*fks regions: procedures*⟩+≡

```
subroutine ftuple_determine_splitting_type_isr (ftuple, flv, i, j)
   class(ftuple_t), intent(inout) :: ftuple
   type(flv_structure_t), intent(in) :: flv
   integer, intent(in) :: i, j
   integer :: em
   em = i; if (i == 0) em = 1
   associate (flst => flv%flst)
      if (flst(em) == GLUON .and. flst(j) == GLUON) then
         ftuple%splitting_type = G_TO_GG
      else if (flst(em) == GLUON .and. is_quark(abs(flst(j)))) then
         ftuple%splitting_type = G_TO_QQ
      else if (is_quark(abs(flst(em))) .and. flst(j) == GLUON) then
         ftuple%splitting_type = Q_TO_QG
      else
         ftuple%splitting_type = UNDEFINED_SPLITTING
      end if
   end associate
end subroutine ftuple_determine_splitting_type_isr
```

Two debug functions to check the consistency of `ftuples`

⟨*fks regions: ftuple: TBP*⟩+≡

```
procedure :: has_negative_elements => ftuple_has_negative_elements
procedure :: has_identical_elements => ftuple_has_identical_elements
```

⟨*fks regions: procedures*⟩+≡

```
elemental function ftuple_has_negative_elements (ftuple) result (value)
   logical :: value
   class(ftuple_t), intent(in) :: ftuple
   value = any (ftuple%ireg < 0)
end function ftuple_has_negative_elements

elemental function ftuple_has_identical_elements (ftuple) result (value)
   logical :: value
   class(ftuple_t), intent(in) :: ftuple
   value = ftuple%ireg(1) == ftuple%ireg(2)
end function ftuple_has_identical_elements
```

Each singular region can have a different number of emitter-radiation pairs. This is coped with using the linked list `ftuple_list`.

⟨*fks regions: types*⟩+≡
```
type :: ftuple_list_t
  integer :: index = 0
  type(ftuple_t) :: ftuple
  type(ftuple_list_t), pointer :: next => null ()
  type(ftuple_list_t), pointer :: prev => null ()
  type(ftuple_list_t), pointer :: equiv => null ()
contains
 ⟨fks regions: ftuple list: TBP⟩
end type ftuple_list_t
```

⟨*fks regions: ftuple list: TBP*⟩≡
```
procedure :: write => ftuple_list_write
```

⟨*fks regions: procedures*⟩+≡
```
subroutine ftuple_list_write (list, unit, verbose)
  class(ftuple_list_t), intent(in), target :: list
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  type(ftuple_list_t), pointer :: current
  logical :: verb
  integer :: u
  u = given_output_unit (unit); if (u < 0) return
  verb = .false.; if (present (verbose)) verb = verbose
  select type (list)
  type is (ftuple_list_t)
     current => list
     do
        call current%ftuple%write (unit = u, newline = .false.)
        if (verb .and. associated (current%equiv)) write (u, '(A)', advance = "no") "'"
        if (associated (current%next)) then
           current => current%next
        else
           exit
        end if
     end do
     write (u, *) ""
  end select
end subroutine ftuple_list_write
```

⟨*fks regions: ftuple list: TBP*⟩+≡
```
procedure :: append => ftuple_list_append
```

⟨*fks regions: procedures*⟩+≡
```
subroutine ftuple_list_append (list, ftuple)
 class(ftuple_list_t), intent(inout), target :: list
 type(ftuple_t), intent(in) :: ftuple
 type(ftuple_list_t), pointer :: current

 select type (list)
 type is (ftuple_list_t)
 if (list%index == 0) then
```

```
      nullify (list%next)
      list%index = 1
      list%ftuple = ftuple
   else
      current => list
      do
       if (associated (current%next)) then
          current => current%next
       else
          allocate (current%next)
          nullify (current%next%next)
          nullify (current%next%equiv)
          current%next%prev => current
          current%next%index = current%index + 1
          current%next%ftuple = ftuple
          exit
       end if
     end do
   end if
   end select
end subroutine ftuple_list_append
```

⟨*fks regions: ftuple list: TBP*⟩+≡

```
procedure :: compare => ftuple_list_compare
```

⟨*fks regions: procedures*⟩+≡

```
function ftuple_list_compare (ftuple_list, i1, i2) result (greater)
  logical :: greater
  class(ftuple_list_t), intent(in) :: ftuple_list
  integer, intent(in) :: i1, i2
  greater = ftuple_compare (ftuple_list%get_ftuple (i1), ftuple_list%get_ftuple (i2))
end function ftuple_list_compare
```

⟨*fks regions: ftuple list: TBP*⟩+≡

```
procedure :: get_n_tuples => ftuple_list_get_n_tuples
```

⟨*fks regions: procedures*⟩+≡

```
impure elemental function ftuple_list_get_n_tuples (list) result(n_tuples)
  integer :: n_tuples
  class(ftuple_list_t), intent(in), target :: list
  type(ftuple_list_t), pointer :: current
  n_tuples = 0
  select type (list)
  type is (ftuple_list_t)
     current => list
     if (current%index > 0) then
        n_tuples = 1
        do
           if (associated (current%next)) then
              current => current%next
              n_tuples = n_tuples + 1
           else
              exit
           end if
```

```
            end do
         end if
      end select
   end function ftuple_list_get_n_tuples
```

⟨*fks regions: ftuple list: TBP*⟩+≡

```
   procedure :: get_entry => ftuple_list_get_entry
```

⟨*fks regions: procedures*⟩+≡

```
   function ftuple_list_get_entry (list, index) result (entry)
     type(ftuple_list_t), pointer :: entry
     class(ftuple_list_t), intent(in), target :: list
     integer, intent(in) :: index
     type(ftuple_list_t), pointer :: current
     integer :: i
     entry => null()
     select type (list)
     type is (ftuple_list_t)
        current => list
        if (index == 1) then
           entry => current
        else
           do i = 1, index - 1
              current => current%next
           end do
           entry => current
        end if
     end select
   end function ftuple_list_get_entry
```

⟨*fks regions: ftuple list: TBP*⟩+≡

```
   procedure :: get_ftuple => ftuple_list_get_ftuple
```

⟨*fks regions: procedures*⟩+≡

```
   function ftuple_list_get_ftuple (list, index)  result (ftuple)
     type(ftuple_t) :: ftuple
     class(ftuple_list_t), intent(in), target :: list
     integer, intent(in) :: index
     type(ftuple_list_t), pointer :: entry
     entry => list%get_entry (index)
     ftuple = entry%ftuple
   end function ftuple_list_get_ftuple
```

⟨*fks regions: ftuple list: TBP*⟩+≡

```
   procedure :: set_equiv => ftuple_list_set_equiv
```

⟨*fks regions: procedures*⟩+≡

```
   subroutine ftuple_list_set_equiv (list, i1, i2)
     class(ftuple_list_t), intent(in) :: list
     integer, intent(in) :: i1, i2
     type(ftuple_list_t), pointer :: list1, list2 => null ()
     type(ftuple_t) :: f1, f2
     select type (list)
     type is (ftuple_list_t)
```

```
      f1 = list%get_ftuple (i1)
      f2 = list%get_ftuple (i2)
         if (list%compare (i1, i2)) then
            list1 => list%get_entry (i2)
            list2 => list%get_entry (i1)
         else
            list1 => list%get_entry (i1)
            list2 => list%get_entry (i2)
         end if
         do
            if (associated (list1%equiv)) then
               list1 => list1%equiv
            else
               exit
            end if
         end do
         list1%equiv => list2
      end select
   end subroutine ftuple_list_set_equiv
```

⟨*fks regions: ftuple list: TBP*⟩+≡

```
   procedure :: check_equiv => ftuple_list_check_equiv
```

⟨*fks regions: procedures*⟩+≡

```
   function ftuple_list_check_equiv(list, i1, i2) result(eq)
      class(ftuple_list_t), intent(in) :: list
      integer, intent(in) :: i1, i2
      logical :: eq
      type(ftuple_list_t), pointer :: current
      eq = .false.
      select type (list)
      type is (ftuple_list_t)
         current => list%get_entry (i1)
         do
            if (associated (current%equiv)) then
               current => current%equiv
               if (current%index == i2) then
                  eq = .true.
                  exit
               end if
            else
               exit
            end if
         end do
      end select
   end function ftuple_list_check_equiv
```

⟨*fks regions: ftuple list: TBP*⟩+≡

```
   procedure :: to_array => ftuple_list_to_array
```

⟨*fks regions: procedures*⟩+≡

```
   subroutine ftuple_list_to_array (ftuple_list, ftuple_array, equivalences, ordered)
      class(ftuple_list_t), intent(in), target :: ftuple_list
      type(ftuple_t), intent(out), dimension(:), allocatable :: ftuple_array
```

```
    logical, intent(out), dimension(:,:), allocatable :: equivalences
    logical, intent(in) :: ordered
    integer :: i_tuple, n
    type(ftuple_list_t), pointer :: current => null ()
    integer :: i1, i2
    type(ftuple_t) :: ftuple_tmp
    logical, dimension(:), allocatable :: eq_tmp
    n = ftuple_list%get_n_tuples ()
    allocate (ftuple_array (n), equivalences (n, n))
    equivalences = .false.
    select type (ftuple_list)
    type is (ftuple_list_t)
       current => ftuple_list
       i_tuple = 1
       do
          ftuple_array(i_tuple) = current%ftuple
          if (associated (current%equiv)) then
             i1 = current%index
             i2 = current%equiv%index
             equivalences (i1, i2) = .true.
          end if
          if (associated (current%next)) then
             current => current%next
             i_tuple = i_tuple + 1
          else
             exit
          end if
       end do
    end select
    if (ordered) then
       allocate (eq_tmp (n))
       do i1 = 2, n
          i2 = i1
          do while (i2 > 1 .and. ftuple_compare (ftuple_array(i2 - 1), ftuple_array(i2)))
             ftuple_tmp = ftuple_array(i2 - 1)
             eq_tmp = equivalences(i2, :)
             ftuple_array(i2 - 1) = ftuple_array(i2)
             ftuple_array(i2) = ftuple_tmp
             equivalences(i2 - 1, :) = equivalences(i2, :)
             equivalences(i2, :) = eq_tmp
             i2 = i2 - 1
          end do
       end do
       deallocate (eq_tmp)
    end if
  end subroutine ftuple_list_to_array
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine print_equivalence_matrix (ftuple_array, equivalences)
    type(ftuple_t), intent(in), dimension(:) :: ftuple_array
    logical, intent(in), dimension(:,:) :: equivalences
    integer :: i, i1, i2
    print *, 'Equivalence matrix: '
    do i = 1, size (ftuple_array)
```

```
      call ftuple_array(i)%get(i1,i2)
      print *, 'i: ', i, '(', i1, i2, '): ', equivalences(i,:)
    end do
  end subroutine print_equivalence_matrix
```

Class for working with the flavor specification arrays.

⟨*fks regions: public*⟩+≡

```
  public :: flv_structure_t
```

⟨*fks regions: types*⟩+≡

```
  type :: flv_structure_t
    integer, dimension(:), allocatable :: flst
    integer, dimension(:), allocatable :: tag
    integer :: nlegs = 0
    integer :: n_in = 0
    logical, dimension(:), allocatable :: massive
    logical, dimension(:), allocatable :: colored
  contains
  ⟨fks regions: flv structure: TBP⟩
  end type flv_structure_t
```

Returns `true` if the two particles at position `i` and `j` in the flavor array can originate from the same splitting. For this purpose, the function first checks whether the splitting is allowed at all. If this is the case, the emitter is removed from the flavor array. If the resulting array is equivalent to the Born flavor structure `flv_born`, the pair is accepted as a valid splitting. We first check whether the splitting is possible. The array `flv_orig` contains all particles which share a vertex with the particles at position `i` and `j`. If its size is equal to zero, no splitting is possible and the subroutine is exited. Otherwise, we loop over all possible underlying Born flavor structures and check if any of them equals the actual underlying Born flavor structure. For a quark emitting a gluon, `flv_orig` contains the PDG code of the anti-quark. To be on the safe side, a second array is created, which contains both the positively and negatively signed PDG codes. Then, the origial tuple (i, j) is removed from the real flavor structure and the particles in `flv_orig2` are inserted. If the resulting Born configuration is equal to the underlying Born configuration, up to a permutation of final-state particles, the tuple (i, j) is accepted as valid.

⟨*fks regions: flv structure: TBP*⟩≡

```
  procedure :: valid_pair => flv_structure_valid_pair
```

⟨*fks regions: procedures*⟩+≡

```
  function flv_structure_valid_pair &
     (flv, i, j, flv_ref, model) result (valid)
    logical :: valid
    class(flv_structure_t), intent(in) :: flv
    integer, intent(in) :: i,j
    type(flv_structure_t), intent(in) :: flv_ref
    type(model_t), intent(in) :: model
    integer :: k, n_orig
    type(flv_structure_t) :: flv_test
    integer, dimension(:), allocatable :: flv_orig
    valid = .false.
    if (all ([i, j] <= flv%n_in)) return
```

```
    call model%match_vertex (flv%flst(i), flv%flst(j), flv_orig)
    n_orig = size (flv_orig)
    if (n_orig == 0) then
       return
    else
      do k = 1, n_orig
         if (any ([i, j] <= flv%n_in)) then
            flv_test = flv%insert_particle_isr (i, j, flv_orig(k))
         else
            flv_test = flv%insert_particle_fsr (i, j, flv_orig(k))
         end if
         valid = flv_ref .equiv. flv_test
         call flv_test%final ()
         if (valid) return
      end do
    end if
    deallocate (flv_orig)
  end function flv_structure_valid_pair
```

This function checks whether two flavor arrays are the same up to a permutation of the final-state particles

⟨*fks regions: procedures*⟩+≡

```
  function flv_structure_equivalent (flv1, flv2, with_tag) result(equiv)
    logical :: equiv
    type(flv_structure_t), intent(in) :: flv1, flv2
    logical, intent(in) :: with_tag
    type(flavor_permutation_t) :: perm
    integer :: n
    n = size (flv1%flst)
    equiv = .true.
    if (n /= size (flv2%flst)) then
       call msg_fatal &
            ('flv_structure_equivalent: flavor arrays do not have equal lengths')
    else if (flv1%n_in /= flv2%n_in) then
       call msg_fatal &
            ('flv_structure_equivalent: flavor arrays do not have equal n_in')
    else
       call perm%init (flv1, flv2, flv1%n_in, flv1%nlegs, with_tag)
       equiv = perm%test (flv2, flv1, with_tag)
       call perm%final ()
    end if
  end function flv_structure_equivalent
```

⟨*fks regions: procedures*⟩+≡

```
  function flv_structure_equivalent_no_tag (flv1, flv2) result(equiv)
    logical :: equiv
    type(flv_structure_t), intent(in) :: flv1, flv2
    equiv = flv_structure_equivalent (flv1, flv2, .false.)
  end function flv_structure_equivalent_no_tag

  function flv_structure_equivalent_with_tag (flv1, flv2) result(equiv)
    logical :: equiv
    type(flv_structure_t), intent(in) :: flv1, flv2
```

```
    equiv = flv_structure_equivalent (flv1, flv2, .true.)
  end function flv_structure_equivalent_with_tag
```

⟨*fks regions: procedures*⟩+≡

```
  pure subroutine flv_structure_assign_flv (flv_out, flv_in)
    type(flv_structure_t), intent(out) :: flv_out
    type(flv_structure_t), intent(in) :: flv_in
    flv_out%nlegs = flv_in%nlegs
    flv_out%n_in = flv_in%n_in
    if (allocated (flv_in%flst)) then
       allocate (flv_out%flst (size (flv_in%flst)))
       flv_out%flst = flv_in%flst
    end if
    if (allocated (flv_in%tag)) then
       allocate (flv_out%tag (size (flv_in%tag)))
       flv_out%tag = flv_in%tag
    end if
    if (allocated (flv_in%massive)) then
       allocate (flv_out%massive (size (flv_in%massive)))
       flv_out%massive = flv_in%massive
    end if
    if (allocated (flv_in%colored)) then
       allocate (flv_out%colored (size (flv_in%colored)))
       flv_out%colored = flv_in%colored
    end if
  end subroutine flv_structure_assign_flv
```

⟨*fks regions: procedures*⟩+≡

```
  pure subroutine flv_structure_assign_integer (flv_out, iarray)
    type(flv_structure_t), intent(out) :: flv_out
    integer, intent(in), dimension(:) :: iarray
    integer :: i
    flv_out%nlegs = size (iarray)
    allocate (flv_out%flst (flv_out%nlegs))
    allocate (flv_out%tag (flv_out%nlegs))
    flv_out%flst = iarray
    flv_out%tag = [(i, i = 1, flv_out%nlegs)]
  end subroutine flv_structure_assign_integer
```

Returs a new flavor array with the particle at position `index` removed.

⟨*fks regions: flv structure: TBP*⟩+≡

```
  procedure :: remove_particle => flv_structure_remove_particle
```

⟨*fks regions: procedures*⟩+≡

```
  function flv_structure_remove_particle (flv, index) result(flv_new)
    type(flv_structure_t) :: flv_new
    class(flv_structure_t), intent(in) :: flv
    integer, intent(in) :: index
    integer :: n1, n2
    integer :: i, removed_tag
    n1 = size (flv%flst); n2 = n1 - 1
    allocate (flv_new%flst (n2), flv_new%tag (n2))
```

```
    flv_new%nlegs = n2
    flv_new%n_in = flv%n_in
    removed_tag = flv%tag(index)
    if (index == 1) then
       flv_new%flst(1 : n2) = flv%flst(2 : n1)
       flv_new%tag(1 : n2) = flv%tag(2 : n1)
    else if (index == n1) then
       flv_new%flst(1 : n2) = flv%flst(1 : n2)
       flv_new%tag(1 : n2) = flv%tag(1 : n2)
    else
       flv_new%flst(1 : index - 1) = flv%flst(1 : index - 1)
       flv_new%flst(index : n2) = flv%flst(index + 1 : n1)
       flv_new%tag(1 : index - 1) = flv%tag(1 : index - 1)
       flv_new%tag(index : n2) = flv%tag(index + 1 : n1)
    end if
    do i = 1, n2
       if (flv_new%tag(i) > removed_tag) &
            flv_new%tag(i) = flv_new%tag(i) - 1
    end do
  end function flv_structure_remove_particle
```

⟨*fks regions: flv structure: TBP*⟩+≡

```
  procedure :: insert_particle_fsr => flv_structure_insert_particle_fsr
```

⟨*fks regions: procedures*⟩+≡

```
  function flv_structure_insert_particle_fsr (flv, i1, i2, flv_add) result (flv_new)
    type(flv_structure_t) :: flv_new
    class(flv_structure_t), intent(in) :: flv
    integer, intent(in) :: i1, i2, flv_add
    if (flv%flst(i1) + flv_add == 0 .or. flv%flst(i2) + flv_add == 0) then
       flv_new = flv%insert_particle (i1, i2, -flv_add)
    else
       flv_new = flv%insert_particle (i1, i2, flv_add)
    end if
  end function flv_structure_insert_particle_fsr
```

For ISR, the two particles are not exchangable.

⟨*fks regions: flv structure: TBP*⟩+≡

```
  procedure :: insert_particle_isr => flv_structure_insert_particle_isr
```

⟨*fks regions: procedures*⟩+≡

```
  function flv_structure_insert_particle_isr (flv, i_in, i_out, flv_add) result (flv_new)
    type(flv_structure_t) :: flv_new
    class(flv_structure_t), intent(in) :: flv
    integer, intent(in) :: i_in, i_out, flv_add
    if (flv%flst(i_in) + flv_add == 0) then
       flv_new = flv%insert_particle (i_in, i_out, -flv_add)
    else
       flv_new = flv%insert_particle (i_in, i_out, flv_add)
    end if
  end function flv_structure_insert_particle_isr
```

Removes the paritcles at position i1 and i2 and inserts a new particle at position i1.

⟨*fks regions: flv structure: TBP*⟩+≡

```
procedure :: insert_particle => flv_structure_insert_particle
```

⟨*fks regions: procedures*⟩+≡

```
function flv_structure_insert_particle (flv, i1, i2, particle) result (flv_new)
  type(flv_structure_t) :: flv_new
  class(flv_structure_t), intent(in) :: flv
  integer, intent(in) :: i1, i2, particle
  type(flv_structure_t) :: flv_tmp
  integer :: n1, n2
  integer :: new_tag
  n1 = size (flv%flst); n2 = n1 - 1
  allocate (flv_new%flst (n2), flv_new%tag (n2))
  flv_new%nlegs = n2
  flv_new%n_in = flv%n_in
  new_tag = maxval(flv%tag) + 1
  if (i1 < i2) then
     flv_tmp = flv%remove_particle (i1)
     flv_tmp = flv_tmp%remove_particle (i2 - 1)
  else if(i2 < i1) then
     flv_tmp = flv%remove_particle(i2)
     flv_tmp = flv_tmp%remove_particle(i1 - 1)
  else
     call msg_fatal ("flv_structure_insert_particle: Indices are identical!")
  end if
  if (i1 == 1) then
     flv_new%flst(1) = particle
     flv_new%flst(2 : n2) = flv_tmp%flst(1 : n2 - 1)
     flv_new%tag(1) = new_tag
     flv_new%tag(2 : n2) = flv_tmp%tag(1 : n2 - 1)
  else if (i1 == n1 .or. i1 == n2) then
     flv_new%flst(1 : n2 - 1) = flv_tmp%flst(1 : n2 - 1)
     flv_new%flst(n2) = particle
     flv_new%tag(1 : n2 - 1) = flv_tmp%tag(1 : n2 - 1)
     flv_new%tag(n2) = new_tag
  else
     flv_new%flst(1 : i1 - 1) = flv_tmp%flst(1 : i1 - 1)
     flv_new%flst(i1) = particle
     flv_new%flst(i1 + 1 : n2) = flv_tmp%flst(i1 : n2 - 1)
     flv_new%tag(1 : i1 - 1) = flv_tmp%tag(1 : i1 - 1)
     flv_new%tag(i1) = new_tag
     flv_new%tag(i1 + 1 : n2) = flv_tmp%tag(i1 : n2 - 1)
  end if
end function flv_structure_insert_particle
```

Counts the number of occurances of a particle in a flavor array

⟨*fks regions: flv structure: TBP*⟩+≡

```
procedure :: count_particle => flv_structure_count_particle
```

⟨*fks regions: procedures*⟩+≡

```
function flv_structure_count_particle (flv, part) result (n)
  class(flv_structure_t), intent(in) :: flv
```

```
    integer, intent(in) :: part
    integer :: n
    n = count (flv%flst == part)
  end function flv_structure_count_particle
```

Initializer for flavor structures

⟨*fks regions: flv structure: TBP*⟩+≡

```
  procedure :: init => flv_structure_init
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine flv_structure_init (flv, aval, n_in, tags)
    class(flv_structure_t), intent(inout) :: flv
    integer, intent(in), dimension(:) :: aval
    integer, intent(in) :: n_in
    integer, intent(in), dimension(:), optional :: tags
    integer :: i, n
    n = size (aval)
    allocate (flv%flst (n), flv%tag (n))
    flv%flst = aval
    if (present (tags)) then
       flv%tag = tags
    else
       do i = 1, n
          flv%tag(i) = i
       end do
    end if
    flv%nlegs = n
    flv%n_in = n_in
  end subroutine flv_structure_init
```

⟨*fks regions: flv structure: TBP*⟩+≡

```
  procedure :: write => flv_structure_write
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine flv_structure_write (flv, unit)
    class(flv_structure_t), intent(in) :: flv
    integer, intent(in), optional :: unit
    integer :: i, u
    u = given_output_unit (unit); if (u < 0) return
    write (u, '(A)') char (flv%to_string ())
  end subroutine flv_structure_write
```

⟨*fks regions: flv structure: TBP*⟩+≡

```
  procedure :: to_string => flv_structure_to_string
```

⟨*fks regions: procedures*⟩+≡

```
  function flv_structure_to_string (flv) result (flv_string)
    type(string_t) :: flv_string
    class(flv_structure_t), intent(in) :: flv
    integer :: i, n
    if (allocated (flv%flst)) then
       flv_string = var_str ("[")
       n = size (flv%flst)
       do i = 1, n - 1
```

```
         flv_string = flv_string // str (flv%flst(i)) // var_str(",")
      end do
      flv_string = flv_string // str (flv%flst(n)) // var_str("]")
   else
      flv_string = var_str ("[not allocated]")
   end if
end function flv_structure_to_string
```

Creates the underlying Born flavor structure for a given real flavor structure if the particle at position `emitter` is removed

⟨*fks regions: flv structure: TBP*⟩+≡

```
procedure :: create_uborn => flv_structure_create_uborn
```

⟨*fks regions: procedures*⟩+≡

```
function flv_structure_create_uborn (flv, emitter) result(flv_uborn)
  type(flv_structure_t) :: flv_uborn
  class(flv_structure_t), intent(in) :: flv
  integer, intent(in) :: emitter
  integer n_legs
  integer :: f1, f2
  n_legs = size(flv%flst)
  allocate (flv_uborn%flst (n_legs - 1), flv_uborn%tag (n_legs - 1))
  if (emitter > flv%n_in) then
     f1 = flv%flst(n_legs); f2 = flv%flst(n_legs - 1)
     if (f1 == 21) then
        !!! Emitted particle is a gluon => just remove it
        flv_uborn = flv%remove_particle(n_legs)
     else if (is_quark (f1) .and. is_quark (f2) .and. f1 + f2 == 0) then
        !!! Emission type is a gluon splitting into two quarks
        flv_uborn = flv%insert_particle(n_legs - 1, n_legs, 21)
     else
        call msg_error ("Create underlying Born: Unsupported splitting type.")
        call msg_error (char (str (flv%flst)))
        call msg_fatal ("FKS - FAIL")
     end if
  else if (emitter > 0) then
     f1 = flv%flst(n_legs); f2 = flv%flst(emitter)
     if (f1 == 21) then
        flv_uborn = flv%remove_particle(n_legs)
     else if (is_quark (f1) .and. is_gluon (f2)) then
        flv_uborn = flv%insert_particle (emitter, n_legs, -f1)
     else if (is_quark (f1) .and. is_quark (f2) .and. f1 == f2) then
        flv_uborn = flv%insert_particle(emitter, n_legs, 21)
     end if
  else
     flv_uborn = flv%remove_particle (n_legs)
  end if
end function flv_structure_create_uborn
```

⟨*fks regions: flv structure: TBP*⟩+≡

```
procedure :: init_mass_and_color => flv_structure_init_mass_and_color
```

⟨*fks regions: procedures*⟩+≡

```
subroutine flv_structure_init_mass_and_color (flv, model)
```

```
    class(flv_structure_t), intent(inout) :: flv
    type(model_t), intent(in) :: model
    integer :: i
    type(flavor_t) :: flavor
    allocate (flv%massive (flv%nlegs), flv%colored(flv%nlegs))
    do i = 1, flv%nlegs
       call flavor%init (flv%flst(i), model)
       flv%massive(i) = flavor%get_mass () > 0
       flv%colored(i) = &
            is_quark (flv%flst(i)) .or. is_gluon (flv%flst(i))
    end do
  end subroutine flv_structure_init_mass_and_color
```

⟨*fks regions: flv structure: TBP*⟩+≡

```
  procedure :: get_last_two => flv_structure_get_last_two
```

⟨*fks regions: procedures*⟩+≡

```
  function flv_structure_get_last_two (flv, n) result (flst_last)
    integer, dimension(2) :: flst_last
    class(flv_structure_t), intent(in) :: flv
    integer, intent(in) :: n
    flst_last = [flv%flst(n - 1), flv%flst(n)]
  end function flv_structure_get_last_two
```

⟨*fks regions: flv structure: TBP*⟩+≡

```
  procedure :: final => flv_structure_final
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine flv_structure_final (flv)
    class(flv_structure_t), intent(inout) :: flv
    if (allocated (flv%flst)) deallocate (flv%flst)
    if (allocated (flv%tag)) deallocate (flv%tag)
    if (allocated (flv%massive)) deallocate (flv%massive)
    if (allocated (flv%colored)) deallocate (flv%colored)
  end subroutine flv_structure_final
```

⟨*fks regions: public*⟩+≡

```
  public :: flavor_permutation_t
```

⟨*fks regions: types*⟩+≡

```
  type :: flavor_permutation_t
    integer, dimension(:,:), allocatable :: perms
  contains
  ⟨fks regions: flavor permutation: TBP⟩
  end type flavor_permutation_t
```

⟨*fks regions: flavor permutation: TBP*⟩≡

```
  procedure :: init => flavor_permutation_init
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine flavor_permutation_init (perm, flv_in, flv_ref, n_first, n_last, with_tag)
    class(flavor_permutation_t), intent(out) :: perm
    type(flv_structure_t), intent(in) :: flv_in, flv_ref
    integer, intent(in) :: n_first, n_last
```

```
    logical, intent(in) :: with_tag
    integer :: flv1, flv2, tmp
    integer :: tag1, tag2
    integer :: i, j, j_min, i_perm
    integer, dimension(:,:), allocatable :: perm_list_tmp
    type(flv_structure_t) :: flv_copy
    logical :: condition
    logical, dimension(:), allocatable :: already_correct
    flv_copy = flv_in
    allocate (perm_list_tmp (factorial (n_last - n_first - 1), 2))
    allocate (already_correct (flv_in%nlegs))
    already_correct = flv_in%flst == flv_ref%flst
    if (with_tag) &
         already_correct = already_correct .and. (flv_in%tag == flv_ref%tag)
    j_min = n_first + 1
    i_perm = 0
    do i = n_first + 1, n_last
       flv1 = flv_ref%flst(i)
       tag1 = flv_ref%tag(i)
       do j = j_min, n_last
          if (already_correct(i) .or. already_correct(j)) cycle
          flv2 = flv_copy%flst(j)
          tag2 = flv_copy%tag(j)
          condition = (flv1 == flv2) .and. i /= j
          if (with_tag) condition = condition .and. (tag1 == tag2)
          if (condition) then
             i_perm = i_perm + 1
             tmp = flv_copy%flst(i)
             flv_copy%flst(i) = flv2
             flv_copy%flst(j) = tmp
             tmp = flv_copy%tag(i)
             flv_copy%tag(i) = tag2
             flv_copy%tag(j) = tmp
             perm_list_tmp (i_perm, 1) = i
             perm_list_tmp (i_perm, 2) = j
             exit
          end if
       end do
       j_min = j_min + 1
    end do
    allocate (perm%perms (i_perm, 2))
    perm%perms = perm_list_tmp (1 : i_perm, :)
    deallocate (perm_list_tmp)
    call flv_copy%final ()
  end subroutine flavor_permutation_init
```

⟨*fks regions: flavor permutation: TBP*⟩+≡

```
  procedure :: write => flavor_permutation_write
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine flavor_permutation_write (perm, unit)
    class(flavor_permutation_t), intent(in) :: perm
    integer, intent(in), optional :: unit
    integer :: i, n, u
```

```
    u = given_output_unit (unit); if (u < 0) return
    write (u, "(A)") "Flavor permutation list: "
    n = size (perm%perms, dim = 1)
    if (n > 0) then
       do i = 1, n
          write (u, "(A1,I1,1X,I1,A1)", advance = "no") "[", perm%perms(i,1), perm%perms(i,2), "]"
          if (i < n) write (u, "(A4)", advance = "no") " // "
       end do
       write (u, "(A)") ""
    else
       write (u, "(A)") "[Empty]"
    end if
  end subroutine flavor_permutation_write
```

⟨*fks regions: flavor permutation: TBP*⟩+≡

```
  procedure :: reset => flavor_permutation_final
  procedure :: final => flavor_permutation_final
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine flavor_permutation_final (perm)
    class(flavor_permutation_t), intent(inout) :: perm
    if (allocated (perm%perms)) deallocate (perm%perms)
  end subroutine flavor_permutation_final
```

⟨*fks regions: flavor permutation: TBP*⟩+≡

```
  generic :: apply => apply_permutation, &
         apply_flavor, apply_integer, apply_ftuple
  procedure :: apply_permutation => flavor_permutation_apply_permutation
  procedure :: apply_flavor => flavor_permutation_apply_flavor
  procedure :: apply_integer => flavor_permutation_apply_integer
  procedure :: apply_ftuple => flavor_permutation_apply_ftuple
```

⟨*fks regions: procedures*⟩+≡

```
  elemental function flavor_permutation_apply_permutation (perm_1, perm_2) &
         result (perm_out)
    type(flavor_permutation_t) :: perm_out
    class(flavor_permutation_t), intent(in) :: perm_1
    type(flavor_permutation_t), intent(in) :: perm_2
    integer :: n1, n2
    n1 = size (perm_1%perms, dim = 1)
    n2 = size (perm_2%perms, dim = 1)
    allocate (perm_out%perms (n1 + n2, 2))
    perm_out%perms (1 : n1, :) = perm_1%perms
    perm_out%perms (n1 + 1: n1 + n2, :) = perm_2%perms
  end function flavor_permutation_apply_permutation
```

⟨*fks regions: procedures*⟩+≡

```
  elemental function flavor_permutation_apply_flavor (perm, flv_in, invert) &
         result (flv_out)
    type(flv_structure_t) :: flv_out
    class(flavor_permutation_t), intent(in) :: perm
    type(flv_structure_t), intent(in) :: flv_in
    logical, intent(in), optional :: invert
```

```
    integer :: i, i1, i2
    integer :: p1, p2, incr
    integer :: flv_tmp, tag_tmp
    logical :: inv
    inv = .false.; if (present(invert)) inv = invert
    flv_out = flv_in
    if (inv) then
       p1 = 1
       p2 = size (perm%perms, dim = 1)
       incr = 1
    else
       p1 = size (perm%perms, dim = 1)
       p2 = 1
       incr = -1
    end if
    do i = p1, p2, incr
       i1 = perm%perms(i,1)
       i2 = perm%perms(i,2)
       flv_tmp = flv_out%flst(i1)
       tag_tmp = flv_out%tag(i1)
       flv_out%flst(i1) = flv_out%flst(i2)
       flv_out%flst(i2) = flv_tmp
       flv_out%tag(i1) = flv_out%tag(i2)
       flv_out%tag(i2) = tag_tmp
    end do
  end function flavor_permutation_apply_flavor
```

⟨*fks regions: procedures*⟩+≡

```
  elemental function flavor_permutation_apply_integer (perm, i_in) result (i_out)
    integer :: i_out
    class(flavor_permutation_t), intent(in) :: perm
    integer, intent(in) :: i_in
    integer :: i, i1, i2
    i_out = i_in
    do i = size (perm%perms(:,1)), 1, -1
       i1 = perm%perms(i,1)
       i2 = perm%perms(i,2)
       if (i_out == i1) then
          i_out = i2
       else if (i_out == i2) then
          i_out = i1
       end if
    end do
  end function flavor_permutation_apply_integer
```

⟨*fks regions: procedures*⟩+≡

```
  elemental function flavor_permutation_apply_ftuple (perm, f_in) result (f_out)
    type(ftuple_t) :: f_out
    class(flavor_permutation_t), intent(in) :: perm
    type(ftuple_t), intent(in) :: f_in
    integer :: i, i1, i2
    f_out = f_in
    do i = size (perm%perms, dim = 1), 1, -1
```

```
      i1 = perm%perms(i,1)
      i2 = perm%perms(i,2)
      if (f_out%ireg(1) == i1) then
         f_out%ireg(1) = i2
      else if (f_out%ireg(1) == i2) then
         f_out%ireg(1) = i1
      end if
      if (f_out%ireg(2) == i1) then
         f_out%ireg(2) = i2
      else if (f_out%ireg(2) == i2) then
         f_out%ireg(2) = i1
      end if
    end do
    if (f_out%ireg(1) > f_out%ireg(2)) f_out%ireg = f_out%ireg([2,1])
  end function flavor_permutation_apply_ftuple
```

⟨*fks regions: flavor permutation: TBP*⟩+≡

```
  procedure :: test => flavor_permutation_test
```

⟨*fks regions: procedures*⟩+≡

```
  function flavor_permutation_test (perm, flv1, flv2, with_tag) result (valid)
    logical :: valid
    class(flavor_permutation_t), intent(in) :: perm
    type(flv_structure_t), intent(in) :: flv1, flv2
    logical, intent(in) :: with_tag
    type(flv_structure_t) :: flv_test
    flv_test = perm%apply (flv2, invert = .true.)
    valid = all (flv_test%flst == flv1%flst)
    if (with_tag) valid = valid .and. all (flv_test%tag == flv1%tag)
    call flv_test%final ()
  end function flavor_permutation_test
```

A singular region is a partition of phase space which is associated with an individual emitter and, if relevant, resonance. It is associated with an α_r- and resonance-index, with a real flavor structure and its underlying Born flavor structure. To compute the FKS weights, it is relevant to know all the other particle indices which can result in a divergenent phase space configuration, which are collected in the `ftuples`-array.

Some singular regions might behave physically identical. E.g. a real flavor structure associated with three-jet production is $[11, -11, 0, 2-2, 0]$. Here, there are two possible `ftuples` which contribute to the same $u \to ug$ splitting, namely $(3, 4)$ and $(4, 6)$. The resulting singular regions will be identical. To avoid this, one singular region is associated with the multiplicity factor `mult`. When computing the subtraction terms for each singular region, the result is then simply multiplied by this factor.
The `double_fsr`-flag indicates whether the singular region should also be supplied by a symmetry factor, explained below.

⟨*fks regions: public*⟩+≡

```
  public :: singular_region_t
```

⟨*fks regions: types*⟩+≡

```
  type :: singular_region_t
    integer :: alr
```

```
  integer :: i_res
  type(flv_structure_t) :: flst_real
  type(flv_structure_t) :: flst_uborn
  integer :: mult
  integer :: emitter
  integer :: nregions
  integer :: real_index
  type(ftuple_t), dimension(:), allocatable :: ftuples
  integer :: uborn_index
  logical :: double_fsr
  logical :: soft_divergence
  logical :: coll_divergence
  integer, dimension(:), allocatable :: i_reg_to_i_con
  logical :: pseudo_isr = .false.
contains
⟨fks regions: singular region: TBP⟩
end type singular_region_t
```

⟨*fks regions: singular region: TBP*⟩≡

```
procedure :: init => singular_region_init
```

⟨*fks regions: procedures*⟩+≡

```
subroutine singular_region_init (sregion, alr, mult, i_res, &
       flst_real, flst_uborn, flv_born, emitter, ftuples, equivalences)
  class(singular_region_t), intent(out) :: sregion
  integer, intent(in) :: alr, mult, i_res
  type(flv_structure_t), intent(in) :: flst_real
  type(flv_structure_t), intent(in) :: flst_uborn
  type(flv_structure_t), dimension(:), intent(in) :: flv_born
  integer, intent(in) :: emitter
  type(ftuple_t), intent(inout), dimension(:) :: ftuples
  logical, intent(inout), dimension(:,:) :: equivalences
  integer :: i, i1, i2
  integer :: n_regions, n_legs
  call debug_input_values ()
  sregion%alr = alr
  sregion%mult = mult
  sregion%i_res = i_res
  sregion%flst_real = flst_real
  sregion%flst_uborn = flst_uborn
  sregion%emitter = emitter
  n_regions = size (ftuples)
  sregion%nregions = size (ftuples)
  n_legs = size (flst_real%flst)
  allocate (sregion%ftuples (sregion%nregions))
  sregion%ftuples = ftuples
  do i = 1, size(flv_born)
     if (flv_born (i) .equiv. sregion%flst_uborn) then
        sregion%uborn_index = i
        exit
     end if
  end do
contains
  subroutine debug_input_values()
```

```
      integer :: i
      call msg_debug2 (D_SUBTRACTION, "singular_region_init")
      if (debug2_active (D_SUBTRACTION)) then
         print *, 'alr =    ', alr
         print *, 'mult =    ', mult
         print *, 'i_res =    ', i_res
         call flst_real%write ()
         call flst_uborn%write ()
         print *, 'emitter =    ', emitter
         call print_equivalence_matrix (ftuples, equivalences)
      end if
    end subroutine debug_input_values
  end subroutine singular_region_init
```

⟨*fks regions: singular region: TBP*⟩+≡

```
  procedure :: write => singular_region_write
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine singular_region_write (sregion, unit, maxnregions)
    class(singular_region_t), intent(in) :: sregion
    integer, intent(in), optional :: unit
    integer, intent(in), optional :: maxnregions
    character(len=7), parameter :: flst_format = "(I3,A1)"
    character(len=16), parameter :: ireg_format = "(A1,I1,A1,I1,A3)"
    character(len=22), parameter :: &
       ireg_format_resonant = "(A1,I1,A1,I1,A1,I1,A3)"
    character(len=7), parameter :: ireg_space_format = "(7X,A1)"
    integer :: nreal, nborn, i, u, mr, i1, i2
    integer :: nleft, nright, nreg, nreg_diff
    integer :: i_res
    character(len=3) :: closing_bracket
    u = given_output_unit (unit); if (u < 0) return
    mr = sregion%nregions; if (present (maxnregions))  mr = maxnregions
    nreal = size (sregion%flst_real%flst)
    nborn = size (sregion%flst_uborn%flst)
    call write_vline (u)
    write (u, '(A1)', advance = 'no') '['
    do i = 1, nreal - 1
       write (u, flst_format, advance = 'no') sregion%flst_real%flst(i), ','
    end do
    write (u, flst_format, advance = 'no') sregion%flst_real%flst(nreal), ']'
    call write_vline (u)
    write (u, '(I6)', advance = 'no') sregion%real_index
    call write_vline (u)
    write (u, '(I3)', advance = 'no') sregion%emitter
    call write_vline (u)
    write (u, '(I3)', advance = 'no') sregion%mult
    call write_vline (u)
    write (u, '(I4)', advance = 'no') sregion%nregions
    call write_vline (u)
    if (sregion%i_res > 0) then
       write (u, '(I3)', advance = 'no') sregion%i_res
       call write_vline (u)
    end if
```

```
    nreg = sregion%nregions
    if (nreg == mr) then
       nleft = 0
       nright = 0
    else
       nreg_diff = mr - nreg
       nleft = nreg_diff / 2
       if (mod(nreg_diff , 2) == 0) then
          nright = nleft
       else
          nright = nleft + 1
       end if
    end if
    if (nleft > 0) then
       do i = 1, nleft
          write(u, ireg_space_format, advance='no') ' '
       end do
    end if
    write (u, '(A)', advance = 'no') char (ftuple_string (sregion%ftuples, .false.))
    call write_vline (u)
    write (u,'(A1)',advance = 'no') '['
    do i = 1, nborn - 1
       write(u, flst_format, advance = 'no') sregion%flst_uborn%flst(i), ','
    end do
    write (u, flst_format, advance = 'no') sregion%flst_uborn%flst(nborn), ']'
    call write_vline (u)
    write (u, '(I7)', advance = 'no') sregion%uborn_index
    write (u, '(A)')
  end subroutine singular_region_write
```

⟨*fks regions: singular region: TBP*⟩+≡

```
  procedure :: write_latex => singular_region_write_latex
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine singular_region_write_latex (region, unit)
    class(singular_region_t), intent(in) :: region
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit); if (u < 0) return
    write (u, "(I2,A3,A,A3,I2,A3,I1,A3,I1,A3,A,A3,I2,A3,A,A3)") &
           region%alr, " & ", char (region%flst_real%to_string ()), &
           " & ", region%real_index, " & ", region%emitter, " & ", &
           region%mult, " & ", char (ftuple_string (region%ftuples, .true.)), &
           " & ", region%uborn_index, " & ",  char (region%flst_uborn%to_string ()), &
           " \\"
  end subroutine singular_region_write_latex
```

In case of a $g \to gg$ or $g \to qq$ splitting, the factor

$$\frac{2E_{\mathrm{em}}}{E_{\mathrm{em}} + E_{\mathrm{rad}}}$$

is multiplied to the real matrix element. This way, the symmetry of the splitting is used and only one singular region has to be taken into account. However, the

factor ensures that there is only a soft singularity if the radiated parton becomes soft.

⟨*fks regions: singular region: TBP*⟩+≡

```
procedure :: set_splitting_info => singular_region_set_splitting_info
```

⟨*fks regions: procedures*⟩+≡

```
subroutine singular_region_set_splitting_info (region)
  class(singular_region_t), intent(inout) :: region
  integer :: i1, i2
  integer :: reg
  region%double_fsr = .false.
  associate (ftuple => region%ftuples)
     do reg = 1, region%nregions
        call ftuple(reg)%get (i1, i2)
        if (i1 /= region%emitter) then
           cycle
        else
           region%soft_divergence = &
                ftuple(reg)%splitting_type /= G_TO_QQ

           if (i1 == 0) then
             region%coll_divergence = .true.
           else
             region%coll_divergence = .not. region%flst_real%massive(i1)
           end if

           if (ftuple(reg)%splitting_type == G_TO_GG) then
              region%double_fsr = .true.
              exit
           else if (ftuple(reg)%splitting_type == UNDEFINED_SPLITTING) then
              call msg_fatal ("All splittings should be defined!")
           end if
        end if
     end do
   end associate
end subroutine singular_region_set_splitting_info
```

⟨*fks regions: singular region: TBP*⟩+≡

```
procedure :: double_fsr_factor => singular_region_double_fsr_factor
```

⟨*fks regions: procedures*⟩+≡

```
function singular_region_double_fsr_factor (region, p) result (val)
  class(singular_region_t), intent(in) :: region
  type(vector4_t), intent(in), dimension(:) :: p
  real(default) :: val
  real(default) :: E_rad, E_em
  if (region%double_fsr) then
     E_em = energy (p(region%emitter))
     E_rad = energy (p(region%flst_real%nlegs))
     val = two * E_em / (E_em + E_rad)
  else
     val = one
  end if
end function singular_region_double_fsr_factor
```

⟨*fks regions: singular region: TBP*⟩+≡
```
procedure :: has_soft_divergence => singular_region_has_soft_divergence
```

⟨*fks regions: procedures*⟩+≡
```
function singular_region_has_soft_divergence (region) result (div)
   logical :: div
   class(singular_region_t), intent(in) :: region
   div = region%soft_divergence
end function singular_region_has_soft_divergence
```

⟨*fks regions: singular region: TBP*⟩+≡
```
procedure :: has_collinear_divergence => &
       singular_region_has_collinear_divergence
```

⟨*fks regions: procedures*⟩+≡
```
function singular_region_has_collinear_divergence (region) result (div)
  logical :: div
  class(singular_region_t), intent(in) :: region
  div = region%coll_divergence
end function singular_region_has_collinear_divergence
```

⟨*fks regions: singular region: TBP*⟩+≡
```
procedure :: has_identical_ftuples => singular_region_has_identical_ftuples
```

⟨*fks regions: procedures*⟩+≡
```
elemental function singular_region_has_identical_ftuples (sregion) result (value)
  logical :: value
  class(singular_region_t), intent(in) :: sregion
  integer :: alr
  value = .false.
  do alr = 1, sregion%nregions
     value = value .or. (count (sregion%ftuples(alr) == sregion%ftuples) > 1)
  end do
end function singular_region_has_identical_ftuples
```

⟨*fks regions: interfaces*⟩+≡
```
interface assignment(=)
   module procedure singular_region_assign
end interface
```

⟨*fks regions: procedures*⟩+≡
```
subroutine singular_region_assign (reg_out, reg_in)
  type(singular_region_t), intent(out) :: reg_out
  type(singular_region_t), intent(in) :: reg_in
  reg_out%alr = reg_in%alr
  reg_out%i_res = reg_in%i_res
  reg_out%flst_real = reg_in%flst_real
  reg_out%flst_uborn = reg_in%flst_uborn
  reg_out%mult = reg_in%mult
  reg_out%emitter = reg_in%emitter
  reg_out%nregions = reg_in%nregions
  reg_out%real_index = reg_in%real_index
  reg_out%uborn_index = reg_in%uborn_index
```

```
    reg_out%double_fsr = reg_in%double_fsr
    reg_out%soft_divergence = reg_in%soft_divergence
    reg_out%coll_divergence = reg_in%coll_divergence
    if (allocated (reg_in%ftuples)) then
       allocate (reg_out%ftuples (size (reg_in%ftuples)))
       reg_out%ftuples = reg_in%ftuples
    else
       call msg_bug ("singular_region_assign: Trying to copy a singular region without allocated f
    end if
  end subroutine singular_region_assign
```

⟨*fks regions: types*⟩+≡

```
  type :: resonance_mapping_t
    type(resonance_history_t), dimension(:), allocatable :: res_histories
    integer, dimension(:), allocatable :: alr_to_i_res
    integer, dimension(:,:), allocatable :: i_res_to_alr
    type(vector4_t), dimension(:), allocatable :: p_res
  contains
  ⟨fks regions: resonance mapping: TBP⟩
  end type resonance_mapping_t
```

Testing: Init resonance mapping for $\mu\mu bb$ final state.

⟨*fks regions: resonance mapping: TBP*⟩≡

```
  procedure :: init => resonance_mapping_init
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine resonance_mapping_init (res_map, res_hist)
    class(resonance_mapping_t), intent(inout) :: res_map
    type(resonance_history_t), intent(in), dimension(:) :: res_hist
    integer :: n_hist, i_hist1, i_hist2, n_contributors
    n_contributors = 0
    n_hist = size (res_hist)
    allocate (res_map%res_histories (n_hist))
    do i_hist1 = 1, n_hist
       if (i_hist1 + 1 <= n_hist) then
          do i_hist2 = i_hist1 + 1, n_hist
             if (.not. (res_hist(i_hist1) .contains. res_hist(i_hist2))) &
                n_contributors = n_contributors + res_hist(i_hist2)%n_resonances
          end do
       else
          n_contributors = n_contributors + res_hist(i_hist1)%n_resonances
       end if
    end do
    allocate (res_map%p_res (n_contributors))
    res_map%res_histories = res_hist
    res_map%p_res = vector4_null
  end subroutine resonance_mapping_init
```

⟨*fks regions: resonance mapping: TBP*⟩+≡

```
  procedure :: set_alr_to_i_res => resonance_mapping_set_alr_to_i_res
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine resonance_mapping_set_alr_to_i_res (res_map, regions, alr_new_to_old)
```

```
      class(resonance_mapping_t), intent(inout) :: res_map
      type(singular_region_t), intent(in), dimension(:) :: regions
      integer, intent(out), dimension(:), allocatable :: alr_new_to_old
      integer :: alr, i_res
      integer :: alr_new, n_alr_res
      integer :: k
      call msg_debug (D_SUBTRACTION, "resonance_mapping_set_alr_to_i_res")
      n_alr_res = 0
      do alr = 1, size (regions)
         do i_res = 1, size (res_map%res_histories)
            if (res_map%res_histories(i_res)%contains_leg (regions(alr)%emitter)) &
               n_alr_res = n_alr_res + 1
         end do
      end do

      allocate (res_map%alr_to_i_res (n_alr_res))
      allocate (res_map%i_res_to_alr (size (res_map%res_histories), 10))
      res_map%i_res_to_alr = 0
      allocate (alr_new_to_old (n_alr_res))
      alr_new = 1
      do alr = 1, size (regions)
         do i_res = 1, size (res_map%res_histories)
            if (res_map%res_histories(i_res)%contains_leg (regions(alr)%emitter)) then
               res_map%alr_to_i_res (alr_new) = i_res
               alr_new_to_old (alr_new) = alr
               alr_new = alr_new  + 1
            end if
         end do
      end do

      do i_res = 1, size (res_map%res_histories)
         k = 1
         do alr = 1, size (regions)
            if (res_map%res_histories(i_res)%contains_leg (regions(alr)%emitter)) then
               res_map%i_res_to_alr (i_res, k) = alr
               k = k + 1
            end if
         end do
      end do
      if (debug_active (D_SUBTRACTION)) then
         print *, 'i_res_to_alr:'
         do i_res = 1, size(res_map%i_res_to_alr, dim=1)
            print *, res_map%i_res_to_alr (i_res, :)
         end do
         print *, 'alr_new_to_old:', alr_new_to_old
      end if
   end subroutine resonance_mapping_set_alr_to_i_res
```

⟨*fks regions: resonance mapping: TBP*⟩+≡

```
   procedure :: get_resonance_history => resonance_mapping_get_resonance_history
```

⟨*fks regions: procedures*⟩+≡

```
   function resonance_mapping_get_resonance_history (res_map, alr) result (res_hist)
      type(resonance_history_t) :: res_hist
```

```
      class(resonance_mapping_t), intent(in) :: res_map
      integer, intent(in) :: alr
      res_hist = res_map%res_histories(res_map%alr_to_i_res (alr))
   end function resonance_mapping_get_resonance_history
```

⟨*fks regions: resonance mapping: TBP*⟩+≡

```
   procedure :: write => resonance_mapping_write
```

⟨*fks regions: procedures*⟩+≡

```
   subroutine resonance_mapping_write (res_map)
     class(resonance_mapping_t), intent(in) :: res_map
     integer :: i_res
     do i_res = 1, size (res_map%res_histories)
        call res_map%res_histories(i_res)%write ()
     end do
   end subroutine resonance_mapping_write
```

⟨*fks regions: resonance mapping: TBP*⟩+≡

```
   procedure :: get_resonance_value => resonance_mapping_get_resonance_value
```

⟨*fks regions: procedures*⟩+≡

```
   function resonance_mapping_get_resonance_value (res_map, i_res, p, i_gluon) result (p_map)
     real(default) :: p_map
     class(resonance_mapping_t), intent(in) :: res_map
     integer, intent(in) :: i_res
     type(vector4_t), intent(in), dimension(:) :: p
     integer, intent(in), optional :: i_gluon
     p_map = res_map%res_histories(i_res)%mapping (p, i_gluon)
   end function resonance_mapping_get_resonance_value
```

⟨*fks regions: resonance mapping: TBP*⟩+≡

```
   procedure :: get_resonance_all => resonance_mapping_get_resonance_all
```

⟨*fks regions: procedures*⟩+≡

```
   function resonance_mapping_get_resonance_all (res_map, alr, p, i_gluon) result (p_map)
     real(default) :: p_map
     class(resonance_mapping_t), intent(in) :: res_map
     integer, intent(in) :: alr
     type(vector4_t), intent(in), dimension(:) :: p
     integer, intent(in), optional :: i_gluon
     integer :: i_res
     p_map = zero
     do i_res = 1, size (res_map%res_histories)
        associate (res => res_map%res_histories(i_res))
           if (any (res_map%i_res_to_alr (i_res, :) == alr)) &
              p_map = p_map + res%mapping (p, i_gluon)
        end associate
     end do
   end function resonance_mapping_get_resonance_all
```

⟨*fks regions: resonance mapping: TBP*⟩+≡

```
   procedure :: get_weight => resonance_mapping_get_weight
```

⟨*fks regions: procedures*⟩+≡

```
function resonance_mapping_get_weight (res_map, alr, p) result (pfr)
  real(default) :: pfr
  class(resonance_mapping_t), intent(in) :: res_map
  integer, intent(in) :: alr
  type(vector4_t), intent(in), dimension(:) :: p
  real(default) :: sumpfr
  integer :: i_res
  sumpfr = zero
  do i_res = 1, size (res_map%res_histories)
     sumpfr = sumpfr + res_map%get_resonance_value (i_res, p)
  end do
  pfr = res_map%get_resonance_value (res_map%alr_to_i_res (alr), p) / sumpfr
end function resonance_mapping_get_weight
```

⟨*fks regions: resonance mapping: TBP*⟩+≡

```
procedure :: get_resonance_alr => resonance_mapping_get_resonance_alr
```

⟨*fks regions: procedures*⟩+≡

```
function resonance_mapping_get_resonance_alr (res_map, alr, p, i_gluon) result (p_map)
  real(default) :: p_map
  class(resonance_mapping_t), intent(in) :: res_map
  integer, intent(in) :: alr
  type(vector4_t), intent(in), dimension(:) :: p
  integer, intent(in), optional :: i_gluon
  integer :: i_res
  i_res = res_map%alr_to_i_res (alr)
  p_map = res_map%res_histories(i_res)%mapping (p, i_gluon)
end function resonance_mapping_get_resonance_alr
```

⟨*fks regions: interfaces*⟩+≡

```
interface assignment(=)
   module procedure resonance_mapping_assign
end interface
```

⟨*fks regions: procedures*⟩+≡

```
subroutine resonance_mapping_assign (res_map_out, res_map_in)
  type(resonance_mapping_t), intent(out) :: res_map_out
  type(resonance_mapping_t), intent(in) :: res_map_in
  if (allocated (res_map_in%res_histories)) then
     allocate (res_map_out%res_histories (size (res_map_in%res_histories)))
     res_map_out%res_histories = res_map_in%res_histories
  end if
  if (allocated (res_map_in%alr_to_i_res)) then
     allocate (res_map_out%alr_to_i_res (size (res_map_in%alr_to_i_res)))
     res_map_out%alr_to_i_res = res_map_in%alr_to_i_res
  end if
  if (allocated (res_map_in%i_res_to_alr)) then
     allocate (res_map_out%i_res_to_alr &
        (size (res_map_in%i_res_to_alr, 1), size (res_map_in%i_res_to_alr, 2)))
     res_map_out%i_res_to_alr = res_map_in%i_res_to_alr
  end if
  if (allocated (res_map_in%p_res)) then
```

```
      allocate (res_map_out%p_res (size (res_map_in%p_res)))
      res_map_out%p_res = res_map_in%p_res
    end if
  end subroutine resonance_mapping_assign
```

Every FKS mapping should store the $\sum_\alpha d_{ij}^{-1}$ and $\sum_\alpha d_{ij,\mathrm{soft}}^{-1}$. Also we keep the option open to use a normlization factor, which ensures $\sum_\alpha S_\alpha = 1$.

⟨*fks regions: types*⟩+≡

```
  type, abstract :: fks_mapping_t
     real(default) :: sumdij
     real(default) :: sumdij_soft
     logical :: pseudo_isr = .false.
     real(default) :: normalization_factor = one
  contains
  ⟨fks regions: fks mapping: TBP⟩
  end type fks_mapping_t
```

⟨*fks regions: public*⟩+≡

```
  public :: fks_mapping_default_t
```

⟨*fks regions: types*⟩+≡

```
  type, extends (fks_mapping_t) :: fks_mapping_default_t
    real(default) :: exp_1, exp_2
    integer :: n_in
  contains
  ⟨fks regions: fks mapping default: TBP⟩
  end type fks_mapping_default_t
```

⟨*fks regions: public*⟩+≡

```
  public :: fks_mapping_resonances_t
```

⟨*fks regions: types*⟩+≡

```
  type, extends (fks_mapping_t) :: fks_mapping_resonances_t
    real(default) :: exp_1, exp_2
    type(resonance_mapping_t) :: res_map
    integer :: i_con = 0
  contains
  ⟨fks regions: fks mapping resonances: TBP⟩
  end type fks_mapping_resonances_t
```

⟨*fks regions: public*⟩+≡

```
  public :: operator(.equiv.)
  public :: operator(.equivtag.)
```

⟨*fks regions: interfaces*⟩+≡

```
  interface operator(.equiv.)
    module procedure flv_structure_equivalent_no_tag
  end interface

  interface operator(.equivtag.)
    module procedure flv_structure_equivalent_with_tag
  end interface
```

```
interface assignment(=)
  module procedure flv_structure_assign_flv
  module procedure flv_structure_assign_integer
end interface
```

⟨*fks regions: public*⟩+≡

```
public :: region_data_t
```

⟨*fks regions: types*⟩+≡

```
type :: region_data_t
  type(singular_region_t), dimension(:), allocatable :: regions
  type(flv_structure_t), dimension(:), allocatable :: flv_born
  type(flv_structure_t), dimension(:), allocatable :: flv_real
  integer, dimension(:), allocatable :: emitters
  integer :: n_regions = 0
  integer :: n_emitters = 0
  integer :: n_flv_born = 0
  integer :: n_flv_real = 0
  integer :: n_in = 0
  integer :: n_legs_born = 0
  integer :: n_legs_real = 0
  integer :: n_phs = 0
  integer, dimension(:), allocatable :: underlying_borns
  class(fks_mapping_t), allocatable :: fks_mapping
  integer, dimension(:), allocatable :: resonances
  type(resonance_contributors_t), dimension(:), allocatable :: alr_contributors
  integer, dimension(:), allocatable :: alr_to_i_contributor
  integer, dimension(:), allocatable :: i_phs_to_i_con
contains
⟨fks regions: reg data: TBP⟩
end type region_data_t
```

⟨*fks regions: reg data: TBP*⟩≡

```
procedure :: allocate_fks_mappings => region_data_allocate_fks_mappings
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_allocate_fks_mappings (reg_data, mapping_type)
  class(region_data_t), intent(inout) :: reg_data
  integer, intent(in) :: mapping_type

  select case (mapping_type)
  case (FKS_DEFAULT)
     allocate (fks_mapping_default_t :: reg_data%fks_mapping)
  case (FKS_RESONANCES)
     allocate (fks_mapping_resonances_t :: reg_data%fks_mapping)
  case default
     call msg_fatal ("Init region_data: FKS mapping not implemented!")
  end select
end subroutine region_data_allocate_fks_mappings
```

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: init => region_data_init
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_init (reg_data, n_in, model, flavor_born, &
    flavor_real)
  class(region_data_t), intent(inout) :: reg_data
  integer, intent(in) :: n_in
  type(model_t), intent(in) :: model
  integer, intent(in), dimension(:,:) :: flavor_born, flavor_real
  type(ftuple_list_t), dimension(:), allocatable :: ftuples
  integer, dimension(:), allocatable :: emitter
  type(flv_structure_t), dimension(:), allocatable :: flst_alr
  integer :: i
  integer :: n_flv_real_before_check
  reg_data%n_in = n_in
  reg_data%n_flv_born = size (flavor_born, dim = 2)
  reg_data%n_legs_born = size (flavor_born, dim = 1)
  reg_data%n_legs_real = reg_data%n_legs_born + 1
  n_flv_real_before_check = size (flavor_real, dim = 2)
  allocate (reg_data%flv_born (reg_data%n_flv_born))
  allocate (reg_data%flv_real (n_flv_real_before_check))
  do i = 1, reg_data%n_flv_born
     call reg_data%flv_born(i)%init (flavor_born (:, i), n_in)
  end do
  do i = 1, n_flv_real_before_check
     call reg_data%flv_real(i)%init (flavor_real (:, i), n_in)
  end do

  call reg_data%find_regions (model, ftuples, emitter, flst_alr)
  call reg_data%init_singular_regions (ftuples, emitter, flst_alr)
  reg_data%n_flv_real = maxval (reg_data%regions%real_index)
  call reg_data%find_emitters ()
  call reg_data%set_underlying_borns ()
  call reg_data%set_mass_and_color (model)
  call reg_data%set_splitting_info ()
end subroutine region_data_init
```

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: init_resonance_information => region_data_init_resonance_information
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_init_resonance_information (reg_data)
  class(region_data_t), intent(inout) :: reg_data
  call reg_data%enlarge_singular_regions_with_resonances ()
  call reg_data%find_resonances ()
end subroutine region_data_init_resonance_information
```

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: set_resonance_mappings => region_data_set_resonance_mappings
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_set_resonance_mappings (reg_data, resonance_histories)
  class(region_data_t), intent(inout) :: reg_data
  type(resonance_history_t), intent(in), dimension(:) :: resonance_histories
  select type (map => reg_data%fks_mapping)
  type is (fks_mapping_resonances_t)
```

```
      call map%res_map%init (resonance_histories)
    end select
  end subroutine region_data_set_resonance_mappings
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: setup_fks_mappings => region_data_setup_fks_mappings
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_setup_fks_mappings (reg_data, template, n_in)
    class(region_data_t), intent(inout) :: reg_data
    type(fks_template_t), intent(in) :: template
    integer, intent(in) :: n_in
    call reg_data%allocate_fks_mappings (template%mapping_type)
    select type (map => reg_data%fks_mapping)
    type is (fks_mapping_default_t)
      call map%set_parameter (n_in, template%fks_dij_exp1, template%fks_dij_exp2)
    end select
  end subroutine region_data_setup_fks_mappings
```

So far, we have only created singular regions for a non-resonant case. When resonance mappings are required, we have more singular regions, since they must now be identified by their emitter-resonance pair index, where the emitter must be compatible with the resonance.

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: enlarge_singular_regions_with_resonances &
     => region_data_enlarge_singular_regions_with_resonances
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_enlarge_singular_regions_with_resonances (reg_data)
    class(region_data_t), intent(inout) :: reg_data
    integer :: alr
    integer, dimension(:), allocatable :: alr_new_to_old
    integer :: n_alr_new
    type(singular_region_t), dimension(:), allocatable :: save_regions
    call msg_debug (D_SUBTRACTION, "region_data_enlarge_singular_regions_with_resonances")
    call debug_input_values ()
    select type (fks_mapping => reg_data%fks_mapping)
    type is (fks_mapping_default_t)
      return
    type is (fks_mapping_resonances_t)
      allocate (save_regions (reg_data%n_regions))
      do alr = 1, reg_data%n_regions
        save_regions(alr) = reg_data%regions(alr)
      end do

      associate (res_map => fks_mapping%res_map)
        call res_map%set_alr_to_i_res (reg_data%regions, alr_new_to_old)
        deallocate (reg_data%regions)
        n_alr_new = size (alr_new_to_old)
        reg_data%n_regions = n_alr_new
        allocate (reg_data%regions (n_alr_new))
        do alr = 1, n_alr_new
          reg_data%regions(alr) = save_regions(alr_new_to_old (alr))
          reg_data%regions(alr)%i_res = res_map%alr_to_i_res (alr)
```

```
        end do
      end associate
    end select

  contains

    subroutine debug_input_values ()
      if (debug2_active (D_SUBTRACTION)) then
         call reg_data%write ()
      end if
    end subroutine debug_input_values

  end subroutine region_data_enlarge_singular_regions_with_resonances
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: set_isr_pseudo_regions => region_data_set_isr_pseudo_regions
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_set_isr_pseudo_regions (reg_data)
    class(region_data_t), intent(inout) :: reg_data
    integer :: alr
    integer :: n_alr_new
    !!! Subroutine called for threshold factorization ->
    !!! Size of singular regions at this point is fixed
    type(singular_region_t), dimension(2) :: save_regions
    integer, dimension(4) :: alr_new_to_old
    do alr = 1, reg_data%n_regions
       save_regions(alr) = reg_data%regions(alr)
    end do
    n_alr_new = reg_data%n_regions * 2
    alr_new_to_old = [1, 1, 2, 2]
    deallocate (reg_data%regions)
    allocate (reg_data%regions (n_alr_new))
    reg_data%n_regions = n_alr_new
    do alr = 1, n_alr_new
       reg_data%regions(alr) = save_regions(alr_new_to_old (alr))
       call add_pseudo_emitters (reg_data%regions(alr))
       if (mod (alr, 2) == 0) reg_data%regions(alr)%pseudo_isr = .true.
    end do
  contains
    subroutine add_pseudo_emitters (sregion)
      type(singular_region_t), intent(inout) :: sregion
      type(ftuple_t), dimension(2) :: ftuples_save
      integer :: alr
      do alr = 1, 2
         ftuples_save(alr) = sregion%ftuples(alr)
      end do
      deallocate (sregion%ftuples)
      sregion%nregions = sregion%nregions * 2
      allocate (sregion%ftuples (sregion%nregions))
      do alr = 1, sregion%nregions
         sregion%ftuples(alr) = ftuples_save (alr_new_to_old(alr))
         if (mod (alr, 2) == 0) sregion%ftuples(alr)%pseudo_isr = .true.
      end do
```

```
    end subroutine add_pseudo_emitters
  end subroutine region_data_set_isr_pseudo_regions
```

This subroutine splits up the ftuple-list of the singular regions into interference-free lists, i.e. lists which only contain the same emitter. This is relevant for factorized NLO calculations. In the current implementation, it is hand-tailored for the threshold computation, but should be generalized further in the future.

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: split_up_interference_regions_for_threshold => &
      region_data_split_up_interference_regions_for_threshold
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_split_up_interference_regions_for_threshold (reg_data)
    class(region_data_t), intent(inout) :: reg_data
    integer :: alr, i_ftuple
    integer :: current_emitter
    integer :: i1, i2
    integer :: n_new_reg
    type(ftuple_t), dimension(2) :: ftuples
    do alr = 1, reg_data%n_regions
       associate (region => reg_data%regions(alr))
          current_emitter = region%emitter
          n_new_reg = 0
          do i_ftuple = 1, region%nregions
             call region%ftuples(i_ftuple)%get (i1, i2)
             if (i1 == current_emitter) then
                n_new_reg = n_new_reg + 1
                ftuples(n_new_reg) = region%ftuples(i_ftuple)
             end if
          end do
          deallocate (region%ftuples)
          allocate (region%ftuples(n_new_reg))
          region%ftuples = ftuples (1 : n_new_reg)
          region%nregions = n_new_reg
       end associate
    end do
    reg_data%fks_mapping%normalization_factor = 0.5_default
  end subroutine region_data_split_up_interference_regions_for_threshold
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: set_mass_and_color => region_data_set_mass_and_color
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_set_mass_and_color (reg_data, model)
    class(region_data_t), intent(inout) :: reg_data
    type(model_t), intent(in) :: model
    integer :: i
    do i = 1, reg_data%n_regions
       associate (region => reg_data%regions(i))
          call region%flst_uborn%init_mass_and_color (model)
          call region%flst_real%init_mass_and_color (model)
       end associate
    end do
    do i = 1, reg_data%n_flv_born
```

```
      call reg_data%flv_born(i)%init_mass_and_color (model)
    end do
    do i = 1, size (reg_data%flv_real)
      call reg_data%flv_real(i)%init_mass_and_color (model)
    end do
  end subroutine region_data_set_mass_and_color
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: uses_resonances => region_data_uses_resonances
```

⟨*fks regions: procedures*⟩+≡

```
  function region_data_uses_resonances (reg_data) result (val)
    logical :: val
    class(region_data_t), intent(in) :: reg_data
    select type (fks_mapping => reg_data%fks_mapping)
    type is (fks_mapping_resonances_t)
      val = .true.
    class default
      val = .false.
    end select
  end function region_data_uses_resonances
```

Creates a list containing the emitter of each singular region.

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: get_emitter_list => region_data_get_emitter_list
```

⟨*fks regions: procedures*⟩+≡

```
  pure function region_data_get_emitter_list (reg_data) result(emitters)
    class(region_data_t), intent(in) :: reg_data
    integer, dimension(:), allocatable :: emitters
    integer :: i
    allocate (emitters (reg_data%n_regions))
    do i = 1, reg_data%n_regions
      emitters(i) = reg_data%regions(i)%emitter
    end do
  end function region_data_get_emitter_list
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: get_associated_resonances => region_data_get_associated_resonances
```

⟨*fks regions: procedures*⟩+≡

```
  function region_data_get_associated_resonances (reg_data, emitter) result (res)
    integer, dimension(:), allocatable :: res
    class(region_data_t), intent(in) :: reg_data
    integer, intent(in) :: emitter
    integer :: alr, i
    integer :: n_res
    select type (fks_mapping => reg_data%fks_mapping)
    type is (fks_mapping_resonances_t)
      n_res = 0

      do alr = 1, reg_data%n_regions
        if (reg_data%regions(alr)%emitter == emitter) &
          n_res = n_res + 1
```

```
      end do

      if (n_res > 0) then
         allocate (res (n_res))
      else
         return
      end if
      i = 1

      do alr = 1, reg_data%n_regions
         if (reg_data%regions(alr)%emitter == emitter) then
            res (i) = fks_mapping%res_map%alr_to_i_res (alr)
            i = i + 1
         end if
      end do
   end select
end function region_data_get_associated_resonances
```

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: emitter_is_compatible_with_resonance => &
   region_data_emitter_is_compatible_with_resonance
```

⟨*fks regions: procedures*⟩+≡

```
function region_data_emitter_is_compatible_with_resonance &
   (reg_data, i_res, emitter) result (compatible)
   logical :: compatible
   class(region_data_t), intent(in) :: reg_data
   integer, intent(in) :: i_res, emitter
   integer :: i_res_alr, alr
   compatible = .false.
   select type (fks_mapping => reg_data%fks_mapping)
   type is (fks_mapping_resonances_t)
      do alr = 1, reg_data%n_regions
         i_res_alr = fks_mapping%res_map%alr_to_i_res (alr)
         if (i_res_alr == i_res .and. reg_data%get_emitter(alr) == emitter) then
            compatible = .true.
            exit
         end if
      end do
   end select
end function region_data_emitter_is_compatible_with_resonance
```

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: emitter_is_in_resonance => region_data_emitter_is_in_resonance
```

⟨*fks regions: procedures*⟩+≡

```
function region_data_emitter_is_in_resonance (reg_data, i_res, emitter) result (exist)
  logical :: exist
  class(region_data_t), intent(in) :: reg_data
  integer, intent(in) :: i_res, emitter
  integer :: i
  exist = .false.
  select type (fks_mapping => reg_data%fks_mapping)
  type is (fks_mapping_resonances_t)
```

```
      associate (res_history => fks_mapping%res_map%res_histories(i_res))
         do i = 1, res_history%n_resonances
            exist = exist .or. any (res_history%resonances(i)%contributors%c == emitter)
         end do
      end associate
    end select
  end function region_data_emitter_is_in_resonance
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: get_contributors => region_data_get_contributors
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_get_contributors (reg_data, i_res, emitter, c, success)
    class(region_data_t), intent(in) :: reg_data
    integer, intent(in) :: i_res, emitter
    integer, intent(inout), dimension(:), allocatable :: c
    logical, intent(out) :: success
    integer :: i
    success = .false.
    select type (fks_mapping => reg_data%fks_mapping)
    type is (fks_mapping_resonances_t)
      associate (res_history => fks_mapping%res_map%res_histories (i_res))
         do i = 1, res_history%n_resonances
            if (any (res_history%resonances(i)%contributors%c == emitter)) then
               allocate (c (size (res_history%resonances(i)%contributors%c)))
               c = res_history%resonances(i)%contributors%c
               success = .true.
               exit
            end if
         end do
      end associate
    end select
  end subroutine region_data_get_contributors
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: get_emitter => region_data_get_emitter
```

⟨*fks regions: procedures*⟩+≡

```
  pure function region_data_get_emitter (reg_data, alr) result (emitter)
    class(region_data_t), intent(in) :: reg_data
    integer, intent(in) :: alr
    integer :: emitter
    emitter = reg_data%regions(alr)%emitter
  end function region_data_get_emitter
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: map_real_to_born_index => region_data_map_real_to_born_index
```

⟨*fks regions: procedures*⟩+≡

```
  function region_data_map_real_to_born_index (reg_data, real_index) result (uborn_index)
    integer :: uborn_index
    class(region_data_t), intent(in) :: reg_data
    integer, intent(in) :: real_index
    integer :: alr
```

```
    uborn_index = 0
    do alr = 1, size (reg_data%regions)
       if (reg_data%regions(alr)%real_index == real_index) then
          uborn_index = reg_data%regions(alr)%uborn_index
          exit
       end if
    end do
  end function region_data_map_real_to_born_index
```

Creates a list of α_r which share the same underlying Born index. We factor out the computation of the size of the underlying Born group so that we do not need to call the whole routine twice when we work around the ifort auto-allocation bug.

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: get_uborn_group_size => region_data_get_uborn_group_size
```

⟨*fks regions: procedures*⟩+≡

```
  function region_data_get_uborn_group_size (reg_data, uborn_index) result (n_uborn)
    integer :: n_uborn
    class(region_data_t), intent(in) :: reg_data
    integer, intent(in) :: uborn_index
    integer :: alr
    n_uborn = 0
    call msg_debug (D_SUBTRACTION, "inside region_data_get_uborn_group_size")
    call msg_debug (D_SUBTRACTION, "n_regions = ", reg_data%n_regions)
    do alr = 1, reg_data%n_regions
       if (reg_data%regions(alr)%i_res > 1) cycle
       if (reg_data%regions(alr)%uborn_index == uborn_index) n_uborn = n_uborn + 1
    end do
  end function region_data_get_uborn_group_size
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: get_uborn_group => region_data_get_uborn_group
```

⟨*fks regions: procedures*⟩+≡

```
  function region_data_get_uborn_group (reg_data, uborn_index) result (uborn_group)
    integer, dimension(:), allocatable :: uborn_group
    class(region_data_t), intent(in) :: reg_data
    integer, intent(in) :: uborn_index
    integer :: alr, n_uborn, i_born
    n_uborn = reg_data%get_uborn_group_size (uborn_index)
    if (n_uborn > 0) then
       allocate (uborn_group (n_uborn))
       i_born = 1
       do alr = 1, reg_data%n_regions
          if (reg_data%regions(alr)%i_res > 1) cycle
          if (reg_data%regions(alr)%uborn_index == uborn_index) then
             uborn_group (i_born) = alr
             i_born = i_born + 1
          end if
       end do
    end if
  end function region_data_get_uborn_group
```

Same as above, for emitters with the same underlying Born index.

⟨*fks regions: reg data: TBP*⟩+≡
```
procedure :: get_emitter_group_size => region_data_get_emitter_group_size
```

⟨*fks regions: procedures*⟩+≡
```
function region_data_get_emitter_group_size (reg_data, emitter) result (n_emitter)
  integer :: n_emitter
  class(region_data_t), intent(in) :: reg_data
  integer, intent(in) :: emitter
  integer :: alr
  n_emitter = 0
  do alr = 1, reg_data%n_regions
     if (reg_data%regions(alr)%i_res > 1) cycle
     if (reg_data%regions(alr)%emitter == emitter) n_emitter = n_emitter + 1
  end do
end function region_data_get_emitter_group_size
```

⟨*fks regions: reg data: TBP*⟩+≡
```
procedure :: get_emitter_group => region_data_get_emitter_group
```

⟨*fks regions: procedures*⟩+≡
```
function region_data_get_emitter_group (reg_data, emitter) result (emitter_group)
  integer, dimension(:), allocatable :: emitter_group
  class(region_data_t), intent(in) :: reg_data
  integer, intent(in) :: emitter
  integer :: alr, n_emitter, i_emitter
  n_emitter = reg_data%get_emitter_group_size (emitter)
  if (n_emitter > 0) then
     allocate (emitter_group (n_emitter))
     i_emitter = 1
     do alr = 1, reg_data%n_regions
        if (reg_data%regions(alr)%i_res > 1) cycle
        if (reg_data%regions(alr)%emitter == emitter) then
           emitter_group (i_emitter) = alr
           i_emitter = i_emitter + 1
        end if
     end do
  end if
end function region_data_get_emitter_group
```

⟨*fks regions: reg data: TBP*⟩+≡
```
generic :: get_flv_states_born => get_flv_states_born_single, get_flv_states_born_array
procedure :: get_flv_states_born_single => region_data_get_flv_states_born_single
procedure :: get_flv_states_born_array => region_data_get_flv_states_born_array
```

⟨*fks regions: procedures*⟩+≡
```
function region_data_get_flv_states_born_array (reg_data) result (flv_states)
  integer, dimension(:,:), allocatable :: flv_states
  class(region_data_t), intent(in) :: reg_data
  integer :: i_flv
  allocate (flv_states (reg_data%n_legs_born, reg_data%n_flv_born))
  do i_flv = 1, reg_data%n_flv_born
     flv_states (:, i_flv) = reg_data%flv_born(i_flv)%flst
  end do
```

```
end function region_data_get_flv_states_born_array

function region_data_get_flv_states_born_single (reg_data, i_flv) result (flv_states)
  integer, dimension(:), allocatable :: flv_states
  class(region_data_t), intent(in) :: reg_data
  integer, intent(in) :: i_flv
  allocate (flv_states (reg_data%n_legs_born))
  flv_states = reg_data%flv_born(i_flv)%flst
end function region_data_get_flv_states_born_single
```

⟨*fks regions: reg data: TBP*⟩+≡

```
generic :: get_flv_states_real => get_flv_states_real_single, get_flv_states_real_array
procedure :: get_flv_states_real_single => region_data_get_flv_states_real_single
procedure :: get_flv_states_real_array => region_data_get_flv_states_real_array
```

⟨*fks regions: procedures*⟩+≡

```
function region_data_get_flv_states_real_single (reg_data, i_flv) result (flv_states)
  integer, dimension(:), allocatable :: flv_states
  class(region_data_t), intent(in) :: reg_data
  integer, intent(in) :: i_flv
  integer :: i_reg
  allocate (flv_states (reg_data%n_legs_real))
  do i_reg = 1, reg_data%n_regions
     if (i_flv == reg_data%regions(i_reg)%real_index) then
        flv_states = reg_data%regions(i_reg)%flst_real%flst
        exit
     end if
  end do
end function region_data_get_flv_states_real_single

function region_data_get_flv_states_real_array (reg_data) result (flv_states)
  integer, dimension(:,:), allocatable :: flv_states
  class(region_data_t), intent(in) :: reg_data
  integer :: i_flv
  allocate (flv_states (reg_data%n_legs_real, reg_data%n_flv_real))
  do i_flv = 1, reg_data%n_flv_real
     flv_states (:, i_flv) = reg_data%get_flv_states_real (i_flv)
  end do
end function region_data_get_flv_states_real_array
```

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: get_all_flv_states => region_data_get_all_flv_states
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_get_all_flv_states (reg_data, flv_born, flv_real)
   class(region_data_t), intent(in) :: reg_data
   integer, dimension(:,:), allocatable, intent(out) :: flv_born, flv_real
   allocate (flv_born (reg_data%n_legs_born, reg_data%n_flv_born))
   flv_born = reg_data%get_flv_states_born ()
   allocate (flv_real (reg_data%n_legs_real, reg_data%n_flv_real))
   flv_real = reg_data%get_flv_states_real ()
end subroutine region_data_get_all_flv_states
```

⟨*fks regions: reg data: TBP*⟩+≡
```
procedure :: get_n_in => region_data_get_n_in
```
⟨*fks regions: procedures*⟩+≡
```
function region_data_get_n_in (reg_data) result (n_in)
  integer :: n_in
  class(region_data_t), intent(in) :: reg_data
  n_in = reg_data%n_in
end function region_data_get_n_in
```

⟨*fks regions: reg data: TBP*⟩+≡
```
procedure :: get_n_legs_real => region_data_get_n_legs_real
```
⟨*fks regions: procedures*⟩+≡
```
function region_data_get_n_legs_real (reg_data) result (n_legs)
  integer :: n_legs
  class(region_data_t), intent(in) :: reg_data
  n_legs = reg_data%n_legs_real
end function region_data_get_n_legs_real
```

⟨*fks regions: reg data: TBP*⟩+≡
```
procedure :: get_n_legs_born => region_data_get_n_legs_born
```
⟨*fks regions: procedures*⟩+≡
```
function region_data_get_n_legs_born (reg_data) result (n_legs)
  integer :: n_legs
  class(region_data_t), intent(in) :: reg_data
  n_legs = reg_data%n_legs_born
end function region_data_get_n_legs_born
```

⟨*fks regions: reg data: TBP*⟩+≡
```
procedure :: get_n_flv_real => region_data_get_n_flv_real
```
⟨*fks regions: procedures*⟩+≡
```
function region_data_get_n_flv_real (reg_data) result (n_flv)
  integer :: n_flv
  class(region_data_t), intent(in) :: reg_data
  n_flv = reg_data%n_flv_real
end function region_data_get_n_flv_real
```

⟨*fks regions: reg data: TBP*⟩+≡
```
procedure :: get_n_flv_born => region_data_get_n_flv_born
```
⟨*fks regions: procedures*⟩+≡
```
function region_data_get_n_flv_born (reg_data) result (n_flv)
  integer :: n_flv
  class(region_data_t), intent(in) :: reg_data
  n_flv = reg_data%n_flv_born
end function region_data_get_n_flv_born
```

Returns $S_i = \frac{1}{\mathcal{D}d_i}$ or $S_{ij} = \frac{1}{\mathcal{D}d_{ij}}$ for one particular singular region. At this point, the flavor array should be rearranged in such a way that the emitted particle is at the last position of the flavor structure list.

⟨*fks regions: reg data: TBP*⟩+≡

```
generic :: get_svalue => get_svalue_last_pos, get_svalue_ij
procedure :: get_svalue_last_pos => region_data_get_svalue_last_pos
procedure :: get_svalue_ij => region_data_get_svalue_ij
```

⟨*fks regions: procedures*⟩+≡

```
function region_data_get_svalue_ij (reg_data, p, alr, i, j, i_res) result (sval)
  class(region_data_t), intent(inout) :: reg_data
  type(vector4_t), intent(in), dimension(:) :: p
  integer, intent(in) :: alr, i, j
  integer, intent(in) :: i_res
  real(default) :: sval
  associate (map => reg_data%fks_mapping)
     call map%compute_sumdij (reg_data%regions(alr), p)
     select type (map)
     type is (fks_mapping_resonances_t)
        map%i_con = reg_data%alr_to_i_contributor (alr)
     end select
     map%pseudo_isr = reg_data%regions(alr)%pseudo_isr
     sval = map%svalue (p, i, j, i_res) * map%normalization_factor
  end associate
end function region_data_get_svalue_ij

function region_data_get_svalue_last_pos (reg_data, p, alr, emitter, i_res) result (sval)
  class(region_data_t), intent(inout) :: reg_data
  type(vector4_t), intent(in), dimension(:) :: p
  integer, intent(in) :: alr, emitter
  integer, intent(in) :: i_res
  real(default) :: sval
  sval = reg_data%get_svalue (p, alr, emitter, reg_data%n_legs_real, i_res)
end function region_data_get_svalue_last_pos
```

The same as above, but for the soft limit.

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: get_svalue_soft => region_data_get_svalue_soft
```

⟨*fks regions: procedures*⟩+≡

```
function region_data_get_svalue_soft &
     (reg_data, p, p_soft, alr, emitter, i_res) result (sval)
  class(region_data_t), intent(inout) :: reg_data
  type(vector4_t), intent(in), dimension(:) :: p
  type(vector4_t), intent(in) :: p_soft
  integer, intent(in) :: alr, emitter, i_res
  real(default) :: sval
  associate (map => reg_data%fks_mapping)
     call map%compute_sumdij_soft (reg_data%regions(alr), p, p_soft)
     select type (map)
     type is (fks_mapping_resonances_t)
        map%i_con = reg_data%alr_to_i_contributor (alr)
     end select
     map%pseudo_isr = reg_data%regions(alr)%pseudo_isr
     sval = map%svalue_soft (p, p_soft, emitter, i_res) * map%normalization_factor
  end associate
end function region_data_get_svalue_soft
```

This subroutine starts with a specification of N- and $N+1$-particle configurations, `flst_born` and `flst_real`, saved in `reg_data`. From these, it creates a list of fundamental tuples, a list of emitters and a list containing the $N+1$-particle configuration, rearranged in such a way that the emitter-radiation pair is last (`flst_alr`). For the $e^+ e^- \to u\,\bar{u}\,g$- example, the generated objects are shown in table 27.2. Note that at this point, `flst_alr` is arranged in such a way that the emitter can only be equal to $n_{legs} - 1$ for final-state radiation or 0, 1, or 2 for initial-state radiation. Further, it occurs that regions can be equivalent. For example in table 27.2 the regions correpsonding to `alr` = 1 and `alr` = 3 as well as `alr` = 2 and `alr` = 4 describe the same physics and are therefore equivalent.

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: find_regions => region_data_find_regions
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_find_regions &
     (reg_data, model, ftuples, emitters, flst_alr)
  class(region_data_t), intent(in) :: reg_data
  type(model_t), intent(in) :: model
  type(ftuple_list_t), intent(out), dimension(:), allocatable :: ftuples
  integer, intent(out), dimension(:), allocatable :: emitters
  type(flv_structure_t), intent(out), dimension(:), allocatable :: flst_alr
  type(ftuple_t) :: current_ftuple
  integer, dimension(:), allocatable :: emitter_tmp
  type(flv_structure_t), dimension(:), allocatable :: flst_alr_tmp
  type(ftuple_list_t), dimension(:,:), allocatable :: ftuples_tmp
  integer, dimension(:,:), allocatable :: ftuple_index
  integer :: n_born, n_real
  integer :: n_legreal
  integer, parameter :: n_regions_start = 20
  integer, parameter :: increment_list = 50
  integer :: i_born, i_real, i_reg, i_ftuple
  integer :: last_registered_i_born, last_registered_i_real

  n_born = size (reg_data%flv_born)
  n_real = size (reg_data%flv_real)
  n_legreal = size (reg_data%flv_real(1)%flst)
  allocate (ftuples_tmp (n_born,n_real))
  allocate (ftuple_index (n_born,n_real))
  allocate (emitter_tmp (n_regions_start))
  allocate (flst_alr_tmp (n_regions_start))
  i_reg = 0
  ftuple_index = 0
  i_ftuple = 0
  last_registered_i_born = 0; last_registered_i_real = 0

  do i_real = 1, n_real
     do i_born = 1, n_born
        call check_final_state_emissions (i_real, i_born, i_reg)
        call check_initial_state_emissions (i_real, i_born, i_reg)
     end do
  end do

  allocate (flst_alr (i_reg))
  flst_alr = flst_alr_tmp(1 : i_reg)
```

```
  allocate (emitters (i_reg))
  emitters = emitter_tmp(1 : i_reg)
  allocate (ftuples (count (ftuples_tmp%get_n_tuples () > 0)))
  do i_born = 1, n_born
     do i_real = 1, n_real
        if (ftuples_tmp(i_born,i_real)%get_n_tuples () > 0) &
             ftuples(ftuple_index(i_born,i_real)) = ftuples_tmp(i_born,i_real)
     end do
  end do
  deallocate (flst_alr_tmp)
  deallocate (emitter_tmp)
  deallocate (ftuples_tmp)
  deallocate (ftuple_index)

contains
  subroutine extend_flv_array (flv)
    type(flv_structure_t), intent(inout), dimension(:), allocatable :: flv
    type(flv_structure_t), dimension(:), allocatable :: flv_store
    integer :: n
    n = size (flv)
    allocate (flv_store (n))
    flv_store = flv
    deallocate (flv)
    allocate (flv (n + increment_list))
    flv(1:n) = flv_store
    deallocate (flv_store)
  end subroutine extend_flv_array

  function incr_i_ftuple_if_required (i_born, i_real, i_ftuple_in) result (i_ftuple)
    integer :: i_ftuple
    integer, intent(in) :: i_born, i_real, i_ftuple_in
    if (last_registered_i_born /= i_born .or. last_registered_i_real /= i_real) then
       last_registered_i_born = i_born
       last_registered_i_real = i_real
       i_ftuple = i_ftuple_in + 1
    else
       i_ftuple = i_ftuple_in
    end if
  end function incr_i_ftuple_if_required

  subroutine check_final_state_emissions (i_real, i_born, i_reg)
    integer, intent(in) :: i_real, i_born
    integer, intent(inout) :: i_reg
    integer :: leg1, leg2
    type(flv_structure_t) :: born_flavor
    logical :: valid1, valid2
    born_flavor = reg_data%flv_born(i_born)
    do leg1 = reg_data%n_in + 1, n_legreal
       do leg2 = leg1 + 1, n_legreal
          associate (flv_real => reg_data%flv_real(i_real))
             valid1 = flv_real%valid_pair(leg1, leg2, born_flavor, model)
             valid2 = flv_real%valid_pair(leg2, leg1, born_flavor, model)
             if (valid1 .or. valid2) then
                i_reg = i_reg + 1
```

```
            if (i_reg > size (flst_alr_tmp)) call extend_flv_array (flst_alr_tmp)
            if(valid1) then
               flst_alr_tmp(i_reg) = create_alr (flv_real, &
                    reg_data%n_in, leg1, leg2)
            else
               flst_alr_tmp(i_reg) = create_alr (flv_real, &
                    reg_data%n_in, leg2, leg1)
            end if
            call current_ftuple%set (leg1, leg2)
            call current_ftuple%determine_splitting_type_fsr &
                 (flv_real, leg1, leg2)
            i_ftuple = incr_i_ftuple_if_required (i_born, i_real, i_ftuple)
            call ftuples_tmp(i_born,i_real)%append (current_ftuple)
            ftuple_index(i_born,i_real) = i_ftuple
            if (i_reg > size (emitter_tmp)) &
                 call extend_integer_array (emitter_tmp, increment_list)
            emitter_tmp(i_reg) = n_legreal - 1
         end if
      end associate
   end do
  end do
end subroutine check_final_state_emissions

subroutine check_initial_state_emissions (i_real, i_born, i_reg)
  integer, intent(in) :: i_real, i_born
  integer, intent(inout) :: i_reg
  integer :: leg, emitter
  type(flv_structure_t) :: born_flavor
  logical :: valid1, valid2
  born_flavor = reg_data%flv_born (i_born)
  do leg = reg_data%n_in + 1, n_legreal
     associate (flv_real => reg_data%flv_real(i_real))
        valid1 = flv_real%valid_pair(1, leg, born_flavor, model)
        if (reg_data%n_in > 1) then
           valid2 = flv_real%valid_pair(2, leg, born_flavor, model)
        else
           valid2 = .false.
        end if
        if (valid1 .and. valid2) then
           emitter = 0
        else if (valid1 .and. .not. valid2) then
           emitter = 1
        else if (.not. valid1 .and. valid2) then
           emitter = 2
        else
           emitter = -1
        end if
        if (valid1 .or. valid2) then
           i_reg = i_reg + 1
           call current_ftuple%set(emitter, leg)
           call current_ftuple%determine_splitting_type_isr &
                (flv_real, emitter, leg)
           i_ftuple = incr_i_ftuple_if_required (i_born, i_real, i_ftuple)
           call ftuples_tmp(i_born,i_real)%append (current_ftuple)
```

```
                ftuple_index(i_born,i_real) = i_real
                if (i_reg > size (emitter_tmp)) &
                     call extend_integer_array (emitter_tmp, increment_list)
                emitter_tmp(i_reg) = emitter
                if (i_reg > size (flst_alr_tmp)) call extend_flv_array (flst_alr_tmp)
                flst_alr_tmp(i_reg) = &
                     create_alr (flv_real, reg_data%n_in, emitter, leg)
            end if
        end associate
      end do
    end subroutine check_initial_state_emissions

  end subroutine region_data_find_regions
```

Creates singular regions according to table 27.2. It scans all regions in table 27.2 and records the real flavor structures. If they are equivalent, the flavor structure is not recorded, but the multiplicity of the present one is increased.

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: init_singular_regions => region_data_init_singular_regions
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_init_singular_regions &
         (reg_data, ftuples, emitter, flv_alr)
    class(region_data_t), intent(inout) :: reg_data
    type(ftuple_list_t), intent(inout), dimension(:), allocatable :: ftuples
    integer :: n_independent_flv
    integer, intent(in), dimension(:) :: emitter
    type(flv_structure_t), intent(in), dimension(:) :: flv_alr
    type(flv_structure_t), dimension(:), allocatable :: flv_uborn, flv_alr_registered
    integer, dimension(:), allocatable :: mult
    integer, dimension(:), allocatable :: flst_emitter
    integer :: n_regions, maxregions
    integer, dimension(:), allocatable :: index
    integer :: i, i_flv, n_legs
    logical :: equiv, valid_fs_splitting
    integer :: i_first, i_reg, i_reg_prev
    integer, dimension(:), allocatable :: region_to_ftuple, alr_limits
    integer, dimension(:), allocatable :: equiv_index

    maxregions = size (emitter)
    n_legs = flv_alr(1)%nlegs

    allocate (flv_uborn (maxregions))
    allocate (flv_alr_registered (maxregions))
    allocate (mult (maxregions))
    mult = 0
    allocate (flst_emitter (maxregions))
    allocate (index (maxregions))
    allocate (region_to_ftuple (maxregions))
    allocate (equiv_index (maxregions))

    call setup_region_mappings (n_independent_flv, alr_limits, region_to_ftuple)
    i_first = 1
    i_reg = 1
```

```
SCAN_FLAVORS: do i_flv = 1, n_independent_flv
   SCAN_FTUPLES: do i = i_first, i_first + alr_limits (i_flv) - 1
      equiv = .false.
      if (i == i_first) then
         flv_alr_registered(i_reg) = flv_alr(i)
         mult(i_reg) = mult(i_reg) + 1
         flv_uborn(i_reg) = flv_alr(i)%create_uborn (emitter(i))
         flst_emitter(i_reg) = emitter(i)
         index (i_reg) = region_to_index(ftuples, i)
         equiv_index (i_reg) = region_to_ftuple(i)
         i_reg  = i_reg + 1
      else
         !!! Check for equivalent flavor structures
         do i_reg_prev = 1, i_reg - 1
            if (emitter(i) == emitter(i_reg_prev) .and. emitter(i) > reg_data%n_in) then
               valid_fs_splitting = check_fs_splitting (flv_alr(i)%get_last_two(n_legs), &
                      flv_alr_registered(i_reg_prev)%get_last_two(n_legs), &
                      flv_alr(i)%tag(n_legs - 1), flv_alr_registered(i_reg_prev)%tag(n_legs -
               if ((flv_alr(i) .equiv. flv_alr_registered(i_reg_prev)) &
                    .and. valid_fs_splitting) then
                  mult(i_reg_prev) = mult(i_reg_prev) + 1
                  equiv = .true.
                  call ftuples (region_to_index(ftuples, i))%set_equiv &
                       (equiv_index(i_reg_prev), region_to_ftuple(i))
                  exit
               end if
            else if (emitter(i) == emitter(i_reg_prev) .and. emitter(i) <= reg_data%n_in) then
               if (flv_alr(i) .equiv. flv_alr_registered(i_reg_prev)) then
                  mult(i_reg_prev) = mult(i_reg_prev) + 1
                  equiv = .true.
                  call ftuples (region_to_index(ftuples, i))%set_equiv &
                       (equiv_index(i_reg_prev), region_to_ftuple(i))
                  exit
               end if
            end if
         end do
         if (.not. equiv) then
            flv_alr_registered(i_reg) = flv_alr(i)
            mult(i_reg) = mult(i_reg) + 1
            flv_uborn(i_reg) = flv_alr(i)%create_uborn (emitter(i))
            flst_emitter(i_reg) = emitter(i)
            index (i_reg) = region_to_index (ftuples, i)
            equiv_index (i_reg) = region_to_ftuple(i)
            i_reg = i_reg + 1
         end if
      end if
   end do SCAN_FTUPLES
   i_first = i_first + alr_limits(i_flv)
end do SCAN_FLAVORS
n_regions = i_reg - 1

allocate (reg_data%regions (n_regions))
reg_data%n_regions = n_regions
call init_regions_with_permuted_flavors ()
```

```
  call assign_real_indices ()

  deallocate (flv_uborn)
  deallocate (flv_alr_registered)
  deallocate (mult)
  deallocate (flst_emitter)
  deallocate (index)
  deallocate (region_to_ftuple)
  deallocate (equiv_index)

contains

  subroutine setup_region_mappings (n_independent_flv, &
         alr_limits, region_to_ftuple)
     integer, intent(inout) :: n_independent_flv
     integer, intent(inout), dimension(:), allocatable :: alr_limits
     integer, intent(inout), dimension(:), allocatable :: region_to_ftuple
     integer :: i, j, i_flv
     n_independent_flv = 0
     do i = 1, size (ftuples)
        if (ftuples(i)%get_n_tuples() > 0) &
           n_independent_flv = n_independent_flv + 1
     end do
     allocate (alr_limits (n_independent_flv))

     j = 1
     do i = 1, size (ftuples)
        if (ftuples(i)%get_n_tuples() > 0) then
           alr_limits(j) = ftuples(i)%get_n_tuples ()
           j = j + 1
        end if
     end do

     if (.not. (sum (alr_limits) == maxregions)) &
          call msg_fatal ("Too many regions!")

     j = 1
     do i_flv = 1, n_independent_flv
        do i = 1, alr_limits(i_flv)
           region_to_ftuple(j) = i
           j = j + 1
        end do
     end do
  end subroutine setup_region_mappings

  subroutine check_permutation (perm, flv_perm, flv_orig, i_reg)
    type(flavor_permutation_t), intent(in) :: perm
    type(flv_structure_t), intent(in) :: flv_perm, flv_orig
    integer, intent(in) :: i_reg
    type(flv_structure_t) :: flv_test
    flv_test = perm%apply (flv_orig, invert = .true.)
    if (.not. all (flv_test%flst == flv_perm%flst)) then
       print *, 'Fail at: ', i_reg
       print *, 'Original flavor structure: ', flv_orig%flst
```

```
      call perm%write ()
      print *, 'Permuted flavor: ', flv_perm%flst
      print *, 'Should be: ', flv_test%flst
      call msg_fatal ("Permutation does not reproduce original flavor!")
   end if
end subroutine check_permutation

subroutine init_regions_with_permuted_flavors ()
   type(flavor_permutation_t) :: perm_list
   type(ftuple_t), dimension(:), allocatable :: ftuple_array
   logical, dimension(:,:), allocatable :: equivalences
   integer :: i, j
   do j = 1, n_regions
      do i = 1, reg_data%n_flv_born
         if (reg_data%flv_born (i) .equiv. flv_uborn (j)) then
            call perm_list%reset ()
            call perm_list%init (reg_data%flv_born(i), flv_uborn(j), &
                 reg_data%n_in, reg_data%n_legs_born, .true.)
            flv_uborn(j) = perm_list%apply (flv_uborn(j))
            flv_alr_registered(j) = perm_list%apply (flv_alr_registered(j))
            flst_emitter(j) = perm_list%apply (flst_emitter(j))
         end if
      end do
      call ftuples(index(j))%to_array (ftuple_array, equivalences, .true.)
      do i = 1, size (reg_data%flv_real)
         if (reg_data%flv_real(i) .equiv. flv_alr_registered(j)) then
            call perm_list%reset ()
            call perm_list%init (flv_alr_registered(j), reg_data%flv_real(i), &
                 reg_data%n_in, reg_data%n_legs_real, .false.)
            if (debug_active (D_SUBTRACTION)) call check_permutation &
                 (perm_list, reg_data%flv_real(i), flv_alr_registered(j), j)
            ftuple_array = perm_list%apply (ftuple_array)
         end if
      end do
      call reg_data%regions(j)%init (j, mult(j), 0, flv_alr_registered(j), &
           flv_uborn(j), reg_data%flv_born, flst_emitter(j), ftuple_array, &
           equivalences)
      if (allocated (ftuple_array)) deallocate (ftuple_array)
      if (allocated (equivalences)) deallocate (equivalences)
   end do
end subroutine init_regions_with_permuted_flavors

subroutine assign_real_indices ()
  type(flv_structure_t) :: current_flv_real
  integer :: i_real, current_uborn_index
  i_real = 1
  associate (regions => reg_data%regions)
     do i = 1, reg_data%n_regions
        if (i == 1) then
           regions(i)%real_index = 1
           current_flv_real = flv_alr_registered(1)
           current_uborn_index = regions(i)%uborn_index
        else if ((flv_alr_registered(i) .equiv. current_flv_real) &
                .and. regions(i)%uborn_index == current_uborn_index) then
```

```
            regions(i)%real_index = i_real
         else
            i_real = i_real + 1
            regions(i)%real_index = i_real
            current_flv_real = flv_alr_registered(i)
            current_uborn_index = regions(i)%uborn_index
         end if
      end do
   end associate
end subroutine assign_real_indices

subroutine write_perm_list (perm_list)
   integer, intent(in), dimension(:,:) :: perm_list
   integer :: i
   do i = 1, size (perm_list(:,1))
      write (*,'(I1,1x,I1,A)', advance = "no" ) perm_list(i,1), perm_list(i,2), '/'
   end do
   print *, ''
end subroutine write_perm_list

function check_fs_splitting (flv1, flv2, tag1, tag2) result (valid)
   logical :: valid
   integer, intent(in), dimension(2) :: flv1, flv2
   integer, intent(in) :: tag1, tag2
   if (flv1(1) + flv1(2) == 0) then
      valid = abs(flv1(1)) == abs(flv2(1)) .and. abs(flv1(2)) == abs(flv2(2))
   else
      valid = flv1(1) == flv2(1) .and. flv1(2) == flv2(2) .and. tag1 == tag2
   end if
end function check_fs_splitting
end subroutine region_data_init_singular_regions
```

Create an array containing all emitters and resonances of `region_data`.

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: find_emitters => region_data_find_emitters
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_find_emitters (reg_data)
   class(region_data_t), intent(inout) :: reg_data
   integer :: alr, j, n_em, em
   integer, dimension(N_MAX_ALR) :: em_count
   em_count = -1
   n_em = 0

   !!!Count the number of different emitters
   do alr = 1, reg_data%n_regions
      em = reg_data%regions(alr)%emitter
      if (.not. any (em_count == em)) then
         n_em = n_em + 1
         em_count(alr) = em
      end if
   end do

   if (n_em < 1) call msg_fatal ("region_data_find_emitters: No emitters found!")
```

```
    reg_data%n_emitters = n_em
    allocate (reg_data%emitters (reg_data%n_emitters))
    reg_data%emitters = -1

    j = 1
    do alr = 1, size (reg_data%regions)
       em = reg_data%regions(alr)%emitter
       if (.not. any (reg_data%emitters == em)) then
          reg_data%emitters(j) = em
          j = j + 1
       end if
    end do
  end subroutine region_data_find_emitters
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: find_resonances => region_data_find_resonances
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_find_resonances (reg_data)
    class(region_data_t), intent(inout) :: reg_data
    integer :: alr, j, k, n_res, n_contr
    integer :: res
    integer, dimension(10) :: res_count
    type(resonance_contributors_t), dimension(10) :: contributors_count
    type(resonance_contributors_t) :: contributors
    integer :: i_res, emitter
    logical :: share_emitter
    res_count = -1
    n_res = 0; n_contr = 0

    !!! Count the number of different resonances
    do alr = 1, reg_data%n_regions
       select type (fks_mapping => reg_data%fks_mapping)
       type is (fks_mapping_resonances_t)
          res = fks_mapping%res_map%alr_to_i_res (alr)
          if (.not. any (res_count == res)) then
             n_res = n_res + 1
             res_count(alr) = res
          end if
       end select
    end do

    if (n_res > 0) allocate (reg_data%resonances (n_res))

    j = 1
    select type (fks_mapping => reg_data%fks_mapping)
    type is (fks_mapping_resonances_t)
       do alr = 1, size (reg_data%regions)
          res = fks_mapping%res_map%alr_to_i_res (alr)
          if (.not. any (reg_data%resonances == res)) then
             reg_data%resonances(j) = res
             j = j + 1
          end if
       end do
```

```
      allocate (reg_data%alr_to_i_contributor (size (reg_data%regions)))
      do alr = 1, size (reg_data%regions)
         i_res = fks_mapping%res_map%alr_to_i_res (alr)
         emitter = reg_data%regions(alr)%emitter
         !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
         call reg_data%get_contributors (i_res, emitter, contributors%c, share_emitter)
         if (.not. share_emitter) cycle
         if (.not. any (contributors_count == contributors)) then
            n_contr = n_contr + 1
            contributors_count(alr) = contributors
         end if
         if (allocated (contributors%c)) deallocate (contributors%c)
      end do
      allocate (reg_data%alr_contributors (n_contr))
      j = 1
      do alr = 1, size (reg_data%regions)
         i_res = fks_mapping%res_map%alr_to_i_res (alr)
         emitter = reg_data%regions(alr)%emitter
         !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
         call reg_data%get_contributors (i_res, emitter, contributors%c, share_emitter)
         if (.not. share_emitter) cycle
         if (.not. any (reg_data%alr_contributors == contributors)) then
            reg_data%alr_contributors(j) = contributors
            reg_data%alr_to_i_contributor (alr) = j
            j = j + 1
         else
            do k = 1, size (reg_data%alr_contributors)
               if (reg_data%alr_contributors(k) == contributors) exit
            end do
            reg_data%alr_to_i_contributor (alr) = k
         end if
         if (allocated (contributors%c)) deallocate (contributors%c)
      end do
   end select
   call reg_data%extend_ftuples (n_res)
   call reg_data%set_contributors ()

end subroutine region_data_find_resonances
```

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: set_i_phs_to_i_con => region_data_set_i_phs_to_i_con
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_set_i_phs_to_i_con (reg_data)
   class(region_data_t), intent(inout) :: reg_data
   integer :: alr
   integer :: i_res, emitter, i_con, i_phs, i_em
   type(phs_identifier_t), dimension(:), allocatable :: phs_id_tmp
   logical :: share_emitter, phs_exist
   type(resonance_contributors_t) :: contributors
   allocate (phs_id_tmp (reg_data%n_phs))
   if (allocated (reg_data%resonances)) then
      allocate (reg_data%i_phs_to_i_con (reg_data%n_phs))
```

```
      do i_em = 1, size (reg_data%emitters)
         emitter = reg_data%emitters(i_em)
         do i_res = 1, size (reg_data%resonances)
            if (reg_data%emitter_is_compatible_with_resonance (i_res, emitter)) then
               alr = find_alr (emitter, i_res)
               if (alr == 0) call msg_fatal ("Could not find requested alpha region!")
               i_con = reg_data%alr_to_i_contributor (alr)
               call reg_data%get_contributors (i_res, emitter, contributors%c, share_emitter)
               if (.not. share_emitter) cycle
               call check_for_phs_identifier &
                  (phs_id_tmp, reg_data%n_in, emitter, contributors%c, phs_exist, i_phs)
               if (phs_id_tmp(i_phs)%emitter < 0) then
                  phs_id_tmp(i_phs)%emitter = emitter
                  allocate (phs_id_tmp(i_phs)%contributors (size (contributors%c)))
                  phs_id_tmp(i_phs)%contributors = contributors%c
               end if
               reg_data%i_phs_to_i_con (i_phs) = i_con
            end if
            if (allocated (contributors%c)) deallocate (contributors%c)
         end do
      end do
   end if
contains
   function find_alr (emitter, i_res) result (alr)
      integer :: alr
      integer, intent(in) :: emitter, i_res
      integer :: i
      do i = 1, reg_data%n_regions
         if (reg_data%regions(i)%emitter == emitter .and. &
             reg_data%regions(i)%i_res == i_res) then
            alr = i
            return
         end if
      end do
      alr = 0
   end function find_alr
end subroutine region_data_set_i_phs_to_i_con
```

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: set_alr_to_i_phs => region_data_set_alr_to_i_phs
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_set_alr_to_i_phs (reg_data, phs_identifiers, alr_to_i_phs)
   class(region_data_t), intent(inout) :: reg_data
   type(phs_identifier_t), intent(in), dimension(:) :: phs_identifiers
   integer, intent(out), dimension(:) :: alr_to_i_phs
   integer :: alr, i_phs
   integer :: emitter, i_res
   type(resonance_contributors_t) :: contributors
   logical :: share_emitter, phs_exist
   do alr = 1, reg_data%n_regions
      associate (region => reg_data%regions(alr))
         emitter = region%emitter
         i_res = region%i_res
```

```
         if (i_res /= 0) then
            !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
            call reg_data%get_contributors (i_res, emitter, &
               contributors%c, share_emitter)
            if (.not. share_emitter) cycle
         end if
         if (allocated (contributors%c)) then
            call check_for_phs_identifier (phs_identifiers, reg_data%n_in, &
               emitter, contributors%c, phs_exist = phs_exist, i_phs = i_phs)
         else
            call check_for_phs_identifier (phs_identifiers, reg_data%n_in, &
               emitter, phs_exist = phs_exist, i_phs = i_phs)
         end if
         if (.not. phs_exist) &
            call msg_fatal ("phs identifiers are not set up correctly!")
         alr_to_i_phs(alr) = i_phs
      end associate
      if (allocated (contributors%c)) deallocate (contributors%c)
   end do
end subroutine region_data_set_alr_to_i_phs
```

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: set_contributors => region_data_set_contributors
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_set_contributors (reg_data)
   class(region_data_t), intent(inout) :: reg_data
   integer :: alr, i_res, i_reg, i_con
   integer :: i1, i2, i_em
   integer, dimension(:), allocatable :: contributors
   logical :: share_emitter
   do alr = 1, size (reg_data%regions)
      associate (sregion => reg_data%regions(alr))
         allocate (sregion%i_reg_to_i_con (sregion%nregions))
         do i_reg = 1, sregion%nregions
            call sregion%ftuples(i_reg)%get (i1, i2)
            i_em = get_emitter_index (i1, i2, reg_data%n_legs_real)
            i_res = sregion%ftuples(i_reg)%i_res
            call reg_data%get_contributors (i_res, i_em, contributors, share_emitter)
            !!! Lookup contributor index
            do i_con = 1, size (reg_data%alr_contributors)
               if (all (reg_data%alr_contributors(i_con)%c == contributors)) then
                  sregion%i_reg_to_i_con (i_reg) = i_con
                  exit
               end if
            end do
            deallocate (contributors)
         end do
      end associate
   end do
contains
   function get_emitter_index (i1, i2, n) result (i_em)
      integer :: i_em
      integer, intent(in) :: i1, i2, n
```

```
      if (i1 == n) then
         i_em = i2
      else
         i_em = i1
      end if
   end function get_emitter_index
end subroutine region_data_set_contributors
```

This extension of the ftuples is still too naive as it assumes that the same resonances are possible for all ftuples

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: extend_ftuples => region_data_extend_ftuples
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_extend_ftuples (reg_data, n_res)
  class(region_data_t), intent(inout) :: reg_data
  integer, intent(in) :: n_res
  integer :: alr, n_reg_save
  integer :: i_reg, i_res, i_em, k
  type(ftuple_t), dimension(:), allocatable :: ftuple_save
  integer :: n_new
  do alr = 1, size (reg_data%regions)
     associate (sregion => reg_data%regions(alr))
        n_reg_save = sregion%nregions
        allocate (ftuple_save (n_reg_save))
        ftuple_save = sregion%ftuples
        n_new = count_n_new_ftuples (sregion, n_res)
        deallocate (sregion%ftuples)
        sregion%nregions = n_new
        allocate (sregion%ftuples (n_new))
        k = 1
        do i_res = 1, n_res
           do i_reg = 1, n_reg_save
              associate (ftuple_new => sregion%ftuples(k))
                 i_em = ftuple_save(i_reg)%ireg(1)
                 if (reg_data%emitter_is_in_resonance (i_res, i_em)) then
                    call ftuple_new%set (i_em, ftuple_save(i_reg)%ireg(2))
                    ftuple_new%i_res = i_res
                    ftuple_new%splitting_type = ftuple_save(i_reg)%splitting_type
                    k = k + 1
                 end if
              end associate
           end do
        end do
     end associate
     deallocate (ftuple_save)
  end do
contains
  function count_n_new_ftuples (sregion, n_res) result (n_new)
    integer :: n_new
    type(singular_region_t), intent(in) :: sregion
    integer, intent(in) :: n_res
    integer :: i_reg, i_res, i_em
    n_new = 0
```

```
      do i_reg = 1, sregion%nregions
         do i_res = 1, n_res
            i_em = sregion%ftuples(i_reg)%ireg(1)
            if (reg_data%emitter_is_in_resonance (i_res, i_em)) &
               n_new = n_new + 1
         end do
      end do
    end function count_n_new_ftuples
  end subroutine region_data_extend_ftuples
```

Create the map real flavor -¿ underlying Born

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: set_underlying_borns => region_data_set_underlying_borns
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_set_underlying_borns (reg_data)
    class(region_data_t), intent(inout) :: reg_data
    integer :: i, alr
    integer, dimension(:), allocatable :: flst_born
    allocate (reg_data%underlying_borns (reg_data%n_flv_born))
    do i = 1, reg_data%n_flv_born
       if (allocated (flst_born))  deallocate (flst_born)
       allocate (flst_born (size (reg_data%flv_born(i)%flst)))
       flst_born = reg_data%flv_born(i)%flst
       do alr = 1, reg_data%n_regions
          if (all (reg_data%regions(alr)%flst_uborn%flst == flst_born)) then
             reg_data%underlying_borns(i) = reg_data%regions(alr)%uborn_index
             exit
          end if
       end do
    end do
  end subroutine region_data_set_underlying_borns
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: get_flavor_indices => region_data_get_flavor_indices
```

⟨*fks regions: procedures*⟩+≡

```
  function region_data_get_flavor_indices (reg_data, born) result (i_flv)
    integer, dimension(:), allocatable :: i_flv
    class(region_data_t), intent(in) :: reg_data
    logical, intent(in) :: born
    allocate (i_flv (reg_data%n_regions))
    if (born) then
       i_flv = reg_data%regions%uborn_index
    else
       i_flv = reg_data%regions%real_index
    end if
  end function region_data_get_flavor_indices
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: get_matrix_element_index => region_data_get_matrix_element_index
```

⟨*fks regions: procedures*⟩+≡

```
function region_data_get_matrix_element_index (reg_data, i_reg) result (i_me)
  integer :: i_me
  class(region_data_t), intent(in) :: reg_data
  integer, intent(in) :: i_reg
  i_me = reg_data%regions(i_reg)%real_index
end function region_data_get_matrix_element_index
```

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: compute_number_of_phase_spaces &
   => region_data_compute_number_of_phase_spaces
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_compute_number_of_phase_spaces (reg_data)
  class(region_data_t), intent(inout) :: reg_data
  integer :: i_em, i_res, i_phs
  integer :: emitter
  type(resonance_contributors_t) :: contributors
  integer, parameter :: n_max_phs = 10
  type(phs_identifier_t), dimension(n_max_phs) :: phs_id_tmp
  logical :: share_emitter, phs_exist
  if (allocated (reg_data%resonances)) then
     reg_data%n_phs = 0
     do i_em = 1, size (reg_data%emitters)
        emitter = reg_data%emitters(i_em)
        do i_res = 1, size (reg_data%resonances)
           if (reg_data%emitter_is_compatible_with_resonance (i_res, emitter)) then
              !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
              call reg_data%get_contributors (i_res, emitter, contributors%c, share_emitter)
              if (.not. share_emitter) cycle
              call check_for_phs_identifier &
                 (phs_id_tmp, reg_data%n_in, emitter, contributors%c, phs_exist, i_phs)
              if (.not. phs_exist) then
                 reg_data%n_phs = reg_data%n_phs + 1
                 if (reg_data%n_phs > n_max_phs) call msg_fatal &
                    ("Buffer of phase space identifieres: Too much phase spaces!")
                 call phs_id_tmp(i_phs)%init (emitter, contributors%c)
              end if
           end if
           if (allocated (contributors%c)) deallocate (contributors%c)
        end do
     end do
  else
     reg_data%n_phs = size (remove_duplicates_from_list (reg_data%emitters))
  end if
end subroutine region_data_compute_number_of_phase_spaces
```

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: get_n_phs => region_data_get_n_phs
```

⟨*fks regions: procedures*⟩+≡

```
function region_data_get_n_phs (reg_data) result (n_phs)
  integer :: n_phs
  class(region_data_t), intent(in) :: reg_data
```

```
    n_phs = reg_data%n_phs
  end function region_data_get_n_phs
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: set_splitting_info => region_data_set_splitting_info
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_set_splitting_info (reg_data)
     class(region_data_t), intent(inout) :: reg_data
     integer :: alr
     do alr = 1, reg_data%n_regions
        call reg_data%regions(alr)%set_splitting_info ()
     end do
   end subroutine region_data_set_splitting_info
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: init_phs_identifiers => region_data_init_phs_identifiers
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_init_phs_identifiers (reg_data, phs_id)
    class(region_data_t), intent(in) :: reg_data
    type(phs_identifier_t), intent(out), dimension(:), allocatable :: phs_id
    integer :: i_em, i_res, i_phs
    integer :: emitter
    type(resonance_contributors_t) :: contributors
    logical :: share_emitter, phs_exist
    allocate (phs_id (reg_data%n_phs))
    do i_em = 1, size (reg_data%emitters)
       emitter = reg_data%emitters(i_em)
       if (allocated (reg_data%resonances)) then
          do i_res = 1, size (reg_data%resonances)
             !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
             call reg_data%get_contributors (i_res, emitter, contributors%c, share_emitter)
             if (.not. share_emitter) cycle
             call check_for_phs_identifier &
                (phs_id, reg_data%n_in, emitter, contributors%c, phs_exist, i_phs)
             if (.not. phs_exist) &
                call phs_id(i_phs)%init (emitter, contributors%c)
             if (allocated (contributors%c)) deallocate (contributors%c)
          end do
       else
          call check_for_phs_identifier (phs_id, reg_data%n_in, emitter, &
             phs_exist = phs_exist, i_phs = i_phs)
          if (.not. phs_exist) call phs_id(i_phs)%init (emitter)
       end if
    end do
  end subroutine region_data_init_phs_identifiers
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: get_all_ftuples => region_data_get_all_ftuples
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_get_all_ftuples (reg_data, ftuples)
    class(region_data_t), intent(in) :: reg_data
```

```
    type(ftuple_t), intent(inout), dimension(:), allocatable :: ftuples
    type(ftuple_t), dimension(:), allocatable :: ftuple_tmp
    integer :: i, j, alr
    !!! Can have at most n * (n-1) ftuples
    j = 0
    allocate (ftuple_tmp (reg_data%n_legs_real * (reg_data%n_legs_real - 1)))
    do i = 1, reg_data%n_regions
       associate (region => reg_data%regions(i))
          do alr = 1, region%nregions
             if (.not. any (region%ftuples(alr) == ftuple_tmp)) then
                j = j + 1
                ftuple_tmp(j) = region%ftuples(alr)
             end if
          end do
       end associate
    end do
    allocate (ftuples (j))
    ftuples = ftuple_tmp(1:j)
    deallocate (ftuple_tmp)
  end subroutine region_data_get_all_ftuples
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: write_to_file => region_data_write_to_file
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_write_to_file (reg_data, proc_id, latex, os_data)
     class(region_data_t), intent(inout) :: reg_data
     type(string_t), intent(in) :: proc_id
     logical, intent(in) :: latex
     type(os_data_t), intent(in) :: os_data
     type(string_t) :: filename
     integer :: u
     integer :: status

     if (latex) then
         filename = proc_id // "_fks_regions.tex"
     else
        filename = proc_id // "_fks_regions.out"
     end if
     u = free_unit ()
     open (u, file=char(filename), action = "write", status="replace")
     if (latex) then
        !call reg_data%write_latex (os_data, proc_id // "_fks_regions", stat_out = status)
        call reg_data%write_latex (u)
        close (u)
        call os_data_build_latex_file (os_data, proc_id // "_fks_regions", stat_out = status)
        if (status /= 0) &
             call msg_error (char ("Failed to compile " // filename))
     else
        call reg_data%write (u)
        close (u)
     end if
  end subroutine region_data_write_to_file
```

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: write_latex => region_data_write_latex
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_write_latex (reg_data, unit)
  class(region_data_t), intent(in) :: reg_data
  integer, intent(in), optional :: unit
  integer :: i, u
  u = given_output_unit (); if (present (unit)) u = unit
  write (u, "(A)") "\documentclass{article}"
  write (u, "(A)") "\begin{document}"
  write (u, "(A)") "%FKS region data, automatically created by WHIZARD"
  write (u, "(A)") "\begin{table}"
  write (u, "(A)") "\begin{center}"
  write (u, "(A)") "\begin{tabular} {|c|c|c|c|c|c|c|c|}"
  write (u, "(A)") "\hline"
  write (u, "(A)") "$\alpha_r$ & $f_r$ & $i_r$ & $\varepsilon$ & $\varsigma$ & $\mathcal{P}_{\rm
  write (u, "(A)") "\hline"
  do i = 1, reg_data%n_regions
     call reg_data%regions(i)%write_latex (u)
  end do
  write (u, "(A)") "\hline"
  write (u, "(A)") "\end{tabular}"
  write (u, "(A)") "\caption{List of singular regions}"
  write (u, "(A)") "\begin{description}"
  write (u, "(A)") "\item[$\alpha_r$] Index of the singular region"
  write (u, "(A)") "\item[$f_r$] Real flavor structure"
  write (u, "(A)") "\item[$i_r$] Index of the associated real flavor structure"
  write (u, "(A)") "\item[$\varepsilon$] Emitter"
  write (u, "(A)") "\item[$\varsigma$] Multiplicity" !!! The symbol used by 0908.4272 for multip
  write (u, "(A)") "\item[$\mathcal{P}_{\rm{FKS}}$] The set of singular FKS-pairs"
  write (u, "(A)") "\item[$i_b$] Underlying Born index"
  write (u, "(A)") "\item[$f_b$] Underlying Born flavor structure"
  write (u, "(A)") "\end{description}"
  write (u, "(A)") "\end{center}"
  write (u, "(A)") "\end{table}"
  write (u, "(A)") "\end{document}"
end subroutine region_data_write_latex
```

Creates a table with information about all singular regions and writes it to a file. Returns the index of the real flavor structure an ftuple belongs to.

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: write => region_data_write
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_write (reg_data, unit)
   class(region_data_t), intent(in) :: reg_data
   integer, intent(in), optional :: unit
   integer :: j
   integer :: maxnregions, i_reg_max
   type(string_t) :: flst_title, ftuple_title
   integer :: n_res, u
   u = given_output_unit (unit); if (u < 0) return
   maxnregions = 1; i_reg_max = 1
   do j = 1, reg_data%n_regions
```

```
      if (size (reg_data%regions(j)%ftuples) > maxnregions) then
         maxnregions = reg_data%regions(j)%nregions
         i_reg_max = j
      end if
   end do
   flst_title = '(A' // flst_title_format(reg_data%n_legs_real) // ')'
   ftuple_title = '(A' // ftuple_title_format() // ')'
   write (u,'(A,1X,I3)') 'Total number of regions: ', size(reg_data%regions)
   write (u, '(A3)', advance = 'no') 'alr'
   call write_vline (u)
   write (u, char (flst_title), advance = 'no') 'flst_real'
   call write_vline (u)
   write (u, '(A6)', advance = 'no') 'i_real'
   call write_vline (u)
   write (u, '(A3)', advance = 'no') 'em'
   call write_vline (u)
   write (u, '(A3)', advance = 'no') 'mult'
   call write_vline (u)
   write (u, '(A4)', advance = 'no') 'nreg'
   call write_vline (u)
   if (allocated (reg_data%fks_mapping)) then
      select type (fks_mapping => reg_data%fks_mapping)
      type is (fks_mapping_resonances_t)
         write (u, '(A3)', advance = 'no') 'res'
         call write_vline (u)
      end select
   end if
   write (u, char (ftuple_title), advance = 'no') 'ftuples'
   call write_vline (u)
   flst_title = '(A' // flst_title_format(reg_data%n_legs_born) // ')'
   write (u, char (flst_title), advance = 'no') 'flst_born'
   call write_vline (u)
   write (u, '(A7)') 'i_born'
   do j = 1, reg_data%n_regions
      write (u, '(I3)', advance = 'no') j
      call reg_data%regions(j)%write (u, maxnregions)
   end do
   call write_separator (u)
   if (allocated (reg_data%fks_mapping)) then
      select type (fks_mapping => reg_data%fks_mapping)
      type is (fks_mapping_resonances_t)
         write (u, '(A)')
         write (u, '(A)') "The FKS regions are combined with resonance information: "
         n_res = size (fks_mapping%res_map%res_histories)
         write (u, '(A,1X,I1)') "Number of QCD resonance histories: ", n_res
         do j = 1, n_res
            write (u, '(A,1X,I1)') "i_res = ", j
            call fks_mapping%res_map%res_histories(j)%write (u)
            call write_separator (u)
         end do
      end select
   end if

contains
```

```
    function flst_title_format (n) result (frmt)
      integer, intent(in) :: n
      type(string_t) :: frmt
      character(len=2) :: frmt_char
      write (frmt_char, '(I2)') 4 * n + 1
      frmt = var_str (frmt_char)
    end function flst_title_format

  function ftuple_title_format () result (frmt)
      type(string_t) :: frmt
      integer :: n_ftuple_char
      !!! An ftuple (x,x) consists of five characters. In the string, they
      !!! are separated by maxregions - 1 commas. In total these are
      !!! 5 * maxnregions + maxnregions - 1 = 6 * maxnregions - 1 characters.
      !!! The {} brackets at add two additional characters.
      n_ftuple_char = 6 * maxnregions + 1
      !!! If there are resonances, each ftuple with a resonance adds a ";x"
      !!! to the ftuple
      n_ftuple_char = n_ftuple_char + 2 * count (reg_data%regions(i_reg_max)%ftuples%i_res > 0)
      !!! Pseudo-ISR regions are denoted with a * at the end
      n_ftuple_char = n_ftuple_char + count (reg_data%regions(i_reg_max)%ftuples%pseudo_isr)
      frmt = str (n_ftuple_char)
  end function ftuple_title_format

end subroutine region_data_write
```

⟨*fks regions: procedures*⟩+≡

```
subroutine write_vline (u)
  integer, intent(in) :: u
  character(len=10), parameter :: sep_format = "(1X,A2,1X)"
  write (u, sep_format, advance = 'no') '||'
end subroutine write_vline
```

⟨*fks regions: public*⟩+≡

```
public :: assignment(=)
```

⟨*fks regions: interfaces*⟩+≡

```
interface assignment(=)
   module procedure region_data_assign
end interface
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_assign (reg_data_out, reg_data_in)
  type(region_data_t), intent(out) :: reg_data_out
  type(region_data_t), intent(in) :: reg_data_in
  integer :: i
  if (allocated (reg_data_in%regions)) then
     allocate (reg_data_out%regions (size (reg_data_in%regions)))
     do i = 1, size (reg_data_in%regions)
        reg_data_out%regions(i) = reg_data_in%regions(i)
     end do
  else
```

```
   call msg_warning ("Copying region data without allocated singular regions!")
end if
if (allocated (reg_data_in%flv_born)) then
   allocate (reg_data_out%flv_born (size (reg_data_in%flv_born)))
   do i = 1, size (reg_data_in%flv_born)
      reg_data_out%flv_born(i) = reg_data_in%flv_born(i)
   end do
else
   call msg_warning ("Copying region data without allocated born flavor structure!")
end if
if (allocated (reg_data_in%flv_real)) then
   allocate (reg_data_out%flv_real (size (reg_data_in%flv_real)))
   do i = 1, size (reg_data_in%flv_real)
      reg_data_out%flv_real(i) = reg_data_in%flv_real(i)
   end do
else
   call msg_warning ("Copying region data without allocated real flavor structure!")
end if
if (allocated (reg_data_in%emitters)) then
   allocate (reg_data_out%emitters (size (reg_data_in%emitters)))
   do i = 1, size (reg_data_in%emitters)
      reg_data_out%emitters(i) = reg_data_in%emitters(i)
   end do
else
   call msg_warning ("Copying region data without allocated emitters!")
end if
reg_data_out%n_regions = reg_data_in%n_regions
reg_data_out%n_emitters = reg_data_in%n_emitters
reg_data_out%n_flv_born = reg_data_in%n_flv_born
reg_data_out%n_flv_real = reg_data_in%n_flv_real
reg_data_out%n_in = reg_data_in%n_in
reg_data_out%n_legs_born = reg_data_in%n_legs_born
reg_data_out%n_legs_real = reg_data_in%n_legs_real
if (allocated (reg_data_in%underlying_borns)) then
   allocate (reg_data_out%underlying_borns (size (reg_data_in%underlying_borns)))
   reg_data_out%underlying_borns = reg_data_in%underlying_borns
else
   call msg_warning ("Copying region data without allocated underlying born flavor indices!")
end if
if (allocated (reg_data_in%fks_mapping)) then
   select type (fks_mapping_in => reg_data_in%fks_mapping)
   type is (fks_mapping_default_t)
      allocate (fks_mapping_default_t :: reg_data_out%fks_mapping)
      select type (fks_mapping_out => reg_data_out%fks_mapping)
      type is (fks_mapping_default_t)
         fks_mapping_out = fks_mapping_in
      end select
   type is (fks_mapping_resonances_t)
      allocate (fks_mapping_resonances_t :: reg_data_out%fks_mapping)
      select type (fks_mapping_out => reg_data_out%fks_mapping)
      type is (fks_mapping_resonances_t)
         fks_mapping_out = fks_mapping_in
      end select
   end select
```

```
    else
       call msg_warning ("Copying region data without allocated FKS regions!")
    end if
    if (allocated (reg_data_in%resonances)) then
       allocate (reg_data_out%resonances (size (reg_data_in%resonances)))
       reg_data_out%resonances = reg_data_in%resonances
    end if
    reg_data_out%n_phs = reg_data_in%n_phs
    if (allocated (reg_data_in%alr_contributors)) then
       allocate (reg_data_out%alr_contributors (size (reg_data_in%alr_contributors)))
       reg_data_out%alr_contributors = reg_data_in%alr_contributors
    end if
    if (allocated (reg_data_in%alr_to_i_contributor)) then
       allocate (reg_data_out%alr_to_i_contributor &
          (size (reg_data_in%alr_to_i_contributor)))
       reg_data_out%alr_to_i_contributor = reg_data_in%alr_to_i_contributor
    end if
  end subroutine region_data_assign
```

Returns the index of the real flavor structure an ftuple belogs to.

⟨*fks regions: procedures*⟩+≡

```
  function region_to_index (list, i) result(index)
    type(ftuple_list_t), intent(inout), dimension(:), allocatable :: list
    integer, intent(in) :: i
    integer :: index, nlist, j
    integer, dimension(:), allocatable :: nreg
    nlist = size(list)
    allocate (nreg (nlist))
    index = 0
    do j = 1, nlist
       if (j == 1) then
          nreg(j) = list(j)%get_n_tuples ()
       else
          nreg(j) = nreg(j - 1) + list(j)%get_n_tuples ()
       end if
    end do
    do j = 1, nlist
       if (j == 1) then
          if (i <= nreg(j)) then
             index = j
             exit
          end if
       else
          if (i > nreg(j - 1) .and. i <= nreg(j)) then
             index = j
             exit
          end if
       end if
    end do
  end function region_to_index
```

Final state emission: Rearrange the flavor array in such a way that the emitted particle is last and the emitter is second last. `i1` is the index of the emitter, `i2`

is the index of the emitted particle.

Initial state emission: Just put the emitted particle to the last position.

⟨*fks regions: procedures*⟩+≡

```
function create_alr (flv1, n_in, i_em, i_rad) result(flv2)
  type(flv_structure_t), intent(in) :: flv1
  integer, intent(in) :: n_in
  integer, intent(in) :: i_em, i_rad
  type(flv_structure_t) :: flv2
  integer :: n
  n = size (flv1%flst)
  allocate (flv2%flst (n), flv2%tag (n))
  flv2%nlegs = n
  flv2%n_in = n_in
  if (i_em > n_in) then
     flv2%flst(1 : n_in) = flv1%flst(1 : n_in)
     flv2%flst(n - 1) = flv1%flst(i_em)
     flv2%flst(n) = flv1%flst(i_rad)
     flv2%tag(1 : n_in) = flv1%tag(1 : n_in)
     flv2%tag(n - 1) = flv1%tag(i_em)
     flv2%tag(n) = flv1%tag(i_rad)
     call fill_remaining_flavors (n_in, .true.)
  else
     flv2%flst(1 : n_in) = flv1%flst(1 : n_in)
     flv2%flst(n) = flv1%flst(i_rad)
     flv2%tag(1 : n_in) = flv1%tag(1 : n_in)
     flv2%tag(n) = flv1%tag(i_rad)
     call fill_remaining_flavors (n_in, .false.)
  end if
contains
```

Order remaining particles according to their original position

⟨*fks regions: procedures*⟩+≡

```
  subroutine fill_remaining_flavors (n_in, final_final)
    integer, intent(in) :: n_in
    logical, intent(in) :: final_final
    integer :: i, j
    logical :: check
    j = n_in + 1
    do i = n_in + 1, n
       if (final_final) then
          check = (i /= i_em .and. i /= i_rad)
       else
          check = (i /= i_rad)
       end if
       if (check) then
          flv2%flst(j) = flv1%flst(i)
          flv2%tag(j) = flv1%tag(i)
          j = j + 1
       end if
    end do
  end subroutine fill_remaining_flavors
end function create_alr
```

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: has_pseudo_isr => region_data_has_pseudo_isr
```

⟨*fks regions: procedures*⟩+≡

```
function region_data_has_pseudo_isr (reg_data) result (val)
  logical :: val
  class(region_data_t), intent(in) :: reg_data
  val = any (reg_data%regions%pseudo_isr)
end function region_data_has_pseudo_isr
```

Performs consistency checks on `region_data`. Up to now only checks that now `futple` appears more than once.

⟨*fks regions: reg data: TBP*⟩+≡

```
procedure :: check_consistency => region_data_check_consistency
```

⟨*fks regions: procedures*⟩+≡

```
subroutine region_data_check_consistency (reg_data, fail_fatal, unit)
  class(region_data_t), intent(in) :: reg_data
  logical, intent(in) :: fail_fatal
  integer, intent(in), optional :: unit
  integer :: u
  integer :: i_reg, alr
  integer :: i1, f1, f2
  logical :: undefined_ftuples, same_ftuple_indices, valid_splitting
  logical, dimension(4) :: no_fail
  u = given_output_unit(unit); if (u < 0) return
  no_fail = .true.
  call msg_message ("Check that no negative ftuple indices occur", unit = u)
  do i_reg = 1, reg_data%n_regions
     if (any (reg_data%regions(i_reg)%ftuples%has_negative_elements ())) then
        !!! This error is so severe that we stop immediately
        call msg_fatal ("Negative ftuple indices!")
     end if
  end do
  call msg_message ("Success!", unit = u)
  call msg_message ("Check that there is no ftuple with identical elements", unit = u)
  do i_reg = 1, reg_data%n_regions
     if (any (reg_data%regions(i_reg)%ftuples%has_identical_elements ())) then
        !!! This error is so severe that we stop immediately
        call msg_fatal ("Identical ftuple indices!")
     end if
  end do
  call msg_message ("Success!", unit = u)
  call msg_message ("Check that there are no duplicate ftuples in a region", unit = u)
  do i_reg = 1, reg_data%n_regions
     if (reg_data%regions(i_reg)%has_identical_ftuples ()) then
        if (no_fail(1)) then
           call msg_error ("FAIL: ", unit = u)
           no_fail(1) = .false.
        end if
        write (u, '(A,1x,I3)') 'i_reg:', i_reg
     end if
  end do
  if (no_fail(1)) call msg_message ("Success!", unit = u)
  call msg_message ("Check that ftuples add up to a valid splitting", unit = u)
  do i_reg = 1, reg_data%n_regions
```

```
      do alr = 1, reg_data%regions(i_reg)%nregions
         associate (region => reg_data%regions(i_reg))
            i1 = region%ftuples(alr)%ireg(1)
            if (i1 == 0) i1 = 1 !!! Gluon emission from both initial-state quarks
            f1 = region%flst_real%flst(i1)
            f2 = region%flst_real%flst(region%ftuples(alr)%ireg(2))
            valid_splitting = f1 + f2 == 0 &
                 .or. (f1 == 21 .and. f2 == 21) &
                 .or. is_quark_gluon_splitting (f1, f2)
            if (.not. valid_splitting) then
               if (no_fail(2)) then
                  call msg_error ("FAIL: ", unit = u)
                  no_fail(2) = .false.
               end if
               write (u, '(A,1x,I3)') 'i_reg:', i_reg
               exit
            end if
         end associate
      end do
   end do
   if (no_fail(2)) call msg_message ("Success!", unit = u)
   call msg_message ("Check that at least one ftuple contains the emitter", unit = u)
   do i_reg = 1, reg_data%n_regions
      associate (region => reg_data%regions(i_reg))
         if (.not. any (region%emitter == region%ftuples%ireg(1))) then
            if (no_fail(3)) then
               call msg_error ("FAIL: ", unit = u)
               no_fail(3) = .false.
            end if
            write (u, '(A,1x,I3)') 'i_reg:', i_reg
         end if
      end associate
   end do
   if (no_fail(3)) call msg_message ("Success!", unit = u)
   call msg_message ("Check that each region has at least one ftuple &
        &with index n + 1", unit = u)
   do i_reg = 1, reg_data%n_regions
      if (.not. any (reg_data%regions(i_reg)%ftuples%ireg(2) == reg_data%n_legs_real)) then
         if (no_fail(4)) then
            call msg_error ("FAIL: ", unit = u)
            no_fail(4) = .false.
         end if
         write (u, '(A,1x,I3)') 'i_reg:', i_reg
      end if
   end do
   if (no_fail(4)) call msg_message ("Success!", unit = u)
   if (.not. all (no_fail)) &
        call abort_with_message ("Stop due to inconsistent region data!")

contains
   subroutine abort_with_message (msg)
     character(len=*), intent(in) :: msg
     if (fail_fatal) then
        call msg_fatal (msg)
```

```
      else
         call msg_error (msg, unit = u)
      end if
    end subroutine abort_with_message

    function is_quark_gluon_splitting (pdg_1, pdg_2) result (value)
      logical :: value
      integer, intent(in) :: pdg_1, pdg_2
      value = (is_quark(pdg_1) .and. is_gluon (pdg_2)) .or. &
           (is_quark(pdg_2) .and. is_gluon (pdg_1))
    end function
  end subroutine region_data_check_consistency
```

⟨*fks regions: reg data: TBP*⟩+≡

```
  procedure :: final => region_data_final
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine region_data_final (reg_data)
    class(region_data_t), intent(inout) :: reg_data
    if (allocated (reg_data%regions)) deallocate (reg_data%regions)
    if (allocated (reg_data%flv_born)) deallocate (reg_data%flv_born)
    if (allocated (reg_data%flv_real)) deallocate (reg_data%flv_real)
    if (allocated (reg_data%emitters)) deallocate (reg_data%emitters)
    if (allocated (reg_data%underlying_borns)) deallocate (reg_data%underlying_borns)
    if (allocated (reg_data%fks_mapping)) deallocate (reg_data%fks_mapping)
    if (allocated (reg_data%resonances)) deallocate (reg_data%resonances)
    if (allocated (reg_data%alr_contributors)) deallocate (reg_data%alr_contributors)
    if (allocated (reg_data%alr_to_i_contributor)) deallocate (reg_data%alr_to_i_contributor)
  end subroutine region_data_final
```

⟨*fks regions: fks mapping: TBP*⟩≡

```
  procedure (fks_mapping_dij), deferred :: dij
```

⟨*fks regions: interfaces*⟩+≡

```
  abstract interface
    function fks_mapping_dij (map, p, i, j, i_con) result (d)
      import
      real(default) :: d
      class(fks_mapping_t), intent(in) :: map
      type(vector4_t), intent(in), dimension(:) :: p
      integer, intent(in) :: i, j
      integer, intent(in), optional :: i_con
    end function fks_mapping_dij
  end interface
```

⟨*fks regions: fks mapping: TBP*⟩+≡

```
  procedure (fks_mapping_compute_sumdij), deferred :: compute_sumdij
```

⟨*fks regions: interfaces*⟩+≡

```
  abstract interface
    subroutine fks_mapping_compute_sumdij (map, sregion, p)
      import
      class(fks_mapping_t), intent(inout) :: map
      type(singular_region_t), intent(in) :: sregion
```

```
      type(vector4_t), intent(in), dimension(:) :: p
    end subroutine fks_mapping_compute_sumdij
  end interface
```

⟨*fks regions: fks mapping: TBP*⟩+≡

```
  procedure (fks_mapping_svalue), deferred :: svalue
```

⟨*fks regions: interfaces*⟩+≡

```
  abstract interface
    function fks_mapping_svalue (map, p, i, j, i_res) result (value)
      import
      real(default) :: value
      class(fks_mapping_t), intent(in) :: map
      type(vector4_t), intent(in), dimension(:) :: p
      integer, intent(in) :: i, j
      integer, intent(in), optional :: i_res
    end function fks_mapping_svalue
  end interface
```

⟨*fks regions: fks mapping: TBP*⟩+≡

```
  procedure (fks_mapping_dij_soft), deferred :: dij_soft
```

⟨*fks regions: interfaces*⟩+≡

```
  abstract interface
    function fks_mapping_dij_soft (map, p_born, p_soft, em, i_con) result (d)
      import
      real(default) :: d
      class(fks_mapping_t), intent(in) :: map
      type(vector4_t), intent(in), dimension(:) :: p_born
      type(vector4_t), intent(in) :: p_soft
      integer, intent(in) :: em
      integer, intent(in), optional :: i_con
    end function fks_mapping_dij_soft
  end interface
```

⟨*fks regions: fks mapping: TBP*⟩+≡

```
  procedure (fks_mapping_compute_sumdij_soft), deferred :: compute_sumdij_soft
```

⟨*fks regions: interfaces*⟩+≡

```
  abstract interface
    subroutine fks_mapping_compute_sumdij_soft (map, sregion, p_born, p_soft)
      import
      class(fks_mapping_t), intent(inout) :: map
      type(singular_region_t), intent(in) :: sregion
      type(vector4_t), intent(in), dimension(:) :: p_born
      type(vector4_t), intent(in) :: p_soft
    end subroutine fks_mapping_compute_sumdij_soft
  end interface
```

⟨*fks regions: fks mapping: TBP*⟩+≡

```
  procedure (fks_mapping_svalue_soft), deferred :: svalue_soft
```

⟨*fks regions: interfaces*⟩+≡

```
abstract interface
  function fks_mapping_svalue_soft (map, p_born, p_soft, em, i_res) result (value)
    import
    real(default) :: value
    class(fks_mapping_t), intent(in) :: map
    type(vector4_t), intent(in), dimension(:) :: p_born
    type(vector4_t), intent(in) :: p_soft
    integer, intent(in) :: em
    integer, intent(in), optional :: i_res
  end function fks_mapping_svalue_soft
end interface
```

⟨*fks regions: fks mapping default: TBP*⟩≡

```
procedure :: set_parameter => fks_mapping_default_set_parameter
```

⟨*fks regions: procedures*⟩+≡

```
subroutine fks_mapping_default_set_parameter (map, n_in, dij_exp1, dij_exp2)
  class(fks_mapping_default_t), intent(inout) :: map
  integer, intent(in) :: n_in
  real(default), intent(in) :: dij_exp1, dij_exp2
  map%n_in = n_in
  map%exp_1 = dij_exp1
  map%exp_2 = dij_exp2
end subroutine fks_mapping_default_set_parameter
```

Computes the d_{ij}-quantities defined als follows:

$$
\begin{aligned}
d_{0i} &= \left[E_i^2\,(1-y_i)\right]^{p_1} \\
,d_{1i} &= \left[2E_i^2\,(1-y_i)\right]^{p_1} \\
,d_{2i} &= \left[2E_i^2\,(1+y_i)\right]^{p_1} \\
&,
\end{aligned}
$$

for initial state regions and

$$
d_{ij} = \left[2(k_i \cdot k_j)\frac{E_i E_j}{(E_i+E_j)^2}\right]^{p_2}
$$

for final state regions. The exponents p_1 and p_2 can be used for tuning the efficiency of the mapping and are set to 1 per default.

⟨*fks regions: fks mapping default: TBP*⟩+≡

```
procedure :: dij => fks_mapping_default_dij
```

⟨*fks regions: procedures*⟩+≡

```
function fks_mapping_default_dij (map, p, i, j, i_con) result (d)
  real(default) :: d
  class(fks_mapping_default_t), intent(in) :: map
  type(vector4_t), intent(in), dimension(:) :: p
  integer, intent(in) :: i, j
  integer, intent(in), optional :: i_con
  d = zero
  if (map%pseudo_isr) then
```

```
      d = dij_threshold_gluon_from_top (i, j, p, map%exp_1)
   else if (i > map%n_in .and. j > map%n_in) then
      d = dij_fsr (p(i), p(j), map%exp_1)
   else
      d = dij_isr (map%n_in, i, j, p, map%exp_2)
   end if
contains

   function dij_fsr (p1, p2, expo) result (d_ij)
     real(default) :: d_ij
     type(vector4_t), intent(in) :: p1, p2
     real(default), intent(in) :: expo
     real(default) :: E1, E2
     E1 = p1%p(0); E2 = p2%p(0)
     d_ij = (two * p1 * p2 * E1 * E2 / (E1 + E2)**2)**expo
   end function dij_fsr

   function dij_threshold_gluon_from_top (i, j, p, expo) result (d_ij)
     real(default) :: d_ij
     integer, intent(in) :: i, j
     type(vector4_t), intent(in), dimension(:) :: p
     real(default), intent(in) :: expo
     type(vector4_t) :: p_top
     if (i == THR_POS_B) then
        p_top = p(THR_POS_WP) + p(THR_POS_B)
     else
        p_top = p(THR_POS_WM) + p(THR_POS_BBAR)
     end if
     d_ij = dij_fsr (p_top, p(j), expo)
   end function dij_threshold_gluon_from_top

   function dij_isr (n_in, i, j, p, expo) result (d_ij)
     real(default) :: d_ij
     integer, intent(in) :: n_in, i, j
     type(vector4_t), intent(in), dimension(:) :: p
     real(default), intent(in) :: expo
     real(default) :: E, y
     select case (n_in)
     case (1)
        call get_emitter_variables (1, i, j, p, E, y)
        d_ij = (E**2 * (one - y**2))**expo
     case (2)
        if ((i == 0 .and. j > 2) .or. (j == 0 .and. i > 2)) then
           call get_emitter_variables (0, i, j, p, E, y)
           d_ij = (E**2 * (one - y**2))**expo
        else if ((i == 1 .and. j > 2) .or. (j == 1 .and. i > 2)) then
           call get_emitter_variables (1, i, j, p, E, y)
           d_ij = (two * E**2 * (one - y))**expo
        else if ((i == 2 .and. j > 2) .or. (j == 2 .and. i > 2)) then
           call get_emitter_variables (2, i, j, p, E, y)
           d_ij = (two * E**2 * (one + y))**expo
        end if
     end select
   end function dij_isr
```

```
    subroutine get_emitter_variables (i_check, i, j, p, E, y)
       integer, intent(in) :: i_check, i, j
       type(vector4_t), intent(in), dimension(:) :: p
       real(default), intent(out) :: E, y
       if (j == i_check) then
           E = energy (p(i))
           y = polar_angle_ct (p(i))
       else
           E = energy (p(j))
           y = polar_angle_ct(p(j))
       end if
    end subroutine get_emitter_variables

  end function fks_mapping_default_dij
```

Computes the quantity

$$\mathcal{D} = \sum_k \frac{1}{d_{0k}} + \sum_{kl} \frac{1}{d_{kl}}.$$

⟨*fks regions: fks mapping default: TBP*⟩+≡

```
  procedure :: compute_sumdij => fks_mapping_default_compute_sumdij
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine fks_mapping_default_compute_sumdij (map, sregion, p)
    class(fks_mapping_default_t), intent(inout) :: map
    type(singular_region_t), intent(in) :: sregion
    type(vector4_t), intent(in), dimension(:) :: p
    real(default) :: d
    integer :: alr, i, j

    associate (ftuples => sregion%ftuples)
      d = zero
      do alr = 1, sregion%nregions
         call ftuples(alr)%get (i, j)
         map%pseudo_isr = ftuples(alr)%pseudo_isr
         d = d + one / map%dij (p, i, j)
      end do
    end associate
    map%sumdij = d
  end subroutine fks_mapping_default_compute_sumdij
```

Computes

$$S_i = \frac{1}{\mathcal{D} d_{0i}}$$

or

$$S_{ij} = \frac{1}{\mathcal{D} d_{ij}},$$

respectively.

⟨*fks regions: fks mapping default: TBP*⟩+≡

```
  procedure :: svalue => fks_mapping_default_svalue
```

⟨*fks regions: procedures*⟩+≡

```
function fks_mapping_default_svalue (map, p, i, j, i_res) result (value)
  real(default) :: value
  class(fks_mapping_default_t), intent(in) :: map
  type(vector4_t), intent(in), dimension(:) :: p
  integer, intent(in) :: i, j
  integer, intent(in), optional :: i_res
  value = one / (map%dij (p, i, j) * map%sumdij)
end function fks_mapping_default_svalue
```

In the soft limit, our treatment of the divergences requires a modification of the mapping functions. Recall that there, the ratios of the d-functions must approach either 1 or 0. This means

$$\frac{d_{lm}}{d_{0m}} = \frac{(2k_l \cdot k_m)\left[E_l E_m/(E_l+E_m)^2\right]}{E_m^2(1-y^2)} \overset{k_m = E_m \hat{k}}{=} \frac{E_l E_m^2}{(E_l+E_m)^2} \frac{2k_l \cdot \hat{k}}{E_m^2(1-y^2)} \overset{E_m \to 0}{=} \frac{2}{k_l \cdot \hat{k}}(1-y^2)E_l,$$

where we have written the gluon momentum in terms of the soft momentum $\hat{k}$. In the same limit

$$\frac{d_{lm}}{d_{nm}} = \frac{k_l \cdot \hat{k}}{k_n \cdot \hat{k}} \frac{E_n}{E_l}.$$

From these equations we can deduce the soft limit of d:

$$\begin{aligned} d_0^{\text{soft}} &= 1 - y^2, \\ d_1^{\text{soft}} &= 2(1-y), \\ d_2^{\text{soft}} &= 2(1+y), \\ d_i^{\text{soft}} &= \frac{2k_i \cdot \hat{k}}{E_i}. \end{aligned}$$

⟨*fks regions: fks mapping default: TBP*⟩+≡

```
procedure :: dij_soft => fks_mapping_default_dij_soft
```

⟨*fks regions: procedures*⟩+≡

```
function fks_mapping_default_dij_soft (map, p_born, p_soft, em, i_con) result (d)
  real(default) :: d
  class(fks_mapping_default_t), intent(in) :: map
  type(vector4_t), intent(in), dimension(:) :: p_born
  type(vector4_t), intent(in) :: p_soft
  integer, intent(in) :: em
  integer, intent(in), optional :: i_con
  if (map%pseudo_isr) then
     d = dij_soft_threshold_gluon_from_top (em, p_born, p_soft, map%exp_1)
  else if (em <= map%n_in) then
     d = dij_soft_isr (map%n_in, p_soft, map%exp_2)
  else
     d = dij_soft_fsr (p_born(em), p_soft, map%exp_1)
  end if
contains

  function dij_soft_threshold_gluon_from_top (em, p, p_soft, expo) result (dij_soft)
    real(default) :: dij_soft
    integer, intent(in) :: em
```

```
    type(vector4_t), intent(in), dimension(:) :: p
    type(vector4_t), intent(in) :: p_soft
    real(default), intent(in) :: expo
    type(vector4_t) :: p_top
    if (em == THR_POS_B) then
       p_top = p(THR_POS_WP) + p(THR_POS_B)
    else
       p_top = p(THR_POS_WM) + p(THR_POS_BBAR)
    end if
    dij_soft = dij_soft_fsr (p_top, p_soft, expo)
  end function dij_soft_threshold_gluon_from_top

  function dij_soft_fsr (p_em, p_soft, expo) result (dij_soft)
    real(default) :: dij_soft
    type(vector4_t), intent(in) :: p_em, p_soft
    real(default), intent(in) :: expo
    dij_soft = (two * p_em * p_soft / p_em%p(0))**expo
  end function dij_soft_fsr

  function dij_soft_isr (n_in, p_soft, expo) result (dij_soft)
     real(default) :: dij_soft
     integer, intent(in) :: n_in
     type(vector4_t), intent(in) :: p_soft
     real(default), intent(in) :: expo
     real(default) :: y
     y = polar_angle_ct (p_soft)
     select case (n_in)
     case (1)
        dij_soft = one - y**2
     case (2)
        select case (em)
        case (0)
           dij_soft = one - y**2
        case (1)
           dij_soft = two * (one - y)
        case (2)
           dij_soft = two * (one + y)
        case default
           dij_soft = zero
           call msg_fatal ("fks_mappings_default_dij_soft: n_in > 2")
        end select
     case default
        dij_soft = zero
        call msg_fatal ("fks_mappings_default_dij_soft: n_in > 2")
     end select
     dij_soft = dij_soft**expo
  end function dij_soft_isr
end function fks_mapping_default_dij_soft
```

⟨*fks regions: fks mapping default: TBP*⟩+≡

```
procedure :: compute_sumdij_soft => fks_mapping_default_compute_sumdij_soft
```

⟨*fks regions: procedures*⟩+≡

```
subroutine fks_mapping_default_compute_sumdij_soft (map, sregion, p_born, p_soft)
```

```
    class(fks_mapping_default_t), intent(inout) :: map
    type(singular_region_t), intent(in) :: sregion
    type(vector4_t), intent(in), dimension(:) :: p_born
    type(vector4_t), intent(in) :: p_soft
    real(default) :: d
    integer :: alr, i, j
    integer :: nlegs
    d = zero
    nlegs = size (sregion%flst_real%flst)
    associate (ftuples => sregion%ftuples)
      do alr = 1, sregion%nregions
         call ftuples(alr)%get (i ,j)
         if (j == nlegs) then
            map%pseudo_isr = ftuples(alr)%pseudo_isr
            d = d + one / map%dij_soft (p_born, p_soft, i)
         end if
      end do
    end associate
    map%sumdij_soft = d
  end subroutine fks_mapping_default_compute_sumdij_soft
```

⟨*fks regions: fks mapping default: TBP*⟩+≡

```
  procedure :: svalue_soft => fks_mapping_default_svalue_soft
```

⟨*fks regions: procedures*⟩+≡

```
  function fks_mapping_default_svalue_soft (map, p_born, p_soft, em, i_res) result (value)
    real(default) :: value
    class(fks_mapping_default_t), intent(in) :: map
    type(vector4_t), intent(in), dimension(:) :: p_born
    type(vector4_t), intent(in) :: p_soft
    integer, intent(in) :: em
    integer, intent(in), optional :: i_res
    value = one / (map%sumdij_soft * map%dij_soft (p_born, p_soft, em))
  end function fks_mapping_default_svalue_soft
```

⟨*fks regions: interfaces*⟩+≡

```
  interface assignment(=)
     module procedure fks_mapping_default_assign
  end interface
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine fks_mapping_default_assign (fks_map_out, fks_map_in)
    type(fks_mapping_default_t), intent(out) :: fks_map_out
    type(fks_mapping_default_t), intent(in) :: fks_map_in
    fks_map_out%exp_1 = fks_map_in%exp_1
    fks_map_out%exp_2 = fks_map_in%exp_2
    fks_map_out%n_in = fks_map_in%n_in
  end subroutine fks_mapping_default_assign
```

The $d_{ij,k}$-functions for the resonance mapping are basically the same as in the default case, but the kinematical values here must be evaluated in the resonance

frame of reference. The energy of parton i in a given resonance frame with momentum p_{res} is

$$E_i = \frac{p_i^0 \cdot p_{res}}{m_{res}}.$$

However, since the expressions only depend on ratios of four-momenta, we leave out the denominator because it will cancel out anyway.

⟨*fks regions: fks mapping resonances: TBP*⟩≡
```
procedure :: dij => fks_mapping_resonances_dij
```

⟨*fks regions: procedures*⟩+≡
```
function fks_mapping_resonances_dij (map, p, i, j, i_con) result (d)
  real(default) :: d
  class(fks_mapping_resonances_t), intent(in) :: map
  type(vector4_t), intent(in), dimension(:) :: p
  integer, intent(in) :: i, j
  integer, intent(in), optional :: i_con
  real(default) :: E1, E2
  integer :: ii_con
  if (present (i_con)) then
     ii_con = i_con
  else
     call msg_fatal ("Resonance mappings require resonance index as input!")
  end if
  d = 0
  if (i /= j) then
     if (i > 2 .and. j > 2) then
        associate (p_res => map%res_map%p_res (ii_con))
           E1 = p(i) * p_res
           E2 = p(j) * p_res
           d = two * p(i) * p(j) * E1 * E2 / (E1 + E2)**2
        end associate
     else
        call msg_fatal ("Resonance mappings are not implemented for ISR")
     end if
  end if
end function fks_mapping_resonances_dij
```

Computes

$$S_\alpha = \frac{P^{f_r(\alpha)} d^{-1}(\alpha)}{\sum_{f_r' \in T(F_r(\alpha))} P^{f_r'} \sum_{\alpha' \in Sr(f_r')} d^{-1}(\alpha)}.$$

⟨*fks regions: fks mapping resonances: TBP*⟩+≡
```
procedure :: compute_sumdij => fks_mapping_resonances_compute_sumdij
```

⟨*fks regions: procedures*⟩+≡
```
subroutine fks_mapping_resonances_compute_sumdij (map, sregion, p)
  class(fks_mapping_resonances_t), intent(inout) :: map
  type(singular_region_t), intent(in) :: sregion
  type(vector4_t), intent(in), dimension(:) :: p
  real(default) :: d, pfr
  integer :: i_res, i_reg, i, j, i_con
  integer :: nlegreal

  nlegreal = size (p)
```

```
    d = zero
    do i_reg = 1, sregion%nregions
       associate (ftuple => sregion%ftuples(i_reg))
          call ftuple%get (i, j)
          i_res = ftuple%i_res
       end associate
       pfr = map%res_map%get_resonance_value (i_res, p, nlegreal)
       i_con = sregion%i_reg_to_i_con (i_reg)
       d = d + pfr / map%dij (p, i, j, i_con)
    end do
    map%sumdij = d
  end subroutine fks_mapping_resonances_compute_sumdij
```

⟨*fks regions: fks mapping resonances: TBP*⟩+≡

```
  procedure :: svalue => fks_mapping_resonances_svalue
```

⟨*fks regions: procedures*⟩+≡

```
  function fks_mapping_resonances_svalue (map, p, i, j, i_res) result (value)
    real(default) :: value
    class(fks_mapping_resonances_t), intent(in) :: map
    type(vector4_t), intent(in), dimension(:) :: p
    integer, intent(in) :: i, j
    integer, intent(in), optional :: i_res
    real(default) :: pfr
    integer :: i_gluon
    i_gluon = size (p)
    pfr = map%res_map%get_resonance_value (i_res, p, i_gluon)
    value = pfr / (map%dij (p, i, j, map%i_con) * map%sumdij)
  end function fks_mapping_resonances_svalue
```

⟨*fks regions: fks mapping resonances: TBP*⟩+≡

```
  procedure :: get_resonance_weight => fks_mapping_resonances_get_resonance_weight
```

⟨*fks regions: procedures*⟩+≡

```
  function fks_mapping_resonances_get_resonance_weight (map, alr, p) result (pfr)
    real(default) :: pfr
    class(fks_mapping_resonances_t), intent(in) :: map
    integer, intent(in) :: alr
    type(vector4_t), intent(in), dimension(:) :: p
    pfr = map%res_map%get_weight (alr, p)
  end function fks_mapping_resonances_get_resonance_weight
```

As above, the soft limit of $d_{ij,k}$ must be computed in the resonance frame of reference.

⟨*fks regions: fks mapping resonances: TBP*⟩+≡

```
  procedure :: dij_soft => fks_mapping_resonances_dij_soft
```

⟨*fks regions: procedures*⟩+≡

```
  function fks_mapping_resonances_dij_soft (map, p_born, p_soft, em, i_con) result (d)
     real(default) :: d
     class(fks_mapping_resonances_t), intent(in) :: map
     type(vector4_t), intent(in), dimension(:) :: p_born
     type(vector4_t), intent(in) :: p_soft
     integer, intent(in) :: em
```

```
      integer, intent(in), optional :: i_con
      real(default) :: E1, E2
      integer :: ii_con
      type(vector4_t) :: pb
      if (present (i_con)) then
         ii_con = i_con
      else
         call msg_fatal ("fks_mapping_resonances requires resonance index")
      end if
      associate (p_res => map%res_map%p_res(ii_con))
         pb = p_born(em)
         E1 = pb * p_res
         E2 = p_soft * p_res
         d = two * pb * p_soft * E1 * E2 / E1**2
      end associate
   end function fks_mapping_resonances_dij_soft
```

⟨*fks regions: fks mapping resonances: TBP*⟩+≡

```
   procedure :: compute_sumdij_soft => fks_mapping_resonances_compute_sumdij_soft
```

⟨*fks regions: procedures*⟩+≡

```
   subroutine fks_mapping_resonances_compute_sumdij_soft (map, sregion, p_born, p_soft)
     class(fks_mapping_resonances_t), intent(inout) :: map
     type(singular_region_t), intent(in) :: sregion
     type(vector4_t), intent(in), dimension(:) :: p_born
     type(vector4_t), intent(in) :: p_soft
     real(default) :: d
     real(default) :: pfr
     integer :: i_res, i, j, i_reg, i_con
     integer :: nlegs

     d = zero
     nlegs = size (sregion%flst_real%flst)
     do i_reg = 1, sregion%nregions
        associate (ftuple => sregion%ftuples(i_reg))
           call ftuple%get(i, j)
           i_res = ftuple%i_res
        end associate
        pfr = map%res_map%get_resonance_value (i_res, p_born)
        i_con = sregion%i_reg_to_i_con (i_reg)
        if (j == nlegs) d = d + pfr / map%dij_soft (p_born, p_soft, i, i_con)
     end do
     map%sumdij_soft = d
   end subroutine fks_mapping_resonances_compute_sumdij_soft
```

⟨*fks regions: fks mapping resonances: TBP*⟩+≡

```
   procedure :: svalue_soft => fks_mapping_resonances_svalue_soft
```

⟨*fks regions: procedures*⟩+≡

```
   function fks_mapping_resonances_svalue_soft (map, p_born, p_soft, em, i_res) result (value)
     real(default) :: value
     class(fks_mapping_resonances_t), intent(in) :: map
     type(vector4_t), intent(in), dimension(:) :: p_born
     type(vector4_t), intent(in) :: p_soft
```

```
    integer, intent(in) :: em
    integer, intent(in), optional :: i_res
    real(default) :: pfr
    pfr = map%res_map%get_resonance_value (i_res, p_born)
    value = pfr / (map%sumdij_soft * map%dij_soft (p_born, p_soft, em, map%i_con))
  end function fks_mapping_resonances_svalue_soft
```

⟨*fks regions: fks mapping resonances: TBP*⟩+≡

```
  procedure :: set_resonance_momentum => fks_mapping_resonances_set_resonance_momentum
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine fks_mapping_resonances_set_resonance_momentum (map, p)
    class(fks_mapping_resonances_t), intent(inout) :: map
    type(vector4_t), intent(in) :: p
    map%res_map%p_res = p
  end subroutine fks_mapping_resonances_set_resonance_momentum
```

⟨*fks regions: fks mapping resonances: TBP*⟩+≡

```
  procedure :: set_resonance_momenta => fks_mapping_resonances_set_resonance_momenta
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine fks_mapping_resonances_set_resonance_momenta (map, p)
    class(fks_mapping_resonances_t), intent(inout) :: map
    type(vector4_t), intent(in), dimension(:) :: p
    map%res_map%p_res = p
  end subroutine fks_mapping_resonances_set_resonance_momenta
```

⟨*fks regions: interfaces*⟩+≡

```
  interface assignment(=)
     module procedure fks_mapping_resonances_assign
  end interface
```

⟨*fks regions: procedures*⟩+≡

```
  subroutine fks_mapping_resonances_assign (fks_map_out, fks_map_in)
    type(fks_mapping_resonances_t), intent(out) :: fks_map_out
    type(fks_mapping_resonances_t), intent(in) :: fks_map_in
    fks_map_out%exp_1 = fks_map_in%exp_1
    fks_map_out%exp_2 = fks_map_in%exp_2
    fks_map_out%res_map = fks_map_in%res_map
  end subroutine fks_mapping_resonances_assign
```

⟨*fks regions: public*⟩+≡

```
  public :: create_resonance_histories_for_threshold
```

⟨*fks regions: procedures*⟩+≡

```
  function create_resonance_histories_for_threshold () result (res_history)
    type(resonance_history_t) :: res_history
    res_history%n_resonances = 2
    allocate (res_history%resonances (2))
    allocate (res_history%resonances(1)%contributors%c(2))
    allocate (res_history%resonances(2)%contributors%c(2))
    res_history%resonances(1)%contributors%c = [THR_POS_WP, THR_POS_B]
    res_history%resonances(2)%contributors%c = [THR_POS_WM, THR_POS_BBAR]
```

```
end function create_resonance_histories_for_threshold
```

⟨*fks regions: public*⟩+≡

```
public :: setup_region_data_for_test
```

⟨*fks regions: procedures*⟩+≡

```
subroutine setup_region_data_for_test (n_in, flv_born, flv_real, reg_data)
  integer, intent(in) :: n_in
  integer, intent(in), dimension(:,:) :: flv_born, flv_real
  type(region_data_t), intent(out) :: reg_data
  type(model_t), pointer :: test_model => null ()
  call create_test_model (var_str ("SM"), test_model)
  call reg_data%init (n_in, test_model, flv_born, flv_real)
end subroutine setup_region_data_for_test
```

27.3.1 Unit tests

⟨fks_regions_ut.f90⟩≡
⟨*File header*⟩

```
module fks_regions_ut
  use unit_tests
  use fks_regions_uti
```

⟨*Standard module head*⟩

⟨*fks regions: public test*⟩

```
contains
```

⟨*fks regions: test driver*⟩

```
end module fks_regions_ut
```

⟨fks_regions_uti.f90⟩≡
⟨*File header*⟩

```
module fks_regions_uti
```

⟨*Use strings*⟩

```
  use format_utils, only: write_separator
  use os_interface
  use models

  use fks_regions
```

⟨*Standard module head*⟩

⟨*fks regions: test declarations*⟩

```
contains
```

⟨*fks regions: tests*⟩

```
end module fks_regions_uti
```

⟨*fks regions: public test*⟩≡

```
  public :: fks_regions_test
```

⟨*fks regions: test driver*⟩≡

```
  subroutine fks_regions_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
    call test(fks_regions_1, "fks_regions_1", &
         "Test flavor structure utilities", u, results)
    call test(fks_regions_2, "fks_regions_2", &
         "Test singular regions for final-state radiation for n = 2", &
         u, results)
    call test(fks_regions_3, "fks_regions_3", &
         "Test singular regions for final-state radiation for n = 3", &
         u, results)
    call test(fks_regions_4, "fks_regions_4", &
         "Test singular regions for final-state radiation for n = 4", &
```

```
        u, results)
    call test(fks_regions_5, "fks_regions_5", &
        "Test singular regions for final-state radiation for n = 5", &
        u, results)
    call test(fks_regions_6, "fks_regions_6", &
        "Test singular regions for initial-state radiation", &
        u, results)
    call test(fks_regions_7, "fks_regions_7", &
        "Check Latex output", u, results)
  end subroutine fks_regions_test
```

⟨*fks regions: test declarations*⟩≡

```
  public :: fks_regions_1
```

⟨*fks regions: tests*⟩≡

```
  subroutine fks_regions_1 (u)
    integer, intent(in) :: u
    type(flv_structure_t) :: flv_born, flv_real
    type(model_t), pointer :: test_model => null ()
    write (u, "(A)") "* Test output: fks_regions_1"
    write (u, "(A)") "* Purpose: Test utilities of flavor structure manipulation"
    write (u, "(A)")

    call create_test_model (var_str ("SM"), test_model)

    flv_born = [11, -11, 2, -2]
    flv_real = [11, -11, 2, -2, 21]
    flv_born%n_in = 2; flv_real%n_in = 2
    write (u, "(A)") "* Valid splittings of ee -> uu"
    write (u, "(A)") "Born Flavors: "
    call flv_born%write (u)
    write (u, "(A)") "Real Flavors: "
    call flv_real%write (u)
    write (u, "(A,L1)") "3, 4 (2, -2) : ", flv_real%valid_pair (3, 4, flv_born, test_model)
    write (u, "(A,L1)") "4, 3 (-2, 2) : ", flv_real%valid_pair (4, 3, flv_born, test_model)
    write (u, "(A,L1)") "3, 5 (2, 21) : ", flv_real%valid_pair (3, 5, flv_born, test_model)
    write (u, "(A,L1)") "5, 3 (21, 2) : ", flv_real%valid_pair (5, 3, flv_born, test_model)
    write (u, "(A,L1)") "4, 5 (-2, 21): ", flv_real%valid_pair (4, 5, flv_born, test_model)
    write (u, "(A,L1)") "5, 4 (21, -2): ", flv_real%valid_pair (5, 4, flv_born, test_model)
    call write_separator (u)

    call flv_born%final ()
    call flv_real%final ()

    flv_born = [2, -2, 11, -11]
    flv_real = [2, -2, 11, -11, 21]
    flv_born%n_in = 2; flv_real%n_in = 2
    write (u, "(A)") "* Valid splittings of uu -> ee"
    write (u, "(A)") "Born Flavors: "
    call flv_born%write (u)
    write (u, "(A)") "Real Flavors: "
    call flv_real%write (u)
    write (u, "(A,L1)") "1, 2 (2, -2) : " , flv_real%valid_pair (1, 2, flv_born, test_model)
    write (u, "(A,L1)") "2, 1 (-2, 2) : " , flv_real%valid_pair (2, 1, flv_born, test_model)
```

```
write (u, "(A,L1)") "5, 2 (21, -2): " , flv_real%valid_pair (5, 2, flv_born, test_model)
write (u, "(A,L1)") "2, 5 (-2, 21): " , flv_real%valid_pair (2, 5, flv_born, test_model)
write (u, "(A,L1)") "1, 5 (21, 2) : " , flv_real%valid_pair (5, 1, flv_born, test_model)
write (u, "(A,L1)") "5, 1 (2, 21) : " , flv_real%valid_pair (1, 5, flv_born, test_model)
call flv_real%final ()
flv_real = [21, -2, 11, -11, -2]
flv_real%n_in = 2
write (u, "(A)") "Real Flavors: "
call flv_real%write (u)
write (u, "(A,L1)") "1, 2 (21, -2): " , flv_real%valid_pair (1, 2, flv_born, test_model)
write (u, "(A,L1)") "2, 1 (-2, 21): " , flv_real%valid_pair (2, 1, flv_born, test_model)
write (u, "(A,L1)") "5, 2 (-2, -2): " , flv_real%valid_pair (5, 2, flv_born, test_model)
write (u, "(A,L1)") "2, 5 (-2, -2): " , flv_real%valid_pair (2, 5, flv_born, test_model)
write (u, "(A,L1)") "5, 1 (-2, 21): " , flv_real%valid_pair (5, 1, flv_born, test_model)
write (u, "(A,L1)") "1, 5 (21, -2): " , flv_real%valid_pair (1, 5, flv_born, test_model)
call flv_real%final ()
flv_real = [2, 21, 11, -11, 2]
flv_real%n_in = 2
write (u, "(A)") "Real Flavors: "
call flv_real%write (u)
write (u, "(A,L1)") "1, 2 (2, 21) : " , flv_real%valid_pair (1, 2, flv_born, test_model)
write (u, "(A,L1)") "2, 1 (21, 2) : " , flv_real%valid_pair (2, 1, flv_born, test_model)
write (u, "(A,L1)") "5, 2 (2, 21) : " , flv_real%valid_pair (5, 2, flv_born, test_model)
write (u, "(A,L1)") "2, 5 (21, 2) : " , flv_real%valid_pair (2, 5, flv_born, test_model)
write (u, "(A,L1)") "5, 1 (2, 2)  : " , flv_real%valid_pair (5, 1, flv_born, test_model)
write (u, "(A,L1)") "1, 5 (2, 2)  : " , flv_real%valid_pair (1, 5, flv_born, test_model)
call write_separator (u)

call flv_born%final ()
call flv_real%final ()

flv_born = [11, -11, 2, -2, 21]
flv_real = [11, -11, 2, -2, 21, 21]
flv_born%n_in = 2; flv_real%n_in = 2
write (u, "(A)") "* Valid splittings of ee -> uug"
write (u, "(A)") "Born Flavors: "
call flv_born%write (u)
write (u, "(A)") "Real Flavors: "
call flv_real%write (u)
write (u, "(A,L1)") "3, 4 (2, -2) : " , flv_real%valid_pair (3, 4, flv_born, test_model)
write (u, "(A,L1)") "4, 3 (-2, 2) : " , flv_real%valid_pair (4, 3, flv_born, test_model)
write (u, "(A,L1)") "3, 5 (2, 21) : " , flv_real%valid_pair (3, 5, flv_born, test_model)
write (u, "(A,L1)") "5, 3 (21, 2) : " , flv_real%valid_pair (5, 3, flv_born, test_model)
write (u, "(A,L1)") "4, 5 (-2, 21): " , flv_real%valid_pair (4, 5, flv_born, test_model)
write (u, "(A,L1)") "5, 4 (21, -2): " , flv_real%valid_pair (5, 4, flv_born, test_model)
write (u, "(A,L1)") "3, 6 (2, 21) : " , flv_real%valid_pair (3, 6, flv_born, test_model)
write (u, "(A,L1)") "6, 3 (21, 2) : " , flv_real%valid_pair (6, 3, flv_born, test_model)
write (u, "(A,L1)") "4, 6 (-2, 21): " , flv_real%valid_pair (4, 6, flv_born, test_model)
write (u, "(A,L1)") "6, 4 (21, -2): " , flv_real%valid_pair (6, 4, flv_born, test_model)
write (u, "(A,L1)") "5, 6 (21, 21): " , flv_real%valid_pair (5, 6, flv_born, test_model)
write (u, "(A,L1)") "6, 5 (21, 21): " , flv_real%valid_pair (6, 5, flv_born, test_model)
call flv_real%final ()
flv_real = [11, -11, 2, -2, 1, -1]
flv_real%n_in = 2
```

```
    write (u, "(A)") "Real Flavors (exemplary g -> dd splitting): "
    call flv_real%write (u)
    write (u, "(A,L1)") "3, 4 (2, -2) : " , flv_real%valid_pair (3, 4, flv_born, test_model)
    write (u, "(A,L1)") "4, 3 (-2, 2) : " , flv_real%valid_pair (4, 3, flv_born, test_model)
    write (u, "(A,L1)") "3, 5 (2, 1)  : " , flv_real%valid_pair (3, 5, flv_born, test_model)
    write (u, "(A,L1)") "5, 3 (1, 2)  : " , flv_real%valid_pair (5, 3, flv_born, test_model)
    write (u, "(A,L1)") "4, 5 (-2, 1) : " , flv_real%valid_pair (4, 5, flv_born, test_model)
    write (u, "(A,L1)") "5, 4 (1, -2) : " , flv_real%valid_pair (5, 4, flv_born, test_model)
    write (u, "(A,L1)") "3, 6 (2, -1) : " , flv_real%valid_pair (3, 6, flv_born, test_model)
    write (u, "(A,L1)") "6, 3 (-1, 2) : " , flv_real%valid_pair (6, 3, flv_born, test_model)
    write (u, "(A,L1)") "4, 6 (-2, -1): " , flv_real%valid_pair (4, 6, flv_born, test_model)
    write (u, "(A,L1)") "6, 4 (-1, -2): " , flv_real%valid_pair (6, 4, flv_born, test_model)
    write (u, "(A,L1)") "5, 6 (1, -1) : " , flv_real%valid_pair (5, 6, flv_born, test_model)
    write (u, "(A,L1)") "6, 5 (-1, 1) : " , flv_real%valid_pair (6, 5, flv_born, test_model)
    call write_separator (u)

    call flv_born%final ()
    call flv_real%final ()

    flv_born = [6, -5, 2, -1 ]
    flv_real = [6, -5, 2, -1, 21]
    flv_born%n_in = 1; flv_real%n_in = 1
    write (u, "(A)") "* Valid splittings of t -> b u d~"
    write (u, "(A)") "Born Flavors: "
    call flv_born%write (u)
    write (u, "(A)") "Real Flavors: "
    call flv_real%write (u)
    write (u, "(A,L1)") "1, 2 (6, -5) : " , flv_real%valid_pair (1, 2, flv_born, test_model)
    write (u, "(A,L1)") "1, 3 (6, 2)  : " , flv_real%valid_pair (1, 3, flv_born, test_model)
    write (u, "(A,L1)") "1, 4 (6, -1) : " , flv_real%valid_pair (1, 4, flv_born, test_model)
    write (u, "(A,L1)") "2, 1 (-5, 6) : " , flv_real%valid_pair (2, 1, flv_born, test_model)
    write (u, "(A,L1)") "3, 1 (2, 6)  : " , flv_real%valid_pair (3, 1, flv_born, test_model)
    write (u, "(A,L1)") "4, 1 (-1, 6) : " , flv_real%valid_pair (4, 1, flv_born, test_model)
    write (u, "(A,L1)") "2, 3 (-5, 2) : " , flv_real%valid_pair (2, 3, flv_born, test_model)
    write (u, "(A,L1)") "2, 4 (-5, -1): " , flv_real%valid_pair (2, 4, flv_born, test_model)
    write (u, "(A,L1)") "3, 2 (2, -5) : " , flv_real%valid_pair (3, 2, flv_born, test_model)
    write (u, "(A,L1)") "4, 2 (-1, -5): " , flv_real%valid_pair (4, 2, flv_born, test_model)
    write (u, "(A,L1)") "3, 4 (2, -1) : " , flv_real%valid_pair (3, 4, flv_born, test_model)
    write (u, "(A,L1)") "4, 3 (-1, 2) : " , flv_real%valid_pair (4, 3, flv_born, test_model)
    write (u, "(A,L1)") "1, 5 (6, 21) : " , flv_real%valid_pair (1, 5, flv_born, test_model)
    write (u, "(A,L1)") "5, 1 (21, 6) : " , flv_real%valid_pair (5, 1, flv_born, test_model)
    write (u, "(A,L1)") "2, 5 (-5, 21): " , flv_real%valid_pair (2, 5, flv_born, test_model)
    write (u, "(A,L1)") "5, 2 (21, 5) : " , flv_real%valid_pair (5, 2, flv_born, test_model)
    write (u, "(A,L1)") "3, 5 (2, 21) : " , flv_real%valid_pair (3, 5, flv_born, test_model)
    write (u, "(A,L1)") "5, 3 (21, 2) : " , flv_real%valid_pair (5, 3, flv_born, test_model)
    write (u, "(A,L1)") "4, 5 (-1, 21): " , flv_real%valid_pair (4, 5, flv_born, test_model)
    write (u, "(A,L1)") "5, 4 (21, -1): " , flv_real%valid_pair (5, 4, flv_born, test_model)

    call flv_born%final ()
    call flv_real%final ()

  end subroutine fks_regions_1
```

⟨*fks regions: test declarations*⟩+≡

```
    public :: fks_regions_2
```

⟨*fks regions: tests*⟩+≡

```
    subroutine fks_regions_2 (u)
      integer, intent(in) :: u
      integer :: n_flv_born, n_flv_real
      integer :: n_legs_born, n_legs_real
      integer :: n_in
      integer, dimension(:,:), allocatable :: flv_born, flv_real
      type(region_data_t) :: reg_data
      write (u, "(A)") "* Test output: fks_regions_2"
      write (u, "(A)") "* Create singular regions for processes with up to four singular regions"
      write (u, "(A)") "* ee -> qq"
      write (u, "(A)")

      n_flv_born = 1; n_flv_real = 1
      n_legs_born = 4; n_legs_real = 5
      n_in = 2

      allocate (flv_born (n_legs_born, n_flv_born))
      allocate (flv_real (n_legs_real, n_flv_real))
      flv_born (:, 1) = [11, -11, 2, -2]
      flv_real (:, 1) = [11, -11, 2, -2, 21]
      call setup_region_data_for_test (n_in, flv_born, flv_real, reg_data)
      call reg_data%check_consistency (.false., u)
      call reg_data%write (u)

      deallocate (flv_born, flv_real)
      call reg_data%final ()
      call write_separator (u)

      write (u, "(A)") "* ee -> tt"
      write (u, "(A)")
      write (u, "(A)") "* This process has four singular regions because they are not equivalent."

      n_flv_born = 1; n_flv_real = 1
      n_legs_born = 6; n_legs_real = 7
      n_in = 2

      allocate (flv_born (n_legs_born, n_flv_born))
      allocate (flv_real (n_legs_real, n_flv_real))
      flv_born (:, 1) = [11, -11, 6, -6, 6, -6]
      flv_real (:, 1) = [11, -11, 6, -6, 6, -6, 21]
      call setup_region_data_for_test (n_in, flv_born, flv_real, reg_data)
      call reg_data%check_consistency (.false., u)
      call reg_data%write (u)

      deallocate (flv_born, flv_real)
      call reg_data%final ()
    end subroutine fks_regions_2
```

⟨*fks regions: test declarations*⟩+≡

```
    public :: fks_regions_3
```

⟨*fks regions: tests*⟩+≡

```
subroutine fks_regions_3 (u)
  integer, intent(in) :: u
  integer :: n_flv_born, n_flv_real
  integer :: n_legs_born, n_legs_real
  integer :: n_in
  integer, dimension(:,:), allocatable :: flv_born, flv_real
  type(region_data_t) :: reg_data
  write (u, "(A)") "* Test output: fks_regions_3"
  write (u, "(A)") "* Create singular regions for processes with three final-state particles"

  write (u, "(A)") "* ee -> qqg"
  write (u, "(A)")
  n_flv_born = 1; n_flv_real = 2
  n_legs_born = 5; n_legs_real = 6
  n_in = 2
  allocate (flv_born (n_legs_born, n_flv_born))
  allocate (flv_real (n_legs_real, n_flv_real))

  flv_born (:, 1) = [11, -11, 2, -2, 21]
  flv_real (:, 1) = [11, -11, 2, -2, 21, 21]
  flv_real (:, 2) = [11, -11, 2, -2, 1, -1]

  call setup_region_data_for_test (n_in, flv_born, flv_real, reg_data)
  call reg_data%check_consistency (.false., u)
  call reg_data%write (u)

  deallocate (flv_born, flv_real)
  call reg_data%final ()

  write (u, "(A)") "* ee -> jet jet jet"
  write (u, "(A)") "* with jet = u:U:d:D:s:S:c:C:b:B:gl"
  write (u, "(A)")
  n_flv_born = 5; n_flv_real = 22
  n_legs_born = 5; n_legs_real = 6
  n_in = 2
  allocate (flv_born (n_legs_born, n_flv_born))
  allocate (flv_real (n_legs_real, n_flv_real))

  flv_born (:, 1)  = [11, -11, -4, 4, 21]
  flv_born (:, 2)  = [11, -11, -2, 2, 21]
  flv_born (:, 3)  = [11, -11, -5, 5, 21]
  flv_born (:, 4)  = [11, -11, -3, 3, 21]
  flv_born (:, 5)  = [11, -11, -1, 1, 21]
  flv_real (:, 1)  = [11, -11, -4, -4, 4, 4]
  flv_real (:, 2)  = [11, -11, -4, -2, 2, 4]
  flv_real (:, 3)  = [11, -11, -4, 4, 21, 21]
  flv_real (:, 4)  = [11, -11, -4, -5, 4, 5]
  flv_real (:, 5)  = [11, -11, -4, -3, 4, 3]
  flv_real (:, 6)  = [11, -11, -4, -1, 2, 3]
  flv_real (:, 7)  = [11, -11, -4, -1, 4, 1]
  flv_real (:, 8)  = [11, -11, -2, -2, 2, 2]
  flv_real (:, 9)  = [11, -11, -2, 2, 21, 21]
  flv_real (:, 10) = [11, -11, -2, -5, 2, 5]
  flv_real (:, 11) = [11, -11, -2, -3, 2, 3]
```

```
    flv_real (:, 12) = [11, -11, -2, -3, 4, 1]
    flv_real (:, 13) = [11, -11, -2, -1, 2, 1]
    flv_real (:, 14) = [11, -11, -5, -5, 5, 5]
    flv_real (:, 15) = [11, -11, -5, -3, 3, 5]
    flv_real (:, 16) = [11, -11, -5, -1, 1, 5]
    flv_real (:, 17) = [11, -11, -5, 5, 21, 21]
    flv_real (:, 18) = [11, -11, -3, -3, 3, 3]
    flv_real (:, 19) = [11, -11, -3, -1, 1, 3]
    flv_real (:, 20) = [11, -11, -3, 3, 21, 21]
    flv_real (:, 21) = [11, -11, -1, -1, 1, 1]
    flv_real (:, 22) = [11, -11, -1, 1, 21, 21]

    call setup_region_data_for_test (n_in, flv_born, flv_real, reg_data)
    call reg_data%check_consistency (.false., u)
    call reg_data%write (u)

    deallocate (flv_born, flv_real)
    call reg_data%final ()

  end subroutine fks_regions_3
```

⟨*fks regions: test declarations*⟩+≡

```
  public :: fks_regions_4
```

⟨*fks regions: tests*⟩+≡

```
  subroutine fks_regions_4 (u)
    integer, intent(in) :: u
    integer :: n_flv_born, n_flv_real
    integer :: n_legs_born, n_legs_real
    integer :: n_in
    integer, dimension(:,:), allocatable :: flv_born, flv_real
    type(region_data_t) :: reg_data
    write (u, "(A)") "* Test output: fks_regions_4"
    write (u, "(A)") "* Create singular regions for processes with four final-state particles"
    write (u, "(A)") "* ee -> 4 jet"
    write (u, "(A)") "* with jet = u:U:d:D:s:S:c:C:b:B:gl"
    write (u, "(A)")
    n_flv_born = 22; n_flv_real = 22
    n_legs_born = 6; n_legs_real = 7
    n_in = 2
    allocate (flv_born (n_legs_born, n_flv_born))
    allocate (flv_real (n_legs_real, n_flv_real))

    flv_born (:, 1)   = [11, -11, -4, -4, 4, 4]
    flv_born (:, 2)   = [11, -11, -4, -2, 2, 4]
    flv_born (:, 3)   = [11, -11, -4, 4, 21, 21]
    flv_born (:, 4)   = [11, -11, -4, -5, 4, 5]
    flv_born (:, 5)   = [11, -11, -4, -3, 4, 3]
    flv_born (:, 6)   = [11, -11, -4, -1, 2, 3]
    flv_born (:, 7)   = [11, -11, -4, -1, 4, 1]
    flv_born (:, 8)   = [11, -11, -2, -2, 2, 2]
    flv_born (:, 9)   = [11, -11, -2, 2, 21, 21]
    flv_born (:, 10)  = [11, -11, -2, -5, 2, 5]
    flv_born (:, 11)  = [11, -11, -2, -3, 2, 3]
```

```
    flv_born (:, 12)  = [11, -11, -2, -3, 4, 1]
    flv_born (:, 13)  = [11, -11, -2, -1, 2, 1]
    flv_born (:, 14)  = [11, -11, -5, -5, 5, 5]
    flv_born (:, 15)  = [11, -11, -5, -3, 3, 5]
    flv_born (:, 16)  = [11, -11, -5, -1, 1, 5]
    flv_born (:, 17)  = [11, -11, -5, 5, 21, 21]
    flv_born (:, 18)  = [11, -11, -3, -3, 3, 3]
    flv_born (:, 19)  = [11, -11, -3, -1, 1, 3]
    flv_born (:, 20)  = [11, -11, -3, -3, 21, 21]
    flv_born (:, 21)  = [11, -11, -1, -1, 1, 1]
    flv_born (:, 22)  = [11, -11, -1, 1, 21, 21]
    flv_real (:, 1)   = [11, -11, -4, -4, 4, 4, 21]
    flv_real (:, 2)   = [11, -11, -4, -2, 2, 4, 21]
    flv_real (:, 3)   = [11, -11, -4, 4, 21, 21, 21]
    flv_real (:, 4)   = [11, -11, -4, -5, 4, 5, 21]
    flv_real (:, 5)   = [11, -11, -4, -3, 4, 3, 21]
    flv_real (:, 6)   = [11, -11, -4, -1, 2, 3, 21]
    flv_real (:, 7)   = [11, -11, -4, -1, 4, 1, 21]
    flv_real (:, 8)   = [11, -11, -2, -2, 2, 2, 21]
    flv_real (:, 9)   = [11, -11, -2, 2, 21, 21, 21]
    flv_real (:, 10)  = [11, -11, -2, -5, 2, 5, 21]
    flv_real (:, 11)  = [11, -11, -2, -3, 2, 3, 21]
    flv_real (:, 12)  = [11, -11, -2, -3, 4, 1, 21]
    flv_real (:, 13)  = [11, -11, -2, -1, 2, 1, 21]
    flv_real (:, 14)  = [11, -11, -5, -5, 5, 5, 21]
    flv_real (:, 15)  = [11, -11, -5, -3, 3, 5, 21]
    flv_real (:, 16)  = [11, -11, -5, -1, 1, 5, 21]
    flv_real (:, 17)  = [11, -11, -5, 5, 21, 21, 21]
    flv_real (:, 18)  = [11, -11, -3, -3, 3, 3, 21]
    flv_real (:, 19)  = [11, -11, -3, -1, 1, 3, 21]
    flv_real (:, 20)  = [11, -11, -3, 3, 21, 21, 21]
    flv_real (:, 21)  = [11, -11, -1, -1, 1, 1, 21]
    flv_real (:, 22)  = [11, -11, -1, 1, 21, 21, 21]

    call setup_region_data_for_test (n_in, flv_born, flv_real, reg_data)
    call reg_data%check_consistency (.false., u)
    call reg_data%write (u)

    deallocate (flv_born, flv_real)
    call reg_data%final ()

  end subroutine fks_regions_4
```

⟨*fks regions: test declarations*⟩+≡

```
  public :: fks_regions_5
```

⟨*fks regions: tests*⟩+≡

```
  subroutine fks_regions_5 (u)
    integer, intent(in) :: u
    integer :: n_flv_born, n_flv_real
    integer :: n_legs_born, n_legs_real
    integer :: n_in
    integer, dimension(:,:), allocatable :: flv_born, flv_real
    type(region_data_t) :: reg_data
```

```
write (u, "(A)") "* Test output: fks_regions_5"
write (u, "(A)") "* Create singular regions for processes with five final-state particles"
write (u, "(A)") "* ee -> 5 jet"
write (u, "(A)") "* with jet = u:U:d:D:s:S:c:C:b:B:gl"
write (u, "(A)")
n_flv_born = 22; n_flv_real = 67
n_legs_born = 7; n_legs_real = 8
n_in = 2
allocate (flv_born (n_legs_born, n_flv_born))
allocate (flv_real (n_legs_real, n_flv_real))


flv_born (:,1)  = [11,-11,-4,-4,4,4,21]
flv_born (:,2)  = [11,-11,-4,-2,2,4,21]
flv_born (:,3)  = [11,-11,-4,4,21,21,21]
flv_born (:,4)  = [11,-11,-4,-5,4,5,21]
flv_born (:,5)  = [11,-11,-4,-3,4,3,21]
flv_born (:,6)  = [11,-11,-4,-1,2,3,21]
flv_born (:,7)  = [11,-11,-4,-1,4,1,21]
flv_born (:,8)  = [11,-11,-2,-2,2,2,21]
flv_born (:,9)  = [11,-11,-2,2,21,21,21]
flv_born (:,10) = [11,-11,-2,-5,2,5,21]
flv_born (:,11) = [11,-11,-2,-3,2,3,21]
flv_born (:,12) = [11,-11,-2,-3,4,1,21]
flv_born (:,13) = [11,-11,-2,-1,2,1,21]
flv_born (:,14) = [11,-11,-5,-5,5,5,21]
flv_born (:,15) = [11,-11,-5,-3,3,5,21]
flv_born (:,16) = [11,-11,-5,-1,1,5,21]
flv_born (:,17) = [11,-11,-5,5,21,21,21]
flv_born (:,18) = [11,-11,-3,-3,3,3,21]
flv_born (:,19) = [11,-11,-3,-1,1,3,21]
flv_born (:,20) = [11,-11,-3,3,21,21,21]
flv_born (:,21) = [11,-11,-1,-1,1,1,21]
flv_born (:,22) = [11,-11,-1,1,21,21,21]

flv_real (:,1)  = [11,-11,-4,-4,-4,4,4,4]
flv_real (:,2)  = [11,-11,-4,-4,-2,2,4,4]
flv_real (:,3)  = [11,-11,-4,-4,4,4,21,21]
flv_real (:,4)  = [11,-11,-4,-4,-5,4,4,5]
flv_real (:,5)  = [11,-11,-4,-4,-3,4,4,3]
flv_real (:,6)  = [11,-11,-4,-4,-1,2,4,3]
flv_real (:,7)  = [11,-11,-4,-4,-1,4,4,1]
flv_real (:,8)  = [11,-11,-4,-2,-2,2,2,4]
flv_real (:,9)  = [11,-11,-4,-2,2,4,21,21]
flv_real (:,10) = [11,-11,-4,-2,-5,2,4,5]
flv_real (:,11) = [11,-11,-4,-2,-3,2,4,3]
flv_real (:,12) = [11,-11,-4,-2,-3,4,4,1]
flv_real (:,13) = [11,-11,-4,-2,-1,2,2,3]
flv_real (:,14) = [11,-11,-4,-2,-1,2,4,1]
flv_real (:,15) = [11,-11,-4,4,21,21,21,21]
flv_real (:,16) = [11,-11,-4,-5,4,5,21,21]
flv_real (:,17) = [11,-11,-4,-5,-5,4,5,5]
flv_real (:,18) = [11,-11,-4,-5,-3,4,3,5]
flv_real (:,19) = [11,-11,-4,-5,-1,2,3,5]
```

```
flv_real (:,20) = [11,-11,-4,-5,-1,4,1,5]
flv_real (:,21) = [11,-11,-4,-3,4,3,21,21]
flv_real (:,22) = [11,-11,-4,-3,-3,4,3,3]
flv_real (:,23) = [11,-11,-4,-3,-1,2,3,3]
flv_real (:,24) = [11,-11,-4,-3,-1,4,1,3]
flv_real (:,25) = [11,-11,-4,-1,2,3,21,21]
flv_real (:,26) = [11,-11,-4,-1,4,1,21,21]
flv_real (:,27) = [11,-11,-4,-1,-1,2,1,3]
flv_real (:,28) = [11,-11,-4,-1,-1,4,1,1]
flv_real (:,29) = [11,-11,-2,-2,-2,2,2,2]
flv_real (:,30) = [11,-11,-2,-2,2,2,21,21]
flv_real (:,31) = [11,-11,-2,-2,-5,2,2,5]
flv_real (:,32) = [11,-11,-2,-2,-3,2,2,3]
flv_real (:,33) = [11,-11,-2,-2,-3,2,4,1]
flv_real (:,34) = [11,-11,-2,-2,-1,2,2,1]
flv_real (:,35) = [11,-11,-2,2,21,21,21,21]
flv_real (:,36) = [11,-11,-2,-5,2,5,21,21]
flv_real (:,37) = [11,-11,-2,-5,-5,2,5,5]
flv_real (:,38) = [11,-11,-2,-5,-3,2,3,5]
flv_real (:,39) = [11,-11,-2,-5,-3,4,1,5]
flv_real (:,40) = [11,-11,-2,-5,-1,2,1,5]
flv_real (:,41) = [11,-11,-2,-3,2,3,21,21]
flv_real (:,42) = [11,-11,-2,-3,4,1,21,21]
flv_real (:,43) = [11,-11,-2,-3,-3,2,3,3]
flv_real (:,44) = [11,-11,-2,-3,-3,4,1,3]
flv_real (:,45) = [11,-11,-2,-3,-1,2,1,3]
flv_real (:,46) = [11,-11,-2,-3,-1,4,1,1]
flv_real (:,47) = [11,-11,-2,-1,2,1,21,21]
flv_real (:,48) = [11,-11,-2,-1,-1,2,1,1]
flv_real (:,49) = [11,-11,-5,-5,-5,5,5,5]
flv_real (:,50) = [11,-11,-5,-5,-3,3,5,5]
flv_real (:,51) = [11,-11,-5,-5,-1,1,5,5]
flv_real (:,52) = [11,-11,-5,-5,5,5,21,21]
flv_real (:,53) = [11,-11,-5,-3,-3,3,3,5]
flv_real (:,54) = [11,-11,-5,-3,-1,1,3,5]
flv_real (:,55) = [11,-11,-5,-3,3,5,21,21]
flv_real (:,56) = [11,-11,-5,-1,-1,1,1,5]
flv_real (:,57) = [11,-11,-5,-1,1,5,21,21]
flv_real (:,58) = [11,-11,-5,5,21,21,21,21]
flv_real (:,59) = [11,-11,-3,-3,-3,3,3,3]
flv_real (:,60) = [11,-11,-3,-3,-1,1,3,3]
flv_real (:,61) = [11,-11,-3,-3,3,3,21,21]
flv_real (:,62) = [11,-11,-3,-1,-1,1,1,3]
flv_real (:,63) = [11,-11,-3,-1,1,3,21,21]
flv_real (:,64) = [11,-11,-3,3,21,21,21,21]
flv_real (:,65) = [11,-11,-1,-1,-1,1,1,1]
flv_real (:,66) = [11,-11,-1,-1,1,1,21,21]
flv_real (:,67) = [11,-11,-1,1,21,21,21,21]

call setup_region_data_for_test (n_in, flv_born, flv_real, reg_data)
call reg_data%check_consistency (.false., u)
call reg_data%write (u)

deallocate (flv_born, flv_real)
```

```
    call reg_data%final ()

  end subroutine fks_regions_5
```

⟨*fks regions: test declarations*⟩+≡

```
  public :: fks_regions_6
```

⟨*fks regions: tests*⟩+≡

```
  subroutine fks_regions_6 (u)
    integer, intent(in) :: u
    integer :: n_flv_born, n_flv_real
    integer :: n_legs_born, n_legs_real
    integer :: n_in
    integer, dimension(:,:), allocatable :: flv_born, flv_real
    type(region_data_t) :: reg_data
    integer :: i, j
    integer, dimension(10) :: flavors
    write (u, "(A)") "* Test output: fks_regions_6"
    write (u, "(A)") "* Create table of singular regions for Drell Yan"
    write (u, "(A)")

    n_flv_born = 10; n_flv_real = 30
    n_legs_born = 4; n_legs_real = 5
    n_in = 2
    allocate (flv_born (n_legs_born, n_flv_born))
    allocate (flv_real (n_legs_real, n_flv_real))
    flavors = [-5, -4, -3, -2, -1, 1, 2, 3, 4, 5]
    do i = 1, n_flv_born
       flv_born (3:4, i) = [11, -11]
    end do
    do j = 1, n_flv_born
       flv_born (1, j) = flavors (j)
       flv_born (2, j) = -flavors (j)
    end do

    do i = 1, n_flv_real
       flv_real (3:4, i) = [11, -11]
    end do
    i = 1
    do j = 1, n_flv_real
       if (mod (j, 3) == 1) then
          flv_real (1, j) = flavors (i)
          flv_real (2, j) = -flavors (i)
          flv_real (5, j) = 21
       else if (mod (j, 3) == 2) then
          flv_real (1, j) = flavors (i)
          flv_real (2, j) = 21
          flv_real (5, j) = flavors (i)
       else
          flv_real (1, j) = 21
          flv_real (2, j) = -flavors (i)
          flv_real (5, j) = -flavors (i)
          i = i + 1
       end if
```

```
    end do

    call setup_region_data_for_test (n_in, flv_born, flv_real, reg_data)
    call reg_data%check_consistency (.false., u)
    call reg_data%write (u)

    call write_separator (u)

    deallocate (flv_born, flv_real)
    call reg_data%final ()

    write (u, "(A)") "* Create table of singular regions for hadronic top decay"
    write (u, "(A)")
    n_flv_born = 1; n_flv_real = 1
    n_legs_born = 4; n_legs_real = 5
    n_in = 1
    allocate (flv_born (n_legs_born, n_flv_born))
    allocate (flv_real (n_legs_real, n_flv_real))

    flv_born (:, 1) = [6, -5, 2, -1]
    flv_real (:, 1) = [6, -5, 2, -1, 21]

    call setup_region_data_for_test (n_in, flv_born, flv_real, reg_data)
    call reg_data%check_consistency (.false., u)
    call reg_data%write (u)
    call reg_data%final ()

  end subroutine fks_regions_6
```

⟨*fks regions: test declarations*⟩+≡

```
  public :: fks_regions_7
```

⟨*fks regions: tests*⟩+≡

```
  subroutine fks_regions_7 (u)
    integer, intent(in) :: u
    integer :: n_flv_born, n_flv_real
    integer :: n_legs_born, n_legs_real
    integer :: n_in
    integer, dimension(:,:), allocatable :: flv_born, flv_real
    type(region_data_t) :: reg_data
    write (u, "(A)") "* Test output: fks_regions_7"
    write (u, "(A)") "* Create table of singular regions for ee -> qq"
    write (u, "(A)")

    n_flv_born = 1; n_flv_real = 1
    n_legs_born = 4; n_legs_real = 5
    n_in = 2

    allocate (flv_born (n_legs_born, n_flv_born))
    allocate (flv_real (n_legs_real, n_flv_real))
    flv_born (:, 1) = [11, -11, 2, -2]
    flv_real (:, 1) = [11, -11, 2, -2, 21]
    call setup_region_data_for_test (n_in, flv_born, flv_real, reg_data)
    call reg_data%write_latex (u)
```

```
      call reg_data%final ()

    end subroutine fks_regions_7
```

27.4 Virtual contribution to the cross section

⟨virtual.f90⟩≡
⟨*File header*⟩

```
  module virtual
```

⟨*Use kinds*⟩
⟨*Use strings*⟩

```
    use numeric_utils
    use constants
    use diagnostics
    use pdg_arrays
    use models
    use physics_defs
    use sm_physics
    use lorentz
    use flavors
    use blha_olp_interfaces, only: blha_loop_positions
    use nlo_data, only: get_threshold_momenta
    use nlo_data, only: ASSOCIATED_LEG_PAIR
    use fks_regions
```

⟨*Standard module head*⟩

⟨*virtual: public*⟩

⟨*virtual: parameters*⟩

⟨*virtual: types*⟩

```
  contains
```

⟨*virtual: procedures*⟩

```
  end module virtual
```

⟨*virtual: public*⟩≡

```
    public :: virtual_t
```

⟨*virtual: types*⟩≡

```
    type :: virtual_t
      real(default), dimension(:,:), allocatable :: gamma_0, gamma_p, c_flv
      real(default) :: ren_scale2, fac_scale, es_scale2
      integer, dimension(:), allocatable :: n_is_neutrinos
      integer :: n_in, n_legs, n_flv
      logical :: bad_point = .false.
      type(string_t) :: selection
      integer :: factorization_mode = NO_FACTORIZATION
```

```
    real(default), dimension(:,:,:), allocatable :: sqme_cc
    logical :: use_internal_cc = .false.
    integer, dimension(:), allocatable :: me_index
    logical :: collinear_resonance_aware = .true.
  contains
  ⟨virtual: virtual: TBP⟩
  end type virtual_t
```

⟨*virtual: virtual: TBP*⟩≡

```
  procedure :: init => virtual_init
```

⟨*virtual: procedures*⟩≡

```
  subroutine virtual_init (virt, flv_born, n_in, use_internal_cc, selection, resonance_aware)
    class(virtual_t), intent(inout) :: virt
    integer, intent(in), dimension(:,:) :: flv_born
    integer, intent(in) :: n_in
    logical, intent(in) :: use_internal_cc
    type(string_t), intent(in) :: selection
    logical, intent(in) :: resonance_aware
    integer :: n_sub
    integer :: i_flv
    virt%n_legs = size (flv_born, 1); virt%n_flv = size (flv_born, 2)
    virt%n_in = n_in
    allocate (virt%sqme_cc (virt%n_legs, virt%n_legs, virt%n_flv))
    allocate (virt%gamma_0 (virt%n_legs, virt%n_flv), &
       virt%gamma_p (virt%n_legs, virt%n_flv), &
       virt%c_flv (virt%n_legs, virt%n_flv))
    call virt%init_constants (flv_born)
    allocate (virt%n_is_neutrinos (virt%n_flv))
    virt%n_is_neutrinos = 0
    do i_flv = 1, virt%n_flv
       if (is_neutrino (flv_born(1, i_flv))) &
          virt%n_is_neutrinos(i_flv) = virt%n_is_neutrinos(i_flv) + 1
       if (is_neutrino (flv_born(2, i_flv))) &
          virt%n_is_neutrinos(i_flv) = virt%n_is_neutrinos(i_flv) + 1
    end do
    virt%use_internal_cc = use_internal_cc
    allocate (virt%me_index (virt%n_flv))
    n_sub = virt%compute_n_sub ()
    do i_flv = 1, virt%n_flv
       virt%me_index (i_flv) = blha_loop_positions (i_flv, n_sub)
    end do
    select case (char (selection))
    case ("Full", "OLP", "Subtraction")
       virt%selection = selection
    case default
       call msg_fatal ('Virtual selection: Possible values are "Full", "OLP" or "Subtraction')
    end select
    virt%collinear_resonance_aware = resonance_aware
  contains

    function is_neutrino (flv) result (neutrino)
      integer, intent(in) :: flv
      logical :: neutrino
```

```
    neutrino = (abs(flv) == 12 .or. abs(flv) == 14 .or. abs(flv) == 16)
  end function is_neutrino

end subroutine virtual_init
```

The virtual subtraction terms contain Casimir operators and derived constants, listed below:

$$C(q) = C(\bar{q}) = C_F, \tag{27.2}$$

$$C(g) = C_A, \tag{27.3}$$

$$\gamma(q) = \gamma(\bar{q}) = \frac{3}{2} C_F, \tag{27.4}$$

$$\gamma(g) = \frac{11}{6} C_A - \frac{2}{3} T_F N_f, \tag{27.5}$$

$$\gamma'(q) = \gamma'(\bar{q}) = \left(\frac{13}{2} - \frac{2\pi^2}{3}\right) C_F, \tag{27.6}$$

$$\gamma'(g) = \left(\frac{67}{9} - \frac{2\pi^2}{3}\right) C_A - \frac{23}{9} T_F N_f. \tag{27.7}$$

For uncolored particles, `virtual_init_constants` sets C, γ and γ' to zero.

⟨*virtual: virtual: TBP*⟩+≡

```
procedure :: init_constants => virtual_init_constants
```

⟨*virtual: procedures*⟩+≡

```
subroutine virtual_init_constants (virt, flv_born)
  class(virtual_t), intent(inout) :: virt
  integer, intent(in), dimension(:,:) :: flv_born
  integer :: i_part, i_flv
  integer, parameter :: nf = 1
  do i_flv = 1, size (flv_born, 2)
     do i_part = 1, size (flv_born, 1)
        if (is_gluon (flv_born(i_part, i_flv))) then
           virt%gamma_0(i_part, i_flv) = (11 * ca - 2 * nf) / 6
           virt%gamma_p(i_part, i_flv) = (67.0 / 9 - 2 * pi**2 / 3) * ca &
              - 23.0 / 18 * nf
           virt%c_flv(i_part, i_flv) = ca
        else if (is_quark (flv_born(i_part, i_flv))) then
           virt%gamma_0(i_part, i_flv) = 1.5 * cf
           virt%gamma_p(i_part, i_flv) = (6.5 - 2 * pi**2 / 3) * cf
           virt%c_flv(i_part, i_flv) = cf
        else
           virt%gamma_0(i_part, i_flv) = zero
           virt%gamma_p(i_part, i_flv) = zero
           virt%c_flv(i_part, i_flv) = zero
        end if
     end do
  end do
end subroutine virtual_init_constants
```

Set the renormalization scale. If the input is zero, use the center-of-mass energy.

⟨*virtual: virtual: TBP*⟩+≡

```
procedure :: set_ren_scale => virtual_set_ren_scale
```

⟨*virtual: procedures*⟩+≡

```
subroutine virtual_set_ren_scale (virt, p, ren_scale)
  class(virtual_t), intent(inout) :: virt
  type(vector4_t), intent(in), dimension(:) :: p
  real(default), intent(in) :: ren_scale
  if (ren_scale > 0) then
    virt%ren_scale2 = ren_scale**2
  else
    virt%ren_scale2 = (p(1) + p(2))**2
  end if
end subroutine virtual_set_ren_scale
```

⟨*virtual: virtual: TBP*⟩+≡

```
procedure :: set_fac_scale => virtual_set_fac_scale
```

⟨*virtual: procedures*⟩+≡

```
subroutine virtual_set_fac_scale (virt, p, fac_scale)
  class(virtual_t), intent(inout) :: virt
  type(vector4_t), dimension(:), intent(in) :: p
  real(default), optional :: fac_scale
  if (present (fac_scale)) then
     virt%fac_scale = fac_scale
  else
     virt%fac_scale = (p(1) + p(2))**1
  end if
end subroutine virtual_set_fac_scale
```

⟨*virtual: virtual: TBP*⟩+≡

```
procedure :: set_ellis_sexton_scale => virtual_set_ellis_sexton_scale
```

⟨*virtual: procedures*⟩+≡

```
subroutine virtual_set_ellis_sexton_scale (virt, Q2)
  class(virtual_t), intent(inout) :: virt
  real(default), intent(in), optional :: Q2
  if (present (Q2)) then
     virt%es_scale2 = Q2
  else
     virt%es_scale2 = virt%ren_scale2
  end if
end subroutine virtual_set_ellis_sexton_scale
```

⟨*virtual: virtual: TBP*⟩+≡

```
procedure :: compute_n_sub => virtual_compute_n_sub
```

⟨*virtual: procedures*⟩+≡

```
function virtual_compute_n_sub (virt) result (n_sub)
  integer :: n_sub
  class(virtual_t), intent(in) :: virt
  n_sub = 1
  if (.not. virt%use_internal_cc) &
       n_sub = n_sub + virt%n_legs * (virt%n_legs - 1 ) / 2
end function virtual_compute_n_sub
```

The virtual-subtracted matrix element is given by the equation

$$\mathcal{V} = \frac{\alpha_s}{2\pi}\left(\mathcal{Q}\mathcal{B} + \sum \mathcal{I}_{ij}\mathcal{B}_{ij} + \mathcal{V}_{fin}\right), \tag{27.8}$$

The expressions for $\mathcal{Q}$ can be found in equations 27.9 and 27.11. The expressions for $\mathcal{I}_{ij}$ can be found in equations (27.12), (27.13), (27.14), depending on whether the particles involved in the radiation process are massive or massless.

⟨*virtual: virtual: TBP*⟩+≡

```
procedure :: evaluate => virtual_evaluate
```

⟨*virtual: procedures*⟩+≡

```
subroutine virtual_evaluate (virt, reg_data, alpha_s, &
       p_born, sqme, separate_alrs, sqme_virt)
  class(virtual_t), intent(inout) :: virt
  type(region_data_t), intent(in) :: reg_data
  real(default), intent(in) :: alpha_s
  type(vector4_t), intent(in), dimension(:)  :: p_born
  real(default), intent(in), dimension(:) :: sqme
  logical, intent(in) :: separate_alrs
  real(default), dimension(:), intent(inout) :: sqme_virt
  real(default) :: sqme_alr
  real(default) :: s, s_o_Q2
  real(default), dimension(reg_data%n_flv_born) :: QB, BI
  integer :: i_flv, ii_flv
  QB = zero; BI = zero
  if (virt%bad_point) return
  if (debug2_active (D_VIRTUAL)) then
     print *, 'Compute virtual component using alpha_s = ', alpha_s
     print *, 'Virtual selection: ', char (virt%selection)
     print *, 'virt%es_scale2 =    ', virt%es_scale2 !!! Debugging
  end if
  s = sum (p_born(1 : virt%n_in))**2
  if (virt%factorization_mode == FACTORIZATION_THRESHOLD) &
       call set_s_for_threshold ()
  s_o_Q2 = s / virt%es_scale2
  do i_flv = 1, reg_data%n_flv_born

     if (separate_alrs) then
        ii_flv = i_flv
     else
        ii_flv = 1
     end if

     if (virt%selection == var_str ("Full") .or. virt%selection == var_str ("OLP")) then
        !!! A factor of alpha_s/twopi is assumed to be included in vfin
        sqme_virt(ii_flv) = sqme_virt(ii_flv) + sqme(virt%me_index(i_flv))
     end if

     if (virt%selection == var_str ("Full") .or. virt%selection == var_str ("Subtraction")) then
        call virt%evaluate_initial_state (i_flv, sqrt(s), reg_data, sqme, QB)
        call virt%compute_collinear_contribution (i_flv, p_born, sqrt(s), reg_data, sqme, QB)

        select case (virt%factorization_mode)
        case (FACTORIZATION_THRESHOLD)
```

```
            call virt%compute_eikonals_threshold (i_flv, p_born, s, s_o_Q2, reg_data, sqme, QB, B
         case default
            call virt%compute_massive_self_eikonals (i_flv, p_born, s, reg_data, sqme, QB)
            call virt%compute_eikonals (i_flv, p_born, s, s_o_Q2, reg_data, sqme, BI)
         end select

         if (debug2_active (D_VIRTUAL)) then
            print *, 'Evaluate i_flv: ', i_flv
            print *, 'Q * sqme_born: ', QB(i_flv)
            print *, 'BI: ', BI(i_flv)
            print *, 'vfin: ', sqme (virt%me_index(i_flv))
         end if
         sqme_virt(ii_flv) = sqme_virt(ii_flv) + alpha_s / twopi * (QB(i_flv) + BI(i_flv))
      end if
   end do

   if (debug2_active (D_VIRTUAL)) then
      call msg_debug2 (D_VIRTUAL, "virtual-subtracted matrix element(s): ")
      print *, sqme_virt
   end if

   do i_flv = 1, reg_data%n_flv_born
      if (virt%n_is_neutrinos(i_flv) > 0) &
           sqme_virt = sqme_virt * virt%n_is_neutrinos(i_flv) * two
   end do
contains
   subroutine set_s_for_threshold ()
     use ttv_formfactors, only: m1s_to_mpole
     real(default) :: mtop2
     mtop2 = m1s_to_mpole (sqrt(s))**2
     if (s < four * mtop2) s = four * mtop2
   end subroutine set_s_for_threshold
end subroutine virtual_evaluate
```

⟨*virtual: virtual: TBP*⟩+≡

```
procedure :: compute_eikonals => virtual_compute_eikonals
```

⟨*virtual: procedures*⟩+≡

```
subroutine virtual_compute_eikonals (virtual, i_flv, &
         p_born, s, s_o_Q2, reg_data, sqme, BI)
   class(virtual_t), intent(inout) :: virtual
   integer, intent(in) :: i_flv
   type(vector4_t), intent(in), dimension(:)  :: p_born
   real(default), intent(in) :: s, s_o_Q2
   type(region_data_t), intent(in) :: reg_data
   real(default), intent(in), dimension(:) :: sqme
   real(default), intent(inout), dimension(:) :: BI
   integer :: i_born
   integer :: i, j, k
   real(default) :: I_ij, BI_tmp
   BI_tmp = zero
   i_born = virtual%me_index(i_flv) + 1
   associate (flst_born => reg_data%flv_born(i_flv))
      do i = 1, virtual%n_legs
```

```
            do j = 1, virtual%n_legs
               if (i /= j) then
                  if (flst_born%colored(i) .and. flst_born%colored(j)) then
                     I_ij = compute_eikonal_factor (p_born, flst_born%massive, &
                          i, j, s_o_Q2)
                     BI_tmp = BI_tmp + virtual%sqme_cc (i, j, i_flv) * I_ij
                     if (debug2_active (D_VIRTUAL)) &
                        print *, 'b_ij: ', virtual%sqme_cc (i, j, i_flv), 'I_ij: ', I_ij
                  end if
               end if
            end do
         end do
      end associate
      if (virtual%use_internal_cc) BI_tmp = BI_tmp * sqme (i_born)
      BI(i_flv) = BI(i_flv) + BI_tmp
   end subroutine virtual_compute_eikonals
```

⟨*virtual: virtual: TBP*⟩+≡

```
   procedure :: compute_eikonals_threshold => virtual_compute_eikonals_threshold
```

⟨*virtual: procedures*⟩+≡

```
   subroutine virtual_compute_eikonals_threshold (virtual, i_flv, &
         p_born, s, s_o_Q2, reg_data, sqme, QB, BI)
      class(virtual_t), intent(in) :: virtual
      integer, intent(in) :: i_flv
      type(vector4_t), intent(in), dimension(:) :: p_born
      real(default), intent(in) :: s, s_o_Q2
      type(region_data_t), intent(in) :: reg_data
      real(default), intent(in), dimension(:) :: sqme
      real(default), intent(inout), dimension(:) :: QB
      real(default), intent(inout), dimension(:) :: BI
      type(vector4_t), dimension(4) :: p_thr
      integer :: i_born, leg
      BI = zero; p_thr = get_threshold_momenta (p_born)
      i_born = virtual%me_index(i_flv) + 1
      call compute_massive_self_eikonals (sqme (i_born), QB(i_flv))
      do leg = 1, 2
         BI(i_flv) = BI(i_flv) + evaluate_leg_pair (ASSOCIATED_LEG_PAIR(leg), i_flv, i_born)
      end do
   contains

      subroutine compute_massive_self_eikonals (sqme_born, QB)
         real(default), intent(in) :: sqme_born
         real(default), intent(inout) :: QB
         real(default) :: term_1, term_2
         integer :: i
         call msg_debug2 (D_VIRTUAL, "compute_massive_self_eikonals")
         call msg_debug2 (D_VIRTUAL, "s_o_Q2", s_o_Q2)
         call msg_debug2 (D_VIRTUAL, "log (s_o_Q2)", log (s_o_Q2))
         do i = 1, 4
            term_1 = log (s_o_Q2)
            term_2 = 0.5_default * I_m_eps (p_thr(i))
            QB = QB - (cf * (term_1 - term_2)) * sqme_born
         end do
```

```
  end subroutine compute_massive_self_eikonals

  function evaluate_leg_pair (i_start, i_flv, i_born) result (b_ij_times_I)
    real(default) :: b_ij_times_I
    integer, intent(in) :: i_start, i_flv, i_born
    real(default) :: I_ij
    integer :: i, j
    b_ij_times_I = zero
    do i = i_start, i_start + 1
       do j = i_start, i_start + 1
          if (i /= j) then
             I_ij = compute_eikonal_factor &
                  (p_thr, [.true., .true., .true., .true.], i, j, s_o_Q2)
             b_ij_times_I = b_ij_times_I + &
                  virtual%sqme_cc (i, j, i_flv) * I_ij
             if (debug2_active (D_VIRTUAL)) &
                print *, 'b_ij: ', virtual%sqme_cc (i, j, i_flv), 'I_ij: ', I_ij
          end if
       end do
    end do
    if (virtual%use_internal_cc) b_ij_times_I = b_ij_times_I * sqme (i_born)
    if (debug2_active (D_VIRTUAL)) then
       print *, 'internal color: ', virtual%use_internal_cc
       print *, 'b_ij_times_I =    ', b_ij_times_I
       print *, 'QB           =    ', QB
    end if
  end function evaluate_leg_pair
end subroutine virtual_compute_eikonals_threshold
```

⟨*virtual: virtual: TBP*⟩+≡

```
procedure :: set_bad_point => virtual_set_bad_point
```

⟨*virtual: procedures*⟩+≡

```
subroutine virtual_set_bad_point (virt, value)
   class(virtual_t), intent(inout) :: virt
   logical, intent(in) :: value
   virt%bad_point = value
end subroutine virtual_set_bad_point
```

The collinear limit of $\tilde{\mathcal{R}}$ can be integrated over the radiation degrees of freedom, giving the collinear contribution to the virtual component. Its general structure is $\mathcal{Q} \cdot \mathcal{B}$. The initial-state contribution to $\mathcal{Q}$ is simply given by

$$\mathcal{Q} = -\log \frac{\mu_F^2}{Q^2} \left(\gamma(\mathcal{I}_1) + \gamma(\mathcal{I}_2)\right), \tag{27.9}$$

where Q^2 is the Ellis-Sexton scale and γ is as in eqns. 27.4 and 27.5. `virtual_evaluate_initial_state` computes this quantity. The loop over the initial-state particles is only executed if we are dealing with a scattering process, because for decays there are no virtual initial-initial interactions.

⟨*virtual: virtual: TBP*⟩+≡

```
procedure :: evaluate_initial_state => virtual_evaluate_initial_state
```

⟨*virtual: procedures*⟩+≡

```
subroutine virtual_evaluate_initial_state (virt, i_flv, sqrts, reg_data, sqme, QB)
  class(virtual_t), intent(inout) :: virt
  integer, intent(in) :: i_flv
  real(default), intent(in) :: sqrts
  type(region_data_t), intent(in) :: reg_data
  real(default), intent(in), dimension(:) :: sqme
  real(default), intent(inout), dimension(:) :: QB
  integer :: i, i_sqme
  if (virt%n_in == 2) then
     i_sqme = virt%me_index(i_flv) + 1
     do i = 1, virt%n_in
        QB(i_flv) = QB(i_flv) - virt%gamma_0 (i, i_flv) * &
             log(virt%fac_scale**2 / virt%es_scale2) * sqme (i_sqme)
     end do
  end if
end subroutine virtual_evaluate_initial_state
```

Same as above, but for final-state particles. The collinear limit for final-state particles follows from the integral

$$I_{+,\alpha_r} = \int d\Phi_{n+1} \frac{\xi_+^{-1-2\epsilon}}{\xi^{-1-2\epsilon}} \mathcal{R}_{\alpha_r}.$$

We can distinguish three situations:

1. α_r contains a massive emitter. In this case, no collinear subtraction terms is required and the integral above irrelevant.

2. α_r contains a massless emitter, but resonances are not taken into account in the subtraction. Here, $\xi_{max} = \frac{2E_{em}}{\sqrt{s}}$ is the upper bound on ξ.

3. α_r contains a massless emitter and resonance-aware subtraction is used. Here, $\xi_{max} = \frac{2E_{em}}{\sqrt{k_{res}^2}}$.

Before version 2.4, only situations 1 and 2 were covered. The difference between situation 2 and 3 comes from the expansion of the plus-distribution in the integral above,

$$\xi_+^{-1-2\epsilon} = \xi^{-1-2\epsilon} + \frac{1}{2\epsilon}\delta(\xi) = \xi_{max}^{-1-2\epsilon}\left[(1-z)^{-1-2\epsilon} + \frac{\xi_{max}^{2\epsilon}}{2\epsilon}\delta(1-z)\right].$$

The expression from the standard FKS literature is given by $\mathcal{Q}$ is given by

$$\begin{aligned}\mathcal{Q} = \sum_{k=n_{in}}^{n_L^{(B)}} \Bigg[& \gamma'(\mathcal{I}_k) - \log\frac{s}{2Q^2}\left(\gamma(\mathcal{I}_k) - 2C(\mathcal{I}_k)\log\frac{2E_k}{\sqrt{s}}\right) \\ & + 2C(\mathcal{I}_k)\log^2\frac{2E_k}{\sqrt{s}} - 2\gamma(\mathcal{I}_k)\log\frac{2E_k}{\sqrt{s}}\Bigg].\end{aligned} \tag{27.10}$$

$n_L^{(B)}$ is the number of legs at Born level. Here, ξ_{max} is implicitly present in the

ratios in the logarithms. Using the resonance-aware ξ_{max} yields

$$\mathcal{Q} = \sum_{k=n_{in}}^{n_L^{(B)}} \left[\gamma'(\mathcal{I}_k) + \left(\log \frac{\sqrt{s}}{2E_{em}} + \log \xi_{max} \right) \left(\log \frac{\sqrt{s}}{2E_{em}} + \log \xi_{max} + \log \frac{Q^2}{s} \right) C(\mathcal{I}_k) \right.$$
$$\left. + 2 \log \xi_{max} \left(\log \xi_{max} - \log \frac{Q^2}{k_{res}^2} \right) C(\mathcal{I}_k) + \left(\frac{Q^2}{k_{res}^2} - 2 \log \xi_{max} \right) \gamma(\mathcal{I}_k) \right] . \tag{27.11}$$

Equation 27.11 leads to 27.10 with the substitutions $\xi_{max} \to \frac{2E_em}{\sqrt{s}}$ and $k_{res}^2 \to s$. `virtual_compute_collinear_contribution` only implements the second one.

⟨*virtual: virtual: TBP*⟩+≡
```
procedure :: compute_collinear_contribution &
   => virtual_compute_collinear_contribution
```

⟨*virtual: procedures*⟩+≡
```
subroutine virtual_compute_collinear_contribution (virt, i_flv, &
         p_born, sqrts, reg_data, sqme, QB)
  class(virtual_t), intent(inout) :: virt
  integer, intent(in) :: i_flv
  type(vector4_t), dimension(:), intent(in) :: p_born
  real(default), intent(in) :: sqrts
  type(region_data_t), intent(in) :: reg_data
  real(default), intent(in), dimension(:) :: sqme
  real(default), intent(inout), dimension(:) :: QB
  real(default) :: s1, s2, s3, s4, s5
  integer :: alr, em, i_sqme
  real(default) :: E_em, log_xi_max, E_tot2
  logical :: massive
  logical, dimension(virt%n_flv, virt%n_legs) :: evaluated
  integer :: i_contr
  type(vector4_t) :: k_res
  type(lorentz_transformation_t) :: L_to_resonance
  evaluated = .false.
  do alr = 1, reg_data%n_regions
     if (i_flv /= reg_data%regions(alr)%uborn_index) cycle
     em = reg_data%regions(alr)%emitter
     if (em == 0) cycle
     if (evaluated(i_flv, em)) cycle
     massive = reg_data%regions(alr)%flst_uborn%massive(em)
     !!! Collinear terms only for massless particles
     if (massive) cycle
     E_em = p_born(em)%p(0)
     i_sqme = virt%me_index(i_flv) + 1
     if (allocated (reg_data%alr_contributors)) then
        i_contr = reg_data%alr_to_i_contributor (alr)
        k_res = get_resonance_momentum (p_born, reg_data%alr_contributors(i_contr)%c)
        E_tot2 = k_res%p(0)**2
        L_to_resonance = inverse (boost (k_res, k_res**1))
        log_xi_max = log (two * space_part_norm (L_to_resonance * p_born(em)) / k_res%p(0))
     else
        E_tot2 = sqrts**2
        log_xi_max = log (two * E_em / sqrts)
     end if
```

```
      if (virt%collinear_resonance_aware) then
         if (debug_active (D_VIRTUAL)) &
             call msg_debug (D_VIRTUAL, "Using resonance-aware collinear subtraction")
         s1 = virt%gamma_p(em, i_flv)
         s2 = two * (log (sqrts / (two * E_em)) + log_xi_max) * &
              (log (sqrts / (two * E_em)) + log_xi_max + log (virt%es_scale2 / sqrts**2)) &
            * virt%c_flv(em, i_flv)
         s3 = two * log_xi_max * &
              (log_xi_max - log (virt%es_scale2 / E_tot2)) * virt%c_flv(em, i_flv)
         s4 = (log (virt%es_scale2 / E_tot2) - two * log_xi_max) * virt%gamma_0(em, i_flv)
         QB(i_flv) = QB(i_flv) + (s1 + s2 + s3 + s4) * sqme(i_sqme)
      else
         if (debug_active (D_VIRTUAL)) &
             call msg_debug (D_VIRTUAL, "Using old-fashioned collinear subtraction")
         s1 = virt%gamma_p(em, i_flv)
         s2 = log (sqrts**2 / virt%es_scale2) * virt%gamma_0(em,i_flv)
         s3 = log (sqrts**2 / virt%es_scale2) * two * virt%c_flv(em,i_flv) * &
              log (two * E_em / sqrts)
         s4 = two * virt%c_flv(em,i_flv) * log (two * E_em / sqrts)**2
         s5 = two * virt%gamma_0(em,i_flv) * log (two * E_em / sqrts)
         QB(i_flv) = QB(i_flv) + (s1 - s2 + s3 + s4 - s5) * sqme(i_sqme)
      end if
      evaluated(i_flv, em) = .true.
   end do
end subroutine virtual_compute_collinear_contribution
```

⟨*virtual: virtual: TBP*⟩+≡

```
procedure :: compute_massive_self_eikonals => virtual_compute_massive_self_eikonals
```

⟨*virtual: procedures*⟩+≡

```
subroutine virtual_compute_massive_self_eikonals (virt, i_flv, &
         p_born, s, reg_data, sqme, QB)
   class(virtual_t), intent(inout) :: virt
   integer, intent(in) :: i_flv
   type(vector4_t), intent(in), dimension(:) :: p_born
   real(default), intent(in) :: s
   type(region_data_t), intent(in) :: reg_data
   real(default), intent(in), dimension(:) :: sqme
   real(default), intent(inout), dimension(:) :: QB
   real(default) :: term1, term2
   integer :: i, i_born
   logical :: massive
   i_born = virt%me_index(i_flv) + 1
   do i = 1, virt%n_legs
      massive = reg_data%flv_born(i_flv)%massive(i)
      if (massive) then
         term1 = log(s / virt%es_scale2)
         term2 = 0.5_default * I_m_eps (p_born(i))
         QB(i_flv) = QB(i_flv) - (virt%c_flv (i, i_flv) * (term1 - term2)) * sqme (i_born)
      end if
   end do
end subroutine virtual_compute_massive_self_eikonals
```

The following code implements the $\mathcal{I}_{ij}$-function appearing in eq. blub. They are defined as follows:

Massles-Massles Case

$$\begin{aligned}\mathcal{I}_{ij} = &\frac{1}{2}\log^2\frac{s}{Q^2} + \log\frac{s}{Q^2}\log\frac{k_ik_j}{2E_iE_j} - \mathrm{Li}_2\left(\frac{\mathrm{k_i k_j}}{\mathrm{2E_i E_j}}\right)\\ &+\frac{1}{2}\log^2\frac{k_ik_j}{2E_iE_j} - \log\left(1-\frac{k_ik_j}{2E_iE_j}\right)\log\frac{k_ik_j}{2E_iE_j}.\end{aligned} \tag{27.12}$$

Massive-Massive Case

$$\mathcal{I}_{ij} = -\frac{1}{2}I_0(k_i,k_j)\log\frac{Q^2}{s} - \frac{1}{2}I_\epsilon(k_i,k_j) \tag{27.13}$$

with

$$I_0(k_i,k_j) = \frac{1}{\beta}\log\frac{1+\beta}{1-\beta}, \qquad \beta = \sqrt{1-\frac{k_i^2k_j^2}{(k_i\cdot k_j)^2}}$$

and a rather involved expression for I_ϵ:

$$\begin{aligned}I_\epsilon(k_i,k_j) &= (K(z_j)-K(z_i))\,\frac{1-\vec{\beta}_i\cdot\vec{\beta}_j}{\sqrt{a(1-b)}},\\ \vec{\beta}_i &= \frac{\vec{k}_i}{k_i^0},\\ a &= \beta_i^2+\beta_j^2-2\vec{\beta}_i\cdot\vec{\beta}_j,\\ x_i &= \frac{\beta_i^2-\vec{\beta}_i\cdot\vec{\beta}_j}{a},\\ x_j &= \frac{\beta_j^2-\vec{\beta}_i\cdot\vec{\beta}_j}{a} = 1-x_j,\\ b &= \frac{\beta_i^2\beta_j^2-(\vec{\beta}_i\cdot\vec{\beta}_j)^2}{a},\\ c &= \sqrt{\frac{b}{4a}},\\ z_+ &= \frac{1+\sqrt{1-b}}{\sqrt{b}},\\ z_- &= \frac{1-\sqrt{1-b}}{\sqrt{b}},\\ z_i &= \frac{\sqrt{x_i^2+4c^2}-x_i}{2c},\\ z_j &= \frac{\sqrt{x_j^2+4c^2}+x_j}{2c},\\ K(z) &= -\frac{1}{2}\log^2\frac{(z-z_-)(z_+-z)}{(z_++z)(z_-+z)} - 2Li_2\left(\frac{2z_-(z_+-z)}{(z_+-z_-)(z_-+z)}\right)\\ &\quad -2Li_2\left(-\frac{2z_+(z_-+z)}{(z_+-z_-)(z_+-z)}\right)\end{aligned}$$

Massive-Massless Case

$$\mathcal{I}_{ij} = \frac{1}{2}\left[\log^2\frac{Q}{s} - \frac{\pi^2}{6}\right] - \frac{1}{2}I_0(k_i,k_j)\log\frac{Q^2}{s} - \frac{1}{2}I_\epsilon(k_i,k_j) \qquad (27.14)$$

with

$$I_0(p,k) = \log\frac{(\hat{p}\cdot\hat{k})^2}{\hat{k}^2},$$

$$I_\varepsilon(p,k) = -2\left[\frac{1}{4}\log^2\frac{1-\beta}{1+\beta} + \log\frac{\hat{p}\cdot\hat{k}}{1+\beta}\log\frac{\hat{p}\cdot\hat{k}}{1-\beta} + Li_2\left(1-\frac{\hat{p}\cdot\hat{k}}{1+\beta}\right) + Li_2\left(1-\frac{\hat{p}\cdot\hat{k}}{1-\beta}\right)\right]$$

using

$$\hat{p} = \frac{p}{p^0}, \quad \hat{k} = \frac{k}{k^0}, \quad \beta = \frac{|\vec{k}|}{k_0}.$$

⟨*virtual: procedures*⟩+≡

```
function compute_eikonal_factor (p_born, massive, i, j, s_o_Q2) result (I_ij)
  real(default) :: I_ij
  type(vector4_t), intent(in), dimension(:) :: p_born
  logical, dimension(:), intent(in) :: massive
  integer, intent(in) :: i, j
  real(default), intent(in) :: s_o_Q2
  if (massive(i) .and. massive(j)) then
     I_ij = compute_Imm (p_born(i), p_born(j), s_o_Q2)
  else if (.not. massive(i) .and. massive(j)) then
     I_ij = compute_I0m (p_born(i), p_born(j), s_o_Q2)
  else if (massive(i) .and. .not. massive(j)) then
     I_ij = compute_I0m (p_born(j), p_born(i), s_o_Q2)
  else
     I_ij = compute_I00 (p_born(i), p_born(j), s_o_Q2)
  end if
end function compute_eikonal_factor

function compute_I00 (pi, pj, s_o_Q2) result (I)
  type(vector4_t), intent(in) :: pi, pj
  real(default), intent(in) :: s_o_Q2
  real(default) :: I
  real(default) :: Ei, Ej
  real(default) :: pij, Eij
  real(default) :: s1, s2, s3, s4, s5
  real(default) :: arglog
  real(default), parameter :: tiny_value = epsilon(1.0)
  s1 = 0; s2 = 0; s3 = 0; s4 = 0; s5 = 0
  Ei = pi%p(0); Ej = pj%p(0)
  pij = pi * pj; Eij = Ei * Ej
  s1 = 0.5 * log(s_o_Q2)**2
  s2 = log(s_o_Q2) * log(pij / (two * Eij))
  s3 = Li2 (pij / (two * Eij))
  s4 = 0.5 * log (pij / (two * Eij))**2
  arglog = one - pij / (2*Eij)
  if (arglog > tiny_value) then
    s5 = log(arglog) * log(pij / (two * Eij))
  else
    s5 = 0
```

```
    end if
    I = s1 + s2 - s3 + s4 - s5
  end function compute_I00

  function compute_I0m (ki, kj, s_o_Q2) result (I)
    type(vector4_t), intent(in) :: ki, kj
    real(default), intent(in) :: s_o_Q2
    real(default) :: I
    real(default) :: logsomu
    real(default) :: s1, s2, s3
    s1 = 0; s2 = 0; s3 = 0
    logsomu = log(s_o_Q2)
    s1 = 0.5 * (0.5 * logsomu**2 - pi**2 / 6)
    s2 = 0.5 * I_0m_0 (ki, kj) * logsomu
    s3 = 0.5 * I_0m_eps (ki, kj)
    I = s1 + s2 - s3
  end function compute_I0m

  function compute_Imm (pi, pj, s_o_Q2) result (I)
    type(vector4_t), intent(in) :: pi, pj
    real(default), intent(in) :: s_o_Q2
    real(default) :: I
    real(default) :: s1, s2
    s1 = 0.5 * log(s_o_Q2) * I_mm_0(pi, pj)
    s2 = 0.5 * I_mm_eps(pi, pj)
    I = s1 - s2
  end function compute_Imm
```

⟨*virtual: procedures*⟩+≡

```
  function I_m_eps (p) result (I)
    type(vector4_t), intent(in) :: p
    real(default) :: I
    real(default) :: beta
    beta = space_part_norm (p)/p%p(0)
    if (beta < tiny_07) then
       I = four * (one + beta**2/3 + beta**4/5 + beta**6/7)
    else
       I = two * log((one + beta) / (one - beta)) / beta
    end if
  end function I_m_eps
```

For $p^2 = 0$ and $k^2 \neq 0$, this computes the expression

$$I_\epsilon(p,k) = -2\left[\frac{1}{4}\log^2\frac{1-\beta}{1+\beta} + \log\frac{\hat{p}\cdot k}{1+\beta}\log\hat{p}\cdot k1-\beta + Li_2\left(1-\frac{\hat{p}\cdot k}{1+\beta}\right) + Li_2\left(1-\frac{\hat{p}\cdot k}{1-\beta}\right)\right],$$

with $\hat{p} = \frac{p}{p^0}$ and $\beta = \frac{|\vec{k}|}{k^0}$.

⟨*virtual: procedures*⟩+≡

```
  function I_0m_eps (p, k) result (I)
    type(vector4_t), intent(in) :: p, k
    real(default) :: I
    type(vector4_t) :: pp, kp
    real(default) :: beta
```

```
    pp = p / p%p(0); kp = k / k%p(0)

    beta = sqrt (one - kp*kp)
    I = -2*(log((one - beta) / (one + beta))**2/4 + log((pp*kp) / (one + beta))*log((pp*kp) / (one
        + Li2(one - (pp*kp) / (one + beta)) + Li2(one - (pp*kp) / (one - beta)))
  end function I_0m_eps
```

For $p^2 = 0$ and $k^2 \neq 0$, computes the expression

$$I_0(p,k) = \log \frac{(\hat{p} \cdot \hat{k})^2}{\hat{k}^2}$$

⟨*virtual: procedures*⟩+≡

```
  function I_0m_0 (p, k) result (I)
    type(vector4_t), intent(in) :: p, k
    real(default) :: I
    type(vector4_t) :: pp, kp

    pp = p / p%p(0); kp = k / k%p(0)
    I = log((pp*kp)**2 / kp**2)
  end function I_0m_0
```

For $k_1^2 \neq 0$ and $k_2^2 \neq 0$, computes the expression

$$I_\epsilon(k_1,k_2) = [K(z_2) - K(z_1)]\,\frac{1 - \vec{\beta}_1 \cdot \vec{\beta}_2}{\sqrt{a(1-b)}},$$

where $\vec{\beta}_i = \frac{\vec{k}_i}{k_i^0}$. Further

$$\begin{aligned}
a &= \beta_1^2 + \beta_2^2 - 2\vec{\beta}_1 \cdot \vec{\beta}_2, \\
b &= \frac{\beta_1^2\beta_2^2 - \left(\vec{\beta}_1 \cdot \vec{\beta}_2\right)^2}{a}, \\
c &= \sqrt{\frac{b}{4a}}, \\
K(z) &= -\frac{1}{2}\log^2\frac{(z-z_-)(z_+-z)}{(z_++z)(z_-+z)} - 2Li_2\left(\frac{2z_-(z_+-z)}{(z_+-z_-)(z_-+z)}\right) \\
&\quad - 2Li_2\left(-\frac{2z_+(z_-+z)}{(z_+-z_-)(z_+-z)}\right), \\
z_+ &= \frac{1+\sqrt{1-b}}{\sqrt{b}}, \\
z_- &= \frac{1-\sqrt{1-b}}{\sqrt{b}}, \\
x_1 &= \frac{\beta_1^2 - \vec{\beta}_1 \cdot \vec{\beta}_2}{a}, \\
x_2 &= 1 - x_1 z_1 \qquad\qquad = \frac{\sqrt{x_1^2+4c^2} - x_1}{2c}, \\
z_2 &= \frac{\sqrt{x_2^2+4c^2} + x_2}{2c}.
\end{aligned}$$

⟨*virtual: procedures*⟩+≡

```
function I_mm_eps (p1, p2) result (I)
  type(vector4_t), intent(in) :: p1, p2
  real(default) :: I
  type(vector3_t) :: beta1, beta2
  real(default) :: a, b, b2
  real(default) :: zp, zm, z1, z2, x1, x2
  real(default) :: zmb, z1b
  real(default) :: K1, K2

  beta1 = space_part (p1) / energy(p1)
  beta2 = space_part (p2) / energy(p2)
  a = beta1**2 + beta2**2 - 2 * beta1 * beta2
  b = beta1**2 * beta2**2 - (beta1 * beta2)**2
  if (beta1**1 > beta2**1) call switch_beta (beta1, beta2)
  if (beta1 == vector3_null) then
     b2 = beta2**1
     I = (-0.5 * log ((one - b2) / (one + b2))**2 - two * Li2 (-two * b2 / (one - b2))) &
         * one / sqrt (a - b)
     return
  end if
  x1 = beta1**2 - beta1 * beta2
  x2 = beta2**2 - beta1 * beta2
  zp = sqrt (a) + sqrt (a - b)
  zm = sqrt (a) - sqrt (a - b)
  zmb = one  / zp
  z1 = sqrt (x1**2 + b) - x1
  z2 = sqrt (x2**2 + b) + x2
  z1b = one / (sqrt (x1**2 + b) + x1)
  K1 = - 0.5 * log (((z1b - zmb) * (zp - z1)) / ((zp + z1) * (z1b + zmb)))**2 &
        - two * Li2 ((two * zmb * (zp - z1)) / ((zp - zm) * (zmb + z1b))) &
        - two * Li2 ((-two * zp * (zm + z1)) / ((zp - zm) * (zp - z1)))
  K2 = - 0.5 * log ((( z2 - zm) * (zp - z2)) / ((zp + z2) * (z2 + zm)))**2 &
        - two * Li2 ((two * zm * (zp - z2)) / ((zp - zm) * (zm + z2))) &
        - two * Li2 ((-two * zp * (zm + z2)) / ((zp - zm) * (zp - z2)))
  I = (K2 - K1) * (one - beta1 * beta2) / sqrt (a - b)
contains
  subroutine switch_beta (beta1, beta2)
    type(vector3_t), intent(inout) :: beta1, beta2
    type(vector3_t) :: beta_tmp
    beta_tmp = beta1
    beta1 = beta2
    beta2 = beta_tmp
  end subroutine switch_beta
end function I_mm_eps
```

For $k_1^2 \neq 0$ and $k_2^2 \neq 0$,computes the expression

$$I_0(k_1, k_2) = \frac{1}{\beta} \log \frac{1+\beta}{1-\beta}, \quad \beta = \sqrt{1 - \frac{k_1^2 k_2^2}{(k_1 \cdot k_2)^2}}$$

⟨*virtual: procedures*⟩+≡

```
function I_mm_0 (k1, k2) result (I)
  type(vector4_t), intent(in) :: k1, k2
```

```
    real(default) :: I
    real(default) :: beta
    beta = sqrt (one - k1**2 * k2**2 / (k1 * k2)**2)
    I = log ((one + beta) / (one - beta)) / beta
  end function I_mm_0
```

⟨*virtual: virtual: TBP*⟩+≡

```
  procedure :: final => virtual_final
```

⟨*virtual: procedures*⟩+≡

```
  subroutine virtual_final (virtual)
    class(virtual_t), intent(inout) :: virtual
    if (allocated (virtual%gamma_0)) deallocate (virtual%gamma_0)
    if (allocated (virtual%gamma_p)) deallocate (virtual%gamma_p)
    if (allocated (virtual%c_flv)) deallocate (virtual%c_flv)
    if (allocated (virtual%n_is_neutrinos)) deallocate (virtual%n_is_neutrinos)
  end subroutine virtual_final
```

27.5 Real Subtraction

⟨real_subtraction.f90⟩≡
⟨*File header*⟩

```
module real_subtraction
```

⟨*Use kinds with double*⟩
⟨*Use strings*⟩

```
  use io_units
  use system_dependencies, only: LHAPDF6_AVAILABLE
  use format_defs, only: FMT_15
  use string_utils
  use constants
  use numeric_utils
  use diagnostics
  use pdg_arrays
  use models
  use physics_defs
  use sm_physics
  use sf_lhapdf
  use pdf
  use lorentz
  use flavors
  use phs_fks, only: real_kinematics_t, isr_kinematics_t
  use phs_fks, only: I_PLUS, I_MINUS
  use phs_fks, only: phs_point_set_t
  use ttv_formfactors, only: m1s_to_mpole

  use fks_regions
  use nlo_data
```

⟨*Standard module head*⟩

⟨*real subtraction: public*⟩

⟨*real subtraction: parameters*⟩

⟨*real subtraction: types*⟩

⟨*real subtraction: interfaces*⟩

```
contains
```

⟨*real subtraction: procedures*⟩

```
end module real_subtraction
```

Soft subtraction terms

⟨*real subtraction: parameters*⟩≡

```
  integer, parameter, public :: INTEGRATION = 0
  integer, parameter, public :: FIXED_ORDER_EVENTS = 1
  integer, parameter, public :: POWHEG = 2
```

⟨*real subtraction: public*⟩≡

```
public :: this_purpose
```

⟨*real subtraction: procedures*⟩≡

```
function this_purpose (purpose)
  type(string_t) :: this_purpose
  integer, intent(in) :: purpose
  select case (purpose)
  case (INTEGRATION)
     this_purpose = var_str ("Integration")
  case (FIXED_ORDER_EVENTS)
     this_purpose = var_str ("Fixed order NLO events")
  case (POWHEG)
     this_purpose = var_str ("Powheg events")
  case default
     this_purpose = var_str ("Undefined!")
  end select
end function this_purpose
```

In the soft limit, the real matrix element behaves as

$$\mathcal{R}_{\rm soft} = 4\pi\alpha_s \left[\sum_{i\neq j} \mathcal{B}_{ij} \frac{k_i \cdot k_j}{(k_i \cdot k)(k_j \cdot k)} - \mathcal{B} \sum_i \frac{k_i^2}{(k_i \cdot k)^2} C_i \right] ,$$

where k denotes the momentum of the emitted parton. The quantity $\mathcal{B}_{ij}$ is called the color-correlated Born matrix element defined as

$$\mathcal{B}_{ij} = \frac{1}{2s} \sum_{\substack{colors \\ spins}} \mathcal{M}_{\{c_k\}} \left(\mathcal{M}^\dagger_{\{c_k\}} \right)_{\substack{c_i \to c_i' \\ c_j \to c_j'}} T^a_{c_i,c_i'} T^a_{c_j,c_j'} .$$

⟨*real subtraction: types*⟩≡

```
type :: soft_subtraction_t
  type(region_data_t), pointer :: reg_data => null ()
  real(default), dimension(:,:), allocatable :: momentum_matrix
  logical :: use_resonance_mappings = .false.
  type(vector4_t) :: p_soft = vector4_null
  logical :: use_internal_color_correlations = .true.
  logical :: use_internal_spin_correlations = .false.
  logical :: xi2_expanded = .true.
  integer :: factorization_mode = NO_FACTORIZATION
contains
⟨real subtraction: soft sub: TBP⟩
end type soft_subtraction_t
```

⟨*real subtraction: soft sub: TBP*⟩≡

```
procedure :: init => soft_subtraction_init
```

⟨*real subtraction: procedures*⟩+≡

```
subroutine soft_subtraction_init (sub_soft, reg_data)
  class(soft_subtraction_t), intent(inout) :: sub_soft
  type(region_data_t), intent(in), target :: reg_data
```

```
    sub_soft%reg_data => reg_data
    allocate (sub_soft%momentum_matrix (reg_data%n_legs_born, &
         reg_data%n_legs_born))
  end subroutine soft_subtraction_init
```

⟨*real subtraction: soft sub: TBP*⟩+≡

```
  procedure :: requires_boost => soft_subtraction_requires_boost
```

⟨*real subtraction: procedures*⟩+≡

```
  function soft_subtraction_requires_boost (sub_soft, sqrts) result (requires_boost)
    logical :: requires_boost
    class(soft_subtraction_t), intent(in) :: sub_soft
    real(default), intent(in) :: sqrts
    real(default) :: mtop
    logical :: above_threshold
    if (sub_soft%factorization_mode == FACTORIZATION_THRESHOLD) then
       mtop = m1s_to_mpole (sqrts)
       above_threshold = sqrts**2 - four * mtop**2 > zero
    else
       above_threshold = .false.
    end if
    requires_boost = sub_soft%use_resonance_mappings .or. above_threshold
  end function soft_subtraction_requires_boost
```

The treatment of the momentum k follows the discussion about the soft limit of the partition functions (ref????). The parton momentum is pulled out, $k = E\hat{k}$. In fact, we will substitute $\hat{k}$ for k throughout the code, because the energy will factor out of the equation when the soft $\mathcal{S}$-function is multiplied. The soft momentum is a unit vector, because $k^2 = \left(k^0\right)^2 - \left(k^0\right)^2 \hat{\vec{k}}^2 = 0$.

The soft momentum is constructed by first creating a unit vector parallel to the emitter's Born momentum. This unit vector is then rotated about the corresponding angles y and ϕ.

⟨*real subtraction: soft sub: TBP*⟩+≡

```
  procedure :: create_softvec_fsr => soft_subtraction_create_softvec_fsr
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine soft_subtraction_create_softvec_fsr &
     (sub_soft, p_born, y, phi, emitter, xi_ref_momentum)
    class(soft_subtraction_t), intent(inout) :: sub_soft
    type(vector4_t), intent(in), dimension(:) :: p_born
    real(default), intent(in) :: y, phi
    integer, intent(in) :: emitter
    type(vector4_t), intent(in) :: xi_ref_momentum
    type(vector3_t) :: dir
    type(vector4_t) :: p_em
    type(lorentz_transformation_t) :: rot
    type(lorentz_transformation_t) :: boost_to_rest_frame
    logical :: requires_boost
    associate (p_soft => sub_soft%p_soft)
       p_soft%p(0) = one
       requires_boost = sub_soft%requires_boost (two * p_born(1)%p(0))
       if (requires_boost) then
          boost_to_rest_frame = inverse (boost (xi_ref_momentum, xi_ref_momentum**1))
```

```
        p_em = boost_to_rest_frame * p_born(emitter)
      else
        p_em = p_born(emitter)
      end if
      p_soft%p(1:3) = p_em%p(1:3) / space_part_norm (p_em)
      dir = create_orthogonal (space_part (p_em))
      rot = rotation (y, sqrt(one - y**2), dir)
      p_soft = rot * p_soft
      if (.not. vanishes (phi)) then
        dir = space_part (p_em) / space_part_norm (p_em)
        rot = rotation (cos(phi), sin(phi), dir)
        p_soft = rot * p_soft
      end if
      if (requires_boost) p_soft = inverse (boost_to_rest_frame) * p_soft
    end associate
  end subroutine soft_subtraction_create_softvec_fsr
```

For initial-state emissions, the soft vector is just a unit vector with the same direction as the radiated particle.

⟨*real subtraction: soft sub: TBP*⟩+≡

```
  procedure :: create_softvec_isr => soft_subtraction_create_softvec_isr
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine soft_subtraction_create_softvec_isr (sub_soft, y, phi)
    class(soft_subtraction_t), intent(inout) :: sub_soft
    real(default), intent(in) :: y, phi
    real(default) :: sin_theta
    sin_theta = sqrt(one - y**2)
    associate (p => sub_soft%p_soft%p)
      p(0) = one
      p(1) = sin_theta * sin(phi)
      p(2) = sin_theta * cos(phi)
      p(3) = y
    end associate
  end subroutine soft_subtraction_create_softvec_isr
```

The soft vector for the real mismatch is basically the same as for usual FSR, except for the scaling with the total gluon energy. Moreover, the resulting vector is rotated into the frame where the 3-axis points along the direction of the emitter. This is necessary because in the collinear limit, the approximation

$$k_i = \frac{k_i^0}{\bar{k}_j^0}\bar{k}_j = \frac{\xi\sqrt{s}}{2\bar{k}_j^0}\bar{k}_j$$

is used. The collinear limit is not included in the soft mismatch yet, but we keep the rotation for future usage here already (the performance loss is negligible).

⟨*real subtraction: soft sub: TBP*⟩+≡

```
  procedure :: create_softvec_mismatch => &
     soft_subtraction_create_softvec_mismatch
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine soft_subtraction_create_softvec_mismatch (sub_soft, E, y, phi, p_em)
    class(soft_subtraction_t), intent(inout) :: sub_soft
```

```
    real(default), intent(in) :: E, phi, y
    type(vector4_t), intent(in) :: p_em
    real(default) :: sin_theta
    type(lorentz_transformation_t) :: rot_em_off_3_axis
    sin_theta = sqrt (one - y**2)
    associate (p => sub_soft%p_soft%p)
       p(0) = E
       p(1) = E * sin_theta * sin(phi)
       p(2) = E * sin_theta * cos(phi)
       p(3) = E * y
    end associate
    rot_em_off_3_axis = rotation_to_2nd (3, space_part (p_em))
    sub_soft%p_soft = rot_em_off_3_axis * sub_soft%p_soft
  end subroutine soft_subtraction_create_softvec_mismatch
```

Computation of the soft limit of R_α. Note that what we are actually integrating (in the case of final-state radiation) is the quantity $f(0,y)/\xi$, where

$$f(\xi,y) = \frac{J(\xi,y,\phi)}{\xi}\xi^2 R_\alpha.$$

J/ξ is computed by the phase space generator. The additional factor of ξ^{-1} is supplied in the `evaluate_region_fsr`-routine. Thus, we are left with a factor of ξ^2. A look on the expression for the soft limit of R_α below reveals that we are factoring out the gluon energy E_i in the denominator. Therefore, we have a factor $\xi^2/E_i^2 = q^2/4$.
Note that the same routine is used also for the computation of the soft mismatch. There, the gluon energy is not factored out from the soft vector, so that we are left with the ξ^2-factor, which will eventually be cancelled out again. So, we just multiply with 1. Both cases are distinguished by the flag `xi2_expanded`.

⟨*real subtraction: soft sub: TBP*⟩+≡

```
  procedure :: compute => soft_subtraction_compute
```

⟨*real subtraction: procedures*⟩+≡

```
  function soft_subtraction_compute (sub_soft, p_born, &
     born_ij, y, q2, alpha_s, alr, emitter, i_res) result (sqme)
    real(default) :: sqme
    class(soft_subtraction_t), intent(inout) :: sub_soft
    type(vector4_t), intent(in), dimension(:) :: p_born
    real(default), intent(in), dimension(:,:) :: born_ij
    real(default), intent(in) :: y
    real(default), intent(in) :: q2, alpha_s
    integer, intent(in) :: alr, emitter, i_res
    real(default) :: s_alpha_soft
    real(default) :: kb
    real(default) :: xi2_factor

    if (.not. vector_set_is_cms (p_born)) then
       call vector4_write_set (p_born, show_mass = .true., &
          check_conservation = .true.)
       call msg_fatal ("Soft subtraction: phase space point must be in CMS")
    end if
    if (debug2_active (D_SUBTRACTION)) then
```

```
      print *, 'Compute soft subtraction using alpha_s = ', alpha_s
   end if

   s_alpha_soft = sub_soft%reg_data%get_svalue_soft (p_born, &
        sub_soft%p_soft, alr, emitter, i_res)
   if (s_alpha_soft > one + tiny_07) call msg_fatal ("s_alpha_soft > 1!")
   if (debug_active (D_SUBTRACTION)) &
        call msg_print_color ('s_alpha_soft', s_alpha_soft, COL_YELLOW)
   select case (sub_soft%factorization_mode)
   case (NO_FACTORIZATION)
      kb = sub_soft%evaluate_factorization_default (p_born, born_ij)
   case (FACTORIZATION_THRESHOLD)
      kb = sub_soft%evaluate_factorization_threshold (thr_leg(emitter), p_born, born_ij)
   end select
   call msg_debug2 (D_SUBTRACTION, 'KB', kb)
   sqme = four * pi * alpha_s * s_alpha_soft * kb
   if (sub_soft%xi2_expanded) then
      xi2_factor = four / q2
   else
      xi2_factor = one
   end if
   if (emitter <= sub_soft%reg_data%n_in) then
      sqme = xi2_factor * (one - y**2) * sqme
   else
      sqme = xi2_factor * (one - y) * sqme
   end if
end function soft_subtraction_compute
```

We loop over all external legs and do not take care to leave out non-colored ones because `born_ij` is constructed in such a way that it is only non-zero for colored entries.

⟨*real subtraction: soft sub: TBP*⟩+≡

```
procedure :: evaluate_factorization_default => &
   soft_subtraction_evaluate_factorization_default
```

⟨*real subtraction: procedures*⟩+≡

```
function soft_subtraction_evaluate_factorization_default &
   (sub_soft, p, born_ij) result (kb)
  real(default) :: kb
  class(soft_subtraction_t), intent(inout) :: sub_soft
  type(vector4_t), intent(in), dimension(:) :: p
  real(default), intent(in), dimension(:,:) :: born_ij
  integer :: i, j
  kb = zero
  call sub_soft%compute_momentum_matrix (p)
  do i = 1, size (p)
     do j = 1, size (p)
        kb = kb + sub_soft%momentum_matrix (i, j) * born_ij (i, j)
     end do
  end do
end function soft_subtraction_evaluate_factorization_default
```

We have to multiply this with $\xi^2(1-y)$. Further, when applying the soft $\mathcal{S}$-function, the energy of the radiated particle is factored out. Thus we have

$\xi^2/E_{em}^2(1-y) = 4/q_0^2(1-y)$. Computes the quantity $\mathcal{K}_{ij} = \frac{k_i \cdot k_j}{(k_i \cdot k)(k_j \cdot k)}$.

⟨*real subtraction: soft sub: TBP*⟩+≡

```
procedure :: compute_momentum_matrix => &
     soft_subtraction_compute_momentum_matrix
```

⟨*real subtraction: procedures*⟩+≡

```
subroutine soft_subtraction_compute_momentum_matrix &
     (sub_soft, p_born)
  class(soft_subtraction_t), intent(inout) :: sub_soft
  type(vector4_t), intent(in), dimension(:) :: p_born
  real(default) :: num, deno1, deno2
  integer :: i, j
  do i = 1, sub_soft%reg_data%n_legs_born
     do j = 1, sub_soft%reg_data%n_legs_born
        if (i <= j) then
           num = p_born(i) * p_born(j)
           deno1 = p_born(i) * sub_soft%p_soft
           deno2 = p_born(j) * sub_soft%p_soft
           sub_soft%momentum_matrix(i, j) = num / (deno1 * deno2)
        else
           !!! momentum matrix is symmetric.
          sub_soft%momentum_matrix(i, j) = sub_soft%momentum_matrix(j, i)
        end if
     end do
  end do
end subroutine soft_subtraction_compute_momentum_matrix
```

⟨*real subtraction: soft sub: TBP*⟩+≡

```
procedure :: evaluate_factorization_threshold => &
   soft_subtraction_evaluate_factorization_threshold
```

⟨*real subtraction: procedures*⟩+≡

```
function soft_subtraction_evaluate_factorization_threshold &
   (sub_soft, leg, p_born, born_ij) result (kb)
  real(default) :: kb
  class(soft_subtraction_t), intent(inout) :: sub_soft
  integer, intent(in) :: leg
  type(vector4_t), intent(in), dimension(:) :: p_born
  real(default), intent(in), dimension(:,:) :: born_ij
  type(vector4_t), dimension(4) :: p
  p = get_threshold_momenta (p_born)
  kb = evaluate_leg_pair (ASSOCIATED_LEG_PAIR (leg))
  if (debug2_active (D_SUBTRACTION))  call show_debug ()

contains

  function evaluate_leg_pair (i_start) result (kbb)
    real(default) :: kbb
    integer, intent(in) :: i_start
    integer :: i1, i2
    real(default) :: numerator, deno1, deno2
    kbb = zero
    do i1 = i_start, i_start + 1
       do i2 = i_start, i_start + 1
```

```
            numerator = p(i1) * p(i2)
            deno1 = p(i1) * sub_soft%p_soft
            deno2 = p(i2) * sub_soft%p_soft
            kbb = kbb +  numerator * born_ij (i1, i2) / deno1 / deno2
         end do
      end do
      if (debug2_active (D_SUBTRACTION)) then
         do i1 = i_start, i_start + 1
            do i2 = i_start, i_start + 1
               call msg_print_color('i1', i1, COL_PEACH)
               call msg_print_color('i2', i2, COL_PEACH)
               call msg_print_color('born_ij (i1,i2)', born_ij (i1,i2), COL_PINK)
               print *, 'Top momentum: ', p(1)%p
            end do
         end do
      end if
    end function evaluate_leg_pair

    subroutine show_debug ()
      integer :: i
      call msg_print_color ('soft_subtraction_evaluate_factorization_threshold', COL_GREEN)
      do i = 1, 4
         print *, 'sqrt(p(i)**2) =    ', sqrt(p(i)**2)
      end do
    end subroutine show_debug

  end function soft_subtraction_evaluate_factorization_threshold
```

⟨*real subtraction: soft sub: TBP*⟩+≡

```
  procedure :: i_xi_ref => soft_subtraction_i_xi_ref
```

⟨*real subtraction: procedures*⟩+≡

```
  function soft_subtraction_i_xi_ref (sub_soft, alr, i_phs) result (i_xi_ref)
    integer :: i_xi_ref
    class(soft_subtraction_t), intent(in) :: sub_soft
    integer, intent(in) :: alr, i_phs
    if (sub_soft%use_resonance_mappings) then
       i_xi_ref = sub_soft%reg_data%alr_to_i_contributor (alr)
    else if (sub_soft%factorization_mode == FACTORIZATION_THRESHOLD) then
       i_xi_ref = i_phs
    else
       i_xi_ref = 1
    end if
  end function soft_subtraction_i_xi_ref
```

⟨*real subtraction: soft sub: TBP*⟩+≡

```
  procedure :: final => soft_subtraction_final
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine soft_subtraction_final (sub_soft)
    class(soft_subtraction_t), intent(inout) :: sub_soft
    if (associated (sub_soft%reg_data)) nullify (sub_soft%reg_data)
    if (allocated (sub_soft%momentum_matrix)) deallocate (sub_soft%momentum_matrix)
  end subroutine soft_subtraction_final
```

27.5.1 Soft mismatch

⟨*real subtraction: public*⟩+≡
```
public :: soft_mismatch_t
```

⟨*real subtraction: types*⟩+≡
```
type :: soft_mismatch_t
  type(region_data_t), pointer :: reg_data => null ()
  real(default), dimension(:), allocatable :: sqme_born
  real(default), dimension(:,:,:), allocatable :: sqme_born_cc
  type(real_kinematics_t), pointer :: real_kinematics => null ()
  type(soft_subtraction_t) :: sub_soft
contains
⟨real subtraction: soft mismatch: TBP⟩
end type soft_mismatch_t
```

⟨*real subtraction: soft mismatch: TBP*⟩≡
```
procedure :: init => soft_mismatch_init
```

⟨*real subtraction: procedures*⟩+≡
```
subroutine soft_mismatch_init (soft_mismatch, reg_data, &
    real_kinematics, factorization_mode)
  class(soft_mismatch_t), intent(inout) :: soft_mismatch
  type(region_data_t), intent(in), target :: reg_data
  type(real_kinematics_t), intent(in), target :: real_kinematics
  integer, intent(in) :: factorization_mode
  soft_mismatch%reg_data => reg_data
  allocate (soft_mismatch%sqme_born (reg_data%n_flv_born))
  allocate (soft_mismatch%sqme_born_cc (reg_data%n_legs_born, &
      reg_data%n_legs_born, reg_data%n_flv_born))
  call soft_mismatch%sub_soft%init (reg_data)
  soft_mismatch%sub_soft%xi2_expanded = .false.
  soft_mismatch%real_kinematics => real_kinematics
  soft_mismatch%sub_soft%factorization_mode = factorization_mode
end subroutine soft_mismatch_init
```

Main routine to compute the soft mismatch. Loops over all singular regions. There, it first creates the soft vector, then the necessary soft real matrix element. These inputs are then used to get the numerical value of the soft mismatch.

⟨*real subtraction: soft mismatch: TBP*⟩+≡
```
procedure :: evaluate => soft_mismatch_evaluate
```

⟨*real subtraction: procedures*⟩+≡
```
function soft_mismatch_evaluate (soft_mismatch, alpha_s) result (sqme_mismatch)
  real(default) :: sqme_mismatch
  class(soft_mismatch_t), intent(inout) :: soft_mismatch
  real(default), intent(in) :: alpha_s
  integer :: alr, i_born, emitter, i_res, i_phs, i_con
  real(default) :: xi, y, q2, s
  real(default) :: E_gluon
  type(vector4_t) :: p_em
  real(default) :: sqme_alr, sqme_soft
  type(vector4_t), dimension(:), allocatable :: p_born
  sqme_mismatch = zero
  associate (real_kinematics => soft_mismatch%real_kinematics)
```

```
xi = real_kinematics%xi_mismatch
y = real_kinematics%y_mismatch
s = real_kinematics%cms_energy2
E_gluon = sqrt (s) * xi / two

if (debug_active (D_MISMATCH)) then
   print *, 'Evaluating soft mismatch: '
   print *, 'Phase space: '
   call vector4_write_set (real_kinematics%p_born_cms%get_momenta(1), &
        show_mass = .true.)
   print *, 'xi: ', xi, 'y: ', y, 's: ', s, 'E_gluon: ', E_gluon
end if

allocate (p_born (soft_mismatch%reg_data%n_legs_born))

do alr = 1, soft_mismatch%reg_data%n_regions

    i_phs = real_kinematics%alr_to_i_phs (alr)
    if (soft_mismatch%reg_data%has_pseudo_isr ()) then
       i_con = 1
       p_born = soft_mismatch%real_kinematics%p_born_onshell%get_momenta(1)
    else
       i_con = soft_mismatch%reg_data%alr_to_i_contributor (alr)
       p_born = soft_mismatch%real_kinematics%p_born_cms%get_momenta(1)
    end if
    q2 = real_kinematics%xi_ref_momenta(i_con)**2
    emitter = soft_mismatch%reg_data%regions(alr)%emitter
    p_em = p_born (emitter)
    i_res = soft_mismatch%reg_data%regions(alr)%i_res
    i_born = soft_mismatch%reg_data%regions(alr)%uborn_index

    call print_debug_alr ()

    call soft_mismatch%sub_soft%create_softvec_mismatch &
         (E_gluon, y, real_kinematics%phi, p_em)
    if (debug_active (D_MISMATCH)) &
         print *, 'Created soft vector: ', soft_mismatch%sub_soft%p_soft%p

    select type (fks_mapping => soft_mismatch%reg_data%fks_mapping)
    type is (fks_mapping_resonances_t)
       call fks_mapping%set_resonance_momentum &
          (real_kinematics%xi_ref_momenta(i_con))
    end select

    sqme_soft = soft_mismatch%sub_soft%compute &
         (p_born, soft_mismatch%sqme_born_cc(:,:,i_born), y, &
         q2, alpha_s, alr, emitter, i_res)

    sqme_alr = soft_mismatch%compute (alr, xi, y, p_em, &
         real_kinematics%xi_ref_momenta(i_con), soft_mismatch%sub_soft%p_soft, &
         soft_mismatch%sqme_born(i_born), sqme_soft, &
         alpha_s, s)

    call msg_debug (D_MISMATCH, 'sqme_alr: ', sqme_alr)
```

```
          sqme_mismatch = sqme_mismatch + sqme_alr

       end do
    end associate
  contains
    subroutine print_debug_alr ()
      if (debug_active (D_MISMATCH)) then
         print *, 'alr: ', alr
         print *, 'i_phs: ', i_phs, 'i_con: ', i_con, 'i_res: ', i_res
         print *, 'emitter: ', emitter, 'i_born: ', i_born
         print *, 'emitter momentum: ', p_em%p
         print *, 'resonance momentum: ', &
            soft_mismatch%real_kinematics%xi_ref_momenta(i_con)%p
         print *, 'q2: ', q2
      end if
    end subroutine print_debug_alr
  end function soft_mismatch_evaluate
```

Computes the soft mismatch in a given α_r,

$$I_{s+,\alpha_r} = \int d\Phi_B \int_0^\infty d\xi \int_{-1}^1 dy \int_0^{2\pi} d\phi \frac{s\xi}{(4\pi)^3} \times \left\{ \tilde{R}_{\alpha_r} \left(e^{-\frac{2k_\gamma \cdot k_{res}}{k_{res}}^2} - e^{-\xi} \right) - \frac{32\pi\alpha_s C_{em}}{s\xi^2} B_{f_b(\alpha_r)} (1-y)^{-1} \left[e^{-\frac{2\tilde{k}_{em} \cdot k_{res}}{k_{res}^2} \frac{k_\gamma^0}{k_{em}^0}} - e^{-\xi} \right] \right\}.$$

⟨*real subtraction: soft mismatch: TBP*⟩+≡

```
procedure :: compute => soft_mismatch_compute
```

⟨*real subtraction: procedures*⟩+≡

```
function soft_mismatch_compute (soft_mismatch, alr, xi, y, p_em, p_res, p_soft, &
   sqme_born, sqme_soft, alpha_s, s) result (sqme_mismatch)
  real(default) :: sqme_mismatch
  class(soft_mismatch_t), intent(in) :: soft_mismatch
  integer, intent(in) :: alr
  real(default), intent(in) :: xi, y
  type(vector4_t), intent(in) :: p_em, p_res, p_soft
  real(default), intent(in) :: sqme_born, sqme_soft
  real(default), intent(in) :: alpha_s, s
  real(default) :: q2, expo, sm1, sm2, jacobian

  q2 = p_res**2
  expo = - two * p_soft * p_res / q2
  !!! Divide by 1 - y to factor out the corresponding
  !!! factor in the soft matrix element
  sm1 = sqme_soft / (one - y) * ( exp(expo) - exp(- xi) )
  call msg_debug2 (D_MISMATCH, 'sqme_soft in mismatch ', sqme_soft)

  sm2 = zero
  if (soft_mismatch%reg_data%regions(alr)%has_collinear_divergence ()) then
     expo = - two * p_em * p_res / q2 * &
        p_soft%p(0) / p_em%p(0)
     sm2 = 32 * pi * alpha_s * cf / (s * xi**2) * sqme_born * &
        ( exp(expo) - exp(- xi) ) / (one - y)
```

```
    end if

    jacobian = soft_mismatch%real_kinematics%jac_mismatch * s * xi / (8 * twopi3)
    sqme_mismatch = (sm1 - sm2) * jacobian

  end function soft_mismatch_compute
```

⟨*real subtraction: soft mismatch: TBP*⟩+≡

```
  procedure :: final => soft_mismatch_final
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine soft_mismatch_final (soft_mismatch)
    class(soft_mismatch_t), intent(inout) :: soft_mismatch
    call soft_mismatch%sub_soft%final ()
    if (associated (soft_mismatch%reg_data)) nullify (soft_mismatch%reg_data)
    if (allocated (soft_mismatch%sqme_born)) deallocate (soft_mismatch%sqme_born)
    if (allocated (soft_mismatch%sqme_born_cc)) deallocate (soft_mismatch%sqme_born_cc)
    if (associated (soft_mismatch%real_kinematics)) nullify (soft_mismatch%real_kinematics)
  end subroutine soft_mismatch_final
```

27.5.2 Collinear and soft-collinear subtraction terms

This data type deals with the calculation of the collinear and soft-collinear contribution to the cross section.

⟨*real subtraction: types*⟩+≡

```
  type :: coll_subtraction_t
    integer :: n_in, n_alr
    logical :: use_resonance_mappings = .false.
  contains
  ⟨real subtraction: coll sub: TBP⟩
  end type coll_subtraction_t
```

⟨*real subtraction: coll sub: TBP*⟩≡

```
  procedure :: init => coll_subtraction_init
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine coll_subtraction_init (coll_sub, n_alr, n_in)
    class(coll_subtraction_t), intent(inout) :: coll_sub
    integer, intent(in) :: n_alr, n_in
    coll_sub%n_in = n_in
    coll_sub%n_alr = n_alr
  end subroutine coll_subtraction_init
```

To compute the collinear limit of $\mathcal{R}$, we follow the original FKS-paper. Here, the real amplitude is supposed to factorize in the collinear limit. considering a splitting $g \to g(i)g(j)$,

$$\mathcal{A}^{(n)}(h_i, h_j, \{h_l\} \stackrel{i\|j}{\to} g_s \sum_{d_e} \sum_{h_e} C(d_e, b, c) S_{gg}^{h_e h_i h_j}(z) \mathcal{A}_{d_e}^{(n-1)}(h_e, \{h_l\})$$

Explain quantities. Evaluating this expression leads to

$$|\mathcal{A}^{(n)}(h_i,h_j,\{h_l\})|^2 \stackrel{i\|j}{\to} |\mathcal{N}^{(n)}(h_i,h_j,\{h_l\}|^2 + \mathcal{R}(h_i,h_j,\{h_l\})|^2,$$

where $\mathcal{R}$ contains all contributions with spin-correlated amplitudes, i.e. terms like $\mathcal{A}_{d_e}^{(n-1)}(+,\{h_l\})\left(\mathcal{A}_{d_e}^{(n-1)}(-,\{h_l\})\right)^*$. Explicitly,

$$\sum_{h_i,h_j,\{h_l\}}\sum_{\{d_l\}}|\mathcal{N}^{(n)}(h_i,h_j,\{h_l\})|^2 = \frac{4\pi\alpha_s}{k_i\cdot k_j}P_{gg}(z)|\mathcal{A}^{(n-1)}|^2,$$

and

$$\sum_{h_i,h_j,\{h_l\}}\sum_{\{d_l\}}\mathcal{R}(h_i,h_j,\{h_l\}) = 32\pi\alpha_s C_A z(1-z)$$
$$\times \underbrace{Re\left\{\frac{\langle ij\rangle}{[ij]}\sum_{\{h_l\}}\sum_{d_e,\{d_l\}}\mathcal{A}_{d_e}^{(n-1)}(+,\{h_l\})\left(\mathcal{A}_{d_e}^{(n-1)}(-,\{h_l\})\right)^*\right\}}_{-\tilde{\mathcal{M}}^{(n-1)}/(2k_i\cdot k_j)}.$$

This yields

$$\mathcal{M}^{(n)} \stackrel{i\|j}{\to} \frac{4\pi\alpha_s}{k_i\cdot k_j}P_{gg}(z)\mathcal{M}^{(n-1)} - \frac{16\pi\alpha_s}{k_i\cdot k_j}C_A z(1-z)\tilde{\mathcal{M}}^{(n-1)}. \tag{27.15}$$

The equivalent expression for a $g\to qq$-splitting is given by

$$\mathcal{M}^{(n)} \stackrel{i\|j}{\to} \frac{4\pi\alpha_s}{k_i\cdot k_j}P_{qg}(z)\mathcal{M}^{(n-1)} + \frac{16\pi\alpha_s}{k_i\cdot k_j}T_F z(1-z)\tilde{\mathcal{M}}^{(n-1)}. \tag{27.16}$$

The function obtains a flag to indicate whether the limit to be computed is also soft. Moreover, this consideration explains the structure of the code. In the soft limit, we find $z\to 0$ as well as $\xi\to 0$. However, the quantity z/ξ is finite, because

$$\frac{z}{\xi} = \frac{p_{rad}^0}{\bar{p}_{em}^0}\frac{q^0}{2p_{rad}^0} = \frac{q^0}{2\bar{p}_{em}^0}.$$

Thus, all expressions are written in terms of this well-behaved quantity. Recalling that there is a prefactor of $(\xi^2 z)^{-1}$, it is necessary to expand with z once.

⟨*real subtraction: coll sub: TBP*⟩+≡
```
procedure :: compute_fsr => coll_subtraction_compute_fsr
```

⟨*real subtraction: procedures*⟩+≡
```
function coll_subtraction_compute_fsr &
   (coll_sub, sregion, p_res, p_born, sqme_born, mom_times_sqme_born_sc, &
   xi, alpha_s, alr, double_fsr) result (sqme)
  real(default) :: sqme
  class(coll_subtraction_t), intent(in) :: coll_sub
  type(singular_region_t), intent(in) :: sregion
  type(vector4_t), intent(in) :: p_res
  type(vector4_t), intent(in), dimension(:) :: p_born
  real(default), intent(in) :: sqme_born, mom_times_sqme_born_sc
```

```
      real(default), intent(in) :: xi, alpha_s
      integer, intent(in) :: alr
      logical, intent(in) :: double_fsr
      real(default) :: q0, z, p0
      real(default) :: z_o_xi, onemz
      real(default) :: pggz
      integer :: nlegs, emitter
      integer :: flv_em, flv_rad

      if (.not. vector_set_is_cms (p_born)) then
         call vector4_write_set (p_born, show_mass = .true., &
              check_conservation = .true., n_in = coll_sub%n_in)
         call msg_fatal ("Collinear subtraction, FSR: Phase space point &
              &must be in CMS")
      end if

      nlegs = size (sregion%flst_real%flst)
      emitter = sregion%emitter
      flv_rad = sregion%flst_real%flst(nlegs)
      flv_em = sregion%flst_real%flst(emitter)
      q0 = p_res**1
      p0 = p_res * p_born(emitter) / q0
      !!! Here, z corresponds to 1-z in the formulas of arXiv:1002.2581;
      !!! the integrand is symmetric under this variable change
      z_o_xi = q0 / (two * p0)
      z = xi * z_o_xi; onemz = one - z
      if (is_gluon(flv_em) .and. is_gluon(flv_rad)) then
         pggz = two * CA * (xi * z / onemz + onemz / z_o_xi)
         sqme = pggz * sqme_born + four * CA * z * onemz * xi * mom_times_sqme_born_sc
      else if (is_quark(abs(flv_em)) .and. is_quark (abs(flv_rad))) then
         sqme = TR * xi * (sqme_born - four * z * onemz * mom_times_sqme_born_sc)
      else if (is_quark (abs(flv_em)) .and. is_gluon (flv_rad)) then
         sqme = sqme_born * CF * (one + onemz**2) / z_o_xi
      else
         sqme = zero
      end if
      sqme = sqme / (p0**2 * onemz * z_o_xi)
      sqme = sqme * four * pi * alpha_s
      if (double_fsr) sqme = sqme * onemz

    end function coll_subtraction_compute_fsr
```

⟨*real subtraction: coll sub: TBP*⟩+≡

```
    procedure :: compute_isr => coll_subtraction_compute_isr
```

⟨*real subtraction: procedures*⟩+≡

```
    function coll_subtraction_compute_isr &
       (coll_sub, sregion, p_born, sqme_born, sqme_born_sc, &
       xi, alpha_s, alr, isr_mode) result (sqme)
      real(default) :: sqme
      class(coll_subtraction_t), intent(in) :: coll_sub
      type(singular_region_t), intent(in) :: sregion
      type(vector4_t), intent(in), dimension(:) :: p_born
      real(default), intent(in) :: sqme_born
```

```
    real(default), intent(in) :: sqme_born_sc
    real(default), intent(in) :: xi, alpha_s
    integer, intent(in) :: alr, isr_mode
    real(default) :: z, onemz
    real(default) :: p02
    integer :: flv_em, flv_rad
    integer :: nlegs

    if (vector_set_is_cms (p_born)) then
       call vector4_write_set (p_born, show_mass = .true., &
          check_conservation = .true.)
       call msg_fatal ("Collinear subtraction, ISR: Phase space point &
          &must be in lab frame")
    end if

    nlegs = size (sregion%flst_real%flst)
    flv_rad = sregion%flst_real%flst(nlegs)
    flv_em = sregion%flst_real%flst(isr_mode)
    !!! No need to pay attention to n_in = 1, because this case always has a
    !!! massive initial-state particle and thus no collinear divergence.
    p02 = p_born(1)%p(0) * p_born(2)%p(0) / two
    z = one - xi; onemz = xi

    if (is_quark(flv_em) .and. is_gluon(flv_rad)) then
       sqme = CF * (one + z**2) * sqme_born
    else if (is_gluon(flv_em) .and. is_quark (flv_rad)) then
       sqme = TR * (z**2 + onemz**2) * onemz * sqme_born
    else
       sqme = zero
    end if
    sqme = sqme * z / p02
    sqme = sqme * four * pi * alpha_s
  end function coll_subtraction_compute_isr
```

⟨*real subtraction: coll sub: TBP*⟩+≡

```
  procedure :: final => coll_subtraction_final
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine coll_subtraction_final (sub_coll)
    class(coll_subtraction_t), intent(inout) :: sub_coll
    sub_coll%use_resonance_mappings = .false.
  end subroutine coll_subtraction_final
```

27.5.3 Real Subtraction

⟨*real subtraction: public*⟩+≡

```
  public :: real_subtraction_t
```

⟨*real subtraction: types*⟩+≡

```
  type :: real_subtraction_t
    type(region_data_t), pointer :: reg_data => null ()
    type(pdf_data_t) :: pdf_data
    type(real_kinematics_t), pointer :: real_kinematics => null ()
```

```
    type(isr_kinematics_t), pointer :: isr_kinematics => null ()
    type(real_scales_t) :: scales
    real(default), dimension(:,:), allocatable :: sqme_real_non_sub
    real(default), dimension(:), allocatable :: sqme_born
    real(default), dimension(:,:,:), allocatable :: sqme_born_cc
    complex(default), dimension(:,:,:,:), allocatable :: sqme_born_sc
    type(soft_subtraction_t) :: sub_soft
    type(coll_subtraction_t) :: sub_coll
    logical, dimension(:), allocatable :: sc_required
    logical :: subtraction_deactivated = .false.
    integer :: purpose = INTEGRATION
    logical :: radiation_event = .true.
    logical :: subtraction_event = .false.
    type(pdf_container_t), dimension(2) :: pdf_born, pdf_scaled, pdf_scaled_coll
    integer, dimension(:), allocatable :: selected_alr
  contains
  ⟨real subtraction: real subtraction: TBP⟩
  end type real_subtraction_t
```

Initializer

⟨*real subtraction: real subtraction: TBP*⟩≡

```
  procedure :: init => real_subtraction_init
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine real_subtraction_init (rsub, reg_data)
    class(real_subtraction_t), intent(inout), target :: rsub
    type(region_data_t), intent(in), target :: reg_data
    integer :: alr, i_born
    call msg_debug (D_SUBTRACTION, "real_subtraction_init")
    call msg_debug (D_SUBTRACTION, "n_in", reg_data%n_in)
    call msg_debug (D_SUBTRACTION, "nlegs_born", reg_data%n_legs_born)
    call msg_debug (D_SUBTRACTION, "nlegs_real", reg_data%n_legs_real)
    call msg_debug (D_SUBTRACTION, "reg_data%n_regions", reg_data%n_regions)

    if (debug2_active (D_SUBTRACTION))  call reg_data%write ()
    rsub%reg_data => reg_data
    allocate (rsub%sqme_born (reg_data%n_flv_born))
    rsub%sqme_born = zero
    allocate (rsub%sqme_born_cc (reg_data%n_legs_born, reg_data%n_legs_born, &
         reg_data%n_flv_born))
    allocate (rsub%sqme_real_non_sub (reg_data%n_flv_real, reg_data%n_phs))
    allocate (rsub%sc_required (reg_data%n_regions))
    do alr = 1, reg_data%n_regions
       rsub%sc_required(alr) = any (reg_data%regions(alr)%flst_uborn%flst == GLUON)
    end do
    if (rsub%requires_spin_correlations ()) then
       allocate (rsub%sqme_born_sc (0:3, 0:3, reg_data%n_legs_born, reg_data%n_flv_born))
       rsub%sqme_born_sc = zero
    end if

    call rsub%sub_soft%init (reg_data)
    call rsub%sub_coll%init (reg_data%n_regions, reg_data%n_in)

    if (rsub%reg_data%n_in > 1 .and. any (rsub%reg_data%get_emitter_list () <= 2)) then
```

```
      call rsub%init_pdfs ()
    end if
  end subroutine real_subtraction_init
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
  procedure :: init_pdfs => real_subtraction_init_pdfs
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine real_subtraction_init_pdfs (rsub)
    class(real_subtraction_t), intent(inout) :: rsub
    type(string_t) :: lhapdf_dir, lhapdf_file
    integer :: lhapdf_member
    call msg_debug (D_SUBTRACTION, "real_subtraction_init_pdfs")
    lhapdf_dir = ""
    lhapdf_file = ""
    lhapdf_member = 0
    if (LHAPDF6_AVAILABLE) then
       call lhapdf_initialize &
          (1, lhapdf_dir, lhapdf_file, lhapdf_member, rsub%pdf_data%pdf)
       associate (pdf_data => rsub%pdf_data)
          pdf_data%type = STRF_LHAPDF6
          pdf_data%xmin = pdf_data%pdf%getxmin ()
          pdf_data%xmax = pdf_data%pdf%getxmax ()
          pdf_data%qmin = sqrt (pdf_data%pdf%getq2min ())
          pdf_data%qmax = sqrt (pdf_data%pdf%getq2max ())
       end associate
    else
       call msg_fatal ("Real subtraction: PDF method must be LHAPDF6")
    end if
  end subroutine real_subtraction_init_pdfs
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
  procedure :: set_resonance_mappings => real_subtraction_set_resonance_mappings
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine real_subtraction_set_resonance_mappings (rsub, use_mappings)
    class(real_subtraction_t), intent(inout) :: rsub
    logical, intent(in) :: use_mappings
    rsub%sub_soft%use_resonance_mappings = use_mappings
    rsub%sub_coll%use_resonance_mappings = use_mappings
  end subroutine real_subtraction_set_resonance_mappings
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
  procedure :: set_real_kinematics => real_subtraction_set_real_kinematics
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine real_subtraction_set_real_kinematics (rsub, real_kinematics)
    class(real_subtraction_t), intent(inout) :: rsub
    type(real_kinematics_t), intent(in), target :: real_kinematics
    rsub%real_kinematics => real_kinematics
  end subroutine real_subtraction_set_real_kinematics
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
  procedure :: set_isr_kinematics => real_subtraction_set_isr_kinematics
```

⟨*real subtraction: procedures*⟩+≡

```
subroutine real_subtraction_set_isr_kinematics (rsub, fractions)
  class(real_subtraction_t), intent(inout) :: rsub
  type(isr_kinematics_t), intent(in), target :: fractions
  rsub%isr_kinematics => fractions
end subroutine real_subtraction_set_isr_kinematics
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
procedure :: get_i_res => real_subtraction_get_i_res
```

⟨*real subtraction: procedures*⟩+≡

```
function real_subtraction_get_i_res (rsub, alr) result (i_res)
  integer :: i_res
  class(real_subtraction_t), intent(inout) :: rsub
  integer, intent(in) :: alr
  select type (fks_mapping => rsub%reg_data%fks_mapping)
  type is (fks_mapping_resonances_t)
     i_res = fks_mapping%res_map%alr_to_i_res (alr)
  class default
     i_res = 0
  end select
end function real_subtraction_get_i_res
```

@

27.5.4 The real contribution to the cross section

In each singular region α, the real contribution to σ is given by the second summand of eqn. **??**,

$$\sigma^\alpha_{\text{real}} = \int d\Phi_n \int_0^{2\pi} d\phi \int_{-1}^{1} dy \int_0^{\xi_{\max}} d\xi \left(\frac{1}{\xi}\right)_+ \left(\frac{1}{1-y}\right)_+ \underbrace{\frac{J(\Phi_n,\xi,y,\phi)}{\xi}\left[(1-y)\xi^2\mathcal{R}^\alpha(\Phi_{n+1})\right]}_{g^\alpha(\xi,y)}. \tag{27.17}$$

Writing out the plus-distribution and introducing $\tilde{\xi} = \xi/\xi_{\max}$ to set the upper integration limit to 1, this turns out to be equal to

$$\sigma^\alpha_{\text{real}} = \int d\Phi_n \int_0^{2/pi} d\phi \int_{-1}^{1} \frac{dy}{1-y} \left\{ \int_0^1 d\tilde{\xi} \left[\frac{g^\alpha(\tilde{\xi}\xi_{\max},y)}{\tilde{\xi}} - \underbrace{\frac{g^\alpha(0,y)}{\tilde{\xi}}}_{\text{soft}} - \underbrace{\frac{g^\alpha(\tilde{\xi}\xi_{\max},1)}{\tilde{\xi}}}_{\text{coll.}} + \underbrace{\frac{g^\alpha(0,1)}{\tilde{\xi}}}_{\text{coll.+soft}} \right] \right.$$
$$\left. + \left[\log\xi_{\max}(y) g^\alpha(0,y) - \log\xi_{\max}(1) g^\alpha(0,1)\right] \right\}. \tag{27.18}$$

This formula is implemented in `compute_sqme_real_fin`

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
procedure :: compute => real_subtraction_compute
```

⟨*real subtraction: procedures*⟩+≡

```
subroutine real_subtraction_compute (rsub, emitter, i_phs, alpha_s, &
         separate_alrs, sqme)
  class(real_subtraction_t), intent(inout) :: rsub
  integer, intent(in) :: emitter, i_phs
  logical, intent(in) :: separate_alrs
  real(default), intent(inout), dimension(:) :: sqme
  real(default) :: sqme_alr, alpha_s
  integer :: alr, i_con, i_res, this_emitter
  logical :: same_emitter
  do alr = 1, rsub%reg_data%n_regions
     if (allocated (rsub%selected_alr)) then
        if (.not. any (rsub%selected_alr == alr)) cycle
     end if
     sqme_alr = zero
     if (emitter > rsub%isr_kinematics%n_in) then
        same_emitter = emitter == rsub%reg_data%regions(alr)%emitter
     else
        same_emitter = rsub%reg_data%regions(alr)%emitter <= rsub%isr_kinematics%n_in
     end if
     if (same_emitter .and. i_phs == rsub%real_kinematics%alr_to_i_phs (alr)) then
        i_res = rsub%get_i_res (alr)
        this_emitter = rsub%reg_data%regions(alr)%emitter
        sqme_alr = rsub%evaluate_emitter_region (alr, this_emitter, i_phs, i_res, alpha_s)
        if (rsub%purpose == INTEGRATION .or. rsub%purpose == FIXED_ORDER_EVENTS) then
           i_con = rsub%get_i_contributor (alr)
           sqme_alr = sqme_alr * rsub%get_phs_factor (i_con)
        end if
     end if
     if (separate_alrs) then
        sqme(alr) = sqme(alr) + sqme_alr
     else
        sqme(1) = sqme(1) + sqme_alr
     end if
  end do
  if (debug2_active (D_SUBTRACTION)) call check_s_alpha_consistency ()
contains
  subroutine check_s_alpha_consistency ()
    real(default) :: sum_s_alpha, sum_s_alpha_soft
    integer :: i_reg, i1, i2
    call msg_debug2 (D_SUBTRACTION, "Check consistency of s_alpha: ")
    do i_reg = 1, rsub%reg_data%n_regions
       sum_s_alpha = zero; sum_s_alpha_soft = zero
       do alr = 1, rsub%reg_data%regions(i_reg)%nregions
          call rsub%reg_data%regions(i_reg)%ftuples(alr)%get (i1, i2)
          call rsub%evaluate_emitter_region_debug (i_reg, alr, i1, i2, i_phs, &
               sum_s_alpha, sum_s_alpha_soft)
       end do
    end do
  end subroutine check_s_alpha_consistency
end subroutine real_subtraction_compute
```

The emitter is fixed. We now have to decide whether we evaluate in ISR or FSR region, and also if resonances are used.

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
procedure :: evaluate_emitter_region => real_subtraction_evaluate_emitter_region
```

⟨*real subtraction: procedures*⟩+≡

```
function real_subtraction_evaluate_emitter_region (rsub, alr, emitter, &
    i_phs, i_res, alpha_s) result (sqme)
  real(default) :: sqme
  class(real_subtraction_t), intent(inout) :: rsub
  integer, intent(in) :: alr, emitter, i_phs, i_res
  real(default), intent(in) :: alpha_s
  if (emitter <= rsub%isr_kinematics%n_in) then
     sqme = rsub%evaluate_region_isr (alr, emitter, i_phs, i_res, alpha_s)
  else
     select type (fks_mapping => rsub%reg_data%fks_mapping)
     type is (fks_mapping_resonances_t)
        call fks_mapping%set_resonance_momenta &
              (rsub%real_kinematics%xi_ref_momenta)
     end select
     sqme = rsub%evaluate_region_fsr (alr, emitter, i_phs, i_res, alpha_s)
  end if
end function real_subtraction_evaluate_emitter_region
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
procedure :: evaluate_emitter_region_debug &
     => real_subtraction_evaluate_emitter_region_debug
```

⟨*real subtraction: procedures*⟩+≡

```
subroutine real_subtraction_evaluate_emitter_region_debug (rsub, i_reg, alr, i1, i2, &
     i_phs, sum_s_alpha, sum_s_alpha_soft)
  class(real_subtraction_t), intent(inout) :: rsub
  integer, intent(in) :: i_reg, alr, i1, i2, i_phs
  real(default), intent(inout) :: sum_s_alpha, sum_s_alpha_soft
  type(vector4_t), dimension(:), allocatable :: p_real, p_born
  integer :: this_emitter, i_res
  allocate (p_real (rsub%reg_data%n_legs_real))
  allocate (p_born (rsub%reg_data%n_legs_born))
  if (rsub%reg_data%has_pseudo_isr ()) then
     p_real = rsub%real_kinematics%p_real_onshell(i_phs)%get_momenta (i_phs)
     p_born = rsub%real_kinematics%p_born_onshell%get_momenta (1)
  else
     p_real = rsub%real_kinematics%p_real_cms%get_momenta (i_phs)
     p_born = rsub%real_kinematics%p_born_cms%get_momenta (1)
  end if
  i_res = rsub%get_i_res (i_reg)
  sum_s_alpha = sum_s_alpha + rsub%reg_data%get_svalue (p_real, i_reg, i1, i2, i_res)
  associate (r => rsub%real_kinematics)
     call rsub%sub_soft%create_softvec_fsr (p_born, r%y_soft(i_phs), r%phi, &
          i1, r%xi_ref_momenta(rsub%sub_soft%i_xi_ref (i_reg, i_phs)))
  end associate
  sum_s_alpha_soft = sum_s_alpha_soft + rsub%reg_data%get_svalue_soft &
       (p_born, rsub%sub_soft%p_soft, i_reg, i1, i_res)
end subroutine real_subtraction_evaluate_emitter_region_debug
```

This subroutine computes the finite part of the real matrix element in an individual singular region. First, the radiation variables are fetched and $\mathcal{R}$ is

multiplied by the appropriate S_α-factors, region multiplicities and double-FSR factors. Then, it computes the soft, collinear, soft-collinear and remnant matrix elements and supplies the corresponding factor $1/\xi/(1-y)$ as well as the corresponding jacobians.

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
procedure :: evaluate_region_fsr => real_subtraction_evaluate_region_fsr
```

⟨*real subtraction: procedures*⟩+≡

```
function real_subtraction_evaluate_region_fsr (rsub, alr, emitter, i_phs, &
     i_res, alpha_s) result (sqme_tot)
  real(default) :: sqme_tot
  class(real_subtraction_t), intent(inout) :: rsub
  integer, intent(in) :: alr, emitter, i_phs, i_res
  real(default), intent(in) :: alpha_s
  real(default) :: sqme_rad, sqme_soft, sqme_coll, sqme_cs, sqme_remn
  sqme_rad = zero; sqme_soft = zero; sqme_coll = zero
  sqme_cs = zero; sqme_remn = zero
  associate (region => rsub%reg_data%regions(alr))
    if (rsub%radiation_event) then
       sqme_rad = rsub%sqme_real_non_sub (rsub%reg_data%get_matrix_element_index (alr), i_phs)
       call evaluate_fks_factors (sqme_rad, rsub%reg_data, rsub%real_kinematics, &
            alr, i_phs, emitter, i_res)
       call apply_kinematic_factors_radiation (sqme_rad, rsub%purpose, &
            rsub%real_kinematics, i_phs, .false., rsub%reg_data%has_pseudo_isr (), &
            emitter)
    end if
    if (rsub%subtraction_event .and. .not. rsub%subtraction_deactivated) then
       call rsub%evaluate_subtraction_terms_fsr (alr, emitter, i_phs, i_res, alpha_s, &
            sqme_soft, sqme_coll, sqme_cs)
       call apply_kinematic_factors_subtraction_fsr (sqme_soft, sqme_coll, sqme_cs, &
            rsub%real_kinematics, i_phs)
       sqme_remn = compute_sqme_remnant_fsr (sqme_soft, sqme_cs, rsub%real_kinematics, i_phs)
       select case (rsub%purpose)
       case (INTEGRATION)
          sqme_tot = sqme_rad - sqme_soft - sqme_coll + sqme_cs + sqme_remn
       case (FIXED_ORDER_EVENTS)
          sqme_tot = - sqme_soft - sqme_coll + sqme_cs + sqme_remn
       case default
          sqme_tot = zero
          call msg_bug ("real_subtraction_evaluate_region_fsr: " // &
               "Undefined rsub%purpose")
       end select
    else
       sqme_tot = sqme_rad
    end if
    sqme_tot = sqme_tot * rsub%real_kinematics%jac_rand(i_phs)
  end associate

  if (debug_active (D_SUBTRACTION) .and. .not. debug2_active (D_SUBTRACTIOn)) then
     call register_debug_sqme ()
  else if (debug2_active (D_SUBTRACTION)) then
     call write_computation_status ()
  end if
```

```
contains

  subroutine register_debug_sqme ()
    real(default), dimension(:), allocatable, save :: sqme_rad_store
    logical :: soft, collinear
    real(default), parameter :: soft_threshold = 0.1_default
    real(default), parameter :: coll_threshold = 0.1_default
    real(default) :: this_sqme_rad, s_alpha, E_gluon
    logical, dimension(:), allocatable, save :: count_alr
    !!! TODO (cw-2017-02-18): Need to be able to set this (?)
    logical :: write_histo = .true.
    if (.not. allocated (sqme_rad_store)) then
       allocate (sqme_rad_store (rsub%reg_data%n_regions))
       sqme_rad_store = zero
    end if
    if (rsub%radiation_event) then
       sqme_rad_store(alr) = sqme_rad
    else
       if (.not. allocated (count_alr)) then
          allocate (count_alr (rsub%reg_data%n_regions))
          count_alr = .false.
       end if
       associate (p_real => rsub%real_kinematics%p_real_cms)
          E_gluon = p_real%get_energy (i_phs, rsub%reg_data%n_legs_real)
          s_alpha = rsub%reg_data%get_svalue (p_real%get_momenta(i_phs), alr, emitter, i_res)
       end associate
       soft = E_gluon < soft_threshold
       collinear = abs (s_alpha - one) < coll_threshold
       this_sqme_rad = sqme_rad_store(alr)
       if (soft) then
          !!! Do not write sqme_rad twice
          if (write_histo .and. .not. rsub%radiation_event) &
               call write_point_to_file (E_gluon, this_sqme_rad, sqme_soft)
          if (abs (this_sqme_rad - sqme_soft) > min (this_sqme_rad, sqme_soft) .and. &
               sqme_soft > tiny_10 .and. this_sqme_rad > tiny_10) then
             call msg_print_color ("Soft MEs do not match", COL_RED)
          else
             call msg_print_color (char ("sqme_soft OK in region " // str (alr)), COL_GREEN)
          end if
       end if
       if (collinear) then
          if (abs (this_sqme_rad - sqme_coll) > min (this_sqme_rad, sqme_coll) .and. &
               sqme_coll > tiny_10 .and. this_sqme_rad > tiny_10) then
             call msg_print_color ("Collinear MEs do not match", COL_RED)
             print *, 'alr: ', alr, 'sqme_rad: ', this_sqme_rad, 'sqme_coll: ', sqme_coll
          else
             call msg_print_color (char ("sqme_coll OK in region " // str (alr)), COL_GREEN)
          end if
       end if
       if (soft .and. collinear) then
          if (abs (this_sqme_rad - sqme_cs) > min (this_sqme_rad, sqme_cs) .and. &
               sqme_cs > tiny_10 .and. this_sqme_rad > tiny_10) then
             call msg_print_color ("Soft-collinear MEs do not match", COL_RED)
          else
```

```
            call msg_print_color (char ("sqme_cs OK in region " // str (alr)), COL_GREEN)
         end if
      end if
      count_alr (alr) = .true.
      if (all (count_alr)) then
         deallocate (count_alr)
         deallocate (sqme_rad_store)
      end if
   end if
 end subroutine register_debug_sqme

 subroutine write_computation_status (passed, total, region_type)
   integer, intent(in), optional :: passed, total
   character(*), intent(in), optional :: region_type
   integer :: i_born
   integer :: u
   real(default) :: xi
   call msg_debug (D_SUBTRACTION, "real_subtraction_evaluate_region_fsr")
   u = given_output_unit (); if (u < 0) return
   i_born = rsub%reg_data%regions(alr)%uborn_index
   xi = rsub%real_kinematics%xi_max (i_phs) * rsub%real_kinematics%xi_tilde
   write (u,'(A,I2)') 'rsub%purpose: ', rsub%purpose
   write (u,'(A,I3)') 'alr: ', alr
   write (u,'(A,I3)') 'emitter: ', emitter
   write (u,'(A,I3)') 'i_phs: ', i_phs
   write (u,'(A,F6.4)') 'xi_max: ', rsub%real_kinematics%xi_max (i_phs)
   write (u,'(A,F6.4,2X,A,F6.4)') 'xi: ', xi, 'y: ', rsub%real_kinematics%y (i_phs)
   write (u,'(A,ES16.9)')  'sqme_born: ', rsub%sqme_born(i_born)
   write (u,'(A,ES16.9)')  'sqme_real: ', sqme_rad
   write (u,'(A,ES16.9)')  'sqme_soft: ', sqme_soft
   write (u,'(A,ES16.9)')  'sqme_coll: ', sqme_coll
   write (u,'(A,ES16.9)')  'sqme_coll-soft: ', sqme_cs
   write (u,'(A,ES16.9)')  'sqme_remn: ', sqme_remn
   write (u,'(A,ES16.9)')  'sqme_tot: ', sqme_tot
   if (present (passed) .and. present (total) .and. &
        present (region_type)) &
      write (u,'(A)') char (str (passed) // " of " // str (total) // &
           " " // region_type // " points passed in total")
 end subroutine write_computation_status

 subroutine write_point_to_file (E_gluon, sqme_rad, sqme_soft)
   real(default), intent(in) :: E_gluon, sqme_rad, sqme_soft
   integer, save :: funit = 0
   type(string_t) :: filename
   filename = var_str ("soft.log")
   if (funit == 0) then
      funit = free_unit ()
      open (funit, file=char(filename), action = "write", status="replace")
      write (funit, "(A,5X,A,5X,A)") "# E_gluon", "Real", "Soft Approx"
   end if
   write (funit,'(3(ES16.9,1X))') E_gluon, sqme_rad, sqme_soft
 end subroutine write_point_to_file

end function real_subtraction_evaluate_region_fsr
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
procedure :: evaluate_region_isr => real_subtraction_evaluate_region_isr
```

⟨*real subtraction: procedures*⟩+≡

```
function real_subtraction_evaluate_region_isr (rsub, alr, emitter, i_phs, i_res, alpha_s) &
     result (sqme_tot)
  real(default) :: sqme_tot
  class(real_subtraction_t), intent(inout) :: rsub
  integer, intent(in) :: alr, emitter, i_phs, i_res
  real(default), intent(in) :: alpha_s
  integer :: i_real
  real(default) :: sqme_rad, sqme_soft, sqme_coll_plus, sqme_coll_minus
  real(default) :: sqme_cs_plus, sqme_cs_minus
  real(default) :: sqme_remn
  logical :: proc_scatter

  proc_scatter = rsub%isr_kinematics%n_in == 2

  sqme_rad = zero; sqme_soft = zero;
  sqme_coll_plus = zero; sqme_coll_minus = zero
  sqme_cs_plus = zero; sqme_cs_minus = zero
  sqme_remn = zero

  if (proc_scatter) call rsub%compute_pdfs ()
  associate (region => rsub%reg_data%regions(alr))
    i_real = region%real_index
    if (rsub%radiation_event) then
       sqme_rad = rsub%sqme_real_non_sub (1, i_phs)
       call evaluate_fks_factors (sqme_rad, rsub%reg_data, rsub%real_kinematics, &
            alr, i_phs, emitter, i_res)
       if (proc_scatter) then
          call rsub%scale_pdfs_real (sqme_rad, i_real, I_PLUS)
          call rsub%scale_pdfs_real (sqme_rad, i_real, I_MINUS)
       end if

       call apply_kinematic_factors_radiation (sqme_rad, rsub%purpose, rsub%real_kinematics, &
          i_phs, .true., .false.)
    end if
    if (rsub%subtraction_event .and. .not. rsub%subtraction_deactivated) then
       call rsub%evaluate_subtraction_terms_isr (alr, emitter, i_phs, i_res, alpha_s, &
            sqme_soft, sqme_coll_plus, sqme_coll_minus, sqme_cs_plus, sqme_cs_minus)
       if (proc_scatter) then
          call rsub%scale_pdfs_collinear (sqme_coll_plus, i_real, &
               region%uborn_index, I_PLUS)
          call rsub%scale_pdfs_collinear (sqme_coll_minus, i_real, &
               region%uborn_index, I_MINUS)
       end if
       call apply_kinematic_factors_subtraction_isr (sqme_soft, sqme_coll_plus, &
            sqme_coll_minus, sqme_cs_plus, sqme_cs_minus, rsub%real_kinematics, i_phs)
       sqme_remn = compute_sqme_remnant_isr (proc_scatter, sqme_soft, sqme_cs_plus, &
          sqme_cs_minus, rsub%isr_kinematics, rsub%real_kinematics, i_phs)

       sqme_tot = sqme_rad - sqme_soft - sqme_coll_plus - sqme_coll_minus &
```

```
            + sqme_cs_plus + sqme_cs_minus + sqme_remn
      else
         sqme_tot = sqme_rad
      end if
   end associate

   sqme_tot = sqme_tot * rsub%real_kinematics%jac_rand (i_phs)

   call debug_output ()

contains

   subroutine debug_output ()
      logical :: soft, collinear
      type(vector4_t) :: p_gluon
      if (debug_active (D_SUBTRACTION)) then
         call msg_debug (D_SUBTRACTION, "real_subtraction_evaluate_region_isr")
         if (debug_active (D_SUBTRACTION)) then
            call write_computation_status ()
         else
            associate (p_real => rsub%real_kinematics%p_real_cms)
               p_gluon = p_real%get_momentum (i_phs, p_real%get_n_momenta (i_phs))
               soft = p_gluon%p(0) < 2.0_default
            end associate
            collinear = abs (rsub%real_kinematics%y (i_phs) - one) < 0.01_default
            if (soft) then
               if (abs (sqme_rad - sqme_soft) > sqme_rad .and. sqme_soft > tiny_10) then
                  call msg_warning ("Soft MEs do not match in soft region")
                  call write_computation_status ()
               end if
            end if
            ! TODO: (bcn 2016-01-13) check coll_plus and coll_minus
            !if (collinear) then
               !if (abs (sqme_rad - sqme_coll) > sqme_rad .and. sqme_coll > tiny_10) then
                  !call msg_warning ("Collinear MEs do not match in collinear region")
                  !call write_computation_status ()
               !end if
            !end if
         end if
      end if
   end subroutine debug_output

   subroutine write_computation_status (unit)
      integer, intent(in), optional :: unit
      integer :: i_born
      integer :: u
      real(default) :: xi
      u = given_output_unit (unit); if (u < 0) return
      i_born = rsub%reg_data%regions(alr)%uborn_index
      xi = rsub%real_kinematics%xi_max (i_phs) * rsub%real_kinematics%xi_tilde
      write (u,'(A,I2)') 'alr: ', alr
      write (u,'(A,I2)') 'emitter: ', emitter
      write (u,'(A,F4.2)') 'xi_max: ', rsub%real_kinematics%xi_max (i_phs)
      print *, 'xi: ', xi, 'y: ', rsub%real_kinematics%y (i_phs)
```

```
      print *, 'xb1: ', rsub%isr_kinematics%x(1), 'xb2: ', rsub%isr_kinematics%x(2)
      print *, 'random jacobian: ', rsub%real_kinematics%jac_rand (i_phs)
      write (u,'(A,ES16.9)')  'sqme_born: ', rsub%sqme_born(i_born)
      write (u,'(A,ES16.9)')  'sqme_real: ', sqme_rad
      write (u,'(A,ES16.9)')  'sqme_soft: ', sqme_soft
      write (u,'(A,ES16.9)')  'sqme_coll_plus: ', sqme_coll_plus
      write (u,'(A,ES16.9)')  'sqme_coll_minus: ', sqme_coll_minus
      write (u,'(A,ES16.9)')  'sqme_cs_plus: ', sqme_cs_plus
      write (u,'(A,ES16.9)')  'sqme_cs_minus: ', sqme_cs_minus
      write (u,'(A,ES16.9)')  'sqme_remn: ', sqme_remn
      write (u,'(A,ES16.9)')  'sqme_tot: ', sqme_tot
    end subroutine write_computation_status

  end function real_subtraction_evaluate_region_isr
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
  procedure :: evaluate_subtraction_terms_fsr => &
      real_subtraction_evaluate_subtraction_terms_fsr
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine real_subtraction_evaluate_subtraction_terms_fsr (rsub, &
      alr, emitter, i_phs, i_res, alpha_s, sqme_soft, sqme_coll, sqme_cs)
    class(real_subtraction_t), intent(inout) :: rsub
    integer, intent(in) :: alr, emitter, i_phs, i_res
    real(default), intent(in) :: alpha_s
    real(default), intent(out) :: sqme_soft, sqme_coll, sqme_cs
    sqme_soft = rsub%compute_sub_soft (alr, emitter, i_phs, i_res, alpha_s)
    sqme_coll = rsub%compute_sub_coll (alr, emitter, i_phs, alpha_s)
    sqme_cs = rsub%compute_sub_coll_soft (alr, emitter, alpha_s)
  end subroutine real_subtraction_evaluate_subtraction_terms_fsr
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine evaluate_fks_factors (sqme, reg_data, real_kinematics, &
      alr, i_phs, emitter, i_res)
    real(default), intent(inout) :: sqme
    type(region_data_t), intent(inout) :: reg_data
    type(real_kinematics_t), intent(in), target :: real_kinematics
    integer, intent(in) :: alr, i_phs, emitter, i_res
    real(default) :: s_alpha
    type(phs_point_set_t), pointer :: p_real => null ()
    if (reg_data%has_pseudo_isr ()) then
       p_real => real_kinematics%p_real_onshell (i_phs)
    else
       p_real => real_kinematics%p_real_cms
    end if
    s_alpha = reg_data%get_svalue (p_real%get_momenta(i_phs), alr, emitter, i_res)
    if (debug_active (D_SUBTRACTION)) call msg_print_color('s_alpha', s_alpha, COL_YELLOW)
    if (s_alpha > one + tiny_07) call msg_fatal ("s_alpha > 1!")
    sqme = sqme * s_alpha
    associate (region => reg_data%regions(alr))
       sqme = sqme * region%mult
       if (emitter > reg_data%n_in) &
            sqme = sqme * region%double_fsr_factor (p_real%get_momenta(i_phs))
```

```
    end associate
  end subroutine evaluate_fks_factors
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine apply_kinematic_factors_radiation (sqme, purpose, real_kinematics, &
        i_phs, isr, threshold, emitter)
    real(default), intent(inout) :: sqme
    integer, intent(in) :: purpose
    type(real_kinematics_t), intent(in) :: real_kinematics
    integer, intent(in) :: i_phs
    logical, intent(in) :: isr, threshold
    integer, intent(in), optional :: emitter
    real(default) :: xi, xi_tilde, s
    xi_tilde = real_kinematics%xi_tilde
    xi = xi_tilde * real_kinematics%xi_max (i_phs)
    select case (purpose)
    case (INTEGRATION, FIXED_ORDER_EVENTS)
       sqme = sqme * xi**2 / xi_tilde * real_kinematics%jac(i_phs)%jac(1)
    case (POWHEG)
       if (.not. isr) then
          s = real_kinematics%cms_energy2
          sqme = sqme * real_kinematics%jac(i_phs)%jac(1) * s / (8 * twopi3) * xi
       else
          call msg_fatal ("POWHEG with initial-state radiation not implemented yet")
       end if
    end select
  end subroutine apply_kinematic_factors_radiation
```

⟨*real subtraction: procedures*⟩+≡

```
  subroutine apply_kinematic_factors_subtraction_fsr &
     (sqme_soft, sqme_coll, sqme_cs, real_kinematics, i_phs)
    real(default), intent(inout) :: sqme_soft, sqme_coll, sqme_cs
    type(real_kinematics_t), intent(in) :: real_kinematics
    integer, intent(in) :: i_phs
    real(default) :: xi_tilde, onemy
    xi_tilde = real_kinematics%xi_tilde
    onemy = one - real_kinematics%y(i_phs)
    sqme_soft = sqme_soft / onemy / xi_tilde
    sqme_coll = sqme_coll / onemy / xi_tilde
    sqme_cs = sqme_cs / onemy / xi_tilde
    associate (jac => real_kinematics%jac(i_phs)%jac)
       sqme_soft = sqme_soft * jac(2)
       sqme_coll = sqme_coll * jac(3)
       sqme_cs = sqme_cs * jac(2)
    end associate
  end subroutine apply_kinematic_factors_subtraction_fsr
```

For final state radiation, the subtraction remnant cross section is

$$\sigma_{\mathrm{remn}} = (\sigma_{\mathrm{soft}} - \sigma_{\mathrm{coll}}) \log \xi_{\max} \cdot \tilde{\xi}. \tag{27.19}$$

⟨*real subtraction: procedures*⟩+≡

```
  function compute_sqme_remnant_fsr (sqme_soft, sqme_cs, &
```

```
    real_kinematics, i_phs) result (sqme_remn)
  real(default) :: sqme_remn
  real(default), intent(in) :: sqme_soft, sqme_cs
  type(real_kinematics_t), intent(in) :: real_kinematics
  integer, intent(in) :: i_phs
  sqme_remn = (sqme_soft - sqme_cs) * &
       log (real_kinematics%xi_max (i_phs)) * real_kinematics%xi_tilde
end function compute_sqme_remnant_fsr
```

⟨*real subtraction: procedures*⟩+≡

```
subroutine apply_kinematic_factors_subtraction_isr &
   (sqme_soft, sqme_coll_plus, sqme_coll_minus, sqme_cs_plus, &
    sqme_cs_minus, real_kinematics, i_phs)
  real(default), intent(inout) :: sqme_soft, sqme_coll_plus, sqme_coll_minus
  real(default), intent(inout) :: sqme_cs_plus, sqme_cs_minus
  type(real_kinematics_t), intent(in) :: real_kinematics
  integer, intent(in) :: i_phs
  real(default) :: xi_tilde, y, onemy, onepy
  xi_tilde = real_kinematics%xi_tilde
  y = real_kinematics%y (i_phs)
  onemy = one - y; onepy = one + y
  associate (jac => real_kinematics%jac(i_phs)%jac)
     sqme_soft = sqme_soft / (one - y**2) / xi_tilde * jac(2)
     sqme_coll_plus = sqme_coll_plus / onemy / xi_tilde / two * jac(3)
     sqme_coll_minus = sqme_coll_minus / onepy / xi_tilde / two * jac(4)
     sqme_cs_plus = sqme_cs_plus / onemy / xi_tilde / two * jac(2)
     sqme_cs_minus = sqme_cs_minus / onepy / xi_tilde / two * jac(2)
  end associate
end subroutine apply_kinematic_factors_subtraction_isr
```

⟨*real subtraction: procedures*⟩+≡

```
function compute_sqme_remnant_isr (proc_scatter, sqme_soft, sqme_cs_plus, sqme_cs_minus, &
   isr_kinematics, real_kinematics, i_phs) result (sqme_remn)
  real(default) :: sqme_remn
  logical, intent(in) :: proc_scatter
  real(default), intent(in) :: sqme_soft, sqme_cs_plus, sqme_cs_minus
  type(isr_kinematics_t), intent(in) :: isr_kinematics
  type(real_kinematics_t), intent(in) :: real_kinematics
  integer, intent(in) :: i_phs
  real(default) :: xi_tilde, xi_max, xi_max_plus, xi_max_minus
  xi_max = real_kinematics%xi_max (i_phs)
  if (proc_scatter) then
     xi_max_plus = one - isr_kinematics%x(I_PLUS)
     xi_max_minus = one - isr_kinematics%x(I_MINUS)
  else
     xi_max_plus = real_kinematics%xi_max (i_phs)
     xi_max_minus = real_kinematics%xi_max (i_phs)
  end if
  xi_tilde = real_kinematics%xi_tilde
  sqme_remn = log(xi_max) * xi_tilde * sqme_soft
  sqme_remn = sqme_remn - log (xi_max_plus) * xi_tilde * sqme_cs_plus &
                        - log (xi_max_minus) * xi_tilde * sqme_cs_minus
end function compute_sqme_remnant_isr
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
procedure :: evaluate_subtraction_terms_isr => &
    real_subtraction_evaluate_subtraction_terms_isr
```

⟨*real subtraction: procedures*⟩+≡

```
subroutine real_subtraction_evaluate_subtraction_terms_isr (rsub, &
    alr, emitter, i_phs, i_res, alpha_s, sqme_soft, sqme_coll_plus, &
    sqme_coll_minus, sqme_cs_plus, sqme_cs_minus)

  class(real_subtraction_t), intent(inout) :: rsub
  integer, intent(in) :: alr, emitter, i_phs, i_res
  real(default), intent(in) :: alpha_s
  real(default), intent(out) :: sqme_soft
  real(default), intent(out) :: sqme_coll_plus, sqme_coll_minus
  real(default), intent(out) :: sqme_cs_plus, sqme_cs_minus
  sqme_soft = rsub%compute_sub_soft (alr, emitter, i_phs, i_res, alpha_s)
  if (emitter /= 2) then
     sqme_coll_plus = rsub%compute_sub_coll (alr, 1, i_phs, alpha_s)
     sqme_cs_plus = rsub%compute_sub_coll_soft (alr, 1, alpha_s)
  else
     sqme_coll_plus = zero
     sqme_cs_plus = zero
  end if
  if (emitter /= 1) then
     sqme_coll_minus = rsub%compute_sub_coll (alr, 2, i_phs, alpha_s)
     sqme_cs_minus = rsub%compute_sub_coll_soft (alr, 2, alpha_s)
  else
     sqme_coll_minus = zero
     sqme_cs_minus = zero
  end if
end subroutine real_subtraction_evaluate_subtraction_terms_isr
```

This is basically the part of the real jacobian corresponding to

$$\frac{q^2}{8(2\pi)^3}.$$

We interpret it as the additional phase space factor of the real component, to be more consistent with the evaluation of the Born phase space.

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
procedure :: get_phs_factor => real_subtraction_get_phs_factor
```

⟨*real subtraction: procedures*⟩+≡

```
function real_subtraction_get_phs_factor (rsub, i_con) result (factor)
  real(default) :: factor
  class(real_subtraction_t), intent(in) :: rsub
  integer, intent(in) :: i_con
  real(default) :: s
  s = rsub%real_kinematics%xi_ref_momenta (i_con)**2
  factor = s / (8 * twopi3)
end function real_subtraction_get_phs_factor
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
procedure :: get_i_contributor => real_subtraction_get_i_contributor
```

⟨*real subtraction: procedures*⟩+≡

```
function real_subtraction_get_i_contributor (rsub, alr) result (i_con)
  integer :: i_con
  class(real_subtraction_t), intent(in) :: rsub
  integer, intent(in) :: alr
  if (allocated (rsub%reg_data%alr_to_i_contributor)) then
     i_con = rsub%reg_data%alr_to_i_contributor (alr)
  else
     i_con = 1
  end if
end function real_subtraction_get_i_contributor
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
procedure :: compute_sub_soft => real_subtraction_compute_sub_soft
```

⟨*real subtraction: procedures*⟩+≡

```
function real_subtraction_compute_sub_soft (rsub, alr, emitter, &
     i_phs, i_res, alpha_s) result (sqme_soft)
  real(default) :: sqme_soft
  class(real_subtraction_t), intent(inout) :: rsub
  integer, intent(in) :: alr, emitter, i_phs, i_res
  real(default), intent(in) :: alpha_s
  integer :: i_xi_ref, i_born
  real(default) :: q2
  type(vector4_t), dimension(:), allocatable :: p_born
  associate (real_kinematics => rsub%real_kinematics)
     if (rsub%reg_data%regions(alr)%has_soft_divergence ()) then
        i_xi_ref = rsub%sub_soft%i_xi_ref (alr, i_phs)
        q2 = real_kinematics%xi_ref_momenta (i_xi_ref)**2
        allocate (p_born (rsub%reg_data%n_legs_born))
        if (rsub%reg_data%has_pseudo_isr ()) then
           p_born = real_kinematics%p_born_onshell%get_momenta(1)
        else
           p_born = real_kinematics%p_born_cms%get_momenta(1)
        end if
        if (emitter > rsub%sub_soft%reg_data%n_in) then
           call rsub%sub_soft%create_softvec_fsr &
                (p_born, real_kinematics%y_soft(i_phs), &
                real_kinematics%phi, emitter, &
                real_kinematics%xi_ref_momenta(i_xi_ref))
        else
           call rsub%sub_soft%create_softvec_isr &
              (real_kinematics%y_soft(i_phs), real_kinematics%phi)
        end if
        i_born = rsub%reg_data%regions(alr)%uborn_index
        sqme_soft = rsub%sub_soft%compute &
             (p_born, rsub%sqme_born_cc(:,:,i_born), &
             real_kinematics%y(i_phs), &
             q2, alpha_s, alr, emitter, i_res)
     else
        sqme_soft = zero
     end if
```

```
    end associate
    if (debug2_active (D_SUBTRACTION)) call check_soft_vector ()
  contains
    subroutine check_soft_vector ()
      type(vector4_t) :: p_gluon
      call msg_debug2 (D_SUBTRACTION, "Compare soft vector: ")
      print *, 'p_soft: ', rsub%sub_soft%p_soft%p
      print *, 'Normalized gluon momentum: '
      if (rsub%reg_data%has_pseudo_isr ()) then
         p_gluon = rsub%real_kinematics%p_real_onshell(thr_leg(emitter))%get_momentum &
              (i_phs, rsub%reg_data%n_legs_real)
      else
         p_gluon = rsub%real_kinematics%p_real_cms%get_momentum &
              (i_phs, rsub%reg_data%n_legs_real)
      end if
      call vector4_write (p_gluon / p_gluon%p(0), show_mass = .true.)
    end subroutine check_soft_vector
  end function real_subtraction_compute_sub_soft
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
  procedure :: get_spin_correlation_term => real_subtraction_get_spin_correlation_term
```

⟨*real subtraction: procedures*⟩+≡

```
  function real_subtraction_get_spin_correlation_term (rsub, alr, i_born, emitter) &
        result (mom_times_sqme)
    real(default) :: mom_times_sqme
    class(real_subtraction_t), intent(in) :: rsub
    integer, intent(in) :: alr, i_born, emitter
    real(default), dimension(0:3) :: k_perp
    integer :: i, mu, nu

    if (rsub%sc_required(alr)) then
       if (debug2_active(D_SUBTRACTION)) call check_me_consistency ()
       associate (real_kin => rsub%real_kinematics)
          if (emitter > rsub%reg_data%n_in) then
             k_perp = k_perp_fsr (real_kin%p_born_cms%get_momentum(1, emitter), &
                  rsub%real_kinematics%phi)
          else
             k_perp = k_perp_isr (real_kin%p_born_cms%get_momentum(1, emitter), &
                  rsub%real_kinematics%phi)
          end if
       end associate
       mom_times_sqme = zero
       do mu = 0, 3
          do nu = 0, 3
             mom_times_sqme = mom_times_sqme + &
                  k_perp(mu) * k_perp(nu) * rsub%sqme_born_sc (mu, nu, emitter, i_born)
          end do
       end do
    else
       mom_times_sqme = zero
    end if
  contains
    subroutine check_me_consistency ()
```

```
      real(default) ::  sqme_sum
      call msg_debug2 (D_SUBTRACTION, "Spin-correlation: Consistency check")
      sqme_sum = rsub%sqme_born_sc(0,0,emitter,i_born) &
               - rsub%sqme_born_sc(1,1,emitter,i_born) &
               - rsub%sqme_born_sc(2,2,emitter,i_born) &
               - rsub%sqme_born_sc(3,3,emitter,i_born)
      if (.not. nearly_equal (sqme_sum, -rsub%sqme_born(i_born), 0.0001_default)) then
         print *, 'Spin-correlated matrix elements are not consistent: '
         print *, 'emitter: ', emitter
         print *, 'g^{mu,nu} B_{mu,nu}: ', -sqme_sum
         print *, 'all Born matrix elements: ', rsub%sqme_born
         call msg_fatal ("FAIL")
      else
         call msg_print_color ("Success", COL_GREEN)
      end if
    end subroutine check_me_consistency

    function k_perp_fsr (p, phi)
      real(default), dimension(0:3) :: k_perp_fsr
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: phi
      type(vector4_t) :: k
      type(vector3_t) :: vec
      type(lorentz_transformation_t) :: rot
      vec = p%p(1:3) / p%p(0)
      k%p(0) = zero
      k%p(1) = p%p(1); k%p(2) = p%p(2)
      k%p(3) = - (p%p(1)**2 + p%p(2)**2) / p%p(3)
      rot = rotation (cos(phi), sin(phi), vec)
      k = rot * k
      k%p(1:3) = k%p(1:3) / space_part_norm (k)
      k_perp_fsr = k%p
    end function k_perp_fsr

    function k_perp_isr (p, phi)
      real(default), dimension(0:3) :: k_perp_isr
      type(vector4_t), intent(in) :: p
      real(default), intent(in) :: phi
      k_perp_isr(0) = zero
      k_perp_isr(1) = cos(phi)
      k_perp_isr(2) = sin(phi)
      k_perp_isr(3) = zero
    end function k_perp_isr
  end function real_subtraction_get_spin_correlation_term
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
  procedure :: compute_sub_coll => real_subtraction_compute_sub_coll
```

⟨*real subtraction: procedures*⟩+≡

```
  function real_subtraction_compute_sub_coll (rsub, alr, em, i_phs, alpha_s) result (sqme_coll)
    real(default) :: sqme_coll
    class(real_subtraction_t), intent(inout) :: rsub
    integer, intent(in) :: alr, em, i_phs
    real(default), intent(in) :: alpha_s
```

```
    real(default) :: xi, xi_max_pm
    real(default) :: mom_times_sqme_sc
    integer :: i_con
    real(default) :: pfr
    associate (sregion => rsub%reg_data%regions(alr))
       if (sregion%has_collinear_divergence ()) then
          xi = rsub%real_kinematics%xi_tilde * rsub%real_kinematics%xi_max (i_phs)
          if (rsub%sub_coll%use_resonance_mappings) then
             i_con = rsub%reg_data%alr_to_i_contributor (alr)
          else
             i_con = 1
          end if
          mom_times_sqme_sc = rsub%get_spin_correlation_term (alr, sregion%uborn_index, em)
          if (em <= rsub%sub_coll%n_in) then
             xi_max_pm = one - rsub%isr_kinematics%x(em)
             xi = rsub%real_kinematics%xi_tilde * xi_max_pm
             sqme_coll = rsub%sub_coll%compute_isr (sregion, &
                  rsub%real_kinematics%p_born_lab%phs_point(1)%p, &
                  rsub%sqme_born(sregion%uborn_index), mom_times_sqme_sc, &
                  xi, alpha_s, alr, em)
          else
             sqme_coll = rsub%sub_coll%compute_fsr (sregion, &
                    rsub%real_kinematics%xi_ref_momenta (i_con), &
                    rsub%real_kinematics%p_born_cms%get_momenta(1), &
                    rsub%sqme_born(sregion%uborn_index), mom_times_sqme_sc, &
                    xi, alpha_s, alr, sregion%double_fsr)
             if (rsub%sub_coll%use_resonance_mappings) then
                select type (fks_mapping => rsub%reg_data%fks_mapping)
                type is (fks_mapping_resonances_t)
                   pfr = fks_mapping%get_resonance_weight (alr, &
                         rsub%real_kinematics%p_born_cms%get_momenta(1))
                end select
                sqme_coll = sqme_coll * pfr
             end if
          end if
       else
          sqme_coll = zero
       end if
    end associate
  end function real_subtraction_compute_sub_coll
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
  procedure :: compute_sub_coll_soft => real_subtraction_compute_sub_coll_soft
```

⟨*real subtraction: procedures*⟩+≡

```
  function real_subtraction_compute_sub_coll_soft (rsub, alr, em, alpha_s) result (sqme_cs)
    real(default) :: sqme_cs
    class(real_subtraction_t), intent(inout) :: rsub
    integer, intent(in) :: alr, em
    real(default), intent(in) :: alpha_s
    real(default) :: mom_times_sqme_sc
    integer :: i_con
    associate (sregion => rsub%reg_data%regions(alr))
       if (sregion%has_collinear_divergence ()) then
```

```
            if (rsub%sub_coll%use_resonance_mappings) then
               i_con = rsub%reg_data%alr_to_i_contributor (alr)
            else
               i_con = 1
            end if
            mom_times_sqme_sc = rsub%get_spin_correlation_term (alr, sregion%uborn_index, em)
            if (em <= rsub%sub_coll%n_in) then
               sqme_cs = rsub%sub_coll%compute_isr (sregion, &
                    rsub%real_kinematics%p_born_lab%phs_point(1)%p, &
                    rsub%sqme_born(sregion%uborn_index), mom_times_sqme_sc, &
                    zero, alpha_s, alr, em)
            else
               sqme_cs = rsub%sub_coll%compute_fsr (sregion, &
                    rsub%real_kinematics%xi_ref_momenta(i_con), &
                    rsub%real_kinematics%p_born_cms%phs_point(1)%p, &
                    rsub%sqme_born(sregion%uborn_index), mom_times_sqme_sc, &
                    zero, alpha_s, alr, sregion%double_fsr)
            end if
         else
            sqme_cs = zero
         end if
      end associate
   end function real_subtraction_compute_sub_coll_soft
```

Gets the pdf value from the `pdf_data` object. Note that LHAPDF delivers $x \cdot f(x)$ as output, so that x has to be factored out.

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
   procedure :: compute_pdfs => real_subtraction_compute_pdfs
```

⟨*real subtraction: procedures*⟩+≡

```
   subroutine real_subtraction_compute_pdfs (rsub)
      class(real_subtraction_t), intent(inout) :: rsub
      integer :: i
      real(default) :: z, z_coll, x, Q
      real(default) :: x_scaled, x_scaled_coll
      real(double), dimension(-6:6) :: f_dble = 0._double
      Q = rsub%isr_kinematics%fac_scale
      do i = 1, 2
         x = rsub%isr_kinematics%x(i)
         z = rsub%isr_kinematics%z(i)
         z_coll = rsub%isr_kinematics%z_coll(i)
         x_scaled = x * z
         x_scaled_coll = x * z_coll
         call rsub%pdf_data%evolve (dble(x), dble(Q), f_dble)
         rsub%pdf_born(i)%f = f_dble / dble(x)
         call rsub%pdf_data%evolve (dble(x_scaled), dble(Q), f_dble)
         rsub%pdf_scaled(i)%f = f_dble / dble(x_scaled)
         call rsub%pdf_data%evolve (dble(x_scaled_coll), dble(Q), f_dble)
         rsub%pdf_scaled_coll(i)%f = f_dble / dble(x_scaled_coll)
      end do
   end subroutine real_subtraction_compute_pdfs
```

The real matrix element is evaluated already multiplied with the parton densities

at Born level, but with the real flavor structure. We multiply out the Born contribution and use the real one instead.

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
procedure :: scale_pdfs_real => real_subtraction_scale_pdfs_real
```

⟨*real subtraction: procedures*⟩+≡

```
subroutine real_subtraction_scale_pdfs_real (rsub, sqme, i_real, i_part)
  class(real_subtraction_t), intent(inout) :: rsub
  real(default), intent(inout) :: sqme
  integer, intent(in) :: i_real, i_part
  integer :: flv
  real(default) :: pdfs, pdfb
  flv = rsub%reg_data%flv_real(i_real)%flst(i_part)
  !!! Gluon has index 0 in the pdf array
  if (flv == GLUON) flv = 0
  pdfb = rsub%pdf_born(i_part)%f(flv)
  pdfs = rsub%pdf_scaled(i_part)%f(flv)
  sqme = sqme * pdfs / pdfb
end subroutine real_subtraction_scale_pdfs_real
```

The same as above, but for the collinear subtraction terms. They are proportional to the Born matrix element so they enter the subroutine with pdfs evaluated for the Born flavor structure. Therefore, we additionally have to take the flavor of the underlying Born particle when dividing out the Born contribution.

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
procedure :: scale_pdfs_collinear => real_subtraction_scale_pdfs_collinear
```

⟨*real subtraction: procedures*⟩+≡

```
subroutine real_subtraction_scale_pdfs_collinear &
   (rsub, sqme, i_real, i_born, i_part)
  class(real_subtraction_t), intent(inout) :: rsub
  real(default), intent(inout) :: sqme
  integer, intent(in) :: i_real, i_born, i_part
  integer :: flv_born, flv_real
  real(default) :: pdfs, pdfb
  flv_born = rsub%reg_data%flv_born(i_born)%flst(i_part)
  flv_real = rsub%reg_data%flv_real(i_real)%flst(i_part)
  if (flv_born == GLUON) flv_born = 0
  if (flv_real == GLUON) flv_real = 0
  pdfb = rsub%pdf_born(i_part)%f(flv_born)
  pdfs = rsub%pdf_scaled_coll(i_part)%f(flv_real)
  sqme = sqme * pdfs / pdfb
end subroutine real_subtraction_scale_pdfs_collinear
```

⟨*real subtraction: real subtraction: TBP*⟩+≡

```
procedure :: requires_spin_correlations => &
     real_subtraction_requires_spin_correlations
```

⟨*real subtraction: procedures*⟩+≡

```
function real_subtraction_requires_spin_correlations (rsub) result (val)
  logical :: val
  class(real_subtraction_t), intent(in) :: rsub
  val = any (rsub%sc_required)
end function real_subtraction_requires_spin_correlations
```

⟨*real subtraction: real subtraction: TBP*⟩+≡
```
procedure :: final => real_subtraction_final
```

⟨*real subtraction: procedures*⟩+≡
```
subroutine real_subtraction_final (rsub)
  class(real_subtraction_t), intent(inout) :: rsub
  call rsub%sub_soft%final ()
  call rsub%sub_coll%final ()
  !!! Finalization of region data is done in pcm_nlo_final
  if (associated (rsub%reg_data)) nullify (rsub%reg_data)
  !!! Finalization of real kinematics is done in pcm_instance_nlo_final
  if (associated (rsub%real_kinematics)) nullify (rsub%real_kinematics)
  if (associated (rsub%isr_kinematics)) nullify (rsub%isr_kinematics)
  if (allocated (rsub%sqme_real_non_sub)) deallocate (rsub%sqme_real_non_sub)
  if (allocated (rsub%sqme_born)) deallocate (rsub%sqme_born)
  if (allocated (rsub%sqme_born_cc)) deallocate (rsub%sqme_born_cc)
  if (allocated (rsub%sc_required)) deallocate (rsub%sc_required)
  if (allocated (rsub%selected_alr)) deallocate (rsub%selected_alr)
end subroutine real_subtraction_final
```

Partitions of the real matrix element and Powheg damping

⟨*real subtraction: public*⟩+≡
```
public :: real_partition_t
```

⟨*real subtraction: types*⟩+≡
```
type, abstract :: real_partition_t
contains
⟨real subtraction: real partition: TBP⟩
end type real_partition_t
```

⟨*real subtraction: real partition: TBP*⟩≡
```
procedure (real_partition_init), deferred :: init
```

⟨*real subtraction: interfaces*⟩≡
```
abstract interface
   subroutine real_partition_init (partition, scale, reg_data)
     import
     class(real_partition_t), intent(out) :: partition
     real(default), intent(in) :: scale
     type(region_data_t), intent(in) :: reg_data
   end subroutine real_partition_init
end interface
```

⟨*real subtraction: real partition: TBP*⟩+≡
```
procedure (real_partition_write), deferred :: write
```

⟨*real subtraction: interfaces*⟩+≡
```
abstract interface
   subroutine real_partition_write (partition, unit)
     import
     class(real_partition_t), intent(in) :: partition
     integer, intent(in), optional :: unit
   end subroutine real_partition_write
```

```
end interface
```

To allow really arbitrary damping functions, `get_f` should get the full real phase space as argument and not just some `pt2` that is extracted higher up.

⟨*real subtraction: real partition: TBP*⟩+≡
```
procedure (real_partition_get_f), deferred :: get_f
```

⟨*real subtraction: interfaces*⟩+≡
```
abstract interface
  function real_partition_get_f (partition, p) result (f)
     import
     real(default) :: f
     class(real_partition_t), intent(in) :: partition
     type(vector4_t), intent(in), dimension(:) :: p
  end function real_partition_get_f
end interface
```

⟨*real subtraction: public*⟩+≡
```
public :: powheg_damping_simple_t
```

⟨*real subtraction: types*⟩+≡
```
type, extends (real_partition_t) :: powheg_damping_simple_t
   real(default) :: h2 = 5._default
   integer :: emitter
contains
⟨real subtraction: powheg damping simple: TBP⟩
end type powheg_damping_simple_t
```

⟨*real subtraction: powheg damping simple: TBP*⟩≡
```
procedure :: get_f => powheg_damping_simple_get_f
```

⟨*real subtraction: procedures*⟩+≡
```
function powheg_damping_simple_get_f (partition, p) result (f)
  real(default) :: f
  class(powheg_damping_simple_t), intent(in) :: partition
  type(vector4_t), intent(in), dimension(:) :: p
  real(default) :: pt2
  !!! TODO (cw-2017-03-01) Compute pt2 from emitter)
  f = partition%h2 / (pt2 + partition%h2)
end function powheg_damping_simple_get_f
```

⟨*real subtraction: powheg damping simple: TBP*⟩+≡
```
procedure :: init => powheg_damping_simple_init
```

⟨*real subtraction: procedures*⟩+≡
```
subroutine powheg_damping_simple_init (partition, scale, reg_data)
  class(powheg_damping_simple_t), intent(out) :: partition
  real(default), intent(in) :: scale
  type(region_data_t), intent(in) :: reg_data
  partition%h2 = scale**2
end subroutine powheg_damping_simple_init
```

⟨*real subtraction: powheg damping simple: TBP*⟩+≡
```
procedure :: write => powheg_damping_simple_write
```

⟨*real subtraction: procedures*⟩+≡
```
subroutine powheg_damping_simple_write (partition, unit)
  class(powheg_damping_simple_t), intent(in) :: partition
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit); if (u < 0) return
  write (u, "(1x,A)") "Powheg damping simple: "
  write (u, "(1x,A, "// FMT_15 // ")") "scale h2: ", partition%h2
end subroutine powheg_damping_simple_write
```

⟨*real subtraction: public*⟩+≡
```
public :: real_partition_fixed_order_t
```

⟨*real subtraction: types*⟩+≡
```
type, extends (real_partition_t) :: real_partition_fixed_order_t
   real(default) :: scale
   type(ftuple_t), dimension(:), allocatable :: fks_pairs
contains
```
⟨*real subtraction: real partition fixed order: TBP*⟩
```
end type real_partition_fixed_order_t
```

⟨*real subtraction: real partition fixed order: TBP*⟩≡
```
procedure :: init => real_partition_fixed_order_init
```

⟨*real subtraction: procedures*⟩+≡
```
subroutine real_partition_fixed_order_init (partition, scale, reg_data)
  class(real_partition_fixed_order_t), intent(out) :: partition
  real(default), intent(in) :: scale
  type(region_data_t), intent(in) :: reg_data
end subroutine real_partition_fixed_order_init
```

⟨*real subtraction: real partition fixed order: TBP*⟩+≡
```
procedure :: write => real_partition_fixed_order_write
```

⟨*real subtraction: procedures*⟩+≡
```
subroutine real_partition_fixed_order_write (partition, unit)
  class(real_partition_fixed_order_t), intent(in) :: partition
  integer, intent(in), optional :: unit
end subroutine real_partition_fixed_order_write
```

⟨*real subtraction: real partition fixed order: TBP*⟩+≡
```
procedure :: get_f => real_partition_fixed_order_get_f
```

⟨*real subtraction: procedures*⟩+≡
```
function real_partition_fixed_order_get_f (partition, p) result (f)
  real(default) :: f
  class(real_partition_fixed_order_t), intent(in) :: partition
  type(vector4_t), intent(in), dimension(:) :: p
  integer :: i
  f = zero
  do i = 1, size (partition%fks_pairs)
```

```
      associate (ii => partition%fks_pairs(i)%ireg)
         if ((p(ii(1)) + p(ii(2)))**1 < p(ii(1))**1 + p(ii(2))**1 + partition%scale) then
            f = one
            exit
         end if
      end associate
   end do
end function real_partition_fixed_order_get_f
```

27.6 Combining the FKS Pieces

⟨nlo_data.f90⟩≡

```
⟨File header⟩

module nlo_data

⟨Use kinds⟩
⟨Use strings⟩
  use diagnostics
  use constants, only: zero
  use string_utils, only: split_string, read_ival, string_contains_word
  use io_units
  use lorentz
  use variables, only: var_list_t
  use format_defs, only: FMT_15
  use physics_defs, only: THR_POS_WP, THR_POS_WM
  use physics_defs, only: THR_POS_B, THR_POS_BBAR
  use physics_defs, only: NO_FACTORIZATION, FACTORIZATION_THRESHOLD

⟨Standard module head⟩

⟨nlo data: public⟩

⟨nlo data: parameters⟩

⟨nlo data: types⟩
⟨nlo data: interfaces⟩

contains

⟨nlo data: procedures⟩

end module nlo_data
```

⟨*nlo data: parameters*⟩≡

```
  integer, parameter, public :: FKS_DEFAULT = 1
  integer, parameter, public :: FKS_RESONANCES = 2

  integer, dimension(2), parameter, public :: ASSOCIATED_LEG_PAIR = [1, 3]
```

⟨*nlo data: public*⟩≡

```
  public :: fks_template_t
```

⟨*nlo data: types*⟩≡

```
  type :: fks_template_t
    logical :: subtraction_disabled = .false.
    integer :: mapping_type = FKS_DEFAULT
    logical :: count_kinematics = .false.
    real(default) :: fks_dij_exp1
    real(default) :: fks_dij_exp2
    real(default) :: xi_min
    real(default) :: y_max
    type(string_t), dimension(:), allocatable :: excluded_resonances
```

```
contains
⟨nlo data: fks template: TBP⟩
end type fks_template_t
```

⟨*nlo data: fks template: TBP*⟩≡

```
procedure :: write => fks_template_write
```

⟨*nlo data: procedures*⟩≡

```
subroutine fks_template_write (template, unit)
  class(fks_template_t), intent(in) :: template
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u,'(1x,A)') 'FKS Template: '
  write (u,'(1x,A)', advance = 'no') 'Mapping Type: '
  select case (template%mapping_type)
  case (FKS_DEFAULT)
     write (u,'(A)') 'Default'
  case (FKS_RESONANCES)
     write (u,'(A)') 'Resonances'
  case default
     write (u,'(A)') 'Unkown'
  end select
  write (u,'(1x,A,ES4.3,ES4.3)') 'd_ij exponentials: ', &
     template%fks_dij_exp1, template%fks_dij_exp2
end subroutine fks_template_write
```

⟨*nlo data: fks template: TBP*⟩+≡

```
procedure :: set_dij_exp => fks_template_set_dij_exp
```

⟨*nlo data: procedures*⟩+≡

```
subroutine fks_template_set_dij_exp (template, exp1, exp2)
  class(fks_template_t), intent(inout) :: template
  real(default), intent(in) :: exp1, exp2
  template%fks_dij_exp1 = exp1
  template%fks_dij_exp2 = exp2
end subroutine fks_template_set_dij_exp
```

⟨*nlo data: fks template: TBP*⟩+≡

```
procedure :: set_xi_and_y_bounds => fks_template_set_xi_and_y_bounds
```

⟨*nlo data: procedures*⟩+≡

```
subroutine fks_template_set_xi_and_y_bounds (template, xi_min, y_max)
  class(fks_template_t), intent(inout) :: template
  real(default), intent(in) :: xi_min, y_max
  template%xi_min = xi_min
  template%y_max = y_max
end subroutine fks_template_set_xi_and_y_bounds
```

⟨*nlo data: fks template: TBP*⟩+≡

```
procedure :: set_mapping_type => fks_template_set_mapping_type
```

⟨*nlo data: procedures*⟩+≡

```
subroutine fks_template_set_mapping_type (template, val)
  class(fks_template_t), intent(inout) :: template
  integer, intent(in) :: val
  template%mapping_type = val
end subroutine fks_template_set_mapping_type
```

⟨*nlo data: fks template: TBP*⟩+≡

```
procedure :: set_counter => fks_template_set_counter
```

⟨*nlo data: procedures*⟩+≡

```
subroutine fks_template_set_counter (template)
  class(fks_template_t), intent(inout) :: template
  template%count_kinematics = .true.
end subroutine fks_template_set_counter
```

⟨*nlo data: public*⟩+≡

```
public :: real_scales_t
```

⟨*nlo data: types*⟩+≡

```
type :: real_scales_t
   real(default) :: scale
   real(default) :: ren_scale
   real(default) :: fac_scale
   real(default) :: scale_born
   real(default) :: fac_scale_born
   real(default) :: ren_scale_born
end type real_scales_t
```

⟨*nlo data: public*⟩+≡

```
public :: pdf_container_t
```

⟨*nlo data: types*⟩+≡

```
type :: pdf_container_t
   real(default), dimension(-6:6) :: f
end type pdf_container_t
```

⟨*nlo data: public*⟩+≡

```
public :: get_threshold_momenta
```

⟨*nlo data: procedures*⟩+≡

```
function get_threshold_momenta (p) result (p_thr)
  type(vector4_t), dimension(4) :: p_thr
  type(vector4_t), intent(in), dimension(:) :: p
  p_thr(1) = p(THR_POS_WP) + p(THR_POS_B)
  p_thr(2) = p(THR_POS_B)
  p_thr(3) = p(THR_POS_WM) + p(THR_POS_BBAR)
  p_thr(4) = p(THR_POS_BBAR)
end function get_threshold_momenta
```

⟨nlo_color_data.f90⟩≡

```
⟨File header⟩

module nlo_color_data

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use constants
  use numeric_utils
  use diagnostics
  use physics_defs
  use process_constants !NODEP!
  use sm_physics
  use os_interface
  use models
  use pdg_arrays
  use particle_specifiers
  use phs_base
  use phs_single
  use cascades
  use state_matrices
  use interactions
  use lorentz
  use prc_core
  use prc_user_defined
  use sf_base
  use colors
  use phs_fks
  use flavors
  use fks_regions
  use nlo_data
  use virtual
  use real_subtraction
  use dglap_remnant

⟨Standard module head⟩

⟨nlo color data: public⟩

⟨nlo color data: types⟩

contains

⟨nlo color data: procedures⟩

end module nlo_color_data
```

⟨*nlo color data: types*⟩≡

```
  type :: color_index_list_t
     integer, dimension(:), allocatable :: col
  end type color_index_list_t
```

We need to associate singular regions with compatible color flows.

⟨*nlo color data: types*⟩+≡

```
type :: ftuple_color_map_t
  integer :: index
  integer :: color_index
  type(ftuple_color_map_t), pointer :: next
  type(ftuple_color_map_t), pointer :: prev
contains
⟨nlo color data: color map: TBP⟩
end type ftuple_color_map_t
```

⟨*nlo color data: color map: TBP*⟩≡

```
procedure :: init => ftuple_color_map_init
```

⟨*nlo color data: procedures*⟩≡

```
subroutine ftuple_color_map_init (icm)
  class(ftuple_color_map_t), intent(inout), target :: icm
  icm%index = 0
  icm%color_index = 0
  nullify (icm%next)
  nullify (icm%prev)
end subroutine ftuple_color_map_init
```

Explain

⟨*nlo color data: color map: TBP*⟩+≡

```
procedure :: present => ftuple_color_map_present
```

⟨*nlo color data: procedures*⟩+≡

```
function ftuple_color_map_present (icm, color_index) result(pres)
  class(ftuple_color_map_t), intent(in), target :: icm
  integer, intent(in) :: color_index
  logical :: pres
  type(ftuple_color_map_t), pointer :: current
  pres = .false.
  select type (icm)
  type is (ftuple_color_map_t)
     current => icm
     do
        if (current%color_index == color_index) then
           pres = .true.
           exit
        else
           if (associated (current%next)) then
              current => current%next
           else
              exit
           end if
        end if
     end do
  end select
end function ftuple_color_map_present
```

Appends a color flow to the list

⟨*nlo color data: color map: TBP*⟩+≡

```
procedure :: append => ftuple_color_map_append
```

⟨*nlo color data: procedures*⟩+≡

```
subroutine ftuple_color_map_append (icm, val)
  class(ftuple_color_map_t), intent(inout), target :: icm
  integer, intent(in) :: val
  type(ftuple_color_map_t), pointer :: current
  select type (icm)
  type is (ftuple_color_map_t)
  if (.not. icm%present (val)) then
    if (icm%index == 0) then
      nullify(icm%next)
      icm%index = 1
      icm%color_index = val
    else
      current => icm
      do
        if (associated (current%next)) then
          current => current%next
        else
          allocate (current%next)
          nullify (current%next%next)
          current%next%prev => current
          current%next%index = current%index + 1
          current%next%color_index = val
          exit
        end if
      end do
    end if
  end if
  end select
end subroutine ftuple_color_map_append
```

⟨*nlo color data: color map: TBP*⟩+≡

```
procedure :: get_n_entries => ftuple_color_map_get_n_entries
```

⟨*nlo color data: procedures*⟩+≡

```
function ftuple_color_map_get_n_entries (icm) result(n_entries)
  class(ftuple_color_map_t), intent(in), target :: icm
  integer :: n_entries
  type(ftuple_color_map_t), pointer :: current
  n_entries = 0
  select type (icm)
  type is (ftuple_color_map_t)
     current => icm
     do
        if (associated (current%next)) then
           current => current%next
        else
           n_entries = current%index
           exit
        end if
     end do
  end select
end function ftuple_color_map_get_n_entries
```

Explain

⟨*nlo color data: color map: TBP*⟩+≡

```
procedure :: get_index_array => ftuple_color_map_get_index_array
```

⟨*nlo color data: procedures*⟩+≡

```
function ftuple_color_map_get_index_array (icm) result(iarr)
  class(ftuple_color_map_t), intent(in), target :: icm
  integer, dimension(:), allocatable :: iarr
  type(ftuple_color_map_t), pointer :: current
  integer :: n_entries
  integer :: i
  call msg_debug &
       (D_SUBTRACTION, "inside ftuple_color_map_get_index_array")
  select type (icm)
  type is (ftuple_color_map_t)
  n_entries = icm%get_n_entries ()
  call msg_debug (D_SUBTRACTION, "n_entries = ", n_entries)
  allocate (iarr(n_entries))
  do i = 1, n_entries
    if (i == 1) then
      current => icm
    else
      current => current%next
    end if
    iarr(i) = current%color_index
    if (debug_active (D_SUBTRACTION)) &
         print *, "i = ", i, "   iarr(i) = ", iarr(i)
  end do
  end select
end function ftuple_color_map_get_index_array
```

⟨*nlo color data: color map: TBP*⟩+≡

```
procedure :: get_entry => ftuple_color_map_get_entry
```

⟨*nlo color data: procedures*⟩+≡

```
function ftuple_color_map_get_entry (icm, index) result(entry)
  class(ftuple_color_map_t), intent(in), target :: icm
  integer, intent(in) :: index
  integer :: entry
  type(ftuple_color_map_t), pointer :: current
  integer :: i
  entry = 0
  select type (icm)
  type is (ftuple_color_map_t)
     if (index <= icm%get_n_entries ()) then
        do i = 1, icm%get_n_entries ()
           if (i == 1) then
              current => icm
           else
              current => current%next
           end if
           if (i == index) entry = current%color_index
        end do
     end if
```

```
    end select
  end function ftuple_color_map_get_entry
```

Explain

⟨*nlo color data: color map: TBP*⟩+≡

```
  procedure :: create_map => ftuple_color_map_create_map
```

⟨*nlo color data: procedures*⟩+≡

```
  recursive subroutine ftuple_color_map_create_map (icm, flst, &
       emitter, allreg, color_states_born, color_states_real, &
       included_color_structures, p_rad_in)
    class(ftuple_color_map_t), intent(inout) :: icm
    type(flv_structure_t), intent(in) :: flst
    integer, intent(in) :: emitter
    type(ftuple_t), intent(in), dimension(:) :: allreg
    integer, intent(in), dimension(:,:,:) :: color_states_born
    integer, intent(in), dimension(:,:,:) :: color_states_real
    integer, intent(in), dimension(:) :: included_color_structures
    integer, intent(in), optional :: p_rad_in
    integer :: nreg, region
    integer :: p1, p2, p_rad
    integer :: flv_em, flv_rad
    integer :: n_col_real
    integer, dimension(2) :: col_em, col_rad
    integer :: i
    !!! splitting type: 1 - q -> qg
    !!!                 2 - g -> qq
    !!!                 3 - g -> gg
    integer :: splitting_type_flv, splitting_type_col
    if (debug_active (D_SUBTRACTION)) then
       print *, "CREATING THE COLOR MAP"
       print *, " ... and the emitter is: ", emitter
       do i = 1, size (allreg)
          call allreg(i)%write ()
       end do
    end if
    nreg = size (allreg)
    call msg_debug (D_SUBTRACTION, "size of all regions = ", nreg)
    n_col_real = size (color_states_real (1,1,:))
    call msg_debug (D_SUBTRACTION, "number color flows real = ", n_col_real)
    do region = 1, nreg
      call allreg(region)%get (p1, p2)
      if (debug_active (D_SUBTRACTION)) then
         print *, "region = ", region
         print *, "   p1  = ", p1
         print *, "   p2  = ", p2
      end if
      if (p1 == emitter .or. p2 == emitter .or. present (p_rad_in)) then
        if (.not. present (p_rad_in)) then
          if (p1 == emitter) then
            p_rad = p2
          else
            p_rad = p1
          end if
```

```
else
  p_rad = p_rad_in
end if
if (emitter /= 0) then
  flv_em = flst%flst (emitter)
  call msg_debug (D_SUBTRACTION, "flv(emitter) = ", flv_em)
else
   !!! In the 3-jet example ee -> jjj this is never entered
  call icm%create_map &
       (flst, 1, allreg, color_states_born, color_states_real, &
       included_color_structures, p_rad)
  call icm%create_map &
       (flst, 2, allreg, color_states_born, color_states_real, &
       included_color_structures, p_rad)
  return
end if
flv_rad = flst%flst (p_rad)
call msg_debug (D_SUBTRACTION, "flv(radiated) = ", flv_rad)
!!! q -> qg or g -> qq
if (is_quark (flv_em) .and. is_gluon (flv_rad) .or. &
    is_gluon (flv_em) .and. is_quark (flv_rad)) then
   splitting_type_flv = 1
!!! q -> gq
else if (is_quark (flv_em) .and. flv_em + flv_rad == 0) then
   splitting_type_flv = 2
!!! g -> gg
else if (is_gluon (flv_em) .and. is_gluon (flv_rad)) then
   splitting_type_flv = 3
else
  splitting_type_flv = 0
end if
call msg_debug &
     (D_SUBTRACTION, "splitting_type = ", splitting_type_flv)
do i = 1, n_col_real
   if (.not. (any (i == included_color_structures))) cycle
   col_em = color_states_real(:,emitter,i)
   col_rad = color_states_real(:,p_rad,i)
  if (is_color_singlet (col_em(1), col_em(2)) &
      .and. (is_color_doublet (col_rad(1), col_rad(2)) &
      .or. is_color_ghost (col_rad(1), col_rad(2)))) then
    splitting_type_col = 1
  else if (is_color_singlet (col_em(1), col_em(2)) .and. &
           is_color_singlet (col_rad(1), col_rad(2))) then
    splitting_type_col = 2
  else if (is_color_doublet (col_em(1), col_em(2)) .and. &
           is_color_doublet (col_rad(1), col_rad(2))) then
    splitting_type_col = 3
  else
    splitting_type_col = 0
  end if
  call msg_debug (D_SUBTRACTION, &
       "splitting_type_col = ", splitting_type_col)
  if (splitting_type_flv == splitting_type_col .and. &
      splitting_type_flv /= 0) then
```

```
            call msg_debug (D_SUBTRACTION, &
                 "we are appending the color structure to region i = ", i)
            call icm%append (i)
          end if
        end do
      end if
    end do
  contains
    function is_color_singlet (c1, c2) result (singlet)
      integer, intent(in) :: c1, c2
      logical :: singlet
      singlet = (c1 == 0 .and. c2 /= 0) .or. (c1 /= 0 .and. c2 == 0)
    end function is_color_singlet
    function is_color_doublet (c1, c2) result (doublet)
      integer, intent(in) :: c1, c2
      logical :: doublet
      doublet = c1 /= 0 .and. c2 /= 0
    end function is_color_doublet
    function is_color_ghost (c1, c2) result (ghost)
      integer, intent(in) :: c1, c2
      logical :: ghost
      ghost = c1 == 0 .and. c2 == 0
    end function is_color_ghost
  end subroutine ftuple_color_map_create_map
```

This data type contains color information, necessary for both soft and virtual counterterms.

⟨*nlo color data: public*⟩≡

```
  public :: nlo_color_data_t
```

⟨*nlo color data: types*⟩+≡

```
  type nlo_color_data_t
    type(ftuple_color_map_t), dimension(:), allocatable :: icm
    integer, dimension(:,:,:), allocatable :: col_state_born, col_state_real
    logical, dimension(:,:), allocatable :: ghost_flag_born, ghost_flag_real
    integer :: n_col_born, n_col_real
    type(color_t), dimension(:,:), allocatable :: color_real
    integer, dimension(:), allocatable :: col_born
    complex(default), dimension(:), allocatable :: color_factors_born
    integer, dimension(:,:), allocatable :: cf_index_real
    real(default), dimension(:,:,:), allocatable :: beta_ij
    logical :: beta_ij_evaluated = .false.
    logical :: color_is_conserved = .false.
    integer, dimension(:), allocatable :: equivalent_color_up_to_sign
    integer, dimension(:), allocatable :: included_color_structures

  contains
  ⟨nlo color data: nlo color data: TBP⟩
  end type nlo_color_data_t
```

⟨*nlo color data: nlo color data: TBP*⟩≡

```
  procedure :: prepare => nlo_color_data_prepare
```

⟨*nlo color data: procedures*⟩+≡

```
subroutine nlo_color_data_prepare (color_data, reg_data, flavor_indices, color_indices)
  class(nlo_color_data_t), intent(inout) :: color_data
  type(region_data_t), intent(in) :: reg_data
  integer, intent(in), dimension(:) :: flavor_indices, color_indices
  integer :: i, i_born, n_flv
  integer, dimension(N_MAX_FLV) :: checked_uborns
  integer, dimension(:), allocatable :: i_flv, i_limit
  type(color_index_list_t), dimension(:), allocatable :: i_col
  call evaluate_flavors (n_flv, i_flv, i_limit)
  call evaluate_colors (n_flv, i_flv, i_limit, i_col)
  checked_uborns = 0
  do i = 1, size (i_flv)
     i_born = reg_data%map_real_to_born_index (i_flv(i))
     if (.not. any (checked_uborns == i_born)) then
        if (.not. allocated (color_data%included_color_structures)) then
           allocate (color_data%included_color_structures (size (i_col(i)%col)))
           color_data%included_color_structures = i_col(i)%col
        end if
        checked_uborns(i) = i_born
     end if
  end do
contains
  subroutine evaluate_flavors (n_flv, i_flavor, i_limit)
    integer, intent(out) :: n_flv
    integer, intent(out), dimension(:), allocatable :: i_flavor
    integer, intent(out), dimension(:), allocatable :: i_limit
    integer :: i, current_flavor
    n_flv = 0; current_flavor = 0
    do i = 1, size (flavor_indices)
       if (flavor_indices (i) /= current_flavor) then
          n_flv = n_flv + 1
          current_flavor = flavor_indices(i)
       end if
    end do
    allocate (i_flavor (n_flv), i_limit (n_flv))
    current_flavor = 0; i_limit(1) = 1
    do i = 1, size (flavor_indices)
       if (flavor_indices(i) /= current_flavor) then
         i_limit(current_flavor + 1) = i
         current_flavor = flavor_indices(i)
         i_flavor(current_flavor) = current_flavor
       end if
    end do
  end subroutine evaluate_flavors

  subroutine evaluate_colors (n_flv, i_flavor, col_limit, i_color)
    integer, intent(in) :: n_flv
    integer, intent(in), dimension(:) :: i_flavor, col_limit
    type(color_index_list_t), intent(out), dimension(:), allocatable :: i_color
    integer :: i, j, k, col_first
    integer :: count_n_col
    allocate (i_color (n_flv))
    j = 1
```

```
      do i = 1, n_flv
         col_first = color_indices(col_limit (i))
         count_n_col = 1
         k = col_limit(i) + 1
         do while (color_indices(k) /= col_first)
            count_n_col = count_n_col + 1
            k = k + 1
         end do
         allocate (i_color(i)%col(count_n_col))
      end do
      do i = 1, n_flv
         j = 1
         col_first = color_indices(col_limit (i))
         i_color(i)%col(1) = color_indices (col_limit(i))
         k = col_limit(i) + 1
         do while (color_indices(k) /= col_first)
            j = j + 1
            i_color(i)%col(j) = color_indices (k)
            k = k + 1
         end do
      end do
    end subroutine evaluate_colors

  end subroutine nlo_color_data_prepare
```

⟨*nlo color data: nlo color data: TBP*⟩+≡

```
  procedure :: init_color_matrix => nlo_color_data_init_color_matrix
```

⟨*nlo color data: procedures*⟩+≡

```
  subroutine nlo_color_data_init_color_matrix (color_data, reg_data, prc_constants, &
     factorization_mode)
    class(nlo_color_data_t), intent(inout) :: color_data
    type(region_data_t), intent(inout) :: reg_data
    type(process_constants_t), intent(in), dimension(2) :: prc_constants
    integer, intent(in) :: factorization_mode
    if (debug_active (D_SUBTRACTION))  call show_input_values ()
    call prc_constants(1)%get_col_state (color_data%col_state_born)
    call prc_constants(2)%get_col_state (color_data%col_state_real)
    call prc_constants(2)%get_cf_index (color_data%cf_index_real)
    call prc_constants(1)%get_color_factors (color_data%color_factors_born)
    color_data%n_col_born = size (color_data%col_state_born (1, 1, :))
    color_data%n_col_real = size (color_data%col_state_real (1 ,1, :))
    call color_data%init_ghost_flags (prc_constants)
    call color_data%init_color (reg_data)
    call color_data%check_equivalences ()
    call color_data%compute_betaij (reg_data, factorization_mode)
  contains
    subroutine show_input_values()
      integer :: i
      call msg_debug (D_SUBTRACTION, "color_data_init")
      call msg_debug (D_SUBTRACTION, "factorization_mode", factorization_mode)
      do i = 1, size(color_data%included_color_structures)
         call msg_debug (D_SUBTRACTION, "included_color_structures(i)", &
            color_data%included_color_structures(i))
```

```
      end do
    end subroutine show_input_values
  end subroutine nlo_color_data_init_color_matrix
```

⟨*nlo color data: nlo color data: TBP*⟩+≡

```
  procedure :: init_ghost_flags => nlo_color_data_init_ghost_flags
```

⟨*nlo color data: procedures*⟩+≡

```
  subroutine nlo_color_data_init_ghost_flags (color_data, prc_constants)
    class(nlo_color_data_t), intent(inout) :: color_data
    type(process_constants_t), intent(in), dimension(2) :: prc_constants
    call prc_constants(1)%get_ghost_flag (color_data%ghost_flag_born)
    call prc_constants(2)%get_ghost_flag (color_data%ghost_flag_real)
  end subroutine nlo_color_data_init_ghost_flags
```

⟨*nlo color data: nlo color data: TBP*⟩+≡

```
  procedure :: check_equivalences => nlo_color_data_check_equivalences
```

⟨*nlo color data: procedures*⟩+≡

```
  subroutine nlo_color_data_check_equivalences (color_data)
    class(nlo_color_data_t), intent(inout) :: color_data
    integer :: i_col1, i_col2
    if (allocated (color_data%equivalent_color_up_to_sign))  &
         deallocate (color_data%equivalent_color_up_to_sign)
    allocate (color_data%equivalent_color_up_to_sign (color_data%n_col_born))
    color_data%equivalent_color_up_to_sign (1) = 1
    do i_col1 = 1, color_data%n_col_born
       do i_col2 = 1, i_col1 - 1
          if (is_equivalent (color_data%col_state_born (:, :, i_col1), &
             color_data%col_state_born (:, :, i_col2))) then
             color_data%equivalent_color_up_to_sign (i_col1) = i_col2
          else
             color_data%equivalent_color_up_to_sign (i_col1) = i_col1
          end if
       end do
    end do
  contains
    function is_equivalent (col1, col2) result (equiv)
      logical :: equiv
      integer, intent(in), dimension(:,:) :: col1, col2
      integer :: leg
      integer, dimension(2) :: c1, c2, c3
      equiv = .true.
      do leg = 1, size (col1, dim = 2)
         c1 = col1(:, leg); c2 = col2(:, leg)
         c3(1) = -c2(2); c3(2) = -c2(1)
         equiv = equiv .and. (all (c1 == c2) .or. all (c1 == c3))
         if (.not. equiv) exit
      end do
    end function is_equivalent
  end subroutine nlo_color_data_check_equivalences
```

⟨*nlo color data: nlo color data: TBP*⟩+≡

```
  procedure :: init_color => nlo_color_data_init_color
```

⟨*nlo color data: procedures*⟩+≡

```
subroutine nlo_color_data_init_color (color_data, reg_data)
  class(nlo_color_data_t), intent(inout) :: color_data
  type(region_data_t), intent(in) :: reg_data
  integer :: i
  call msg_debug (D_SUBTRACTION, "nlo_color_data_init_color")
  if (allocated (color_data%color_real))  &
       deallocate (color_data%color_real)
  if (allocated (color_data%icm))  deallocate (color_data%icm)
  allocate (color_data%color_real (reg_data%n_legs_real, &
       color_data%n_col_real))
  allocate (color_data%icm (reg_data%n_regions))
  do i = 1, color_data%n_col_real
    call color_init_from_array (color_data%color_real (:, i), &
         color_data%col_state_real (:, :, i), &
         color_data%ghost_flag_real (:, i))
    call color_data%color_real (1 : reg_data%n_in, i)%invert ()
  end do
  do i = 1, reg_data%n_regions
    call color_data%icm(i)%init
    associate (region => reg_data%regions(i))
      if (debug_active (D_SUBTRACTION)) then
         print *, "index i = ", i
         print *, "creating map"
      end if
      call color_data%icm(i)%create_map (region%flst_real, region%emitter, &
           region%ftuples, color_data%col_state_born, &
           color_data%col_state_real, color_data%included_color_structures)
    end associate
  end do
end subroutine nlo_color_data_init_color
```

Allocate and compute β_{ij}:

⟨*nlo color data: nlo color data: TBP*⟩+≡

```
procedure :: compute_betaij => nlo_color_data_compute_betaij
```

⟨*nlo color data: procedures*⟩+≡

```
subroutine nlo_color_data_compute_betaij (color_data, reg_data, factorization_mode)
  class(nlo_color_data_t), intent(inout) :: color_data
  type(region_data_t), intent(inout) :: reg_data
  integer, intent(in) :: factorization_mode
  integer :: alr, i_born
  logical, dimension(reg_data%n_flv_born) :: checked_uborn
  if (debug_active (D_SUBTRACTION)) then
     call msg_print_color ("nlo_color_data_compute_betaij", COL_AQUA)
     call msg_print_color ("factorization_mode", factorization_mode, COL_AQUA)
  end if
  if (allocated (color_data%beta_ij)) &
       deallocate (color_data%beta_ij)
  allocate (color_data%beta_ij (reg_data%n_legs_born, &
       reg_data%n_legs_born, reg_data%n_flv_born))
  select case (factorization_mode)
  case (NO_FACTORIZATION)
     call msg_debug (D_SUBTRACTION, "No factorization")
```

```
      checked_uborn = .false.
      do alr = 1, reg_data%n_regions
         i_born = reg_data%regions(alr)%uborn_index
         if (checked_uborn (i_born)) cycle
         call color_data%fill_betaij_matrix (reg_data%n_legs_born, i_born, &
            reg_data%regions(alr)%flst_real, reg_data)
         checked_uborn (i_born) = .true.
      end do
   case (FACTORIZATION_THRESHOLD)
      call msg_debug (D_SUBTRACTION, "Threshold factorization")
      call color_data%fill_betaij_matrix_threshold ()
   end select
end subroutine nlo_color_data_compute_betaij
```

Actual computation of β_{ij}.

⟨*nlo color data: nlo color data: TBP*⟩+≡

```
procedure :: fill_betaij_matrix => nlo_color_data_fill_betaij_matrix
```

⟨*nlo color data: procedures*⟩+≡

```
subroutine nlo_color_data_fill_betaij_matrix &
     (color_data, n_legs, uborn_index, flst_real, reg_data)
   class(nlo_color_data_t), intent(inout) :: color_data
   integer, intent(in) :: n_legs, uborn_index
   type(flv_structure_t), intent(in) :: flst_real
   type(region_data_t), intent(inout) :: reg_data
   integer :: em1, em2
   if (debug_active (D_SUBTRACTION)) then
      print *, "FILLING BETA_IJ MATRIX"
      print *, "n_legs = ", n_legs
      print *, "uborn_index = ", uborn_index
   end if
   associate (flv_born => reg_data%flv_born (uborn_index))
      do em1 = 1, n_legs
         do em2 = 1, n_legs
            if (debug2_active (D_SUBTRACTION)) then
               print *, "em1 = ", em1, " em2 = ", em2
               print *, "col1/2 = ", flv_born%colored(em1), flv_born%colored(em2)
            end if
            if (flv_born%colored(em1) .and. flv_born%colored(em2)) then
               if (em1 < em2) then
                  if (debug2_active (D_SUBTRACTION)) then
                     print *, "calling compute_bij for em1 < em2"
                  end if
                  color_data%beta_ij (em1, em2, uborn_index) &
                     = color_data%compute_bij &
                     (reg_data, uborn_index, flst_real, em1, em2)
               else if (em1 > em2) then
                  !!! B_ij is symmetric
                  color_data%beta_ij (em1, em2, uborn_index) = &
                     color_data%beta_ij (em2, em1, uborn_index)
               else
                  if (is_quark (abs (flv_born%flst (em1)))) then
                      color_data%beta_ij (em1, em2, uborn_index) = - cf
                  else
```

```
                    color_data%beta_ij (em1, em2, uborn_index) = - ca
                 end if
              end if
           else
              color_data%beta_ij (em1, em2, uborn_index) = zero
           end if
        end do
     end do
  end associate
  color_data%beta_ij_evaluated = .true.
  call check_color_conservation (color_data%beta_ij (:, :, uborn_index), &
       n_legs, color_data%color_is_conserved)
  if (debug_active (D_SUBTRACTION)) then
     call msg_debug (D_SUBTRACTION, "nlo_color_data_fill_betaij_matrix")
     do em1 = 1, n_legs
        do em2 = 1, n_legs
           print *, 'em1, em2, color_data%beta_ij(em1,em2,uborn_index) = ', &
                em1, em2, color_data%beta_ij(em1, em2, uborn_index)
        end do
     end do
  end if
contains
  subroutine check_color_conservation (bij_matrix, n_legs, success)
    real(default), intent(in), dimension(:,:) :: bij_matrix
    integer, intent(in) :: n_legs
    logical, intent(out) :: success
    logical, dimension(:), allocatable :: check
    integer :: i, j
    real(default) :: bcheck
    real(default), parameter :: tol = 0.0001_default
    allocate (check (n_legs))
    do i = 1, n_legs
       bcheck = 0.0
       do j = 1, n_legs
          if (i /= j) bcheck = bcheck + bij_matrix (i, j)
       end do
       if (is_quark (abs(flst_real%flst (i))) .or. &
           is_gluon (flst_real%flst (i))) then
          if (is_quark (abs(flst_real%flst (i))) .and. &
               (bcheck - cf) < tol) then
             check (i) = .true.
          else if (is_gluon (flst_real%flst (i)) .and. &
               (bcheck - ca) < tol) then
             check (i) = .true.
          else
             check (i) = .false.
          end if
       else
          if (bcheck < tol) then
             check (i) = .true.
          else
             check (i) = .false.
          end if
       end if
```

```
      end do
      success = all (check)
    end subroutine check_color_conservation
  end subroutine nlo_color_data_fill_betaij_matrix
```

⟨*nlo color data: nlo color data: TBP*⟩+≡

```
  procedure :: fill_betaij_matrix_threshold &
     => nlo_color_data_fill_betaij_matrix_threshold
```

⟨*nlo color data: procedures*⟩+≡

```
  subroutine nlo_color_data_fill_betaij_matrix_threshold (color_data)
    class(nlo_color_data_t), intent(inout) :: color_data
    integer :: i, j
    color_data%beta_ij = zero
    associate (beta_ij => color_data%beta_ij (:,:,1))
       do i = 1, 4
          beta_ij (i,i) = -CF
       end do
       beta_ij (1,2) = CF; beta_ij (2,1) = CF
       beta_ij (3,4) = CF; beta_ij (4,3) = CF
    end associate
    if (debug_active (D_SUBTRACTION)) then
       call msg_debug (D_SUBTRACTION, "nlo_color_data_fill_betaij_matrix_threshold")
       do i = 1, size(color_data%beta_ij, dim=1)
          do j = 1, size(color_data%beta_ij, dim=1)
             print *, 'i, j, color_data%beta_ij(i,j,1) = ', &
                  i, j, color_data%beta_ij(i,j,1)
          end do
       end do
    end if
  end subroutine nlo_color_data_fill_betaij_matrix_threshold
```

⟨*nlo color data: nlo color data: TBP*⟩+≡

```
  procedure :: compute_bij => nlo_color_data_compute_bij
```

⟨*nlo color data: procedures*⟩+≡

```
  function nlo_color_data_compute_bij (color_data, reg_data, uborn_index, &
      flst_real, em1, em2) result (bij)
    real(default) :: bij
    class(nlo_color_data_t), intent(inout) :: color_data
    type(region_data_t), intent(inout) :: reg_data
    integer, intent(in) :: uborn_index
    type(flv_structure_t), intent(in) :: flst_real
    integer, intent(in) :: em1, em2
    logical, dimension(:,:), allocatable :: cf_present
    type(singular_region_t), dimension(2,100) :: reg
    integer ::  i, j, k, l
    integer :: alr, n_alr
    type(ftuple_color_map_t) :: icm1, icm2
    integer :: i1, i2
    real(default) :: color_factor, color_factor_born
    integer, dimension(2) :: i_reg
    logical , dimension(2) :: found
    integer, dimension(2,100) :: map_em_col_tmp
```

```
integer, dimension(:), allocatable :: map_em_col1, map_em_col2
integer, dimension(2) :: col1, col2
integer, dimension(:), allocatable :: iarray1, iarray2
integer, dimension(:), allocatable :: iisec1, iisec2
integer :: sign
integer, dimension(:), allocatable :: uborn_group, em1_group, em2_group
integer, dimension(:), allocatable :: compute_alrs
if (debug_active (D_SUBTRACTION)) then
   call color_data%write (verbose = .true.)
   print *, "em1 = ", em1
   print *, "em2 = ", em2
end if
bij = zero
color_factor = zero; color_factor_born = zero
found = .false.
allocate (uborn_group (reg_data%get_uborn_group_size (uborn_index)))
if (debug_active (D_SUBTRACTION)) &
     print *, "size (uborn_group) = ", size (uborn_group)
uborn_group = reg_data%get_uborn_group (uborn_index)
if (debug_active (D_SUBTRACTION)) &
     print *, "uborn_group = ", uborn_group
allocate (em1_group (reg_data%get_emitter_group_size (em1)))
em1_group = reg_data%get_emitter_group (em1)
if (debug_active (D_SUBTRACTION)) &
     print *, "em1_group = ", em1_group
allocate (em2_group (reg_data%get_emitter_group_size (em2)))
em2_group = reg_data%get_emitter_group (em2)
if (debug_active (D_SUBTRACTION)) &
     print *, "em2_group = ", em2_group
n_alr = 0
do i = 1, size (uborn_group)
   n_alr = n_alr + count (uborn_group(i) == em1_group) &
        + count (uborn_group(i) == em2_group)
end do
if (n_alr == 0) return
allocate (compute_alrs (n_alr))
compute_alrs = &
   pack (uborn_group, [(any (uborn_group(i) == em1_group) .or. any (uborn_group(i) == em2_grou
   i = 1, size (uborn_group))])
!!! Include distinction between Born flavors
do i = 1, size (color_data%color_factors_born)
   if (color_data%equivalent_color_up_to_sign (i) == i) &
      color_factor_born = color_factor_born + real (color_data%color_factors_born (i))
end do
i1 = 1; i2 = 1
if (debug_active (D_SUBTRACTION)) &
     print *, "Number of color flows in the real = ", color_data%n_col_real
do i = 1, color_data%n_col_real
   if (any (color_data%included_color_structures == i)) then
      col1 = color_data%col_state_real (:, em1, i)
      col2 = color_data%col_state_real (:, reg_data%n_legs_real, i)
      if (share_line (col1, col2)) then
         map_em_col_tmp(1, i1) = i
         i1 = i1 + 1
```

```
      end if
      col1 = color_data%col_state_real (:, em2, i)
      if (share_line (col1, col2)) then
         map_em_col_tmp(2, i2) = i
         i2 = i2 + 1
      end if
   end if
end do
if (debug_active (D_SUBTRACTION)) then
   print *, "i1 = ", i1
   print *, "i2 = ", i2
end if
allocate (map_em_col1 (i1), map_em_col2 (i2))
map_em_col1 = map_em_col_tmp (1, 1 : i1 - 1)
map_em_col2 = map_em_col_tmp (2, 1 : i2 - 1)

i_reg = 1

if (debug_active (D_SUBTRACTION)) &
     print *, "Number of regions = ", reg_data%n_regions

do alr = 1, reg_data%n_regions
    if (.not. any (compute_alrs == alr)) cycle
    if (em1 == reg_data%regions(alr)%emitter .or. &
       (em1 <= 2 .and. reg_data%regions(alr)%emitter == 0)) then
       reg(1, i_reg(1)) = reg_data%regions(alr)
       i_reg(1) = i_reg(1) + 1
       found(1) = .true.
    end if
    if (em2 == reg_data%regions(alr)%emitter .or. &
       (em2 <= 2 .and. reg_data%regions(alr)%emitter == 0)) then
       reg(2, i_reg(2)) = reg_data%regions(alr)
       i_reg(2) = i_reg(2) + 1
       found(2) = .true.
    end if
end do
if (.not. (found(1) .and. found(2))) then
   return
end if

if (debug_active (D_SUBTRACTION)) &
     print *, "i_reg = ", i_reg

do i = 1, i_reg(1) - 1
   do j = 1, i_reg(2) - 1
      if (debug_active (D_SUBTRACTION)) then
         print *, "i=", i, " reg(1,i)%alr = ", reg(1,i)%alr
         print *, "i=", i, " reg(2,i)%alr = ", reg(2,i)%alr
      end if
      icm1 = color_data%icm (reg(1, i)%alr)
      icm2 = color_data%icm (reg(2, j)%alr)

      allocate (iarray1 (size (icm1%get_index_array ())))
      allocate (iarray2 (size (icm2%get_index_array ())))
```

```
         if (debug_active (D_SUBTRACTION)) then
            print *, "iarray1 = ", iarray1
            print *, "iarray2 = ", iarray2
         end if

         iarray1 = icm1%get_index_array ()
         iarray2 = icm2%get_index_array ()

         allocate (iisec1 (count (iarray1 == map_em_col1)))
         allocate (iisec2 (count (iarray2 == map_em_col2)))

         iisec1 = pack (iarray1, [ (any(iarray1(i) == map_em_col1), &
              i = 1, size(iarray1)) ])
         iisec2 = pack (iarray2, [ (any(iarray2(i) == map_em_col2), &
              i = 1, size(iarray2)) ])

         allocate (cf_present (size (color_index_present &
            (color_data%cf_index_real), 1), size (color_index_present &
            (color_data%cf_index_real), 2)))

         cf_present = color_index_present (color_data%cf_index_real)

         do k = 1, size (iisec1)
            do l = 1, size (iisec2)
               i1 = iisec1(k)
               i2 = iisec2(l)
               if (cf_present (i1, i2)) then
                  if (is_gluon (flst_real%flst (em1)) .or. &
                     is_gluon (flst_real%flst (em2))) then
                     sign = get_sign (color_data%col_state_real (:, :, i1)) * &
                        get_sign (color_data%col_state_real (:, :, i2))
                  else
                     sign = 1
                  end if
                  color_factor = color_factor + sign * compute_color_factor &
                     (color_data%color_real(:, i1), &
                     color_data%color_real(:, i2))
               end if
            end do
         end do
      deallocate (iarray1, iarray2, iisec1, iisec2, cf_present)
      end do
   end do
   !!! The real color factor always differs from the Born one
   !!! by one vertex factor. Thus, apply the factor 1/2
   bij = color_factor / (two * color_factor_born)

contains
   function share_line (col1, col2) result (share)
     integer, intent(in), dimension(2) :: col1, col2
     logical :: share
     logical :: id1, id2, id3
     id1 = (abs(col1(1)) == abs(col2(1)) .and. col1(1) /= 0) .or. &
```

```
        (abs(col1(2)) == abs(col2(2)) .and. col1(2) /= 0)
  id2 = (abs(col1(1)) == abs(col2(2)) .and. col1(1) /= 0) .or. &
        (abs(col1(2)) == abs(col2(1)) .and. col1(2) /= 0)
  id3 = col2(1) == 0 .and. col2(2) == 0
  if (id1 .or. id2 .or. id3) then
    share = .true.
  else
    share = .false.
  end if
end function share_line

function get_sign (col) result (sign)
  integer, intent(in), dimension(:,:) :: col
  integer :: sign
  integer, dimension(:), allocatable :: iref, iperm
  integer :: iref1, iperm1
  integer :: n, i, i_first, j
  integer :: i1, i2
  integer :: p1, p2
  p1 = 2; p2 = 2
  iref1 = 0; iperm1 = 0; i_first = 0; i1 = 0
  do i = 1, size(col(1,:))
    if (.not. all (col(:,i) == 0)) then
      if (col(1,i) == 0) then
        i1 = col(2,i)
        iref1 = i; iperm1 = i
        i_first = i
      else
        i1 = col(1,i)
        iref1 = i; iperm1 = i
        i_first = i
      end if
      exit
    end if
  end do
  if (iref1 == 0 .or. iperm1 == 0 .or. i_first == 0) &
     call msg_fatal ("Invalid color structure")
  n = size(col(1,:)) - i_first + 1
  allocate (iref(n), iperm(n))
  iref(1) = iref1; iperm(1) = iperm1
  do i = i_first+1, size(col(1,:))
    if (all (col(:,i) == 0)) cycle
    if (i == size(col(1,:))) then
      iref(p1) = i_first + 1
    else
      iref(p1) = i + 1
      p1 = p1 + 1
    end if
    do j = i_first+1, size(col(1,:))
      if (col(1,j) == -i1) then
        i1 = col(2,j)
        iperm(p2) = j
        p2 = p2 + 1
        exit
```

```
        else if (col(2,j) == -i1) then
          i1 = col(1,j)
          iperm(p2) = j
          p2 = p2 + 1
          exit
        end if
      end do
    end do
    sign = 1
    do i = 1, n
      if (iperm(i) == iref(i)) then
        cycle
      else
        do j = i+1, n
          if (iperm(j) == iref(i)) then
            i1 = j
            exit
          end if
        end do
        i2 = iperm(i)
        iperm(i) = iperm(i1)
        iperm(i1) = i2
        sign = -sign
      end if
    end do
  end function get_sign

  function color_index_present (cf_index) result (cf_present)
    integer, intent(in), dimension(:,:), allocatable :: cf_index
    logical, dimension(:,:), allocatable :: cf_present
    integer :: n_col
    integer :: c, i1, i2
    n_col = size (cf_index(1,:))
    allocate (cf_present (n_col, n_col))
    cf_present = .false.
    do c = 1, n_col
      i1 = cf_index (1, c)
      i2 = cf_index (2, c)
      cf_present (i1, i2) = .true.
      if (i1 /= i2) cf_present(i2, i1) = .true.
    end do
  end function color_index_present
end function nlo_color_data_compute_bij
```

⟨*nlo color data: nlo color data: TBP*⟩+≡

```
procedure :: write => nlo_color_data_write
```

⟨*nlo color data: procedures*⟩+≡

```
subroutine nlo_color_data_write (color_data, unit, verbose)
  class(nlo_color_data_t), intent(in) :: color_data
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u, i, i1, i2
  integer :: n_legs, n_up
```

```
  logical :: print_second_off_diagonal
  u = given_output_unit (unit); if (u < 0) return
  print_second_off_diagonal = .false.
  if (present (verbose)) print_second_off_diagonal = verbose
  n_legs = size (color_data%beta_ij, dim = 2)
  n_up = n_legs
  write (u, "(1x,A)") "Color information: "
  write (u, "(1x,A,1x,I1)") "Number of Born color states: ", &
       color_data%n_col_born
  write (u, "(1x,A,1x,I1)") "Number of real color states: ", &
       color_data%n_col_real
  if (.not. color_data%beta_ij_evaluated) return
  write (u, "(1x,A)") "Color correlation: "
  do i = 1, size (color_data%beta_ij, dim = 3)
    write (u, "(1x,A,1x,I1)") "State nr. ", i
    write (u, "(1x,A)") "-------------"
    write (u, "(1x,A,1x,A,1x,A)") "i1", "i2", "color factor"
    do i1 = 1, n_legs
      if (.not. print_second_off_diagonal) n_up = i1
      do i2 = 1, n_up
        write (u, "(1x,I1,1x,I1,1x,F5.2)") &
             i1, i2, color_data%beta_ij (i1, i2, i)
      end do
    end do
    write (u, "(1x,A)") "========================================"
  end do
  if (color_data%color_is_conserved) then
    write (u, "(1x,A)") "Color is conserved."
  else
    write (u, "(1x,A)") "Fatal error: Color conversation is violated."
  end if
end subroutine nlo_color_data_write
```

27.6.1 Unit tests

⟨nlo_color_data_ut.f90⟩≡
⟨*File header*⟩

```
module nlo_color_data_ut
  use unit_tests
  use nlo_color_data_uti
```

⟨*Standard module head*⟩

⟨*nlo color data: public test*⟩

```
contains
```

⟨*nlo color data: test driver*⟩

```
end module nlo_color_data_ut
```

⟨nlo_color_data_uti.f90⟩≡
⟨*File header*⟩

```
module nlo_color_data_uti
  use physics_defs, only: NO_FACTORIZATION
  use nlo_color_data, only: nlo_color_data_t
  use fks_regions, only: region_data_t, setup_region_data_for_test
```

⟨*Standard module head*⟩

⟨*nlo color data: test declarations*⟩

```
contains
```

⟨*nlo color data: tests*⟩

```
end module nlo_color_data_uti
```

⟨*nlo color data: public test*⟩≡

```
  public :: nlo_color_data_test
```

⟨*nlo color data: test driver*⟩≡

```
  subroutine nlo_color_data_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
    call test(nlo_color_data_1, "nlo_color_data_1", &
       "Test creation of the beta_ij-matrix for FSR", &
       u, results)
    call test(nlo_color_data_2, "nlo_color_data_2", &
       "Test creation of the beta_ij-matrix for ISR", &
       u, results)
    call test(nlo_color_data_3, "nlo_color_data_3", &
       "Test creation of the beta_ij-matrix for ISR and multiple Born structures (Drell Yan)", &
       u, results)
  end subroutine nlo_color_data_test
```

⟨*nlo color data: test declarations*⟩≡

```
public :: nlo_color_data_1
```

⟨*nlo color data: tests*⟩≡

```
subroutine nlo_color_data_1 (u)
  integer, intent(in) :: u
  type(nlo_color_data_t) :: color_data
  integer :: n_legs_born, n_legs_real
  integer :: n_flv_born, n_flv_real
  integer :: n_col_born, n_col_real
  integer :: n_in
  integer, dimension(:,:), allocatable :: flv_born, flv_real
  type(region_data_t) :: reg_data
  write (u, "(A)") "* Test output: nlo_color_data_1"
  write (u, "(A)") "* Purpose: Test internal computation of color-correlated matrix elements"
  write (u, "(A)") "* in the color-flow scheme"
  write (u, "(A)")
  write (u, "(A)") "* Process: e- e+ -> u u~"

  n_legs_born = 4; n_legs_real = 5
  n_flv_born = 1; n_flv_real = 1
  n_col_born = 1; n_col_real = 2
  allocate (color_data%col_state_born (2, n_legs_born, n_col_born))
  associate (col_state_born => color_data%col_state_born)
     col_state_born (:, 1, 1) = [0, 0]
     col_state_born (:, 2, 1) = [0, 0]
     col_state_born (:, 3, 1) = [1, 0]
     col_state_born (:, 4, 1) = [0, -1]
  end associate
  allocate (color_data%col_state_real (2, n_legs_real, n_col_real))
  associate (col_state_real => color_data%col_state_real)
     col_state_real (:, 1, 1) = [0, 0]
     col_state_real (:, 2, 1) = [0, 0]
     col_state_real (:, 1, 2) = [0, 0]
     col_state_real (:, 2, 2) = [0, 0]
     col_state_real (:, 3, 1) = [1, 0]
     col_state_real (:, 4, 1) = [0, -2]
     col_state_real (:, 5, 1) = [2, -1]
     col_state_real (:, 3, 2) = [1, 0]
     col_state_real (:, 4, 2) = [0, -1]
     col_state_real (:, 5, 2) = [0, 0]
  end associate
  allocate (color_data%cf_index_real (2, n_col_real))
  color_data%cf_index_real(:, 1) = [1, 1]
  color_data%cf_index_real(:, 2) = [2, 2]
  allocate (color_data%color_factors_born (n_col_born))
  color_data%color_factors_born (1) = cmplx (3, 0)
  color_data%n_col_born = n_col_born
  color_data%n_col_real = n_col_real

  n_in = 2
  allocate (flv_born (n_legs_born, n_flv_born))
  allocate (flv_real (n_legs_real, n_flv_real))
  flv_born (:, 1) = [11, -11, 2, -2]
  flv_real (:, 1) = [11, -11, 2, -2, 21]
```

```
    call setup_region_data_for_test (n_in, flv_born, flv_real, reg_data)
    allocate (color_data%ghost_flag_born (n_legs_born, n_col_born))
    allocate (color_data%ghost_flag_real (n_legs_real, n_col_real))
    color_data%ghost_flag_born (:, 1) = .false.
    color_data%ghost_flag_real (:, 1) = .false.
    color_data%ghost_flag_real (1 : n_legs_born, 2) = .false.
    color_data%ghost_flag_real (n_legs_real, 2) = .true.
    allocate (color_data%included_color_structures (2))
    color_data%included_color_structures = [1, 2]
    call color_data%init_color (reg_data)
    call color_data%check_equivalences ()
    call color_data%compute_betaij (reg_data, NO_FACTORIZATION)
    call color_data%write (u, verbose = .true.)
  end subroutine nlo_color_data_1
```

⟨*nlo color data: test declarations*⟩+≡

```
  public :: nlo_color_data_2
```

⟨*nlo color data: tests*⟩+≡

```
  subroutine nlo_color_data_2 (u)
    integer, intent(in) :: u
    type(nlo_color_data_t) :: color_data
    integer :: n_legs_born, n_legs_real
    integer :: n_flv_born, n_flv_real
    integer :: n_col_born, n_col_real
    integer :: n_in
    integer, dimension(:,:), allocatable :: flv_born, flv_real
    type(region_data_t) :: reg_data
    integer :: i
    write (u, "(A)") "* Test output: nlo_color_data_2"
    write (u, "(A)") "* Purpose: Test internal computation of color-correlated matrix elements"
    write (u, "(A)") "* in the color-flow scheme"
    write (u, "(A)")
    write (u, "(A)") "* Process: u u~ -> Z Z"

    n_legs_born = 4; n_legs_real = 5
    n_flv_born = 1; n_flv_real = 3
    n_col_born = 1; n_col_real = 6
    allocate (color_data%col_state_born (2, n_legs_born, n_col_born))
    associate (col_state_born => color_data%col_state_born)
       col_state_born (:, 1, 1) = [1, 0]
       col_state_born (:, 2, 1) = [0, -1]
       col_state_born (:, 3, 1) = [0, 0]
       col_state_born (:, 4, 1) = [0, 0]
    end associate
    allocate (color_data%col_state_real (2, n_legs_real, n_col_real))
    associate (col_state_real => color_data%col_state_real)
       col_state_real (:, 1, 1) = [0, 0]
       col_state_real (:, 2, 1) = [0, -2]
       col_state_real (:, 3, 1) = [0, 0]
       col_state_real (:, 4, 1) = [0, 0]
       col_state_real (:, 5, 1) = [0, -2]
       col_state_real (:, 1, 2) = [1, 0]
       col_state_real (:, 2, 2) = [0, -1]
```

```
    col_state_real (:, 3, 2) = [0, 0]
    col_state_real (:, 4, 2) = [0, 0]
    col_state_real (:, 5, 2) = [0, 0]
    col_state_real (:, 1, 3) = [1, 0]
    col_state_real (:, 2, 3) = [2, -1]
    col_state_real (:, 3, 3) = [0, 0]
    col_state_real (:, 4, 3) = [0, 0]
    col_state_real (:, 5, 3) = [2, 0]
    col_state_real (:, 1, 4) = [2, -1]
    col_state_real (:, 2, 4) = [0, -2]
    col_state_real (:, 3, 4) = [0, 0]
    col_state_real (:, 4, 4) = [0, 0]
    col_state_real (:, 5, 4) = [0, -1]
    col_state_real (:, 1, 5) = [2, 0]
    col_state_real (:, 2, 5) = [0, 0]
    col_state_real (:, 3, 5) = [0, 0]
    col_state_real (:, 4, 5) = [0, 0]
    col_state_real (:, 5, 5) = [2, 0]
    col_state_real (:, 1, 6) = [2, 0]
    col_state_real (:, 2, 6) = [0, -1]
    col_state_real (:, 3, 6) = [0, 0]
    col_state_real (:, 4, 6) = [0, 0]
    col_state_real (:, 5, 6) = [2, -1]
end associate
allocate (color_data%cf_index_real (2, n_col_real))
do i = 1, n_col_real
   color_data%cf_index_real(:, i) = [i, i]
end do
allocate (color_data%color_factors_born (n_col_born))
color_data%color_factors_born (1) = cmplx (3, 0)
color_data%n_col_born = n_col_born
color_data%n_col_real = n_col_real

n_in = 2
allocate (flv_born (n_legs_born, n_flv_born))
allocate (flv_real (n_legs_real, n_flv_real))
flv_born (:, 1) = [2, -2, 23, 23]
flv_real (:, 1) = [2, -2, 23, 23, 21]
flv_real (:, 2) = [2, 21, 23, 23, 2]
flv_real (:, 3) = [21, -2, 23, 23, -2]
call setup_region_data_for_test (n_in, flv_born, flv_real, reg_data)
allocate (color_data%ghost_flag_born (n_legs_born, n_col_born))
allocate (color_data%ghost_flag_real (n_legs_real, n_col_real))
color_data%ghost_flag_born (:, 1) = .false.
color_data%ghost_flag_real (1, 1) = .true.
color_data%ghost_flag_real (2 : n_legs_real, 1) = .false.
color_data%ghost_flag_real (1 : n_legs_born, 2) = .false.
color_data%ghost_flag_real (n_legs_real, 2) = .true.
color_data%ghost_flag_real (:, 3) = .false.
color_data%ghost_flag_real (:, 4) = .false.
color_data%ghost_flag_real (1, 5) = .false.
color_data%ghost_flag_real (2, 5) = .false.
color_data%ghost_flag_real (3 : n_legs_real, 5) = .true.
color_data%ghost_flag_real (:, 6) = .false.
```

```
    allocate (color_data%included_color_structures (2))
    color_data%included_color_structures = [2, 6]
    call color_data%init_color (reg_data)
    call color_data%check_equivalences ()
    call color_data%compute_betaij (reg_data, NO_FACTORIZATION)
    call color_data%write (u, verbose = .true.)
   end subroutine nlo_color_data_2
```

⟨*nlo color data: test declarations*⟩+≡

```
   public :: nlo_color_data_3
```

⟨*nlo color data: tests*⟩+≡

```
   subroutine nlo_color_data_3 (u)
     integer, intent(in) :: u
     type(nlo_color_data_t) :: color_data
     integer :: n_legs_born, n_legs_real
     integer :: n_flv_born, n_flv_real
     integer :: n_col_born, n_col_real
     integer :: n_in
     integer, dimension(:,:), allocatable :: flv_born, flv_real
     type(region_data_t) :: reg_data
     integer :: i_col, i_flv, pdg1, pdg2
     write (u, "(A)") "* Test output: nlo_color_data_2"
     write (u, "(A)") "* Purpose: Test internal computation of color-correlated matrix elements"
     write (u, "(A)") "* in the color-flow scheme"
     write (u, "(A)")
     write (u, "(A)") "* Process: u u~ -> Z Z"

     n_legs_born = 4; n_legs_real = 5
     n_flv_born = 10; n_flv_real = 30
     n_col_born = 2; n_col_real = 12
     allocate (color_data%col_state_born (2, n_legs_born, n_col_born))
     associate (col_state_born => color_data%col_state_born)
        do i_col = 1, n_col_born
           col_state_born (:, 1, i_col) = [1, 0]
           col_state_born (:, 2, i_col) = [0, -1]
           col_state_born (:, 3, i_col) = [0, 0]
           col_state_born (:, 4, i_col) = [0, 0]
        end do
     end associate
     allocate (color_data%col_state_real (2, n_legs_real, n_col_real))
     associate (col_state_real => color_data%col_state_real)
        do i_col = 1, n_col_real, 6
           col_state_real (:, 1, i_col) = [0, 0]
           col_state_real (:, 2, i_col) = [0, -2]
           col_state_real (:, 3, i_col) = [0, 0]
           col_state_real (:, 4, i_col) = [0, 0]
           col_state_real (:, 5, i_col) = [0, -2]
           col_state_real (:, 1, i_col + 1) = [1, 0]
           col_state_real (:, 2, i_col + 1) = [0, -1]
           col_state_real (:, 3, i_col + 1) = [0, 0]
           col_state_real (:, 4, i_col + 1) = [0, 0]
           col_state_real (:, 5, i_col + 1) = [0, 0]
           col_state_real (:, 1, i_col + 2) = [1, 0]
```

```
      col_state_real (:, 2, i_col + 2) = [2, -1]
      col_state_real (:, 3, i_col + 2) = [0, 0]
      col_state_real (:, 4, i_col + 2) = [0, 0]
      col_state_real (:, 5, i_col + 2) = [2, 0]
      col_state_real (:, 1, i_col + 3) = [2, -1]
      col_state_real (:, 2, i_col + 3) = [0, -2]
      col_state_real (:, 3, i_col + 3) = [0, 0]
      col_state_real (:, 4, i_col + 3) = [0, 0]
      col_state_real (:, 5, i_col + 3) = [0, -1]
      col_state_real (:, 1, i_col + 4) = [2, 0]
      col_state_real (:, 2, i_col + 4) = [0, 0]
      col_state_real (:, 3, i_col + 4) = [0, 0]
      col_state_real (:, 4, i_col + 4) = [0, 0]
      col_state_real (:, 5, i_col + 4) = [2, 0]
      col_state_real (:, 1, i_col + 5) = [2, 0]
      col_state_real (:, 2, i_col + 5) = [0, -1]
      col_state_real (:, 3, i_col + 5) = [0, 0]
      col_state_real (:, 4, i_col + 5) = [0, 0]
      col_state_real (:, 5, i_col + 5) = [2, -1]
   end do
end associate
allocate (color_data%cf_index_real (2, n_col_real))
do i_col = 1, n_col_real
   color_data%cf_index_real(:, i_col) = [i_col, i_col]
end do
allocate (color_data%color_factors_born (n_col_born))
color_data%color_factors_born (:) = cmplx (3, 0)
color_data%n_col_born = n_col_born
color_data%n_col_real = n_col_real

n_in = 2
allocate (flv_born (n_legs_born, n_flv_born))
allocate (flv_real (n_legs_real, n_flv_real))
pdg1 = -5; pdg2 = 5
do i_flv = 1, n_flv_born
   if (pdg1 == 0 .and. pdg2 == 0) then
      pdg1 = pdg1 + 1; pdg2 = pdg2 - 1
   end if
   flv_born (:, i_flv) = [pdg1, pdg2, 23, 23]
   pdg1 = pdg1 + 1; pdg2 = pdg2 - 1
end do
pdg1 = -5; pdg2 = 5
do i_flv = 1, n_flv_real, 3
   if (pdg1 == 0 .and. pdg2 == 0) then
      pdg1 = pdg1 + 1; pdg2 = pdg2 - 1
   end if
   flv_real (:, i_flv) = [pdg1, pdg2, 23, 23, 21]
   flv_real (:, i_flv + 1) = [pdg1, 21, 23, 23, pdg1]
   flv_real (:, i_flv + 2) = [21, pdg2, 23, 23, pdg2]
   pdg1 = pdg1 + 1; pdg2 = pdg2 - 1
end do
call setup_region_data_for_test (n_in, flv_born, flv_real, reg_data)
allocate (color_data%ghost_flag_born (n_legs_born, n_col_born))
allocate (color_data%ghost_flag_real (n_legs_real, n_col_real))
```

```
    do i_col = 1, n_col_born
       color_data%ghost_flag_born (:, i_col) = .false.
    end do
    do i_col = 1, n_col_real, 6
       color_data%ghost_flag_real (1, i_col) = .true.
       color_data%ghost_flag_real (2 : n_legs_real, i_col) = .false.
       color_data%ghost_flag_real (1 : n_legs_born, i_col + 1) = .false.
       color_data%ghost_flag_real (n_legs_real, i_col + 1) = .true.
       color_data%ghost_flag_real (:, i_col + 2) = .false.
       color_data%ghost_flag_real (:, i_col + 3) = .false.
       color_data%ghost_flag_real (1, i_col + 4) = .false.
       color_data%ghost_flag_real (2, i_col + 4) = .false.
       color_data%ghost_flag_real (3 : n_legs_real, i_col + 4) = .true.
       color_data%ghost_flag_real (:, i_col + 5) = .false.
    end do
    allocate (color_data%included_color_structures (2))
    color_data%included_color_structures = [2, 6]
    call color_data%init_color (reg_data)
    call color_data%check_equivalences ()
    call color_data%compute_betaij (reg_data, NO_FACTORIZATION)
    call color_data%write (u, verbose = .true.)
   end subroutine nlo_color_data_3
```

27.6.2 Putting it together

⟨*nlo data: public*⟩+≡

```
   public :: nlo_settings_t
```

⟨*nlo data: types*⟩+≡

```
   type :: nlo_settings_t
      logical :: use_internal_color_correlations = .true.
      logical :: use_internal_spin_correlations = .false.
      logical :: use_resonance_mappings = .false.
      logical :: combined_integration = .false.
      logical :: fixed_order_nlo = .false.
      logical :: test_soft_limit = .false.
      logical :: test_coll_limit = .false.
      logical :: test_anti_coll_limit = .false.
      integer, dimension(:), allocatable :: selected_alr
      integer :: factorization_mode = NO_FACTORIZATION
      !!! Probably not the right place for this. Revisit after refactoring
      real(default) :: powheg_damping_scale = zero
      type(fks_template_t) :: fks_template
      type(string_t) :: virtual_selection
      logical :: virtual_resonance_aware_collinear = .true.
      logical :: use_born_scale = .true.
      logical :: cut_all_sqmes = .true.
   contains
   ⟨nlo data: nlo settings: TBP⟩
   end type nlo_settings_t
```

⟨*nlo data: nlo settings: TBP*⟩≡

```
  procedure :: init => nlo_settings_init
```

⟨*nlo data: procedures*⟩+≡

```
subroutine nlo_settings_init (nlo_settings, var_list, fks_template)
  class(nlo_settings_t), intent(inout) :: nlo_settings
  type(var_list_t), intent(in) :: var_list
  type(fks_template_t), intent(in), optional :: fks_template
  type(string_t) :: color_method
  if (present (fks_template)) nlo_settings%fks_template = fks_template
  color_method = var_list%get_sval (var_str ('$correlation_me_method'))
  nlo_settings%use_internal_color_correlations = color_method == 'omega' &
         .or. color_method == 'threshold'
  nlo_settings%combined_integration = var_list%get_lval &
         (var_str ("?combined_nlo_integration"))
  nlo_settings%fixed_order_nlo = var_list%get_lval &
         (var_str ("?fixed_order_nlo_events"))
  nlo_settings%test_soft_limit = var_list%get_lval (var_str ('?test_soft_limit'))
  nlo_settings%test_coll_limit = var_list%get_lval (var_str ('?test_coll_limit'))
  nlo_settings%test_anti_coll_limit = var_list%get_lval (var_str ('?test_anti_coll_limit'))
  call setup_alr_selection ()
  nlo_settings%virtual_selection = var_list%get_sval (var_str ('$virtual_selection'))
  nlo_settings%virtual_resonance_aware_collinear = &
       var_list%get_lval (var_str ('?virtual_collinear_resonance_aware'))
  nlo_settings%powheg_damping_scale = &
       var_list%get_rval (var_str ('powheg_damping_scale'))
  nlo_settings%use_born_scale = &
       var_list%get_lval (var_str ("?nlo_use_born_scale"))
  nlo_settings%cut_all_sqmes = &
       var_list%get_lval (var_str ("?nlo_cut_all_sqmes"))
contains
  subroutine setup_alr_selection ()
    type(string_t) :: alr_selection
    type(string_t), dimension(:), allocatable :: alr_split
    integer :: i, i1, i2
    alr_selection = var_list%get_sval (var_str ('$select_alpha_regions'))
    if (string_contains_word (alr_selection, var_str (","))) then
       call split_string (alr_selection, var_str (","), alr_split)
       allocate (nlo_settings%selected_alr (size (alr_split)))
       do i = 1, size (alr_split)
          nlo_settings%selected_alr(i) = read_ival(alr_split(i))
       end do
    else if (string_contains_word (alr_selection, var_str (":"))) then
       call split_string (alr_selection, var_str (":"), alr_split)
       if (size (alr_split) == 2) then
          i1 = read_ival (alr_split(1))
          i2 = read_ival (alr_split(2))
          allocate (nlo_settings%selected_alr (i2 - i1 + 1))
          do i = 1, i2 - i1 + 1
             nlo_settings%selected_alr(i) = read_ival (alr_split(i))
          end do
       else
          call msg_fatal ("select_alpha_regions: ':' specifies a range!")
       end if
    else if (len(alr_selection) == 1) then
       allocate (nlo_settings%selected_alr (1))
       nlo_settings%selected_alr(1) = read_ival (alr_selection)
```

```
      end if
      if (allocated (alr_split)) deallocate (alr_split)
    end subroutine setup_alr_selection
  end subroutine nlo_settings_init
```

⟨*nlo data: nlo settings: TBP*⟩+≡

```
  procedure :: write => nlo_settings_write
```

⟨*nlo data: procedures*⟩+≡

```
  subroutine nlo_settings_write (nlo_settings, unit)
    class(nlo_settings_t), intent(in) :: nlo_settings
    integer, intent(in), optional :: unit
    integer :: i, u
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, '(A)') 'nlo_settings:'
    write (u, '(3X,A,L1)') 'internal_color_correlations = ', &
         nlo_settings%use_internal_color_correlations
    write (u, '(3X,A,L1)') 'internal_spin_correlations = ', &
         nlo_settings%use_internal_spin_correlations
    write (u, '(3X,A,L1)') 'use_resonance_mappings = ', &
         nlo_settings%use_resonance_mappings
    write (u, '(3X,A,L1)') 'combined_integration = ', &
         nlo_settings%combined_integration
    write (u, '(3X,A,L1)') 'test_soft_limit = ', &
         nlo_settings%test_soft_limit
    write (u, '(3X,A,L1)') 'test_coll_limit = ', &
         nlo_settings%test_coll_limit
    write (u, '(3X,A,L1)') 'test_anti_coll_limit = ', &
         nlo_settings%test_anti_coll_limit
    if (allocated (nlo_settings%selected_alr)) then
       write (u, '(3x,A)', advance = "no") 'selected alpha regions = ['
       do i = 1, size (nlo_settings%selected_alr)
          write (u, '(A,I0)', advance = "no") ",", nlo_settings%selected_alr(i)
       end do
       write (u, '(A)') "]"
    end if
    write (u, '(3X,A,' // FMT_15 // ')') 'powheg_damping_scale = ', &
         nlo_settings%powheg_damping_scale
    write (u, '(3X,A,A)') 'virtual_selection = ', &
         char (nlo_settings%virtual_selection)
    write (u, '(3X,A,A)') 'Real factorization mode = ', &
         char (factorization_mode (nlo_settings%factorization_mode))
  contains
    function factorization_mode (fm)
      type(string_t) :: factorization_mode
      integer, intent(in) :: fm
      select case (fm)
      case (NO_FACTORIZATION)
         factorization_mode = var_str ("None")
      case (FACTORIZATION_THRESHOLD)
         factorization_mode = var_str ("Threshold")
      case default
         factorization_mode = var_str ("Undefined!")
      end select
```

```
  end function factorization_mode
end subroutine nlo_settings_write
```

27.7 Contribution of divergencies due to PDF Evolution

The origin of these contributions is a bit tricky to understand, because we cannot understand the initial-state real collinear contributions without also taking into account the evolution of parton density functions at $\mathcal{O}(\alpha_s)$. The NLO PDF evolution is given by

$$f(\bar{x}) = \int_0^1 \int_0^1 dx dz f(x)\Gamma(z)\delta(\bar{x} - xz), \tag{27.20}$$

where Γ are the DGLAP evolution kernels for an $a \to d$ splitting,

$$\Gamma_a^{(d)} = \delta_{ad}\delta(1-x) - \frac{\alpha_s}{2\pi}\left(\frac{1}{\epsilon}P_{ad}(x,0) - K_{ad}(x)\right) + \mathcal{O}(\alpha_s). \tag{27.21}$$

K_{ad} is a renormalization scheme matching factor, which is exactly zero in $\bar{MS}$. Let the leading-order hadronic cross section be given by

$$d\sigma^{(0)}(s) = \int dx_\oplus dx_\ominus f_\oplus(x_\oplus) f_\ominus(x_\ominus) d\tilde{\sigma}^{(0)}(x_\oplus x_\ominus s), \tag{27.22}$$

then the NLO hadronic cross section is

$$d\sigma^{(1)}(s) = \int dx_\oplus dx_\ominus dz_\oplus dz_\ominus f_\oplus(x_\oplus) f_\ominus(x_\ominus) \underbrace{\Gamma_\oplus(z_\oplus)\Gamma_\ominus(z_\ominus) d\tilde{\sigma}^{(1)}(z_\oplus z_\ominus s)}_{d\hat{\sigma}^{(1)}}. \tag{27.23}$$

$d\hat{\sigma}$ is called the subtracted partonic cross section. Expanding in α_s we find

$$d\hat{\sigma}_{ab}^{(0)}(k_1,k_2) = d\tilde{\sigma}_{ab}^{(0)}(k_1,k_2), \tag{27.24}$$

$$d\hat{\sigma}_{ab}^{(1)}(k_1,k_2) = d\tilde{\sigma}_{ab}^{(1)}(k_1,k_2) \tag{27.25}$$

$$+ \frac{\alpha_s}{2\pi}\sum_d \int dx \left(\frac{1}{\epsilon}P_{da}(x,0) - K_{da}(x)\right) d\tilde{\sigma}_{db}^{(0)}(xk_1,k_2) \tag{27.26}$$

$$+ \frac{\alpha_s}{2\pi}\sum_d \int \left(\frac{1}{\epsilon}P_{db}(x,0) - K_{db}(x)\right) d\tilde{\sigma}_{ad}^{(0)}(k_1,xk_2). \tag{27.27}$$

$$= d\tilde{\sigma}_{ab}^{(1)} + d\tilde{\sigma}_{ab}^{(cnt,+)} + d\tilde{\sigma}_{ab}^{(cnt,-)} \tag{27.28}$$

Let us now turn the soft-subtracted real part of the cross section. For ease of notation, it is constrained to one singular region,

$$d\sigma_\alpha^{(in)} = \left[\left(\frac{1}{\xi}\right)_+ - 2\epsilon\left(\frac{\log\xi}{\xi}\right)_+\right](1-y^2)\xi^2\mathcal{R}_\alpha\mathcal{S}_\alpha$$
$$\times \frac{1}{2(2\pi)^{3-2\epsilon}}\left(\frac{\sqrt{s}}{2}\right)^{2-2\epsilon}\left(1-y^2\right)^{-1-\epsilon} d\phi d\xi dy d\Omega^{2-2\epsilon},$$

where we regularize collinear divergencies using the identity

$$\left(1-y^2\right)^{-1-\epsilon} = -\frac{2^{-\epsilon}}{\epsilon}\left(\delta(1-y) + \delta(1+y)\right) + \underbrace{\frac{1}{2}\left[\left(\frac{1}{1-y}\right)_+ + \left(\frac{1}{1+y}\right)_+\right]}_{\mathcal{P}(y)}.$$

This enables us to split the cross section into a finite and a singular part. The latter can further be separated into a contribution of the incoming and of the outgoing particles,

$$d\sigma_\alpha^{(in)} = d\sigma_\alpha^{(in,+)} + d\sigma_\alpha^{(in,-)} + d\sigma_\alpha^{(in,f)}.$$

They are given by

$$d\sigma_\alpha^{(in,f)} = \mathcal{P}(y)\left[\left(\frac{1}{\xi}\right)_+ - 2\epsilon\left(\frac{\log\xi}{\xi}\right)_+\right]\frac{1}{2(2\pi)^{3-2\epsilon}}\left(\frac{\sqrt{s}}{2}\right)^{2-2\epsilon} \tag{27.29}$$

$$\times (1-y^2)\xi^2\mathcal{R}_\alpha\mathcal{S}_\alpha d\phi d\xi dy d\Omega^{2-2\epsilon} \tag{27.30}$$

and

$$d\sigma_\alpha^{(in,\pm)} = -\frac{2^{-\epsilon}}{\epsilon}\delta(1\mp y)\left[\left(\frac{1}{\xi}\right)_+ - 2\epsilon\left(\frac{\log\xi}{\xi}\right)_+\right] \tag{27.31}$$

$$\times \frac{1}{2(2\pi)^{3-2\epsilon}}\left(\frac{\sqrt{s}}{2}\right)^{2-2\epsilon}(1-y^2)\xi^2\mathcal{R}_\alpha\mathcal{S}_\alpha d\phi d\xi dy d\Omega^{2-2\epsilon}. \tag{27.32}$$

Equation 27.29 is the contribution to the real cross section which is computed in `evaluate_region_isr`. It is regularized both in the soft and collinear limit via the plus distributions. Equation 27.31 is a different contribution. It is only present exactly in the collinear limit, due to the delta function. The divergences present in this term do not completely cancel out divergences in the virtual matrix element, because the beam axis is distinguished. Thus, the conditions in which the KLM theorem applies are not met. To see this, we carry out the collinear limit, obtaining

$$\lim_{y\to 1}(1-y^2)\xi^2\mathcal{R}_\alpha = 8\pi\alpha_s\mu^{2\epsilon}\left(\frac{2}{\sqrt{s}}\right)^2\xi P^<(1-\xi,\epsilon)\mathcal{R}_\alpha,$$

with the Altarelli-Parisi splitting kernel for $z<1$, $P^<(z,\epsilon)$. Moreover, $\lim_{\vec{k}\|\vec{k}_1} d\phi = d\phi_3$ violates spatial averaging. The integration over the spherical angle $d\Omega$ can be carried out easily, yielding a factor of $2\pi^{1-\epsilon}/\Gamma(1-\epsilon)$. This allows us to redefine ϵ,

$$\frac{1}{\epsilon} - \gamma_E + \log(4\pi) \to \frac{1}{\epsilon}. \tag{27.33}$$

In order to make a connection to $d\tilde{\sigma}^{(cnt,\pm)}$, we relate $P_{ab}(z,0)$ to $P_{ab}^<(z,0)$ via the equation

$$P_{ab}(z,0) = (1-z)P_{ab}^<(z,0)\left(\frac{1}{1-z}\right)_+ + \gamma(a)\delta_{ab}\delta(1-z),$$

which yields

$$d\tilde{\sigma}^{(cnt,+)} = \frac{\alpha_s}{2\pi}\sum_d\left\{-K_{da}(1-\xi) + \frac{1}{\epsilon}\left[\left(\frac{1}{\xi}\right)_+\xi P_{da}^<(1-\xi,0) + \delta_{da}\delta(\xi)\gamma(d)\right]\right\}\mathcal{R}_\alpha\mathcal{S}_\alpha. \tag{27.34}$$

This term has the same pole structure as eqn. 27.31. This makes clear that the quantity

$$d\hat{\sigma}^{(in,+)} = d\tilde{\sigma}^{(in,+)} + \frac{1}{4}d\tilde{\sigma}^{(cnt,+)} \tag{27.35}$$

has no collinear poles. Therefore, our task is to add up eqns. 27.31 and **??** in order to compute the finite remainder. This is the integrand which is evaluated in the `dglap_remnant` component.
So, we have to perform an expansion of $d\hat{\sigma}^{(in,+)}$ in ϵ. Hereby, we must not neglect the implicit ϵ-dependence of $P^{<}$, which leads to additional terms involving the first derivative,

$$P^{<}_{ab}(z,\epsilon) = P^{<}_{ab}(z,0) + \epsilon \frac{\partial P^{<}_{ab}(z,\epsilon)}{\partial \epsilon}|_{\epsilon=0} + \mathcal{O}(\alpha_s^2).$$

This finally gives us the equation for the collinear remnant. Note that there is still one soft $1/\epsilon$-pole, which cancels out with the corresponding expression in the soft-virtual terms.

$$d\hat{\sigma}^{(in,+)} = \frac{\alpha_s}{2\pi}\frac{1}{\epsilon}\gamma(a)\mathcal{R}_\alpha\mathcal{S}_\alpha \tag{27.36}$$

$$+ \frac{\alpha_s}{2\pi}\sum_d \left\{ (1-z)P^{<}_{da}(z,0)\left[\left(\frac{1}{1-z}\right)_+ \log\frac{s}{\mu^2} + 2\left(\frac{\log(1-z)}{1-z}\right)_+\right]\right. \tag{27.37}$$

$$\left. -(1-z)\frac{\partial P^{<}_{da}(z,\epsilon)}{\partial\epsilon}\left(\frac{1}{1-z}\right)_+ - K_{da}(z)\right\}\mathcal{R}_\alpha\mathcal{S}_\alpha \tag{27.38}$$

⟨`dglap_remnant.f90`⟩≡
⟨*File header*⟩

```
module dglap_remnant
```

⟨*Use kinds with double*⟩
⟨*Use strings*⟩

```
  use numeric_utils
  use system_dependencies, only: LHAPDF6_AVAILABLE
  use diagnostics
  use constants
  use physics_defs
  use pdg_arrays
  use sf_lhapdf
  use pdf
  use phs_fks, only: isr_kinematics_t

  use nlo_data
```

⟨*Standard module head*⟩

⟨*dglap remnant: public*⟩

⟨*dglap remnant: types*⟩

```
contains
```

⟨*dglap remnant: procedures*⟩

```
end module dglap_remnant
```

⟨*dglap remnant: public*⟩≡

```
public :: dglap_remnant_t
```

⟨*dglap remnant: types*⟩≡

```
type :: dglap_remnant_t
  type(pdf_data_t) :: pdf_data
  type(isr_kinematics_t), pointer :: isr_kinematics => null ()
  integer, dimension(:), allocatable :: i_light_quarks
  integer, dimension(2) :: flv_in
  type(pdf_container_t), dimension(2) :: pdf_scaled
  type(pdf_container_t), dimension(2) :: pdf_born
  real(default), dimension(:), pointer :: sqme_born => null ()
contains
⟨dglap remnant: dglap remnant: TBP⟩
end type dglap_remnant_t
```

⟨*dglap remnant: dglap remnant: TBP*⟩≡

```
procedure :: init => dglap_remnant_init
```

⟨*dglap remnant: procedures*⟩≡

```
subroutine dglap_remnant_init (dglap, isr_kinematics, flv, n_alr)
  class(dglap_remnant_t), intent(inout) :: dglap
  type(isr_kinematics_t), intent(in), target :: isr_kinematics
  integer, dimension(:,:), intent(in) :: flv
  integer, intent(in) :: n_alr
  integer :: i, j, n_quarks
  logical, dimension(-6:6) :: quark_checked = .false.

  dglap%isr_kinematics => isr_kinematics
  call dglap%set_incoming_flavor (flv(1,1), flv(2,1))
  n_quarks = 0
  do i = 1, size (flv, dim = 1)
     if (is_quark(flv(i,1))) then
        n_quarks = n_quarks + 1
        quark_checked(flv(i, 1)) = .true.
     end if
  end do
  allocate (dglap%i_light_quarks (n_quarks))
  j = 1
  do i = -6, 6
     if (quark_checked(i)) then
        dglap%i_light_quarks(j) = i
        j = j + 1
     end if
  end do

  call dglap%init_pdfs ()
end subroutine dglap_remnant_init
```

⟨*dglap remnant: dglap remnant: TBP*⟩+≡

```
procedure :: set_incoming_flavor => dglap_remnant_set_incoming_flavor
```

⟨*dglap remnant: procedures*⟩+≡

```
subroutine dglap_remnant_set_incoming_flavor (dglap, flv1, flv2)
  class(dglap_remnant_t), intent(inout) :: dglap
```

```
    integer, intent(in) :: flv1, flv2
    dglap%flv_in(1) = flv1; dglap%flv_in(2) = flv2
  end subroutine dglap_remnant_set_incoming_flavor
```

⟨*dglap remnant: dglap remnant: TBP*⟩+≡

```
  procedure :: init_pdfs => dglap_remnant_init_pdfs
```

⟨*dglap remnant: procedures*⟩+≡

```
  subroutine dglap_remnant_init_pdfs (dglap)
    class(dglap_remnant_t), intent(inout) :: dglap
    type(string_t) :: lhapdf_dir, lhapdf_file
    integer :: lhapdf_member
    lhapdf_dir = ""
    lhapdf_file = ""
    lhapdf_member = 0
    if (LHAPDF6_AVAILABLE) then
       call lhapdf_initialize &
            (1, lhapdf_dir, lhapdf_file, lhapdf_member, dglap%pdf_data%pdf)
       associate (pdf_data => dglap%pdf_data)
          pdf_data%type = STRF_LHAPDF6
          pdf_data%xmin = pdf_data%pdf%getxmin ()
          pdf_data%xmax = pdf_data%pdf%getxmax ()
          pdf_data%qmin = sqrt(pdf_data%pdf%getq2min ())
          pdf_data%qmax = sqrt(pdf_data%pdf%getq2max ())
       end associate
    else
       call msg_fatal ("PDF subtraction: PDFs could not be initialized")
    end if
  end subroutine dglap_remnant_init_pdfs
```

⟨*dglap remnant: dglap remnant: TBP*⟩+≡

```
  procedure :: compute_pdfs => dglap_remnant_compute_pdfs
```

⟨*dglap remnant: procedures*⟩+≡

```
  subroutine dglap_remnant_compute_pdfs (dglap)
    class(dglap_remnant_t), intent(inout) :: dglap
    integer :: i
    real(default) :: z, x, Q
    real(double), dimension(-6:6) :: f_dble = 0._double
    Q = dglap%isr_kinematics%fac_scale
    do i = 1, 2
       x = dglap%isr_kinematics%x(i)
       z = dglap%isr_kinematics%z(i)
       call dglap%pdf_data%evolve (dble(x), dble(Q), f_dble)
       dglap%pdf_born(i)%f = f_dble / dble(x)
       call dglap%pdf_data%evolve (dble(x / z), dble(Q), f_dble)
       dglap%pdf_scaled(i)%f = f_dble / dble(x / z)
    end do
  end subroutine dglap_remnant_compute_pdfs
```

⟨*dglap remnant: dglap remnant: TBP*⟩+≡

```
  procedure :: get_gluon_pdf => dglap_remnant_get_gluon_pdf
```

⟨*dglap remnant: procedures*⟩+≡

```
function dglap_remnant_get_gluon_pdf (dglap, em, scaled) result (pdf)
  class(dglap_remnant_t), intent(in) :: dglap
  integer, intent(in) :: em
  logical, intent(in) :: scaled
  real(default) :: pdf
  if (scaled) then
     pdf = dglap%pdf_scaled(em)%f(0)
  else
     pdf = dglap%pdf_born(em)%f(0)
  end if
end function dglap_remnant_get_gluon_pdf
```

⟨*dglap remnant: dglap remnant: TBP*⟩+≡

```
procedure :: get_quark_pdf => dglap_remnant_get_quark_pdf
```

⟨*dglap remnant: procedures*⟩+≡

```
function dglap_remnant_get_quark_pdf (dglap, em, i, scaled) result (pdf)
  class(dglap_remnant_t), intent(in) :: dglap
  integer, intent(in) :: em, i
  logical, intent(in) :: scaled
  real(default) :: pdf
  if (scaled) then
     pdf = dglap%pdf_scaled(em)%f(i)
  else
     pdf = dglap%pdf_born(em)%f(i)
  end if
end function dglap_remnant_get_quark_pdf
```

⟨*dglap remnant: dglap remnant: TBP*⟩+≡

```
procedure :: get_summed_quark_pdf => dglap_remnant_get_summed_quark_pdf
```

⟨*dglap remnant: procedures*⟩+≡

```
function dglap_remnant_get_summed_quark_pdf (dglap, em) result (pdf)
  class(dglap_remnant_t), intent(in) :: dglap
  integer, intent(in) :: em
  real(default) :: pdf
  integer :: i_quark
  pdf = 0._default
  do i_quark = -6, 6
     if (any(i_quark == dglap%i_light_quarks)) &
        pdf = pdf + dglap%get_quark_pdf(em, i_quark, scaled = .true.)
  end do
end function dglap_remnant_get_summed_quark_pdf
```

Evaluates formula (...). Note that, as also is the case for the real subtraction, we have to take into account an additional term, occuring because the integral the plus distribution is evaluated over is not constrained on the interval $[0, 1]$. Explicitly, this means

$$\int_{\bar{x}_\oplus}^1 dz \left(\frac{1}{1-z}\right)_+ = \log(1-\bar{x}_\oplus) f(1) + \int_{\bar{x}_\oplus}^1 \frac{f(z)-f(1)}{1-z}, \tag{27.39}$$

and

$$\int_{\bar{x}_\oplus}^1 dz \left(\frac{\log(1-z)}{1-z}\right)_+ f(z) = \frac{1}{2}\log^2(1-\bar{x}_\oplus)f(1) + \int_{\bar{x}_\oplus}^1 \frac{\log(1-z)[f(z)-f(1)]}{1-z}, \tag{27.40}$$

and the same of course for $\bar{x}_\ominus$. These two terms are stored in the `plus_dist_remnant` variable below.

⟨*dglap remnant: dglap remnant: TBP*⟩+≡
```
procedure :: evaluate => dglap_remnant_evaluate
```

⟨*dglap remnant: procedures*⟩+≡
```
function dglap_remnant_evaluate (dglap, alpha_s, sqme_born) result (sqme_dglap)
  real(default), dimension(:), allocatable :: sqme_dglap
  class(dglap_remnant_t), intent(inout) :: dglap
  real(default), intent(in) :: alpha_s
  real(default), dimension(:), intent(in) :: sqme_born
  real(default) :: factor, factor_soft, plus_dist_remnant
  real(default) :: pdfs, pdfb
  integer :: emitter
  real(default), dimension(2) :: tmp
  real(default) :: sb, xb, onemz
  real(default) :: fac_scale2, jac

  sb = dglap%isr_kinematics%sqrts_born**2
  tmp = zero
  fac_scale2 = dglap%isr_kinematics%fac_scale**2

  call dglap%compute_pdfs ()

  do emitter = 1, 2
     associate (z => dglap%isr_kinematics%z(emitter))
        jac = dglap%isr_kinematics%jacobian(emitter)
        onemz = one - z
        factor = log(sb / z / fac_scale2) / onemz + two * log(onemz) / onemz
        factor_soft = log(sb / fac_scale2) / onemz + two * log(onemz) / onemz

        xb = dglap%isr_kinematics%x(emitter)
        plus_dist_remnant = log(one - xb) * log(sb / fac_scale2) + log(one - xb)**2

        if (is_gluon(dglap%flv_in(emitter))) then
           pdfs = dglap%get_gluon_pdf (emitter, scaled = .true.)
           pdfb = dglap%get_gluon_pdf (emitter, scaled = .false.)
           tmp(emitter) = p_hat_gg(z) * factor / z * pdfs / pdfb * jac &
                - p_hat_gg(one) * factor_soft * jac &
                + p_hat_gg(one) * plus_dist_remnant
           pdfs = dglap%get_summed_quark_pdf (emitter)
           tmp(emitter) = tmp(emitter) + &
                (p_hat_qg(z) * factor - p_derived_qg(z)) / z * pdfs / pdfb * jac
        else if (is_quark(dglap%flv_in(emitter))) then
           pdfs = dglap%get_quark_pdf (emitter, dglap%flv_in(emitter), scaled = .true.)
           pdfb = dglap%get_quark_pdf (emitter, dglap%flv_in(emitter), scaled = .false.)
           if (vanishes (pdfb)) then
              sqme_dglap = zero
              return
```

```
            end if
            tmp(emitter) = p_hat_qq(z) * factor / z * pdfs / pdfb * jac &
                 - p_derived_qq(z) / z * pdfs / pdfb * jac &
                 - p_hat_qq(one) * factor_soft * jac &
                 + p_hat_qq(one) * plus_dist_remnant
            pdfs = dglap%get_gluon_pdf (emitter, scaled = .true.)
            tmp(emitter) = tmp(emitter) + &
                 (p_hat_gq(z) * factor - p_derived_gq(z)) / z * pdfs / pdfb * jac
         end if
      end associate
   end do
   sqme_dglap = alpha_s / twopi * (tmp(1) + tmp(2)) * sqme_born
end function dglap_remnant_evaluate
```

$$\hat{P}^{gg}(z) = 2C_A \left[z + \frac{(1-z)^2}{z} + z(1-z)^2 \right] \tag{27.41}$$

⟨*dglap remnant: procedures*⟩+≡

```
function p_hat_gg (z)
  real(default) :: p_hat_gg
⟨p variables⟩
  p_hat_gg = two * CA * (z + onemz**2 / z + z * onemz**2)
end function p_hat_gg
```

$$\hat{P}^{qg}(z) = C_F(1-z)\frac{1+(1-z)^2}{z} \tag{27.42}$$

⟨*dglap remnant: procedures*⟩+≡

```
function p_hat_qg (z)
  real(default) :: p_hat_qg
⟨p variables⟩
  p_hat_qg = CF * onemz / z * (one + onemz**2)
end function p_hat_qg
```

$$\hat{P}^{gq}(z) = T_F(1 - z - 2z(1-z)^2) \tag{27.43}$$

⟨*dglap remnant: procedures*⟩+≡

```
function p_hat_gq (z)
  real(default) :: p_hat_gq
⟨p variables⟩
  p_hat_gq = TR * (onemz - two * z * onemz**2)
end function p_hat_gq
```

$$\hat{P}^{qq}(z) = C_F(1 + z^2) \tag{27.44}$$

⟨*dglap remnant: procedures*⟩+≡

```
function p_hat_qq (z)
  real(default) :: p_hat_qq
  real(default), intent(in) :: z
  p_hat_qq = CF * (one + z**2)
end function p_hat_qq
```

$$\frac{\partial P^{gg}(z,\epsilon)}{\partial \epsilon}|_{\epsilon=0} = 0 \tag{27.45}$$

⟨*dglap remnant: procedures*⟩+≡

```
function p_derived_gg (z)
  real(default) :: p_derived_gg
  real(default), intent(in) :: z
  p_derived_gg = zero
end function p_derived_gg
```

$$\frac{\partial P^{qg}(z,\epsilon)}{\partial \epsilon}|_{\epsilon=0} = -C_F z \tag{27.46}$$

⟨*dglap remnant: procedures*⟩+≡

```
function p_derived_qg (z)
  real(default) :: p_derived_qg
  real(default), intent(in) :: z
  p_derived_qg = -CF * z
end function p_derived_qg
```

$$\frac{\partial P^{gq}(z,\epsilon)}{\partial \epsilon}|_{\epsilon=0} = -2T_F z(1-z) \tag{27.47}$$

⟨*dglap remnant: procedures*⟩+≡

```
function p_derived_gq (z)
  real(default) :: p_derived_gq
⟨p variables⟩
  p_derived_gq = -two * TR * z * onemz
end function p_derived_gq
```

$$\frac{\partial P^{gq}(z,\epsilon)}{\partial \epsilon}|_{\epsilon=0} = -C_F(1-z) \tag{27.48}$$

⟨*dglap remnant: procedures*⟩+≡

```
function p_derived_qq (z)
  real(default) :: p_derived_qq
⟨p variables⟩
  p_derived_qq = -CF * onemz
end function p_derived_qq
```

⟨*p variables*⟩≡

```
real(default), intent(in) :: z
real(default) :: onemz
onemz = one - z
```

⟨*dglap remnant: dglap remnant: TBP*⟩+≡

```
procedure :: final => dglap_remnant_final
```

⟨*dglap remnant: procedures*⟩+≡

```
subroutine dglap_remnant_final (dglap)
  class(dglap_remnant_t), intent(inout) :: dglap
  if (associated (dglap%isr_kinematics)) nullify (dglap%isr_kinematics)
  if (allocated (dglap%i_light_quarks)) deallocate (dglap%i_light_quarks)
  if (associated (dglap%sqme_born)) deallocate (dglap%sqme_born)
end subroutine dglap_remnant_final
```

27.8 Dispatch

⟨dispatch_fks.f90⟩≡
```
⟨File header⟩

module dispatch_fks

⟨Use kinds⟩
⟨Use strings⟩
  use string_utils, only: split_string
  use variables, only: var_list_t
  use nlo_data, only: fks_template_t, FKS_DEFAULT, FKS_RESONANCES

⟨Standard module head⟩

⟨Dispatch fks: public⟩

contains

⟨Dispatch fks: procedures⟩

end module dispatch_fks
```

Initialize parameters used to optimize FKS calculations.

⟨*Dispatch fks: public*⟩≡
```
public :: dispatch_fks_s
```

⟨*Dispatch fks: procedures*⟩≡
```
subroutine dispatch_fks_s (fks_template, var_list)
  type(fks_template_t), intent(inout) :: fks_template
  type(var_list_t), intent(in) :: var_list
  real(default) :: fks_dij_exp1, fks_dij_exp2
  type(string_t) :: fks_mapping_type
  logical :: subtraction_disabled
  type(string_t) :: exclude_from_resonance
  fks_dij_exp1 = &
     var_list%get_rval (var_str ("fks_dij_exp1"))
  fks_dij_exp2 = &
     var_list%get_rval (var_str ("fks_dij_exp2"))
  fks_mapping_type = &
     var_list%get_sval (var_str ("$fks_mapping_type"))
  subtraction_disabled = &
     var_list%get_lval (var_str ("?disable_subtraction"))
  exclude_from_resonance = &
     var_list%get_sval (var_str ("$resonances_exclude_particles"))
  if (exclude_from_resonance /= var_str ("default")) &
     call split_string (exclude_from_resonance, var_str (":"), &
     fks_template%excluded_resonances)
  call fks_template%set_dij_exp (fks_dij_exp1, fks_dij_exp2)
  call fks_template%set_xi_and_y_bounds &
       (var_list%get_rval (var_str ("fks_xi_min")), &
       var_list%get_rval (var_str ("fks_y_max")))
  select case (char (fks_mapping_type))
  case ("default")
     call fks_template%set_mapping_type (FKS_DEFAULT)
```

```
  case ("resonances")
     call fks_template%set_mapping_type (FKS_RESONANCES)
  end select
  fks_template%subtraction_disabled = subtraction_disabled
end subroutine dispatch_fks_s
```

Chapter 28

Model Handling and Features

These modules deal with process definitions and physics models.

These modules use the `model_data` methods to automatically generate process definitions.

auto_components Generic process-definition generator. We can specify a basic process or initial particle(s) and some rules to extend this process, given a model definition with particle names and vertex structures.

radiation_generator Applies the generic generator to the specific problem of generating NLO corrections in a restricted setup.

Model construction:

eval_trees Implementation of the generic `expr_t` type for the concrete evaluation of expressions that access user variables.

This module is actually part of the Sindarin language implementation, and should be moved elsewhere. Currently, the `models` module relies on it.

models Extends the `model_data_t` structure by user-variable objects for easy access, and provides the means to read a model definition from file.

slha_interface Read/write a SUSY model in the standardized SLHA format. The format defines fields and parameters, but no vertices.

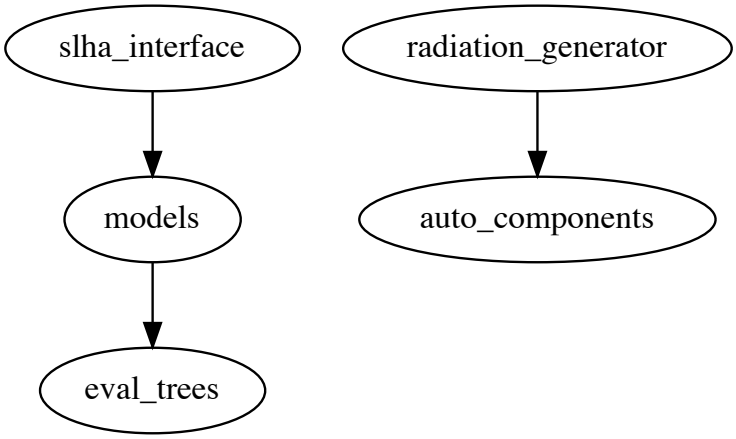

Figure 28.1: Module dependencies in `src/model_features`.

28.1 Automatic generation of process components

This module provides the functionality for automatically generating radiation corrections or decays, provided as lists of PDG codes.

⟨auto_components.f90⟩≡

```
⟨File header⟩

module auto_components

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use diagnostics
  use model_data
  use pdg_arrays
  use physics_defs, only: PHOTON, GLUON, Z_BOSON, W_BOSON
  use numeric_utils, only: extend_integer_array

⟨Standard module head⟩

⟨Auto components: public⟩

⟨Auto components: parameters⟩

⟨Auto components: types⟩

⟨Auto components: interfaces⟩

contains

⟨Auto components: procedures⟩

end module auto_components
```

28.1.1 Constraints: Abstract types

An abstract type that denotes a constraint on the automatically generated states. The concrete objects are applied as visitor objects at certain hooks during the splitting algorithm.

⟨Auto components: types⟩≡

```
  type, abstract :: split_constraint_t
  contains
  ⟨Auto components: split constraint: TBP⟩
  end type split_constraint_t
```

By default, all checks return true.

⟨Auto components: split constraint: TBP⟩≡

```
  procedure :: check_before_split  => split_constraint_check_before_split
  procedure :: check_before_insert => split_constraint_check_before_insert
  procedure :: check_before_record => split_constraint_check_before_record
```

⟨*Auto components: procedures*⟩≡

```
subroutine split_constraint_check_before_split (c, table, pl, k, passed)
  class(split_constraint_t), intent(in) :: c
  class(ps_table_t), intent(in) :: table
  type(pdg_list_t), intent(in) :: pl
  integer, intent(in) :: k
  logical, intent(out) :: passed
  passed = .true.
end subroutine split_constraint_check_before_split

subroutine split_constraint_check_before_insert (c, table, pa, pl, passed)
  class(split_constraint_t), intent(in) :: c
  class(ps_table_t), intent(in) :: table
  type(pdg_array_t), intent(in) :: pa
  type(pdg_list_t), intent(inout) :: pl
  logical, intent(out) :: passed
  passed = .true.
end subroutine split_constraint_check_before_insert

subroutine split_constraint_check_before_record (c, table, pl, n_loop, passed)
  class(split_constraint_t), intent(in) :: c
  class(ps_table_t), intent(in) :: table
  type(pdg_list_t), intent(in) :: pl
  integer, intent(in) :: n_loop
  logical, intent(out) :: passed
  passed = .true.
end subroutine split_constraint_check_before_record
```

A transparent wrapper, so we can collect constraints of different type.

⟨*Auto components: types*⟩+≡

```
type :: split_constraint_wrap_t
   class(split_constraint_t), allocatable :: c
end type split_constraint_wrap_t
```

A collection of constraints.

⟨*Auto components: public*⟩≡

```
public :: split_constraints_t
```

⟨*Auto components: types*⟩+≡

```
type :: split_constraints_t
   class(split_constraint_wrap_t), dimension(:), allocatable :: cc
 contains
 ⟨Auto components: split constraints: TBP⟩
end type split_constraints_t
```

Initialize the constraints set with a specific number of elements.

⟨*Auto components: split constraints: TBP*⟩≡

```
procedure :: init => split_constraints_init
```

⟨*Auto components: procedures*⟩+≡

```
subroutine split_constraints_init (constraints, n)
  class(split_constraints_t), intent(out) :: constraints
  integer, intent(in) :: n
```

```
    allocate (constraints%cc (n))
  end subroutine split_constraints_init
```

Set a constraint.

⟨*Auto components: split constraints: TBP*⟩+≡

```
  procedure :: set => split_constraints_set
```

⟨*Auto components: procedures*⟩+≡

```
  subroutine split_constraints_set (constraints, i, c)
    class(split_constraints_t), intent(inout) :: constraints
    integer, intent(in) :: i
    class(split_constraint_t), intent(in) :: c
    allocate (constraints%cc(i)%c, source = c)
  end subroutine split_constraints_set
```

Apply checks.

`check_before_split` is applied to the particle list that we want to split.

`check_before_insert` is applied to the particle list `pl` that is to replace the particle `pa` that is split. This check may transform the particle list.

`check_before_record` is applied to the complete new particle list that results from splitting before it is recorded.

⟨*Auto components: split constraints: TBP*⟩+≡

```
  procedure :: check_before_split  => split_constraints_check_before_split
  procedure :: check_before_insert => split_constraints_check_before_insert
  procedure :: check_before_record => split_constraints_check_before_record
```

⟨*Auto components: procedures*⟩+≡

```
  subroutine split_constraints_check_before_split &
       (constraints, table, pl, k, passed)
    class(split_constraints_t), intent(in) :: constraints
    class(ps_table_t), intent(in) :: table
    type(pdg_list_t), intent(in) :: pl
    integer, intent(in) :: k
    logical, intent(out) :: passed
    integer :: i
    passed = .true.
    do i = 1, size (constraints%cc)
       call constraints%cc(i)%c%check_before_split (table, pl, k, passed)
       if (.not. passed)  return
    end do
  end subroutine split_constraints_check_before_split

  subroutine split_constraints_check_before_insert &
       (constraints, table, pa, pl, passed)
    class(split_constraints_t), intent(in) :: constraints
    class(ps_table_t), intent(in) :: table
    type(pdg_array_t), intent(in) :: pa
    type(pdg_list_t), intent(inout) :: pl
    logical, intent(out) :: passed
    integer :: i
    passed = .true.
    do i = 1, size (constraints%cc)
       call constraints%cc(i)%c%check_before_insert (table, pa, pl, passed)
```

```
      if (.not. passed)  return
    end do
  end subroutine split_constraints_check_before_insert

  subroutine split_constraints_check_before_record &
      (constraints, table, pl, n_loop, passed)
    class(split_constraints_t), intent(in) :: constraints
    class(ps_table_t), intent(in) :: table
    type(pdg_list_t), intent(in) :: pl
    integer, intent(in) :: n_loop
    logical, intent(out) :: passed
    integer :: i
    passed = .true.
    do i = 1, size (constraints%cc)
      call constraints%cc(i)%c%check_before_record (table, pl, n_loop, passed)
      if (.not. passed)  return
    end do
  end subroutine split_constraints_check_before_record
```

28.1.2 Specific constraints

Number of particles

Specific constraint: The number of particles plus the number of loops, if any, must remain less than the given limit. Note that the number of loops is defined only when we are recording the entry.

⟨*Auto components: types*⟩+≡

```
  type, extends (split_constraint_t) :: constraint_n_tot
     private
     integer :: n_max = 0
   contains
     procedure :: check_before_split => constraint_n_tot_check_before_split
     procedure :: check_before_record => constraint_n_tot_check_before_record
  end type constraint_n_tot
```

⟨*Auto components: public*⟩+≡

```
  public :: constrain_n_tot
```

⟨*Auto components: procedures*⟩+≡

```
  function constrain_n_tot (n_max) result (c)
    integer, intent(in) :: n_max
    type(constraint_n_tot) :: c
    c%n_max = n_max
  end function constrain_n_tot

  subroutine constraint_n_tot_check_before_split (c, table, pl, k, passed)
    class(constraint_n_tot), intent(in) :: c
    class(ps_table_t), intent(in) :: table
    type(pdg_list_t), intent(in) :: pl
    integer, intent(in) :: k
    logical, intent(out) :: passed
    passed = pl%get_size () < c%n_max
  end subroutine constraint_n_tot_check_before_split
```

```
subroutine constraint_n_tot_check_before_record (c, table, pl, n_loop, passed)
  class(constraint_n_tot), intent(in) :: c
  class(ps_table_t), intent(in) :: table
  type(pdg_list_t), intent(in) :: pl
  integer, intent(in) :: n_loop
  logical, intent(out) :: passed
  passed = pl%get_size () + n_loop <= c%n_max
end subroutine constraint_n_tot_check_before_record
```

Number of loops

Specific constraint: The number of loops is limited, independent of the total number of particles.

⟨*Auto components: types*⟩+≡

```
type, extends (split_constraint_t) :: constraint_n_loop
   private
   integer :: n_loop_max = 0
 contains
   procedure :: check_before_record => constraint_n_loop_check_before_record
end type constraint_n_loop
```

⟨*Auto components: public*⟩+≡

```
public :: constrain_n_loop
```

⟨*Auto components: procedures*⟩+≡

```
function constrain_n_loop (n_loop_max) result (c)
  integer, intent(in) :: n_loop_max
  type(constraint_n_loop) :: c
  c%n_loop_max = n_loop_max
end function constrain_n_loop

subroutine constraint_n_loop_check_before_record &
     (c, table, pl, n_loop, passed)
  class(constraint_n_loop), intent(in) :: c
  class(ps_table_t), intent(in) :: table
  type(pdg_list_t), intent(in) :: pl
  integer, intent(in) :: n_loop
  logical, intent(out) :: passed
  passed = n_loop <= c%n_loop_max
end subroutine constraint_n_loop_check_before_record
```

Particles allowed in splitting

Specific constraint: The entries in the particle list ready for insertion are matched to a given list of particle patterns. If a match occurs, the entry is replaced by the corresponding pattern. If there is no match, the check fails. If a massless gauge boson splitting is detected, the splitting partners are checked against a list of excluded particles. If a match occurs, the check fails.

⟨*Auto components: types*⟩+≡

```
type, extends (split_constraint_t) :: constraint_splittings
   private
   type(pdg_list_t) :: pl_match, pl_excluded_gauge_splittings
 contains
   procedure :: check_before_insert => constraint_splittings_check_before_insert
end type constraint_splittings
```

⟨*Auto components: public*⟩+≡

```
public :: constrain_splittings
```

⟨*Auto components: procedures*⟩+≡

```
function constrain_splittings (pl_match, pl_excluded_gauge_splittings) result (c)
  type(pdg_list_t), intent(in) :: pl_match
  type(pdg_list_t), intent(in) :: pl_excluded_gauge_splittings
  type(constraint_splittings) :: c
  c%pl_match = pl_match
  c%pl_excluded_gauge_splittings = pl_excluded_gauge_splittings
end function constrain_splittings

subroutine constraint_splittings_check_before_insert (c, table, pa, pl, passed)
  class(constraint_splittings), intent(in) :: c
  class(ps_table_t), intent(in) :: table
  type(pdg_array_t), intent(in) :: pa
  type(pdg_list_t), intent(inout) :: pl
  logical, intent(out) :: passed
  logical :: has_massless_vector
  integer :: i
  has_massless_vector = .false.
  do i = 1, pa%get_length ()
     if (is_massless_vector(pa%get(i))) then
        has_massless_vector = .true.
        exit
     end if
  end do
  passed = .false.
  if (has_massless_vector .and. count (is_fermion(pl%a%get ())) == 2) then
     do i = 1, c%pl_excluded_gauge_splittings%get_size ()
        if (pl .match. c%pl_excluded_gauge_splittings%a(i)) return
     end do
     call pl%match_replace (c%pl_match, passed)
     passed = .true.
  else
     call pl%match_replace (c%pl_match, passed)
  end if
end subroutine constraint_splittings_check_before_insert
```

Specific constraint: The entries in the particle list ready for insertion are matched to a given list of particle patterns. If a match occurs, the entry is replaced by the corresponding pattern. If there is no match, the check fails.

⟨*Auto components: types*⟩+≡

```
type, extends (split_constraint_t) :: constraint_insert
   private
   type(pdg_list_t) :: pl_match
```

```
 contains
   procedure :: check_before_insert => constraint_insert_check_before_insert
end type constraint_insert
```

⟨*Auto components: public*⟩+≡

```
public :: constrain_insert
```

⟨*Auto components: procedures*⟩+≡

```
function constrain_insert (pl_match) result (c)
  type(pdg_list_t), intent(in) :: pl_match
  type(constraint_insert) :: c
  c%pl_match = pl_match
end function constrain_insert

subroutine constraint_insert_check_before_insert (c, table, pa, pl, passed)
  class(constraint_insert), intent(in) :: c
  class(ps_table_t), intent(in) :: table
  type(pdg_array_t), intent(in) :: pa
  type(pdg_list_t), intent(inout) :: pl
  logical, intent(out) :: passed
  call pl%match_replace (c%pl_match, passed)
end subroutine constraint_insert_check_before_insert
```

Particles required in final state

Specific constraint: The entries in the recorded state must be a superset of the entries in the given list (for instance, the lowest-order state).

⟨*Auto components: types*⟩+≡

```
type, extends (split_constraint_t) :: constraint_require
   private
   type(pdg_list_t) :: pl
 contains
   procedure :: check_before_record => constraint_require_check_before_record
end type constraint_require
```

We check the current state by matching all particle entries against the stored particle list, and crossing out the particles in the latter list when a match is found. The constraint passed if all entries have been crossed out.

For an `if_table` in particular, we check the final state only.

⟨*Auto components: public*⟩+≡

```
public :: constrain_require
```

⟨*Auto components: procedures*⟩+≡

```
function constrain_require (pl) result (c)
  type(pdg_list_t), intent(in) :: pl
  type(constraint_require) :: c
  c%pl = pl
end function constrain_require

subroutine constraint_require_check_before_record &
     (c, table, pl, n_loop, passed)
  class(constraint_require), intent(in) :: c
```

```
    class(ps_table_t), intent(in) :: table
    type(pdg_list_t), intent(in) :: pl
    integer, intent(in) :: n_loop
    logical, intent(out) :: passed
    logical, dimension(:), allocatable :: mask
    integer :: i, k, n_in
    select type (table)
    type is (if_table_t)
       if (table%proc_type > 0) then
          select case (table%proc_type)
          case (PROC_DECAY)
             n_in = 1
          case (PROC_SCATTER)
             n_in = 2
          end select
       else
          call msg_fatal ("Neither a decay nor a scattering process")
       end if
    class default
       n_in = 0
    end select
    allocate (mask (c%pl%get_size ()), source = .true.)
    do i = n_in + 1, pl%get_size ()
       k = c%pl%find_match (pl%get (i), mask)
       if (k /= 0)  mask(k) = .false.
    end do
    passed = .not. any (mask)
  end subroutine constraint_require_check_before_record
```

Radiation

Specific constraint: We have radiation pattern if the original particle matches an entry in the list of particles that should replace it. The constraint prohibits this situation.

⟨*Auto components: public*⟩+≡

```
  public :: constrain_radiation
```

⟨*Auto components: types*⟩+≡

```
  type, extends (split_constraint_t) :: constraint_radiation
     private
   contains
     procedure :: check_before_insert => &
          constraint_radiation_check_before_insert
  end type constraint_radiation
```

⟨*Auto components: procedures*⟩+≡

```
  function constrain_radiation () result (c)
    type(constraint_radiation) :: c
  end function constrain_radiation

  subroutine constraint_radiation_check_before_insert (c, table, pa, pl, passed)
    class(constraint_radiation), intent(in) :: c
```

```
    class(ps_table_t), intent(in) :: table
    type(pdg_array_t), intent(in) :: pa
    type(pdg_list_t), intent(inout) :: pl
    logical, intent(out) :: passed
    passed = .not. (pl .match. pa)
  end subroutine constraint_radiation_check_before_insert
```

Mass sum

Specific constraint: The sum of masses within the particle list must be smaller than a given limit. For in/out state combinations, we check initial and final state separately.

If we specify `margin` in the initialization, the sum must be strictly less than the limit minus the given margin (which may be zero). If not, equality is allowed.

⟨*Auto components: public*⟩+≡

```
  public :: constrain_mass_sum
```

⟨*Auto components: types*⟩+≡

```
  type, extends (split_constraint_t) :: constraint_mass_sum
     private
     real(default) :: mass_limit = 0
     logical :: strictly_less = .false.
     real(default) :: margin = 0
   contains
     procedure :: check_before_record => constraint_mass_sum_check_before_record
  end type constraint_mass_sum
```

⟨*Auto components: procedures*⟩+≡

```
  function constrain_mass_sum (mass_limit, margin) result (c)
    real(default), intent(in) :: mass_limit
    real(default), intent(in), optional :: margin
    type(constraint_mass_sum) :: c
    c%mass_limit = mass_limit
    if (present (margin)) then
       c%strictly_less = .true.
       c%margin = margin
    end if
  end function constrain_mass_sum

  subroutine constraint_mass_sum_check_before_record &
       (c, table, pl, n_loop, passed)
    class(constraint_mass_sum), intent(in) :: c
    class(ps_table_t), intent(in) :: table
    type(pdg_list_t), intent(in) :: pl
    integer, intent(in) :: n_loop
    logical, intent(out) :: passed
    real(default) :: limit
    if (c%strictly_less) then
       limit = c%mass_limit - c%margin
       select type (table)
       type is (if_table_t)
          passed = mass_sum (pl, 1, 2, table%model) < limit  &
```

```
            .and. mass_sum (pl, 3, pl%get_size (), table%model) < limit
      class default
         passed = mass_sum (pl, 1, pl%get_size (), table%model) < limit
      end select
   else
      limit = c%mass_limit
      select type (table)
      type is (if_table_t)
         passed = mass_sum (pl, 1, 2, table%model) <= limit &
            .and. mass_sum (pl, 3, pl%get_size (), table%model) <= limit
      class default
         passed = mass_sum (pl, 1, pl%get_size (), table%model) <= limit
      end select
   end if
end subroutine constraint_mass_sum_check_before_record
```

Initial state particles

Specific constraint: The two incoming particles must both match the given particle list. This is checked for the generated particle list, just before it is recorded.

⟨*Auto components: public*⟩+≡

```
public :: constrain_in_state
```

⟨*Auto components: types*⟩+≡

```
type, extends (split_constraint_t) :: constraint_in_state
   private
   type(pdg_list_t) :: pl
 contains
   procedure :: check_before_record => constraint_in_state_check_before_record
end type constraint_in_state
```

⟨*Auto components: procedures*⟩+≡

```
function constrain_in_state (pl) result (c)
  type(pdg_list_t), intent(in) :: pl
  type(constraint_in_state) :: c
  c%pl = pl
end function constrain_in_state

subroutine constraint_in_state_check_before_record &
     (c, table, pl, n_loop, passed)
  class(constraint_in_state), intent(in) :: c
  class(ps_table_t), intent(in) :: table
  type(pdg_list_t), intent(in) :: pl
  integer, intent(in) :: n_loop
  logical, intent(out) :: passed
  integer :: i
  select type (table)
  type is (if_table_t)
     passed = .false.
     do i = 1, 2
        if (.not. (c%pl .match. pl%get (i)))  return
```

```
      end do
    end select
    passed = .true.
  end subroutine constraint_in_state_check_before_record
```

Coupling constraint

Filters vertices which do not match the desired NLO pattern.

⟨*Auto components: types*⟩+≡

```
  type, extends (split_constraint_t) :: constraint_coupling_t
    private
    logical :: qed = .false.
    logical :: qcd = .true.
    logical :: ew = .false.
    integer :: n_nlo_correction_types
  contains
  ⟨Auto components: constraint coupling: TBP⟩
  end type constraint_coupling_t
```

⟨*Auto components: public*⟩+≡

```
  public :: constrain_couplings
```

⟨*Auto components: procedures*⟩+≡

```
  function constrain_couplings (qcd, qed, n_nlo_correction_types) result (c)
    type(constraint_coupling_t) :: c
    logical, intent(in) :: qcd, qed
    integer, intent(in) :: n_nlo_correction_types
    c%qcd = qcd; c%qed = qed
    c%n_nlo_correction_types = n_nlo_correction_types
  end function constrain_couplings
```

⟨*Auto components: constraint coupling: TBP*⟩≡

```
  procedure :: check_before_insert => constraint_coupling_check_before_insert
```

⟨*Auto components: procedures*⟩+≡

```
  subroutine constraint_coupling_check_before_insert (c, table, pa, pl, passed)
    class(constraint_coupling_t), intent(in) :: c
    class(ps_table_t), intent(in) :: table
    type(pdg_array_t), intent(in) :: pa
    type(pdg_list_t), intent(inout) :: pl
    logical, intent(out) :: passed
    type(pdg_list_t) :: pl_vertex
    type(pdg_array_t) :: pdg_gluon, pdg_photon, pdg_W_Z, pdg_gauge_bosons
    integer :: i, j
    pdg_gluon = GLUON; pdg_photon = PHOTON
    pdg_W_Z = [W_BOSON,-W_BOSON, Z_BOSON]
    if (c%qcd) pdg_gauge_bosons = pdg_gauge_bosons // pdg_gluon
    if (c%qed) pdg_gauge_bosons = pdg_gauge_bosons // pdg_photon
    if (c%ew) pdg_gauge_bosons = pdg_gauge_bosons // pdg_W_Z
    do j = 1, pa%get_length ()
       call pl_vertex%init (pl%get_size () + 1)
       call pl_vertex%set (1, pa%get(j))
```

```
      do i = 1, pl%get_size ()
         call pl_vertex%set (i + 1, pl%get(i))
      end do
      if (is_massless_vector(pa%get(j))) then
         if (.not. table%model%check_vertex &
              (pl_vertex%a(1)%get (), pl_vertex%a(2)%get (), pl_vertex%a(3)%get ())) then
            passed = .false.
            cycle
         end if
      else if (.not. table%model%check_vertex &
           (- pl_vertex%a(1)%get (), pl_vertex%a(2)%get (), pl_vertex%a(3)%get ())) then
         passed = .false.
         cycle
      end if
      if (.not. (pl_vertex .match. pdg_gauge_bosons)) then
         passed = .false.
         cycle
      end if
      if (.not. c%ew .and. (pl_vertex .match. pdg_W_Z)) then
         passed = .false.
         cycle
      end if
      passed = .true.
      exit
    end do
  end subroutine constraint_coupling_check_before_insert
```

28.1.3 Tables of states

Automatically generate a list of possible process components for a given initial set (a single massive particle or a preset list of states).

The set of process components are generated by recursive splitting, applying constraints on the fly that control and limit the process. The generated states are accumulated in a table that we can read out after completion.

⟨*Auto components: types*⟩+≡

```
  type, extends (pdg_list_t) :: ps_entry_t
     integer :: n_loop = 0
     integer :: n_rad = 0
     type(ps_entry_t), pointer :: previous => null ()
     type(ps_entry_t), pointer :: next => null ()
  end type ps_entry_t
```

⟨*Auto components: parameters*⟩≡

```
  integer, parameter :: PROC_UNDEFINED = 0
  integer, parameter :: PROC_DECAY = 1
  integer, parameter :: PROC_SCATTER = 2
```

This is the wrapper type for the decay tree for the list of final states and the final array. First, an abstract base type:

⟨*Auto components: public*⟩+≡

```
  public :: ps_table_t
```

⟨*Auto components: types*⟩+≡
```
type, abstract :: ps_table_t
   private
   class(model_data_t), pointer :: model => null ()
   logical :: loops = .false.
   type(ps_entry_t), pointer :: first => null ()
   type(ps_entry_t), pointer :: last => null ()
   integer :: proc_type
 contains
 ⟨Auto components: ps table: TBP⟩
end type ps_table_t
```

The extensions: one for decay, one for generic final states. The decay-state table stores the initial particle. The final-state table is indifferent, and the initial/final state table treats the first two particles in its list as incoming antiparticles.

⟨*Auto components: public*⟩+≡
```
public :: ds_table_t
public :: fs_table_t
public :: if_table_t
```

⟨*Auto components: types*⟩+≡
```
type, extends (ps_table_t) :: ds_table_t
   private
   integer :: pdg_in = 0
 contains
 ⟨Auto components: ds table: TBP⟩
end type ds_table_t

type, extends (ps_table_t) :: fs_table_t
 contains
 ⟨Auto components: fs table: TBP⟩
end type fs_table_t

type, extends (fs_table_t) :: if_table_t
 contains
 ⟨Auto components: if table: TBP⟩
end type if_table_t
```

Finalizer: we must deallocate the embedded list.

⟨*Auto components: ps table: TBP*⟩≡
```
procedure :: final => ps_table_final
```

⟨*Auto components: procedures*⟩+≡
```
subroutine ps_table_final (object)
  class(ps_table_t), intent(inout) :: object
  type(ps_entry_t), pointer :: current
  do while (associated (object%first))
     current => object%first
     object%first => current%next
     deallocate (current)
  end do
  nullify (object%last)
end subroutine ps_table_final
```

Write the table. A base writer for the body and specific writers for the headers.

⟨*Auto components: ps table: TBP*⟩+≡
```
procedure :: base_write => ps_table_base_write
procedure (ps_table_write), deferred :: write
```

⟨*Auto components: interfaces*⟩≡
```
interface
   subroutine ps_table_write (object, unit)
     import
     class(ps_table_t), intent(in) :: object
     integer, intent(in), optional :: unit
   end subroutine ps_table_write
end interface
```

⟨*Auto components: ds table: TBP*⟩≡
```
procedure :: write => ds_table_write
```

⟨*Auto components: fs table: TBP*⟩≡
```
procedure :: write => fs_table_write
```

⟨*Auto components: if table: TBP*⟩≡
```
procedure :: write => if_table_write
```

The first `n_in` particles will be replaced by antiparticles in the output, and we write an arrow if `n_in` is present.

⟨*Auto components: procedures*⟩+≡
```
subroutine ps_table_base_write (object, unit, n_in)
  class(ps_table_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer, intent(in), optional :: n_in
  integer, dimension(:), allocatable :: pdg
  type(ps_entry_t), pointer :: entry
  type(field_data_t), pointer :: prt
  integer :: u, i, j, n0
  u = given_output_unit (unit)
  entry => object%first
  do while (associated (entry))
     write (u, "(2x)", advance = "no")
     if (present (n_in)) then
        do i = 1, n_in
           write (u, "(1x)", advance = "no")
           pdg = entry%get (i)
           do j = 1, size (pdg)
              prt => object%model%get_field_ptr (pdg(j))
              if (j > 1)  write (u, "(':')", advance = "no")
              write (u, "(A)", advance = "no") &
                   char (prt%get_name (pdg(j) >= 0))
           end do
        end do
        write (u, "(1x,A)", advance = "no")  "=>"
        n0 = n_in + 1
     else
        n0 = 1
     end if
     do i = n0, entry%get_size ()
        write (u, "(1x)", advance = "no")
```

```
        pdg = entry%get (i)
        do j = 1, size (pdg)
           prt => object%model%get_field_ptr (pdg(j))
           if (j > 1)  write (u, "(':')", advance = "no")
           write (u, "(A)", advance = "no") &
                char (prt%get_name (pdg(j) < 0))
        end do
     end do
     if (object%loops) then
        write (u, "(2x,'[',I0,',',I0,']')")  entry%n_loop, entry%n_rad
     else
        write (u, *)
     end if
     entry => entry%next
  end do
end subroutine ps_table_base_write

subroutine ds_table_write (object, unit)
  class(ds_table_t), intent(in) :: object
  integer, intent(in), optional :: unit
  type(field_data_t), pointer :: prt
  integer :: u
  u = given_output_unit (unit)
  prt => object%model%get_field_ptr (object%pdg_in)
  write (u, "(1x,A,1x,A)")  "Decays for particle:", &
       char (prt%get_name (object%pdg_in < 0))
  call object%base_write (u)
end subroutine ds_table_write

subroutine fs_table_write (object, unit)
  class(fs_table_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Table of final states:"
  call object%base_write (u)
end subroutine fs_table_write

subroutine if_table_write (object, unit)
  class(if_table_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Table of in/out states:"
  select case (object%proc_type)
  case (PROC_DECAY)
     call object%base_write (u, n_in = 1)
  case (PROC_SCATTER)
     call object%base_write (u, n_in = 2)
   end select
end subroutine if_table_write
```

Obtain a particle string for a given index in the pdg list

⟨*Auto components: ps table: TBP*⟩+≡

```
procedure :: get_particle_string => ps_table_get_particle_string
```

⟨*Auto components: procedures*⟩+≡

```
subroutine ps_table_get_particle_string (object, index, prt_in, prt_out)
  class(ps_table_t), intent(in) :: object
  integer, intent(in) :: index
  type(string_t), intent(out), dimension(:), allocatable :: prt_in, prt_out
  integer :: n_in
  type(field_data_t), pointer :: prt
  type(ps_entry_t), pointer :: entry
  integer, dimension(:), allocatable :: pdg
  integer :: n0
  integer :: i, j
  entry => object%first
  i = 1
  do while (i < index)
    if (associated (entry%next)) then
      entry => entry%next
      i=i+1
    else
      call msg_fatal ("ps_table: entry with requested index does not exist!")
    end if
  end do

  if (object%proc_type > 0) then
     select case (object%proc_type)
     case (PROC_DECAY)
        n_in = 1
     case (PROC_SCATTER)
        n_in = 2
     end select
  else
     call msg_fatal ("Neither decay nor scattering process")
  end if

  n0 = n_in + 1
  allocate (prt_in (n_in), prt_out (entry%get_size () - n_in))
  do i = 1, n_in
    prt_in(i) = ""
    pdg = entry%get(i)
    do j = 1, size(pdg)
      prt => object%model%get_field_ptr (pdg(j))
      prt_in(i) = prt_in(i) // prt%get_name (pdg(j) >= 0)
      if (j /= size(pdg)) &
         prt_in(i) = prt_in(i) // ":"
    end do
  end do
  do i = n0, entry%get_size ()
    prt_out(i-n_in) = ""
    pdg = entry%get(i)
    do j = 1, size(pdg)
       prt => object%model%get_field_ptr (pdg(j))
       prt_out(i-n_in) = prt_out(i-n_in) // prt%get_name (pdg(j) < 0)
       if (j /= size(pdg)) &
          prt_out(i-n_in) = prt_out(i-n_in) // ":"
```

```
      end do
    end do
  end subroutine ps_table_get_particle_string
```

Initialize with a predefined set of final states, or in/out state lists.

⟨*Auto components: ps table: TBP*⟩+≡

```
  generic :: init => ps_table_init
  procedure, private :: ps_table_init
```

⟨*Auto components: if table: TBP*⟩+≡

```
  generic :: init => if_table_init
  procedure, private :: if_table_init
```

⟨*Auto components: procedures*⟩+≡

```
  subroutine ps_table_init (table, model, pl, constraints, n_in)
    class(ps_table_t), intent(out) :: table
    class(model_data_t), intent(in), target :: model
    type(pdg_list_t), dimension(:), intent(in) :: pl
    type(split_constraints_t), intent(in) :: constraints
    integer, intent(in), optional :: n_in
    logical :: passed
    integer :: i
    table%model => model

    if (present (n_in)) then
       select case (n_in)
       case (1)
          table%proc_type = PROC_DECAY
       case (2)
          table%proc_type = PROC_SCATTER
       case default
          table%proc_type = PROC_UNDEFINED
       end select
    else
       table%proc_type = PROC_UNDEFINED
    end if

    do i = 1, size (pl)
       call table%record (pl(i), 0, 0, constraints, passed)
       if (.not. passed) then
          call msg_fatal ("ps_table: Registering process components failed")
       end if
    end do
  end subroutine ps_table_init

  subroutine if_table_init (table, model, pl_in, pl_out, constraints)
    class(if_table_t), intent(out) :: table
    class(model_data_t), intent(in), target :: model
    type(pdg_list_t), dimension(:), intent(in) :: pl_in, pl_out
    type(split_constraints_t), intent(in) :: constraints
    integer :: i, j, k, p, n_in, n_out
    type(pdg_array_t), dimension(:), allocatable :: pa_in
    type(pdg_list_t), dimension(:), allocatable :: pl
    allocate (pl (size (pl_in) * size (pl_out)))
    k = 0
```

```
    do i = 1, size (pl_in)
       n_in = pl_in(i)%get_size ()
       allocate (pa_in (n_in))
       do p = 1, n_in
          pa_in(p) = pl_in(i)%get (p)
       end do
       do j = 1, size (pl_out)
          n_out = pl_out(j)%get_size ()
          k = k + 1
          call pl(k)%init (n_in + n_out)
          do p = 1, n_in
             call pl(k)%set (p, invert_pdg_array (pa_in(p), model))
          end do
          do p = 1, n_out
             call pl(k)%set (n_in + p, pl_out(j)%get (p))
          end do
       end do
       deallocate (pa_in)
    end do
    n_in = size (pl_in(1)%a)
    call table%init (model, pl, constraints, n_in)
  end subroutine if_table_init
```

Enable loops for the table. This affects both splitting and output.

⟨*Auto components: ps table: TBP*⟩+≡

```
  procedure :: enable_loops => ps_table_enable_loops
```

⟨*Auto components: procedures*⟩+≡

```
  subroutine ps_table_enable_loops (table)
    class(ps_table_t), intent(inout) :: table
    table%loops = .true.
  end subroutine ps_table_enable_loops
```

28.1.4 Top-level methods

Create a table for a single-particle decay. Construct all possible final states from a single particle with PDG code `pdg_in`. The construction is limited by the given `constraints`.

⟨*Auto components: ds table: TBP*⟩+≡

```
  procedure :: make => ds_table_make
```

⟨*Auto components: procedures*⟩+≡

```
  subroutine ds_table_make (table, model, pdg_in, constraints)
    class(ds_table_t), intent(out) :: table
    class(model_data_t), intent(in), target :: model
    integer, intent(in) :: pdg_in
    type(split_constraints_t), intent(in) :: constraints
    type(pdg_list_t) :: pl_in
    type(pdg_list_t), dimension(0) :: pl
    call table%init (model, pl, constraints)
    table%pdg_in = pdg_in
    call pl_in%init (1)
```

```
    call pl_in%set (1, [pdg_in])
    call table%split (pl_in, 0, constraints)
  end subroutine ds_table_make
```

Split all entries in a growing table, starting from a table that may already contain states. Add and record split states on the fly.

⟨*Auto components: fs table: TBP*⟩+≡

```
  procedure :: radiate => fs_table_radiate
```

⟨*Auto components: procedures*⟩+≡

```
  subroutine fs_table_radiate (table, constraints)
    class(fs_table_t), intent(inout) :: table
    type(split_constraints_t) :: constraints
    type(ps_entry_t), pointer :: current
    current => table%first
    do while (associated (current))
       call table%split (current, 0, constraints, record = .true.)
       current => current%next
    end do
  end subroutine fs_table_radiate
```

28.1.5 Splitting algorithm

Recursive splitting. First of all, we record the current `pdg_list` in the table, subject to `constraints`, if requested. We also record copies of the list marked as loop corrections.

When we record a particle list, we sort it first.

If there is room for splitting, We take a PDG array list and the index of an element, and split this element in all possible ways. The split entry is inserted into the list, which we split further.

The recursion terminates whenever the split array would have a length greater than $n_{\max}$.

⟨*Auto components: ps table: TBP*⟩+≡

```
  procedure :: split => ps_table_split
```

⟨*Auto components: procedures*⟩+≡

```
  recursive subroutine ps_table_split (table, pl, n_rad, constraints, &
        record)
    class(ps_table_t), intent(inout) :: table
    class(pdg_list_t), intent(in) :: pl
    integer, intent(in) :: n_rad
    type(split_constraints_t), intent(in) :: constraints
    logical, intent(in), optional :: record
    integer :: n_loop, i
    logical :: passed, save_pdg_index
    type(vertex_iterator_t) :: vit
    integer, dimension(:), allocatable :: pdg1
    integer, dimension(:), allocatable :: pdg2
    if (present (record)) then
       if (record) then
          n_loop = 0
          INCR_LOOPS: do
```

```
            call table%record_sorted (pl, n_loop, n_rad, constraints, passed)
            if (.not. passed)  exit INCR_LOOPS
            if (.not. table%loops)  exit INCR_LOOPS
            n_loop = n_loop + 1
         end do INCR_LOOPS
      end if
   end if
   select type (table)
   type is (if_table_t)
      save_pdg_index = .true.
   class default
      save_pdg_index = .false.
   end select
   do i = 1, pl%get_size ()
      call constraints%check_before_split (table, pl, i, passed)
      if (passed) then
         pdg1 = pl%get (i)
         call vit%init (table%model, pdg1, save_pdg_index)
         SCAN_VERTICES: do
            call vit%get_next_match (pdg2)
            if (allocated (pdg2)) then
               call table%insert (pl, n_rad, i, pdg2, constraints)
            else
               exit SCAN_VERTICES
            end if
         end do SCAN_VERTICES
      end if
   end do
end subroutine ps_table_split
```

The worker part: insert the list of particles found by vertex matching in place of entry `i` in the PDG list. Then split/record further.

The `n_in` parameter tells the replacement routine to insert the new particles after entry `n_in`. Otherwise, they follow index `i`.

⟨*Auto components: ps table: TBP*⟩+≡

```
procedure :: insert => ps_table_insert
```

⟨*Auto components: procedures*⟩+≡

```
recursive subroutine ps_table_insert &
     (table, pl, n_rad, i, pdg, constraints, n_in)
   class(ps_table_t), intent(inout) :: table
   class(pdg_list_t), intent(in) :: pl
   integer, intent(in) :: n_rad, i
   integer, dimension(:), intent(in) :: pdg
   type(split_constraints_t), intent(in) :: constraints
   integer, intent(in), optional :: n_in
   type(pdg_list_t) :: pl_insert
   logical :: passed
   integer :: k, s
   s = size (pdg)
   call pl_insert%init (s)
   do k = 1, s
      call pl_insert%set (k, pdg(k))
   end do
```

```
    call constraints%check_before_insert (table, pl%get (i), pl_insert, passed)
    if (passed) call table%split (pl%replace (i, pl_insert, n_in), n_rad + s - 1, &
            constraints, record = .true.)
  end subroutine ps_table_insert
```

Special case: If we are splitting an initial particle, there is slightly more to do. We loop over the particles from the vertex match and replace the initial particle by each of them in turn. The remaining particles must be appended after the second initial particle, so they will end up in the out state. This is done by providing the `n_in` argument to the base method as an optional argument.

Note that we must call the base-method procedure explicitly, so the `table` argument keeps its dynamic type as `if_table` inside this procedure.

⟨*Auto components: if table: TBP*⟩+≡

```
  procedure :: insert => if_table_insert
```

⟨*Auto components: procedures*⟩+≡

```
  recursive subroutine if_table_insert  &
      (table, pl, n_rad, i, pdg, constraints, n_in)
    class(if_table_t), intent(inout) :: table
    class(pdg_list_t), intent(in) :: pl
    integer, intent(in) :: n_rad, i
    integer, dimension(:), intent(in) :: pdg
    type(split_constraints_t), intent(in) :: constraints
    integer, intent(in), optional :: n_in
    integer, dimension(:), allocatable :: pdg_work
    integer :: p
    if (i > 2) then
       call ps_table_insert (table, pl, n_rad, i, pdg, constraints)
    else
       allocate (pdg_work (size (pdg)))
       do p = 1, size (pdg)
          pdg_work(1) = pdg(p)
          pdg_work(2:p) = pdg(1:p-1)
          pdg_work(p+1:) = pdg(p+1:)
          select case (table%proc_type)
          case (PROC_DECAY)
             call ps_table_insert (table, &
               pl, n_rad, i, pdg_work, constraints, n_in = 1)
          case (PROC_SCATTER)
             call ps_table_insert (table, &
               pl, n_rad, i, pdg_work, constraints, n_in = 2)
          end select
       end do
    end if
  end subroutine if_table_insert
```

Sort before recording. In the case of the `if_table`, we do not sort the first `n_in` particle entries. Instead, we check whether they are allowed in the `pl_beam` PDG list, if that is provided.

⟨*Auto components: ps table: TBP*⟩+≡

```
  procedure :: record_sorted => ps_table_record_sorted
```

⟨*Auto components: if table: TBP*⟩+≡

```
  procedure :: record_sorted => if_table_record_sorted
```

⟨*Auto components: procedures*⟩+≡

```
subroutine ps_table_record_sorted &
     (table, pl, n_loop, n_rad, constraints, passed)
  class(ps_table_t), intent(inout) :: table
  type(pdg_list_t), intent(in) :: pl
  integer, intent(in) :: n_loop, n_rad
  type(split_constraints_t), intent(in) :: constraints
  logical, intent(out) :: passed
  call table%record (pl%sort_abs (), n_loop, n_rad, constraints, passed)
end subroutine ps_table_record_sorted

subroutine if_table_record_sorted &
     (table, pl, n_loop, n_rad, constraints, passed)
  class(if_table_t), intent(inout) :: table
  type(pdg_list_t), intent(in) :: pl
  integer, intent(in) :: n_loop, n_rad
  type(split_constraints_t), intent(in) :: constraints
  logical, intent(out) :: passed
  call table%record (pl%sort_abs (2), n_loop, n_rad, constraints, passed)
end subroutine if_table_record_sorted
```

Record an entry: insert into the list. Check the ordering and insert it at the correct place, unless it is already there.

We record an array only if its mass sum is less than the total available energy. This restriction is removed by setting `constrained` to false.

⟨*Auto components: ps table: TBP*⟩+≡

```
procedure :: record => ps_table_record
```

⟨*Auto components: procedures*⟩+≡

```
subroutine ps_table_record (table, pl, n_loop, n_rad, constraints, passed)
  class(ps_table_t), intent(inout) :: table
  type(pdg_list_t), intent(in) :: pl
  integer, intent(in) :: n_loop, n_rad
  type(split_constraints_t), intent(in) :: constraints
  logical, intent(out) :: passed
  type(ps_entry_t), pointer :: current
  passed = .false.
  if (.not. pl%is_regular ()) then
     call msg_warning ("Record ps_table entry: Irregular pdg-list encountered!")
     return
  end if
  call constraints%check_before_record (table, pl, n_loop, passed)
  if (.not. passed)  then
     return
  end if
  current => table%first
  do while (associated (current))
     if (pl == current) then
        if (n_loop == current%n_loop)  return
     else if (pl < current) then
        call insert
        return
     end if
     current => current%next
```

```
    end do
    call insert
  contains
    subroutine insert ()
      type(ps_entry_t), pointer :: entry
      allocate (entry)
      entry%pdg_list_t = pl
      entry%n_loop = n_loop
      entry%n_rad = n_rad
      if (associated (current)) then
         if (associated (current%previous)) then
            current%previous%next => entry
            entry%previous => current%previous
         else
            table%first => entry
         end if
         entry%next => current
         current%previous => entry
      else
         if (associated (table%last)) then
            table%last%next => entry
            entry%previous => table%last
         else
            table%first => entry
         end if
         table%last => entry
      end if
    end subroutine insert
  end subroutine ps_table_record
```

28.1.6 Tools

Compute the mass sum for a PDG list object, counting the entries with indices between (including) `n1` and `n2`. Rely on the requirement that if an entry is a PDG array, this array must be degenerate in mass.

⟨*Auto components: procedures*⟩+≡

```
  function mass_sum (pl, n1, n2, model) result (m)
    type(pdg_list_t), intent(in) :: pl
    integer, intent(in) :: n1, n2
    class(model_data_t), intent(in), target :: model
    integer, dimension(:), allocatable :: pdg
    real(default) :: m
    type(field_data_t), pointer :: prt
    integer :: i
    m = 0
    do i = n1, n2
       pdg = pl%get (i)
       prt => model%get_field_ptr (pdg(1))
       m = m + prt%get_mass ()
    end do
  end function mass_sum
```

Invert a PDG array, replacing particles by antiparticles. This depends on the model.

⟨*Auto components: procedures*⟩+≡

```
function invert_pdg_array (pa, model) result (pa_inv)
  type(pdg_array_t), intent(in) :: pa
  class(model_data_t), intent(in), target :: model
  type(pdg_array_t) :: pa_inv
  type(field_data_t), pointer :: prt
  integer :: i, pdg
  pa_inv = pa
  do i = 1, pa_inv%get_length ()
     pdg = pa_inv%get (i)
     prt => model%get_field_ptr (pdg)
     if (prt%has_antiparticle ())  call pa_inv%set (i, -pdg)
  end do
end function invert_pdg_array
```

28.1.7 Access results

Return the number of generated decays.

⟨*Auto components: ps table: TBP*⟩+≡

```
procedure :: get_length => ps_table_get_length
```

⟨*Auto components: procedures*⟩+≡

```
function ps_table_get_length (ps_table) result (n)
  class(ps_table_t), intent(in) :: ps_table
  integer :: n
  type(ps_entry_t), pointer :: entry
  n = 0
  entry => ps_table%first
  do while (associated (entry))
     n = n + 1
     entry => entry%next
  end do
end function ps_table_get_length
```

⟨*Auto components: ps table: TBP*⟩+≡

```
procedure :: get_emitters => ps_table_get_emitters
```

⟨*Auto components: procedures*⟩+≡

```
subroutine ps_table_get_emitters (table, constraints, emitters)
   class(ps_table_t), intent(in) :: table
   type(split_constraints_t), intent(in) :: constraints
   integer, dimension(:), allocatable, intent(out) :: emitters
   class(pdg_list_t), pointer :: pl
   integer :: i
   logical :: passed
   type(vertex_iterator_t) :: vit
   integer, dimension(:), allocatable :: pdg1, pdg2
   integer :: n_emitters
   integer, dimension(:), allocatable :: emitters_tmp
   integer, parameter :: buf0 = 6
   n_emitters = 0
```

```
    pl => table%first
    allocate (emitters_tmp (buf0))
    do i = 1, pl%get_size ()
       call constraints%check_before_split (table, pl, i, passed)
       if (passed) then
          pdg1 = pl%get(i)
          call vit%init (table%model, pdg1, .false.)
          do
             call vit%get_next_match(pdg2)
             if (allocated (pdg2)) then
                if (n_emitters + 1 > size (emitters_tmp)) &
                     call extend_integer_array (emitters_tmp, 10)
                emitters_tmp (n_emitters + 1) = pdg1(1)
                n_emitters = n_emitters + 1
             else
                exit
             end if
          end do
       end if
    end do
    allocate (emitters (n_emitters))
    emitters = emitters_tmp (1:n_emitters)
    deallocate (emitters_tmp)
end subroutine ps_table_get_emitters
```

Return an allocated array of decay products (PDG codes). If requested, return also the loop and radiation order count.

⟨*Auto components: ps table: TBP*⟩+≡

```
procedure :: get_pdg_out => ps_table_get_pdg_out
```

⟨*Auto components: procedures*⟩+≡

```
subroutine ps_table_get_pdg_out (ps_table, i, pa_out, n_loop, n_rad)
  class(ps_table_t), intent(in) :: ps_table
  integer, intent(in) :: i
  type(pdg_array_t), dimension(:), allocatable, intent(out) :: pa_out
  integer, intent(out), optional :: n_loop, n_rad
  type(ps_entry_t), pointer :: entry
  integer :: n, j
  n = 0
  entry => ps_table%first
  FIND_ENTRY: do while (associated (entry))
     n = n + 1
     if (n == i) then
        allocate (pa_out (entry%get_size ()))
        do j = 1, entry%get_size ()
           pa_out(j) = entry%get (j)
           if (present (n_loop))  n_loop = entry%n_loop
           if (present (n_rad))  n_rad = entry%n_rad
        end do
        exit FIND_ENTRY
     end if
     entry => entry%next
  end do FIND_ENTRY
end subroutine ps_table_get_pdg_out
```

28.1.8 Unit tests

Test module, followed by the corresponding implementation module.

⟨auto_components_ut.f90⟩≡

```
⟨File header⟩

module auto_components_ut
  use unit_tests
  use auto_components_uti

⟨Standard module head⟩

⟨Auto components: public test⟩

contains

⟨Auto components: test driver⟩

end module auto_components_ut
```

⟨auto_components_uti.f90⟩≡

```
⟨File header⟩

module auto_components_uti

⟨Use kinds⟩
⟨Use strings⟩
  use pdg_arrays
  use model_data
  use model_testbed, only: prepare_model, cleanup_model

  use auto_components

⟨Standard module head⟩

⟨Auto components: test declarations⟩

contains

⟨Auto components: tests⟩

end module auto_components_uti
```

API: driver for the unit tests below.

⟨*Auto components: public test*⟩≡

```
  public :: auto_components_test
```

⟨*Auto components: test driver*⟩≡

```
  subroutine auto_components_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨Auto components: execute tests⟩
  end subroutine auto_components_test
```

Generate Decay Table

Determine all kinematically allowed decay channels for a Higgs boson, using default parameter values.

⟨*Auto components: execute tests*⟩≡
```
call test (auto_components_1, "auto_components_1", &
     "generate decay table", &
     u, results)
```

⟨*Auto components: test declarations*⟩≡
```
public :: auto_components_1
```

⟨*Auto components: tests*⟩≡
```
subroutine auto_components_1 (u)
  integer, intent(in) :: u
  class(model_data_t), pointer :: model
  type(field_data_t), pointer :: prt
  type(ds_table_t) :: ds_table
  type(split_constraints_t) :: constraints

  write (u, "(A)")  "* Test output: auto_components_1"
  write (u, "(A)")  "*   Purpose: determine Higgs decay table"
  write (u, *)

  write (u, "(A)")  "* Read Standard Model"

  call prepare_model (model, var_str ("SM"))

  prt => model%get_field_ptr (25)

  write (u, *)
  write (u, "(A)")  "* Higgs decays n = 2"
  write (u, *)

  call constraints%init (2)
  call constraints%set (1, constrain_n_tot (2))
  call constraints%set (2, constrain_mass_sum (prt%get_mass ()))

  call ds_table%make (model, 25, constraints)
  call ds_table%write (u)
  call ds_table%final ()

  write (u, *)
  write (u, "(A)")  "* Higgs decays n = 3 (w/o radiative)"
  write (u, *)

  call constraints%init (3)
  call constraints%set (1, constrain_n_tot (3))
  call constraints%set (2, constrain_mass_sum (prt%get_mass ()))
  call constraints%set (3, constrain_radiation ())

  call ds_table%make (model, 25, constraints)
```

```
    call ds_table%write (u)
    call ds_table%final ()

    write (u, *)
    write (u, "(A)")  "* Higgs decays n = 3 (w/ radiative)"
    write (u, *)

    call constraints%init (2)
    call constraints%set (1, constrain_n_tot (3))
    call constraints%set (2, constrain_mass_sum (prt%get_mass ()))

    call ds_table%make (model, 25, constraints)
    call ds_table%write (u)
    call ds_table%final ()

    write (u, *)
    write (u, "(A)")  "* Cleanup"

    call cleanup_model (model)

    write (u, *)
    write (u, "(A)")  "* Test output end: auto_components_1"

  end subroutine auto_components_1
```

Generate radiation

Given a final state, add radiation (NLO and NNLO). We provide a list of particles that is allowed to occur in the generated final states.

⟨*Auto components: execute tests*⟩+≡

```
  call test (auto_components_2, "auto_components_2", &
       "generate NLO corrections, final state", &
       u, results)
```

⟨*Auto components: test declarations*⟩+≡

```
  public :: auto_components_2
```

⟨*Auto components: tests*⟩+≡

```
  subroutine auto_components_2 (u)
    integer, intent(in) :: u
    class(model_data_t), pointer :: model
    type(pdg_list_t), dimension(:), allocatable :: pl, pl_zzh
    type(pdg_list_t) :: pl_match
    type(fs_table_t) :: fs_table
    type(split_constraints_t) :: constraints
    real(default) :: sqrts
    integer :: i

    write (u, "(A)")  "* Test output: auto_components_2"
    write (u, "(A)")  "*   Purpose: generate radiation (NLO)"
    write (u, *)
```

```
write (u, "(A)")  "* Read Standard Model"

call prepare_model (model, var_str ("SM"))

write (u, *)
write (u, "(A)")  "* LO final state"
write (u, *)

allocate (pl (2))
call pl(1)%init (2)
call pl(1)%set (1, 1)
call pl(1)%set (2, -1)
call pl(2)%init (2)
call pl(2)%set (1, 21)
call pl(2)%set (2, 21)
do i = 1, 2
   call pl(i)%write (u);  write (u, *)
end do

write (u, *)
write (u, "(A)")  "* Initialize FS table"
write (u, *)

call constraints%init (1)
call constraints%set (1, constrain_n_tot (3))

call fs_table%init (model, pl, constraints)
call fs_table%write (u)

write (u, *)
write (u, "(A)")  "* Generate NLO corrections, unconstrained"
write (u, *)

call fs_table%radiate (constraints)
call fs_table%write (u)
call fs_table%final ()

write (u, *)
write (u, "(A)")  "* Generate NLO corrections, &
     &complete but mass-constrained"
write (u, *)

sqrts = 50

call constraints%init (2)
call constraints%set (1, constrain_n_tot (3))
call constraints%set (2, constrain_mass_sum (sqrts))

call fs_table%init (model, pl, constraints)
call fs_table%radiate (constraints)
call fs_table%write (u)
call fs_table%final ()

write (u, *)
```

```
write (u, "(A)")  "* Generate NLO corrections, restricted"
write (u, *)

call pl_match%init ([1, -1, 21])

call constraints%init (2)
call constraints%set (1, constrain_n_tot (3))
call constraints%set (2, constrain_insert (pl_match))

call fs_table%init (model, pl, constraints)
call fs_table%radiate (constraints)
call fs_table%write (u)
call fs_table%final ()

write (u, *)
write (u, "(A)")  "* Generate NNLO corrections, restricted, with one loop"
write (u, *)

call pl_match%init ([1, -1, 21])

call constraints%init (3)
call constraints%set (1, constrain_n_tot (4))
call constraints%set (2, constrain_n_loop (1))
call constraints%set (3, constrain_insert (pl_match))

call fs_table%init (model, pl, constraints)
call fs_table%enable_loops ()
call fs_table%radiate (constraints)
call fs_table%write (u)
call fs_table%final ()

write (u, *)
write (u, "(A)")  "* Generate NNLO corrections, restricted, with loops"
write (u, *)

call constraints%init (2)
call constraints%set (1, constrain_n_tot (4))
call constraints%set (2, constrain_insert (pl_match))

call fs_table%init (model, pl, constraints)
call fs_table%enable_loops ()
call fs_table%radiate (constraints)
call fs_table%write (u)
call fs_table%final ()

write (u, *)
write (u, "(A)")  "* Generate NNLO corrections, restricted, to Z Z H, &
     &no loops"
write (u, *)

allocate (pl_zzh (1))
call pl_zzh(1)%init (3)
call pl_zzh(1)%set (1, 23)
call pl_zzh(1)%set (2, 23)
```

```
    call pl_zzh(1)%set (3, 25)

    call constraints%init (3)
    call constraints%set (1, constrain_n_tot (5))
    call constraints%set (2, constrain_mass_sum (500._default))
    call constraints%set (3, constrain_require (pl_zzh(1)))

    call fs_table%init (model, pl_zzh, constraints)
    call fs_table%radiate (constraints)
    call fs_table%write (u)
    call fs_table%final ()

    call cleanup_model (model)

    write (u, *)
    write (u, "(A)")  "* Test output end: auto_components_2"

  end subroutine auto_components_2
```

Generate radiation from initial and final state

Given a process, add radiation (NLO and NNLO). We provide a list of particles that is allowed to occur in the generated final states.

⟨*Auto components: execute tests*⟩+≡

```
  call test (auto_components_3, "auto_components_3", &
       "generate NLO corrections, in and out", &
       u, results)
```

⟨*Auto components: test declarations*⟩+≡

```
  public :: auto_components_3
```

⟨*Auto components: tests*⟩+≡

```
  subroutine auto_components_3 (u)
    integer, intent(in) :: u
    class(model_data_t), pointer :: model
    type(pdg_list_t), dimension(:), allocatable :: pl_in, pl_out
    type(pdg_list_t) :: pl_match, pl_beam
    type(if_table_t) :: if_table
    type(split_constraints_t) :: constraints
    real(default) :: sqrts
    integer :: i

    write (u, "(A)")  "* Test output: auto_components_3"
    write (u, "(A)")  "*   Purpose: generate radiation (NLO)"
    write (u, *)

    write (u, "(A)")  "* Read Standard Model"

    call prepare_model (model, var_str ("SM"))

    write (u, *)
    write (u, "(A)")  "* LO initial state"
    write (u, *)
```

```
allocate (pl_in (2))
call pl_in(1)%init (2)
call pl_in(1)%set (1, 1)
call pl_in(1)%set (2, -1)
call pl_in(2)%init (2)
call pl_in(2)%set (1, -1)
call pl_in(2)%set (2, 1)
do i = 1, 2
   call pl_in(i)%write (u);  write (u, *)
end do

write (u, *)
write (u, "(A)")  "* LO final state"
write (u, *)

allocate (pl_out (1))
call pl_out(1)%init (1)
call pl_out(1)%set (1, 23)
call pl_out(1)%write (u);  write (u, *)

write (u, *)
write (u, "(A)")  "* Initialize FS table"
write (u, *)

call constraints%init (1)
call constraints%set (1, constrain_n_tot (4))

call if_table%init (model, pl_in, pl_out, constraints)
call if_table%write (u)

write (u, *)
write (u, "(A)")  "* Generate NLO corrections, unconstrained"
write (u, *)

call if_table%radiate (constraints)
call if_table%write (u)
call if_table%final ()

write (u, *)
write (u, "(A)")  "* Generate NLO corrections, &
     &complete but mass-constrained"
write (u, *)

sqrts = 100
call constraints%init (2)
call constraints%set (1, constrain_n_tot (4))
call constraints%set (2, constrain_mass_sum (sqrts))

call if_table%init (model, pl_in, pl_out, constraints)
call if_table%radiate (constraints)
call if_table%write (u)
call if_table%final ()
```

```
write (u, *)
write (u, "(A)")  "* Generate NLO corrections, &
     &mass-constrained, restricted beams"
write (u, *)

call pl_beam%init (3)
call pl_beam%set (1, 1)
call pl_beam%set (2, -1)
call pl_beam%set (3, 21)

call constraints%init (3)
call constraints%set (1, constrain_n_tot (4))
call constraints%set (2, constrain_in_state (pl_beam))
call constraints%set (3, constrain_mass_sum (sqrts))

call if_table%init (model, pl_in, pl_out, constraints)
call if_table%radiate (constraints)
call if_table%write (u)
call if_table%final ()

write (u, *)
write (u, "(A)")  "* Generate NLO corrections, restricted"
write (u, *)

call pl_match%init ([1, -1, 21])

call constraints%init (4)
call constraints%set (1, constrain_n_tot (4))
call constraints%set (2, constrain_in_state (pl_beam))
call constraints%set (3, constrain_mass_sum (sqrts))
call constraints%set (4, constrain_insert (pl_match))

call if_table%init (model, pl_in, pl_out, constraints)
call if_table%radiate (constraints)
call if_table%write (u)
call if_table%final ()

write (u, *)
write (u, "(A)")  "* Generate NNLO corrections, restricted, Z preserved, &
     &with loops"
write (u, *)

call constraints%init (5)
call constraints%set (1, constrain_n_tot (5))
call constraints%set (2, constrain_in_state (pl_beam))
call constraints%set (3, constrain_mass_sum (sqrts))
call constraints%set (4, constrain_insert (pl_match))
call constraints%set (5, constrain_require (pl_out(1)))

call if_table%init (model, pl_in, pl_out, constraints)
call if_table%enable_loops ()
call if_table%radiate (constraints)
call if_table%write (u)
call if_table%final ()
```

```
      call cleanup_model (model)

      write (u, *)
      write (u, "(A)")  "* Test output end: auto_components_3"

    end subroutine auto_components_3
```

28.2 Creating the real flavor structure

⟨radiation_generator.f90⟩≡
  ⟨*File header*⟩

```
  module radiation_generator
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩

```
    use diagnostics
    use io_units
    use physics_defs, only: PHOTON, GLUON
    use pdg_arrays
    use flavors
    use model_data
    use auto_components
    use string_utils, only: split_string

    implicit none
    private
```

  ⟨*radiation generator: public*⟩

  ⟨*radiation generator: types*⟩

```
  contains
```

  ⟨*radiation generator: procedures*⟩

```
  end module radiation_generator
```

⟨*radiation generator: types*⟩≡

```
    type :: pdg_sorter_t
       integer :: pdg
       logical :: checked = .false.
       integer :: associated_born = 0
    end type pdg_sorter_t
```

⟨*radiation generator: types*⟩+≡

```
    type :: pdg_states_t
      type(pdg_array_t), dimension(:), allocatable :: pdg
      type(pdg_states_t), pointer :: next
      integer :: n_particles
    contains
```

```
⟨radiation generator: pdg states: TBP⟩
end type pdg_states_t
```

⟨*radiation generator: pdg states: TBP*⟩≡

```
procedure :: init => pdg_states_init
```

⟨*radiation generator: procedures*⟩≡

```
subroutine pdg_states_init (states)
  class(pdg_states_t), intent(inout) :: states
  nullify (states%next)
end subroutine pdg_states_init
```

⟨*radiation generator: pdg states: TBP*⟩+≡

```
procedure :: add => pdg_states_add
```

⟨*radiation generator: procedures*⟩+≡

```
subroutine pdg_states_add (states, pdg)
  class(pdg_states_t), intent(inout), target :: states
  type(pdg_array_t), dimension(:), intent(in) :: pdg
  type(pdg_states_t), pointer :: current_state
  select type (states)
  type is (pdg_states_t)
    current_state => states
    do
      if (associated (current_state%next)) then
        current_state => current_state%next
      else
        allocate (current_state%next)
        nullify(current_state%next%next)
        current_state%pdg = pdg
        exit
      end if
    end do
  end select
end subroutine pdg_states_add
```

⟨*radiation generator: pdg states: TBP*⟩+≡

```
procedure :: get_n_states => pdg_states_get_n_states
```

⟨*radiation generator: procedures*⟩+≡

```
function pdg_states_get_n_states (states) result (n)
  class(pdg_states_t), intent(in), target :: states
  integer :: n
  type(pdg_states_t), pointer :: current_state
  n = 0
  select type(states)
  type is (pdg_states_t)
    current_state => states
    do
      if (associated (current_state%next)) then
        n = n+1
        current_state => current_state%next
      else
        exit
```

```
        end if
      end do
    end select
  end function pdg_states_get_n_states
```

⟨*radiation generator: types*⟩+≡

```
  type :: prt_queue_t
    type(string_t), dimension(:), allocatable :: prt_string
    type(prt_queue_t), pointer :: next => null ()
    type(prt_queue_t), pointer :: previous => null ()
    type(prt_queue_t), pointer :: front => null ()
    type(prt_queue_t), pointer :: current_prt => null ()
    type(prt_queue_t), pointer :: back => null ()
    integer :: n_lists = 0
  contains
  ⟨radiation generator: prt queue: TBP⟩
  end type prt_queue_t
```

⟨*radiation generator: prt queue: TBP*⟩≡

```
  procedure :: null => prt_queue_null
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine prt_queue_null (queue)
    class(prt_queue_t), intent(out) :: queue
    queue%next => null ()
    queue%previous => null ()
    queue%front => null ()
    queue%current_prt => null ()
    queue%back => null ()
    queue%n_lists = 0
    if (allocated (queue%prt_string))  deallocate (queue%prt_string)
  end subroutine prt_queue_null
```

⟨*radiation generator: prt queue: TBP*⟩+≡

```
  procedure :: append => prt_queue_append
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine prt_queue_append (queue, prt_string)
    class(prt_queue_t), intent(inout) :: queue
    type(string_t), intent(in), dimension(:) :: prt_string
    type(prt_queue_t), pointer :: new_element => null ()
    type(prt_queue_t), pointer :: current_back => null ()
    allocate (new_element)
    allocate (new_element%prt_string(size (prt_string)))
    new_element%prt_string = prt_string
    if (associated (queue%back)) then
       current_back => queue%back
       current_back%next => new_element
       new_element%previous => current_back
       queue%back => new_element
    else
       !!! Initial entry
       queue%front => new_element
```

```
      queue%back => queue%front
      queue%current_prt => queue%front
    end if
    queue%n_lists = queue%n_lists + 1
  end subroutine prt_queue_append
```

`gfortran 4.7.4` does not support allocate-on-assignment for the caller when this is a function.

⟨*radiation generator: prt queue: TBP*⟩+≡

```
  procedure :: get => prt_queue_get
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine prt_queue_get (queue, prt_string)
    class(prt_queue_t), intent(inout) :: queue
    type(string_t), dimension(:), allocatable, intent(out) :: prt_string
    if (associated (queue%current_prt)) then
      allocate (prt_string(size (queue%current_prt%prt_string)))
      prt_string = queue%current_prt%prt_string
      if (associated (queue%current_prt%next)) &
        queue%current_prt => queue%current_prt%next
    else
      prt_string = " "
    end if
  end subroutine prt_queue_get
```

As above.

⟨*radiation generator: prt queue: TBP*⟩+≡

```
  procedure :: get_last => prt_queue_get_last
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine prt_queue_get_last (queue, prt_string)
    class(prt_queue_t), intent(in) :: queue
    type(string_t), dimension(:), allocatable, intent(out) :: prt_string
    if (associated (queue%back)) then
      allocate (prt_string(size (queue%back%prt_string)))
      prt_string = queue%back%prt_string
    else
      prt_string = " "
    end if
  end subroutine prt_queue_get_last
```

⟨*radiation generator: prt queue: TBP*⟩+≡

```
  procedure :: reset => prt_queue_reset
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine prt_queue_reset (queue)
    class(prt_queue_t), intent(inout) :: queue
    queue%current_prt => queue%front
  end subroutine prt_queue_reset
```

⟨*radiation generator: prt queue: TBP*⟩+≡

```
  procedure :: check_for_same_prt_strings => prt_queue_check_for_same_prt_strings
```

⟨*radiation generator: procedures*⟩+≡

```
function prt_queue_check_for_same_prt_strings (queue) result (val)
  class(prt_queue_t), intent(inout) :: queue
  logical :: val
  type(string_t), dimension(:), allocatable :: prt_string
  integer, dimension(:,:), allocatable :: i_particle
  integer :: n_d, n_dbar, n_u, n_ubar, n_s, n_sbar, n_gl, n_e, n_ep, n_mu, n_mup, n_A
  integer :: i, j
  call queue%reset ()
  allocate (i_particle (queue%n_lists, 12))
  do i = 1, queue%n_lists
     call queue%get (prt_string)
     n_d = count_particle (prt_string, 1)
     n_dbar = count_particle (prt_string, -1)
     n_u = count_particle (prt_string, 2)
     n_ubar = count_particle (prt_string, -2)
     n_s = count_particle (prt_string, 3)
     n_sbar = count_particle (prt_string, -3)
     n_gl = count_particle (prt_string, 21)
     n_e = count_particle (prt_string, 11)
     n_ep = count_particle (prt_string, -11)
     n_mu = count_particle (prt_string, 13)
     n_mup = count_particle (prt_string, -13)
     n_A = count_particle (prt_string, 22)
     i_particle (i, 1) = n_d
     i_particle (i, 2) = n_dbar
     i_particle (i, 3) = n_u
     i_particle (i, 4) = n_ubar
     i_particle (i, 5) = n_s
     i_particle (i, 6) = n_sbar
     i_particle (i, 7) = n_gl
     i_particle (i, 8) = n_e
     i_particle (i, 9) = n_ep
     i_particle (i, 10) = n_mu
     i_particle (i, 11) = n_mup
     i_particle (i, 12) = n_A
  end do
  val = .false.
  do i = 1, queue%n_lists
     do j = 1, queue%n_lists
        if  (i == j) cycle
        val = val .or. all (i_particle (i,:) == i_particle(j,:))
     end do
  end do
contains
  function count_particle (prt_string, pdg) result (n)
    type(string_t), dimension(:), intent(in) :: prt_string
    integer, intent(in) :: pdg
    integer :: n
    integer :: i
    type(string_t) :: prt_ref
    n = 0
    select case (pdg)
    case (1)
```

```
         prt_ref = "d"
      case (-1)
         prt_ref = "dbar"
      case (2)
         prt_ref = "u"
      case (-2)
         prt_ref = "ubar"
      case (3)
         prt_ref = "s"
      case (-3)
         prt_ref = "sbar"
      case (21)
         prt_ref = "gl"
      case (11)
         prt_ref = "e-"
      case (-11)
         prt_ref = "e+"
      case (13)
         prt_ref = "mu-"
      case (-13)
         prt_ref = "mu+"
      case (22)
         prt_ref = "A"
      end select
      do i = 1, size (prt_string)
         if (prt_string(i) == prt_ref) n = n+1
      end do
    end function count_particle

  end function prt_queue_check_for_same_prt_strings
```

⟨*radiation generator: prt queue: TBP*⟩+≡

```
  procedure :: contains => prt_queue_contains
```

⟨*radiation generator: procedures*⟩+≡

```
  function prt_queue_contains (queue, prt_string) result (val)
    class(prt_queue_t), intent(in) :: queue
    type(string_t), intent(in), dimension(:) :: prt_string
    logical :: val
    type(prt_queue_t), pointer :: current => null()
    if (associated (queue%front)) then
       current => queue%front
    else
       call msg_fatal ("Trying to access empty particle queue")
    end if
    val = .false.
    do
       if (size (current%prt_string) == size (prt_string)) then
          if (all (current%prt_string == prt_string)) then
             val = .true.
             exit
          end if
       end if
       if (associated (current%next)) then
```

```
            current => current%next
         else
            exit
         end if
      end do
   end function prt_queue_contains
```

⟨*radiation generator: prt queue: TBP*⟩+≡

```
   procedure :: write => prt_queue_write
```

⟨*radiation generator: procedures*⟩+≡

```
   subroutine prt_queue_write (queue, unit)
      class(prt_queue_t), intent(in) :: queue
      integer, optional :: unit
      type(prt_queue_t), pointer :: current => null ()
      integer :: i, j, u
      u = given_output_unit (unit)
      if (associated (queue%front)) then
         current => queue%front
      else
         write (u, "(A)") "[Particle queue is empty]"
         return
      end if
      j = 1
      do
         write (u, "(I2,A,1X)", advance = 'no') j , ":"
         do i = 1, size (current%prt_string)
            write (u, "(A,1X)", advance = 'no') char (current%prt_string(i))
         end do
         write (u, "(A)")
         if (associated (current%next)) then
            current => current%next
            j = j+1
         else
            exit
         end if
      end do
   end subroutine prt_queue_write
```

⟨*radiation generator: procedures*⟩+≡

```
   subroutine sort_prt (prt, model)
      type(string_t), dimension(:), intent(inout) :: prt
      class(model_data_t), intent(in), target :: model
      type(pdg_array_t), dimension(:), allocatable :: pdg
      type(flavor_t) :: flv
      integer :: i
      call create_pdg_array (prt, model, pdg)
      call sort_pdg (pdg)
      do i = 1, size (pdg)
         call flv%init (pdg(i)%get(), model)
         prt(i) = flv%get_name ()
      end do
   end subroutine sort_prt
```

```
subroutine sort_pdg (pdg)
  type(pdg_array_t), dimension(:), intent(inout) :: pdg
  integer, dimension(:), allocatable :: i_pdg
  integer :: i
  allocate (i_pdg (size (pdg)))
  !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
  do i = 1, size (pdg)
     i_pdg(i) = pdg(i)%get ()
  end do
  i_pdg = sort_abs (i_pdg)
  do i = 1, size (pdg)
     call pdg(i)%set (1, i_pdg(i))
  end do
end subroutine sort_pdg

subroutine create_pdg_array (prt, model, pdg)
  type (string_t), dimension(:), intent(in) :: prt
  class (model_data_t), intent(in), target :: model
  type(pdg_array_t), dimension(:), allocatable, intent(out) :: pdg
  type(flavor_t) :: flv
  integer :: i
  allocate (pdg (size (prt)))
  do i = 1, size (prt)
     call flv%init (prt(i), model)
     pdg(i) = flv%get_pdg ()
  end do
end subroutine create_pdg_array
```

This is used in unit tests:

⟨*radiation generator: test auxiliary*⟩≡

```
subroutine write_pdg_array (pdg, u)
  use pdg_arrays
  type(pdg_array_t), dimension(:), intent(in) :: pdg
  integer, intent(in) :: u
  integer :: i
  do i = 1, size (pdg)
     call pdg(i)%write (u)
  end do
  write (u, "(A)")
end subroutine write_pdg_array

subroutine write_particle_string (prt, u)
⟨Use strings⟩
  type(string_t), dimension(:), intent(in) :: prt
  integer, intent(in) :: u
  integer :: i
  do i = 1, size (prt)
     write (u, "(A,1X)", advance = "no") char (prt(i))
  end do
  write (u, "(A)")
end subroutine write_particle_string
```

⟨*radiation generator: types*⟩+≡

```
type :: reshuffle_list_t
   integer, dimension(:), allocatable :: ii
   type(reshuffle_list_t), pointer :: next => null ()
contains
⟨radiation generator: reshuffle list: TBP⟩
end type reshuffle_list_t
```

⟨*radiation generator: reshuffle list: TBP*⟩≡

```
procedure :: write => reshuffle_list_write
```

⟨*radiation generator: procedures*⟩+≡

```
subroutine reshuffle_list_write (rlist)
  class(reshuffle_list_t), intent(in) :: rlist
  type(reshuffle_list_t), pointer :: current => null ()
  integer :: i
  print *, 'Content of reshuffling list: '
  if (associated (rlist%next)) then
     current => rlist%next
     i = 1
     do
       print *, 'i: ', i, 'list: ', current%ii
       i = i + 1
       if (associated (current%next)) then
          current => current%next
       else
          exit
       end if
     end do
  else
     print *, '[EMPTY]'
  end if
end subroutine reshuffle_list_write
```

⟨*radiation generator: reshuffle list: TBP*⟩+≡

```
procedure :: append => reshuffle_list_append
```

⟨*radiation generator: procedures*⟩+≡

```
subroutine reshuffle_list_append (rlist, ii)
  class(reshuffle_list_t), intent(inout) :: rlist
  integer, dimension(:), allocatable, intent(in) :: ii
  type(reshuffle_list_t), pointer :: current
  if (associated (rlist%next)) then
     current => rlist%next
     do
        if (associated (current%next)) then
           current => current%next
        else
           allocate (current%next)
           allocate (current%next%ii (size (ii)))
           current%next%ii = ii
           exit
        end if
     end do
```

```
    else
       allocate (rlist%next)
       allocate (rlist%next%ii (size (ii)))
       rlist%next%ii = ii
    end if
  end subroutine reshuffle_list_append
```

⟨*radiation generator: reshuffle list: TBP*⟩+≡

```
  procedure :: is_empty => reshuffle_list_is_empty
```

⟨*radiation generator: procedures*⟩+≡

```
  elemental function reshuffle_list_is_empty (rlist) result (is_empty)
    logical :: is_empty
    class(reshuffle_list_t), intent(in) :: rlist
    is_empty = .not. associated (rlist%next)
  end function reshuffle_list_is_empty
```

⟨*radiation generator: reshuffle list: TBP*⟩+≡

```
  procedure :: get => reshuffle_list_get
```

⟨*radiation generator: procedures*⟩+≡

```
  function reshuffle_list_get (rlist, index) result (ii)
    integer, dimension(:), allocatable :: ii
    class(reshuffle_list_t), intent(inout) :: rlist
    integer, intent(in) :: index
    type(reshuffle_list_t), pointer :: current => null ()
    integer :: i
    current => rlist%next
    do i = 1, index - 1
       if (associated (current%next)) then
          current => current%next
       else
          call msg_fatal ("Index exceeds size of reshuffling list")
       end if
    end do
    allocate (ii (size (current%ii)))
    ii = current%ii
  end function reshuffle_list_get
```

We need to reset the `reshuffle_list` in order to deal with subsequent usages of the `radiation_generator`. Below is obviously the lazy and dirty solution. Otherwise, we would have to equip this auxiliary type with additional information about `last` and `previous` pointers. Considering that at most n_{legs} integers are saved in the lists, and that the subroutine is only called during the initialization phase (more precisely: at the moment only in the `radiation_generator` unit tests), I think this quick fix is justified.

⟨*radiation generator: reshuffle list: TBP*⟩+≡

```
  procedure :: reset => reshuffle_list_reset
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine reshuffle_list_reset (rlist)
    class(reshuffle_list_t), intent(inout) :: rlist
    rlist%next => null ()
```

```
    end subroutine reshuffle_list_reset
```

⟨*radiation generator: public*⟩≡

```
    public :: radiation_generator_t
```

⟨*radiation generator: types*⟩+≡

```
    type :: radiation_generator_t
      logical :: qcd_enabled = .false.
      logical :: qed_enabled = .false.
      logical :: is_gluon = .false.
      logical :: fs_gluon = .false.
      logical :: is_photon = .false.
      logical :: fs_photon = .false.
      logical :: only_final_state = .true.
      type(pdg_list_t) :: pl_in, pl_out
      type(pdg_list_t) :: pl_excluded_gauge_splittings
      type(split_constraints_t) :: constraints
      integer :: n_tot
      integer :: n_in, n_out
      integer :: n_loops
      integer :: n_light_quarks
      real(default) :: mass_sum
      type(prt_queue_t) :: prt_queue
      type(pdg_states_t) :: pdg_raw
      type(pdg_array_t), dimension(:), allocatable :: pdg_in_born, pdg_out_born
      type(if_table_t) :: if_table
      type(reshuffle_list_t) :: reshuffle_list
    contains
    ⟨radiation generator: radiation generator: TBP⟩
    end type radiation_generator_t
```

⟨*radiation generator: radiation generator: TBP*⟩≡

```
    generic :: init => init_pdg_list, init_pdg_array
    procedure :: init_pdg_list => radiation_generator_init_pdg_list
    procedure :: init_pdg_array => radiation_generator_init_pdg_array
```

⟨*radiation generator: procedures*⟩+≡

```
    subroutine radiation_generator_init_pdg_list &
         (generator, pl_in, pl_out, pl_excluded_gauge_splittings, qcd, qed)
      class(radiation_generator_t), intent(inout) :: generator
      type(pdg_list_t), intent(in) :: pl_in, pl_out
      type(pdg_list_t), intent(in) :: pl_excluded_gauge_splittings
      logical, intent(in), optional :: qcd, qed
      if (present (qcd))  generator%qcd_enabled = qcd
      if (present (qed))  generator%qed_enabled = qed
      generator%pl_in = pl_in
      generator%pl_out = pl_out
      generator%pl_excluded_gauge_splittings = pl_excluded_gauge_splittings
      generator%is_gluon = pl_in%search_for_particle (GLUON)
      generator%fs_gluon = pl_out%search_for_particle (GLUON)
      generator%is_photon = pl_in%search_for_particle (PHOTON)
      generator%fs_photon = pl_out%search_for_particle (PHOTON)
      !!!generator%only_final_state = .not. (&
      !!!   generator%qcd_enabled .and. pl_in%contains_colored_particles ())
```

```
    generator%mass_sum = 0._default
    call generator%pdg_raw%init ()
  end subroutine radiation_generator_init_pdg_list

  subroutine radiation_generator_init_pdg_array &
       (generator, pdg_in, pdg_out, pdg_excluded_gauge_splittings, qcd, qed)
    class(radiation_generator_t), intent(inout) :: generator
    type(pdg_array_t), intent(in), dimension(:) :: pdg_in, pdg_out
    type(pdg_array_t), intent(in), dimension(:) :: pdg_excluded_gauge_splittings
    logical, intent(in), optional :: qcd, qed
    type(pdg_list_t) :: pl_in, pl_out
    type(pdg_list_t) :: pl_excluded_gauge_splittings
    integer :: i
    call pl_in%init(size (pdg_in))
    call pl_out%init(size (pdg_out))
    do i = 1, size (pdg_in)
       call pl_in%set (i, pdg_in(i))
    end do
    do i = 1, size (pdg_out)
       call pl_out%set (i, pdg_out(i))
    end do
    call pl_excluded_gauge_splittings%init(size (pdg_excluded_gauge_splittings))
    do i = 1, size (pdg_excluded_gauge_splittings)
       call pl_excluded_gauge_splittings%set &
            (i, pdg_excluded_gauge_splittings(i))
    end do
    call generator%init (pl_in, pl_out, pl_excluded_gauge_splittings, qcd, qed)
  end subroutine radiation_generator_init_pdg_array
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
  procedure :: set_initial_state_emissions => &
     radiation_generator_set_initial_state_emissions
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine radiation_generator_set_initial_state_emissions (generator)
     class(radiation_generator_t), intent(inout) :: generator
     generator%only_final_state = .false.
  end subroutine radiation_generator_set_initial_state_emissions
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
  procedure :: setup_if_table => radiation_generator_setup_if_table
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine radiation_generator_setup_if_table (generator, model)
    class(radiation_generator_t), intent(inout) :: generator
    class(model_data_t), intent(in), target :: model
    type(pdg_list_t), dimension(:), allocatable :: pl_in, pl_out

    allocate (pl_in(1), pl_out(1))

    pl_in(1) = generator%pl_in
    pl_out(1) = generator%pl_out

    call generator%if_table%init &
```

```
      (model, pl_in, pl_out, generator%constraints)
  end subroutine radiation_generator_setup_if_table
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
  generic :: reset_particle_content => reset_particle_content_pdg_array, &
                                       reset_particle_content_pdg_list
  procedure :: reset_particle_content_pdg_list => &
    radiation_generator_reset_particle_content_pdg_list
  procedure :: reset_particle_content_pdg_array => &
    radiation_generator_reset_particle_content_pdg_array
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine radiation_generator_reset_particle_content_pdg_list (generator, pl)
    class(radiation_generator_t), intent(inout) :: generator
    type(pdg_list_t), intent(in) :: pl
    generator%pl_out = pl
    generator%fs_gluon = pl%search_for_particle (GLUON)
    generator%fs_photon = pl%search_for_particle (PHOTON)
  end subroutine radiation_generator_reset_particle_content_pdg_list

  subroutine radiation_generator_reset_particle_content_pdg_array (generator, pdg)
    class(radiation_generator_t), intent(inout) :: generator
    type(pdg_array_t), intent(in), dimension(:) :: pdg
    type(pdg_list_t) :: pl
    integer :: i
    call pl%init (size (pdg))
    do i = 1, size (pdg)
       call pl%set (i, pdg(i))
    end do
    call generator%reset_particle_content (pl)
  end subroutine radiation_generator_reset_particle_content_pdg_array
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
  procedure :: reset_reshuffle_list=> radiation_generator_reset_reshuffle_list
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine radiation_generator_reset_reshuffle_list (generator)
    class(radiation_generator_t), intent(inout) :: generator
    call generator%reshuffle_list%reset ()
  end subroutine radiation_generator_reset_reshuffle_list
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
  procedure :: set_n => radiation_generator_set_n
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine radiation_generator_set_n (generator, n_in, n_out, n_loops)
    class(radiation_generator_t), intent(inout) :: generator
    integer, intent(in) :: n_in, n_out, n_loops
    generator%n_tot = n_in + n_out + 1
    generator%n_in = n_in
    generator%n_out = n_out
    generator%n_loops = n_loops
  end subroutine radiation_generator_set_n
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
procedure :: set_constraints => radiation_generator_set_constraints
```

⟨*radiation generator: procedures*⟩+≡

```
subroutine radiation_generator_set_constraints &
     (generator, set_n_loop, set_mass_sum, &
      set_selected_particles, set_required_particles)
  class(radiation_generator_t), intent(inout), target :: generator
  logical, intent(in) :: set_n_loop
  logical, intent(in) :: set_mass_sum
  logical, intent(in) :: set_selected_particles
  logical, intent(in) :: set_required_particles
  integer :: i, j, n, n_constraints
  type(pdg_list_t) :: pl_req, pl_insert
  type(pdg_list_t) :: pl_antiparticles
  type(pdg_array_t) :: pdg_gluon, pdg_photon
  type(pdg_array_t) :: pdg_add, pdg_tmp
  integer :: last_index
  integer :: n_new_particles, n_skip
  integer, dimension(:), allocatable :: i_skip
  integer :: n_nlo_correction_types

  n_nlo_correction_types = count ([generator%qcd_enabled, generator%qed_enabled])

  allocate (i_skip (generator%n_tot))
  i_skip = -1

  n_constraints = 2 + count([set_n_loop, set_mass_sum, &
       set_selected_particles, set_required_particles])
  associate (constraints => generator%constraints)
    n = 1
    call constraints%init (n_constraints)
    call constraints%set (n, constrain_n_tot (generator%n_tot))
    n = 2
    call constraints%set (n, constrain_couplings (generator%qcd_enabled, &
         generator%qed_enabled, n_nlo_correction_types))
    n = n + 1
    if (set_n_loop) then
       call constraints%set (n, constrain_n_loop(generator%n_loops))
       n = n + 1
    end if
    if (set_mass_sum) then
       call constraints%set (n, constrain_mass_sum(generator%mass_sum))
       n = n + 1
    end if
    if (set_required_particles) then
       if (generator%fs_gluon .or. generator%fs_photon) then
          do i = 1, generator%n_out
             pdg_tmp = generator%pl_out%get(i)
             if (pdg_tmp%search_for_particle (GLUON) &
                  .or. pdg_tmp%search_for_particle (PHOTON)) then
                i_skip(i) = i
             end if
          end do
```

```
            n_skip = count (i_skip > 0)
            call pl_req%init (generator%n_out-n_skip)
         else
            call pl_req%init (generator%n_out)
         end if
         j = 1
         do i = 1, generator%n_out
            if (any (i == i_skip)) cycle
            call pl_req%set (j, generator%pl_out%get(i))
            j = j + 1
         end do
         call constraints%set (n, constrain_require (pl_req))
         n = n + 1
      end if
      if (set_selected_particles) then
         if (generator%only_final_state ) then
            call pl_insert%init (generator%n_out + n_nlo_correction_types)
            do i = 1, generator%n_out
               call pl_insert%set(i, generator%pl_out%get(i))
            end do
            last_index = generator%n_out + 1
         else
            call generator%pl_in%create_antiparticles (pl_antiparticles, n_new_particles)
            call pl_insert%init (generator%n_tot + n_new_particles &
                 + n_nlo_correction_types)
            do i = 1, generator%n_in
               call pl_insert%set(i, generator%pl_in%get(i))
            end do
            do i = 1, generator%n_out
               j = i + generator%n_in
               call pl_insert%set(j, generator%pl_out%get(i))
            end do
            do i = 1, n_new_particles
               j = i + generator%n_in + generator%n_out
               call pl_insert%set(j, pl_antiparticles%get(i))
            end do
            last_index = generator%n_tot + n_new_particles + 1
         end if
         pdg_gluon = GLUON; pdg_photon = PHOTON
         if (generator%qcd_enabled) then
            pdg_add = pdg_gluon
            call pl_insert%set (last_index, pdg_add)
            last_index = last_index + 1
         end if
         if (generator%qed_enabled) then
            pdg_add = pdg_photon
            call pl_insert%set (last_index, pdg_add)
         end if
         call constraints%set (n, constrain_splittings (pl_insert, &
              generator%pl_excluded_gauge_splittings))
      end if
   end associate
end subroutine radiation_generator_set_constraints
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
procedure :: find_splittings => radiation_generator_find_splittings
```

⟨*radiation generator: procedures*⟩+≡

```
subroutine radiation_generator_find_splittings (generator)
  class(radiation_generator_t), intent(inout) :: generator
  integer :: i
  type(pdg_array_t), dimension(:), allocatable :: pdg_in, pdg_out, pdg_tmp
  integer, dimension(:), allocatable :: reshuffle_list

  call generator%pl_in%create_pdg_array (pdg_in)
  call generator%pl_out%create_pdg_array (pdg_out)

  associate (if_table => generator%if_table)
     call if_table%radiate (generator%constraints)

     do i = 1, if_table%get_length ()
        call if_table%get_pdg_out (i, pdg_tmp)
        if (size (pdg_tmp) == generator%n_tot) then
           call pdg_reshuffle (pdg_out, pdg_tmp, reshuffle_list)
           call generator%reshuffle_list%append (reshuffle_list)
        end if
     end do
  end associate

contains

  subroutine pdg_reshuffle (pdg_born, pdg_real, list)
    type(pdg_array_t), intent(in), dimension(:) :: pdg_born, pdg_real
    integer, intent(out), dimension(:), allocatable :: list
    type(pdg_sorter_t), dimension(:), allocatable :: sort_born
    type(pdg_sorter_t), dimension(:), allocatable :: sort_real
    integer :: i_min, n_in, n_born, n_real
    integer :: ib, ir

    n_in = generator%n_in
    n_born = size (pdg_born)
    n_real = size (pdg_real)
    allocate (list (n_real - n_in))
    allocate (sort_born (n_born))
    allocate (sort_real (n_real - n_in))
    sort_born%pdg = pdg_born%get ()
    sort_real%pdg = pdg_real(n_in + 1 : n_real)%get()
    do ib = 1, n_born
       if (any (sort_born(ib)%pdg == sort_real%pdg)) &
          call associate_born_indices (sort_born(ib), sort_real, ib, n_real)
    end do
    i_min = maxval (sort_real%associated_born) + 1
    do ir = 1, n_real - n_in
       if (sort_real(ir)%associated_born == 0) then
          sort_real(ir)%associated_born = i_min
          i_min = i_min + 1
       end if
    end do
    list = sort_real%associated_born
```

```
    end subroutine pdg_reshuffle

    subroutine associate_born_indices (sort_born, sort_real, ib, n_real)
      type(pdg_sorter_t), intent(in) :: sort_born
      type(pdg_sorter_t), intent(inout), dimension(:) :: sort_real
      integer, intent(in) :: ib, n_real
      integer :: ir
      do ir = 1, n_real - generator%n_in
         if (sort_born%pdg == sort_real(ir)%pdg &
            .and..not. sort_real(ir)%checked) then
            sort_real(ir)%associated_born = ib
            sort_real(ir)%checked = .true.
            exit
        end if
      end do
    end subroutine associate_born_indices
  end subroutine radiation_generator_find_splittings
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
  procedure :: generate_real_particle_strings &
       => radiation_generator_generate_real_particle_strings
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine radiation_generator_generate_real_particle_strings &
       (generator, prt_tot_in, prt_tot_out)
    type :: prt_array_t
       type(string_t), dimension(:), allocatable :: prt
    end type
    integer, parameter :: n_flv_max = 10
    class(radiation_generator_t), intent(inout) :: generator
    type(string_t), intent(out), dimension(:), allocatable :: prt_tot_in, prt_tot_out
    type(prt_array_t), dimension(n_flv_max) :: prt_in, prt_out
    type(prt_array_t), dimension(n_flv_max) :: prt_out0, prt_in0
    type(pdg_array_t), dimension(:), allocatable :: pdg_tmp, pdg_out, pdg_in
    type(pdg_list_t), dimension(:), allocatable :: pl_in, pl_out
    integer :: i, j
    integer, dimension(:), allocatable :: reshuffle_list_local
    type(reshuffle_list_t) :: reshuffle_list
    integer :: flv
    type(string_t), dimension(:), allocatable :: buf
    integer :: i_buf

    flv = 0
    associate (if_table => generator%if_table)
       do i = 1, if_table%get_length ()
          call if_table%get_pdg_out (i, pdg_tmp)
          if (size (pdg_tmp) == generator%n_tot) then
             call if_table%get_particle_string (i, &
                prt_in0(flv + 1)%prt, prt_out0(flv + 1)%prt)
             flv = flv + 1
          end if
       end do
    end associate
```

```
  do i = 1, flv
     allocate (prt_in(i)%prt (generator%n_in))
     allocate (prt_out(i)%prt (generator%n_tot - generator%n_in))
  end do
  allocate (prt_tot_in (generator%n_in))
  allocate (prt_tot_out (generator%n_tot - generator%n_in))
  allocate (buf (generator%n_tot))
  buf = ""

  do j = 1, flv
     do i = 1, generator%n_in
        prt_in(j)%prt(i) = prt_in0(j)%prt(i)
        call fill_buffer (buf(i), prt_in0(j)%prt(i))
     end do
  end do
  prt_tot_in = buf(1 : generator%n_in)

  do j = 1, flv
     !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
     allocate (reshuffle_list_local (size (generator%reshuffle_list%get(j))))
     reshuffle_list_local = generator%reshuffle_list%get(j)
     do i = 1, size (reshuffle_list_local)
        prt_out(j)%prt(reshuffle_list_local(i)) = prt_out0(j)%prt(i)
        i_buf = reshuffle_list_local(i) + generator%n_in
        call fill_buffer (buf(i_buf), &
             prt_out(j)%prt(reshuffle_list_local(i)))
     end do
     !!! Need to deallocate here because in the next iteration the reshuffling
     !!! list can have a different size
     deallocate (reshuffle_list_local)
  end do
  prt_tot_out = buf(generator%n_in + 1 : generator%n_tot)

contains

  subroutine fill_buffer (buffer, particle)
    type(string_t), intent(inout) :: buffer
    type(string_t), intent(in) :: particle
    logical :: particle_present
    if (len (buffer) > 0) then
       particle_present = check_for_substring (char(buffer), char(particle))
       if (.not. particle_present) buffer = buffer // ":" // particle
    else
       buffer = buffer // particle
    end if
  end subroutine fill_buffer

  function check_for_substring (buffer, substring) result (exist)
    character(len=*), intent(in) :: buffer
    character(len=*), intent(in) :: substring
    character(len=50) :: buffer_internal
    logical :: exist
    integer :: i_first, i_last
    exist = .false.
```

```
      i_first = 1; i_last = 1
      do
         if (buffer(i_last:i_last) == ":") then
            buffer_internal = buffer (i_first : i_last - 1)
            if (buffer_internal == substring) then
               exist = .true.
               exit
            end if
            i_first = i_last + 1; i_last = i_first + 1
            if (i_last > len(buffer)) exit
         else if (i_last == len(buffer)) then
            buffer_internal = buffer (i_first : i_last)
            exist = (buffer_internal == substring)
            exit
         else
            i_last = i_last + 1
            if (i_last > len(buffer)) exit
         end if
      end do
    end function check_for_substring
  end subroutine radiation_generator_generate_real_particle_strings
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
  procedure :: contains_emissions => radiation_generator_contains_emissions
```

⟨*radiation generator: procedures*⟩+≡

```
  function radiation_generator_contains_emissions (generator) result (has_em)
    logical :: has_em
    class(radiation_generator_t), intent(in) :: generator
    has_em = .not. generator%reshuffle_list%is_empty ()
  end function radiation_generator_contains_emissions
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
  procedure :: generate => radiation_generator_generate
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine radiation_generator_generate (generator, prt_in, prt_out)
    class(radiation_generator_t), intent(inout) :: generator
    type(string_t), intent(out), dimension(:), allocatable :: prt_in, prt_out
    call generator%find_splittings ()
    call generator%generate_real_particle_strings (prt_in, prt_out)
  end subroutine radiation_generator_generate
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
  procedure :: generate_multiple => radiation_generator_generate_multiple
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine radiation_generator_generate_multiple (generator, max_multiplicity, model)
    class(radiation_generator_t), intent(inout) :: generator
    integer, intent(in) :: max_multiplicity
    class(model_data_t), intent(in), target :: model
    if (max_multiplicity <= generator%n_out) &
         call msg_fatal ("GKS states: Multiplicity is not large enough!")
    call generator%first_emission (model)
```

```
    call generator%reset_reshuffle_list ()
    if (max_multiplicity - generator%n_out > 1) &
         call generator%append_emissions (max_multiplicity, model)
  end subroutine radiation_generator_generate_multiple
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
  procedure :: first_emission => radiation_generator_first_emission
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine radiation_generator_first_emission (generator, model)
    class(radiation_generator_t), intent(inout) :: generator
    class(model_data_t), intent(in), target :: model
    type(string_t), dimension(:), allocatable :: prt_in, prt_out
    call generator%setup_if_table (model)
    call generator%generate (prt_in, prt_out)
    call generator%prt_queue%null ()
    call generator%prt_queue%append (prt_out)
  end subroutine radiation_generator_first_emission
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
  procedure :: append_emissions => radiation_generator_append_emissions
```

⟨*radiation generator: procedures*⟩+≡

```
  subroutine radiation_generator_append_emissions (generator, max_multiplicity, model)
    class(radiation_generator_t), intent(inout) :: generator
    integer, intent(in) :: max_multiplicity
    class(model_data_t), intent(in), target :: model
    type(string_t), dimension(:), allocatable :: prt_fetched
    type(string_t), dimension(:), allocatable :: prt_in
    type(string_t), dimension(:), allocatable :: prt_out
    type(pdg_array_t), dimension(:), allocatable :: pdg_new_out
    integer :: current_multiplicity, i, j, n_longest_length
    type :: prt_table_t
       type(string_t), dimension(:), allocatable :: prt
    end type prt_table_t
    type(prt_table_t), dimension(:), allocatable :: prt_table_out
    do
       call generator%prt_queue%get (prt_fetched)
       current_multiplicity = size (prt_fetched)
       if (current_multiplicity == max_multiplicity) exit
       call create_pdg_array (prt_fetched, model, &
            pdg_new_out)
       call generator%reset_particle_content (pdg_new_out)
       call generator%set_n (2, current_multiplicity, 0)
       call generator%set_constraints (.false., .false., .true., .true.)
       call generator%setup_if_table (model)
       call generator%generate (prt_in, prt_out)
       n_longest_length = get_length_of_longest_tuple (prt_out)
       call separate_particles (prt_out, prt_table_out)
       do i = 1, n_longest_length
          if (.not. any (prt_table_out(i)%prt == " ")) then
             call sort_prt (prt_table_out(i)%prt, model)
             if (.not. generator%prt_queue%contains (prt_table_out(i)%prt)) then
                call generator%prt_queue%append (prt_table_out(i)%prt)
```

```
          end if
        end if
     end do
     call generator%reset_reshuffle_list ()
  end do

contains
  subroutine separate_particles (prt, prt_table)
    type(string_t), intent(in), dimension(:) :: prt
    type(string_t), dimension(:), allocatable :: prt_tmp
    type(prt_table_t), intent(out), dimension(:), allocatable :: prt_table
    integer :: i, j
    logical, dimension(:), allocatable :: tuples_occured
    allocate (prt_table (n_longest_length))
    do i = 1, n_longest_length
       allocate (prt_table(i)%prt (size (prt)))
    end do
    allocate (tuples_occured (size (prt)))
    do j = 1, size (prt)
       call split_string (prt(j), var_str (":"), prt_tmp)
       do i = 1, n_longest_length
          if (i <= size (prt_tmp)) then
             prt_table(i)%prt(j) = prt_tmp(i)
          else
             prt_table(i)%prt(j) = " "
          end if
       end do
       if (n_longest_length > 1) &
            tuples_occured(j) = prt_table(1)%prt(j) /= " " &
            .and. prt_table(2)%prt(j) /= " "
    end do
    if (any (tuples_occured)) then
       do j = 1, size (tuples_occured)
          if (.not. tuples_occured(j)) then
             do i = 2, n_longest_length
                prt_table(i)%prt(j) = prt_table(1)%prt(j)
             end do
          end if
       end do
    end if
  end subroutine separate_particles

function get_length_of_longest_tuple (prt) result (longest_length)
    type(string_t), intent(in), dimension(:) :: prt
    integer :: longest_length, i
    type(prt_table_t), dimension(:), allocatable :: prt_table
    allocate (prt_table (size (prt)))
    longest_length = 0
    do i = 1, size (prt)
       call split_string (prt(i), var_str (":"), prt_table(i)%prt)
       if (size (prt_table(i)%prt) > longest_length) &
            longest_length = size (prt_table(i)%prt)
    end do
  end function get_length_of_longest_tuple
```

```
    end subroutine radiation_generator_append_emissions
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
    procedure :: reset_queue => radiation_generator_reset_queue
```

⟨*radiation generator: procedures*⟩+≡

```
    subroutine radiation_generator_reset_queue (generator)
      class(radiation_generator_t), intent(inout) :: generator
      call generator%prt_queue%reset ()
    end subroutine radiation_generator_reset_queue
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
    procedure :: get_n_gks_states => radiation_generator_get_n_gks_states
```

⟨*radiation generator: procedures*⟩+≡

```
    function radiation_generator_get_n_gks_states (generator) result (n)
      class(radiation_generator_t), intent(in) :: generator
      integer :: n
      n = generator%prt_queue%n_lists
    end function radiation_generator_get_n_gks_states
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
    procedure :: get_next_state => radiation_generator_get_next_state
```

⟨*radiation generator: procedures*⟩+≡

```
    function radiation_generator_get_next_state (generator) result (prt_string)
      class(radiation_generator_t), intent(inout) :: generator
      type(string_t), dimension(:), allocatable :: prt_string
      call generator%prt_queue%get (prt_string)
    end function radiation_generator_get_next_state
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
    procedure :: get_emitter_indices => radiation_generator_get_emitter_indices
```

⟨*radiation generator: procedures*⟩+≡

```
    subroutine radiation_generator_get_emitter_indices (generator, indices)
       class(radiation_generator_t), intent(in) :: generator
       integer, dimension(:), allocatable, intent(out) :: indices
       type(pdg_array_t), dimension(:), allocatable :: pdg_in, pdg_out
       integer, dimension(:), allocatable :: flv_in, flv_out
       integer, dimension(:), allocatable :: emitters
       integer :: i, j
       integer :: n_in, n_out

       call generator%pl_in%create_pdg_array (pdg_in)
       call generator%pl_out%create_pdg_array (pdg_out)

       n_in = size (pdg_in); n_out = size (pdg_out)
       allocate (flv_in (n_in), flv_out (n_out))
       forall (i=1:n_in) flv_in(i) = pdg_in(i)%get()
       forall (i=1:n_out) flv_out(i) = pdg_out(i)%get()

       call generator%if_table%get_emitters (generator%constraints, emitters)
       allocate (indices (size (emitters)))
```

```
      j = 1
      do i = 1, n_in + n_out
         if (i <= n_in) then
            if (any (flv_in(i) == emitters)) then
               indices (j) = i
               j = j + 1
            end if
         else
            if (any (flv_out(i-n_in) == emitters)) then
               indices (j) = i
               j = j + 1
            end if
         end if
      end do
   end subroutine radiation_generator_get_emitter_indices
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
   procedure :: get_raw_states => radiation_generator_get_raw_states
```

⟨*radiation generator: procedures*⟩+≡

```
   function radiation_generator_get_raw_states (generator) result (raw_states)
     class(radiation_generator_t), intent(in), target :: generator
     integer, dimension(:,:), allocatable :: raw_states
     type(pdg_states_t), pointer :: state
     integer :: n_states, n_particles
     integer :: i_state
     integer :: j
     state => generator%pdg_raw
     n_states = generator%pdg_raw%get_n_states ()
     n_particles = size (generator%pdg_raw%pdg)
     allocate (raw_states (n_particles, n_states))
     do i_state = 1, n_states
       do j = 1, n_particles
         raw_states (j, i_state) = state%pdg(j)%get ()
       end do
         state => state%next
     end do
   end function radiation_generator_get_raw_states
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
   procedure :: save_born_raw => radiation_generator_save_born_raw
```

⟨*radiation generator: procedures*⟩+≡

```
   subroutine radiation_generator_save_born_raw (generator, pdg_in, pdg_out)
     class(radiation_generator_t), intent(inout) :: generator
     type(pdg_array_t), dimension(:), allocatable, intent(in) :: pdg_in, pdg_out
     integer :: i
     !!! !!! !!! Explicit allocation due to gfortran 4.7.4
     allocate (generator%pdg_in_born (size (pdg_in)))
     do i = 1, size (pdg_in)
        generator%pdg_in_born(i) = pdg_in(i)
     end do
     allocate (generator%pdg_out_born (size (pdg_out)))
     do i = 1, size (pdg_out)
```

```
      generator%pdg_out_born(i) = pdg_out(i)
    end do
  end subroutine radiation_generator_save_born_raw
```

⟨*radiation generator: radiation generator: TBP*⟩+≡

```
  procedure :: get_born_raw => radiation_generator_get_born_raw
```

⟨*radiation generator: procedures*⟩+≡

```
  function radiation_generator_get_born_raw (generator) result (flv_born)
    class(radiation_generator_t), intent(in) :: generator
    integer, dimension(:,:), allocatable :: flv_born
    integer :: i_part, n_particles
    n_particles = size (generator%pdg_in_born) + size (generator%pdg_out_born)
    allocate (flv_born (n_particles, 1))
    flv_born(1,1) = generator%pdg_in_born(1)%get ()
    flv_born(2,1) = generator%pdg_in_born(2)%get ()
    do i_part = 3, n_particles
      flv_born(i_part, 1) = generator%pdg_out_born(i_part-2)%get ()
    end do
  end function radiation_generator_get_born_raw
```

28.2.1 Unit tests

Test module, followed by the corresponding implementation module.

⟨radiation_generator_ut.f90⟩≡

```
⟨File header⟩

module radiation_generator_ut
  use unit_tests
  use radiation_generator_uti

⟨Standard module head⟩

⟨radiation generator: public test⟩

contains

⟨radiation generator: test driver⟩

end module radiation_generator_ut
```

⟨radiation_generator_uti.f90⟩≡

```
⟨File header⟩

module radiation_generator_uti

⟨Use strings⟩
  use format_utils, only: write_separator
  use os_interface
  use pdg_arrays
  use models
  use kinds, only: default

  use radiation_generator
```

⟨*Standard module head*⟩

⟨*radiation generator: test declarations*⟩

```
contains
```

⟨*radiation generator: tests*⟩

⟨*radiation generator: test auxiliary*⟩

```
end module radiation_generator_uti
```

API: driver for the unit tests below.

⟨*radiation generator: public test*⟩≡

```
public :: radiation_generator_test
```

⟨*radiation generator: test driver*⟩≡

```
subroutine radiation_generator_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
  call test(radiation_generator_1, "radiation_generator_1", &
     "Test the generator of N+1-particle flavor structures in QCD", &
     u, results)
  call test(radiation_generator_2, "radiation_generator_2", &
     "Test multiple splittings in QCD", &
     u, results)
  call test(radiation_generator_3, "radiation_generator_3", &
     "Test the generator of N+1-particle flavor structures in QED", &
     u, results)
  call test(radiation_generator_4, "radiation_generator_4", &
     "Test multiple splittings in QED", &
     u, results)
end subroutine radiation_generator_test
```

⟨*radiation generator: test declarations*⟩≡

```
public :: radiation_generator_1
```

⟨*radiation generator: tests*⟩≡

```
subroutine radiation_generator_1 (u)
  integer, intent(in) :: u
  type(radiation_generator_t) :: generator
  type(pdg_array_t), dimension(:), allocatable :: pdg_in, pdg_out
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model => null ()

  write (u, "(A)") "* Test output: radiation_generator_1"
  write (u, "(A)") "* Purpose: Create N+1-particle flavor structures &
       &from predefined N-particle flavor structures"
  write (u, "(A)") "* One additional strong coupling, no additional electroweak coupling"
  write (u, "(A)")
  write (u, "(A)") "* Loading radiation model: SM.mdl"

  call syntax_model_file_init ()
```

```
call os_data_init (os_data)
call model_list%read_model &
   (var_str ("SM"), var_str ("SM.mdl"), &
    os_data, model)
write (u, "(A)") "* Success"

allocate (pdg_in (2))
pdg_in(1) = 11; pdg_in(2) = -11

write (u, "(A)") "* Start checking processes"
call write_separator (u)

write (u, "(A)") "* Process 1: Top pair-production with additional gluon"
allocate (pdg_out(3))
pdg_out(1) = 6; pdg_out(2) = -6; pdg_out(3) = 21
call test_process (generator, pdg_in, pdg_out, u)
deallocate (pdg_out)

write (u, "(A)") "* Process 2: Top pair-production with additional jet"
allocate (pdg_out(3))
pdg_out(1) = 6; pdg_out(2) = -6; pdg_out(3) = [-1,1,-2,2,-3,3,-4,4,-5,5,21]
call test_process (generator, pdg_in, pdg_out, u)
deallocate (pdg_out)

write (u, "(A)") "* Process 3: Quark-antiquark production"
allocate (pdg_out(2))
pdg_out(1) = 2; pdg_out(2) = -2
call test_process (generator, pdg_in, pdg_out, u)
deallocate (pdg_out)

write (u, "(A)") "* Process 4: Quark-antiquark production with additional gluon"
allocate (pdg_out(3))
pdg_out(1) = 2; pdg_out(2) = -2; pdg_out(3) = 21
call test_process (generator, pdg_in, pdg_out, u)
deallocate (pdg_out)

write (u, "(A)") "* Process 5: Z + jets"
allocate (pdg_out(3))
pdg_out(1) = 2; pdg_out(2) = -2; pdg_out(3) = 23
call test_process (generator, pdg_in, pdg_out, u)
deallocate (pdg_out)

write (u, "(A)") "* Process 6: Top Decay"
allocate (pdg_out(4))
pdg_out(1) = 24; pdg_out(2) = -24
pdg_out(3) = 5; pdg_out(4) = -5
call test_process (generator, pdg_in, pdg_out, u)
deallocate (pdg_out)

write (u, "(A)") "* Process 7: Production of four quarks"
allocate (pdg_out(4))
pdg_out(1) = 2; pdg_out(2) = -2;
pdg_out(3) = 2; pdg_out(4) = -2
call test_process (generator, pdg_in, pdg_out, u)
```

```
  deallocate (pdg_out); deallocate (pdg_in)

  write (u, "(A)") "* Process 8: Drell-Yan lepto-production"
  allocate (pdg_in (2)); allocate (pdg_out (2))
  pdg_in(1) = 2; pdg_in(2) = -2
  pdg_out(1) = 11; pdg_out(2) = -11
  call test_process (generator, pdg_in, pdg_out, u, .true.)
  deallocate (pdg_out); deallocate (pdg_in)

  write (u, "(A)") "* Process 9: WZ production at hadron-colliders"
  allocate (pdg_in (2)); allocate (pdg_out (2))
  pdg_in(1) = 1; pdg_in(2) = -2
  pdg_out(1) = -24; pdg_out(2) = 23
  call test_process (generator, pdg_in, pdg_out, u, .true.)
  deallocate (pdg_out); deallocate (pdg_in)

contains
  subroutine test_process (generator, pdg_in, pdg_out, u, include_initial_state)
    type(radiation_generator_t), intent(inout) :: generator
    type(pdg_array_t), dimension(:), intent(in) :: pdg_in, pdg_out
    integer, intent(in) :: u
    logical, intent(in), optional :: include_initial_state
    type(string_t), dimension(:), allocatable :: prt_strings_in
    type(string_t), dimension(:), allocatable :: prt_strings_out
    type(pdg_array_t), dimension(10) :: pdg_excluded
    logical :: yorn
    yorn = .false.
    pdg_excluded = [-4, 4, 5, -5, 6, -6, 13, -13, 15, -15]
    if (present (include_initial_state)) yorn = include_initial_state
    write (u, "(A)") "* Leading order: "
    write (u, "(A)", advance = 'no') '* Incoming: '
    call write_pdg_array (pdg_in, u)
    write (u, "(A)", advance = 'no') '* Outgoing: '
    call write_pdg_array (pdg_out, u)

    call generator%init (pdg_in, pdg_out, &
         pdg_excluded_gauge_splittings = pdg_excluded, qcd = .true., qed = .false.)
    call generator%set_n (2, size(pdg_out), 0)
    if (yorn) call generator%set_initial_state_emissions ()
    call generator%set_constraints (.false., .false., .true., .true.)
    call generator%setup_if_table (model)
    call generator%generate (prt_strings_in, prt_strings_out)
    write (u, "(A)") "* Additional radiation: "
    write (u, "(A)") "* Incoming: "
    call write_particle_string (prt_strings_in, u)
    write (u, "(A)") "* Outgoing: "
    call write_particle_string (prt_strings_out, u)
    call write_separator(u)
    call generator%reset_reshuffle_list ()
  end subroutine test_process

end subroutine radiation_generator_1
```

⟨*radiation generator: test declarations*⟩+≡

```
public :: radiation_generator_2
```

⟨*radiation generator: tests*⟩+≡

```
subroutine radiation_generator_2 (u)
  integer, intent(in) :: u
  type(radiation_generator_t) :: generator
  type(pdg_array_t), dimension(:), allocatable :: pdg_in, pdg_out
  type(pdg_array_t), dimension(:), allocatable :: pdg_excluded
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model => null ()
  integer, parameter :: max_multiplicity = 10
  type(string_t), dimension(:), allocatable :: prt_last

  write (u, "(A)") "* Test output: radiation_generator_2"
  write (u, "(A)") "* Purpose: Test the repeated application of &
       &a radiation generator splitting in QCD"
  write (u, "(A)") "* Only Final state emissions! "
  write (u, "(A)")
  write (u, "(A)") "* Loading radiation model: SM.mdl"

  call syntax_model_file_init ()
  call os_data_init (os_data)
  call model_list%read_model &
     (var_str ("SM"), var_str ("SM.mdl"), &
      os_data, model)
  write (u, "(A)") "* Success"

  allocate (pdg_in (2))
  pdg_in(1) = 11; pdg_in(2) = -11
  allocate (pdg_out(2))
  pdg_out(1) = 2; pdg_out(2) = -2
  allocate (pdg_excluded (10))
  pdg_excluded = [-4, 4, 5, -5, 6, -6, 13, -13, 15, -15]

  write (u, "(A)") "* Leading order"
  write (u, "(A)", advance = 'no') "* Incoming: "
  call write_pdg_array (pdg_in, u)
  write (u, "(A)", advance = 'no') "* Outgoing: "
  call write_pdg_array (pdg_out, u)

  call generator%init (pdg_in, pdg_out, &
       pdg_excluded_gauge_splittings = pdg_excluded, qcd = .true., qed = .false.)
  call generator%set_n (2, 2, 0)
  call generator%set_constraints (.false., .false., .true., .true.)

  call write_separator (u)
  write (u, "(A)") "Generate higher-multiplicity states"
  write (u, "(A,I0)") "Desired multiplicity: ", max_multiplicity
  call generator%generate_multiple (max_multiplicity, model)
  call generator%prt_queue%write (u)
  call write_separator (u)
  write (u, "(A,I0)") "Number of higher-multiplicity states: ", generator%prt_queue%n_lists

  write (u, "(A)") "Check that no particle state occurs twice or more"
```

```
    if (.not. generator%prt_queue%check_for_same_prt_strings()) then
       write (u, "(A)") "SUCCESS"
    else
       write (u, "(A)") "FAIL"
    end if
    call write_separator (u)
    write (u, "(A,I0,A)") "Check that there are ", max_multiplicity, " particles in the last entry
    call generator%prt_queue%get_last (prt_last)
    if (size (prt_last) == max_multiplicity) then
       write (u, "(A)") "SUCCESS"
    else
       write (u, "(A)") "FAIL"
    end if
  end subroutine radiation_generator_2
```

⟨*radiation generator: test declarations*⟩+≡

```
  public :: radiation_generator_3
```

⟨*radiation generator: tests*⟩+≡

```
  subroutine radiation_generator_3 (u)
    integer, intent(in) :: u
    type(radiation_generator_t) :: generator
    type(pdg_array_t), dimension(:), allocatable :: pdg_in, pdg_out
    type(os_data_t) :: os_data
    type(model_list_t) :: model_list
    type(model_t), pointer :: model => null ()

    write (u, "(A)") "* Test output: radiation_generator_3"
    write (u, "(A)") "* Purpose: Create N+1-particle flavor structures &
         &from predefined N-particle flavor structures"
    write (u, "(A)") "* One additional electroweak coupling, no additional strong coupling"
    write (u, "(A)")
    write (u, "(A)") "* Loading radiation model: SM.mdl"

    call syntax_model_file_init ()
    call os_data_init (os_data)
    call model_list%read_model &
         (var_str ("SM"), var_str ("SM.mdl"), &
         os_data, model)
    write (u, "(A)") "* Success"

    allocate (pdg_in (2))
    pdg_in(1) = 11; pdg_in(2) = -11

    write (u, "(A)") "* Start checking processes"
    call write_separator (u)

    write (u, "(A)") "* Process 1: Tau pair-production with additional photon"
    allocate (pdg_out(3))
    pdg_out(1) = 15; pdg_out(2) = -15; pdg_out(3) = 22
    call test_process (generator, pdg_in, pdg_out, u)
    deallocate (pdg_out)

    write (u, "(A)") "* Process 2: Tau pair-production with additional leptons or photon"
```

```
allocate (pdg_out(3))
pdg_out(1) = 15; pdg_out(2) = -15; pdg_out(3) = [-13, 13, 22]
call test_process (generator, pdg_in, pdg_out, u)
deallocate (pdg_out)

write (u, "(A)") "* Process 3: Electron-positron production"
allocate (pdg_out(2))
pdg_out(1) = 11; pdg_out(2) = -11
call test_process (generator, pdg_in, pdg_out, u)
deallocate (pdg_out)

write (u, "(A)") "* Process 4: Quark-antiquark production with additional photon"
allocate (pdg_out(3))
pdg_out(1) = 2; pdg_out(2) = -2; pdg_out(3) = 22
call test_process (generator, pdg_in, pdg_out, u)
deallocate (pdg_out)

write (u, "(A)") "* Process 5: Z + jets "
allocate (pdg_out(3))
pdg_out(1) = 2; pdg_out(2) = -2; pdg_out(3) = 23
call test_process (generator, pdg_in, pdg_out, u)
deallocate (pdg_out)

write (u, "(A)") "* Process 6: W + jets"
allocate (pdg_out(3))
pdg_out(1) = 1; pdg_out(2) = -2; pdg_out(3) = -24
call test_process (generator, pdg_in, pdg_out, u)
deallocate (pdg_out)

write (u, "(A)") "* Process 7: Top Decay"
allocate (pdg_out(4))
pdg_out(1) = 24; pdg_out(2) = -24
pdg_out(3) = 5; pdg_out(4) = -5
call test_process (generator, pdg_in, pdg_out, u)
deallocate (pdg_out)

write (u, "(A)") "* Process 8: Production of four quarks"
allocate (pdg_out(4))
pdg_out(1) = 2; pdg_out(2) = -2;
pdg_out(3) = 2; pdg_out(4) = -2
call test_process (generator, pdg_in, pdg_out, u)
deallocate (pdg_out)

write (u, "(A)") "* Process 9: Neutrino pair-production"
allocate (pdg_out(2))
pdg_out(1) = 12; pdg_out(2) = -12
call test_process (generator, pdg_in, pdg_out, u, .true.)
deallocate (pdg_out); deallocate (pdg_in)

write (u, "(A)") "* Process 10: Drell-Yan lepto-production"
allocate (pdg_in (2)); allocate (pdg_out (2))
pdg_in(1) = 2; pdg_in(2) = -2
pdg_out(1) = 11; pdg_out(2) = -11
call test_process (generator, pdg_in, pdg_out, u, .true.)
```

```
    deallocate (pdg_out); deallocate (pdg_in)

    write (u, "(A)") "* Process 11: WZ production at hadron-colliders"
    allocate (pdg_in (2)); allocate (pdg_out (2))
    pdg_in(1) = 1; pdg_in(2) = -2
    pdg_out(1) = -24; pdg_out(2) = 23
    call test_process (generator, pdg_in, pdg_out, u, .true.)
    deallocate (pdg_out); deallocate (pdg_in)

    write (u, "(A)") "* Process 12: Positron-neutrino production"
    allocate (pdg_in (2)); allocate (pdg_out (2))
    pdg_in(1) = -1; pdg_in(2) = 2
    pdg_out(1) = -11; pdg_out(2) = 12
    call test_process (generator, pdg_in, pdg_out, u)
    deallocate (pdg_out); deallocate (pdg_in)

  contains
    subroutine test_process (generator, pdg_in, pdg_out, u, include_initial_state)
      type(radiation_generator_t), intent(inout) :: generator
      type(pdg_array_t), dimension(:), intent(in) :: pdg_in, pdg_out
      integer, intent(in) :: u
      logical, intent(in), optional :: include_initial_state
      type(string_t), dimension(:), allocatable :: prt_strings_in
      type(string_t), dimension(:), allocatable :: prt_strings_out
      type(pdg_array_t), dimension(10) :: pdg_excluded
      logical :: yorn
      yorn = .false.
      pdg_excluded = [-4, 4, 5, -5, 6, -6, 13, -13, 15, -15]
      if (present (include_initial_state)) yorn = include_initial_state
      write (u, "(A)") "* Leading order: "
      write (u, "(A)", advance = 'no') '* Incoming: '
      call write_pdg_array (pdg_in, u)
      write (u, "(A)", advance = 'no') '* Outgoing: '
      call write_pdg_array (pdg_out, u)

      call generator%init (pdg_in, pdg_out, &
           pdg_excluded_gauge_splittings = pdg_excluded, qcd = .false., qed = .true.)
      call generator%set_n (2, size(pdg_out), 0)
      if (yorn) call generator%set_initial_state_emissions ()
      call generator%set_constraints (.false., .false., .true., .true.)
      call generator%setup_if_table (model)
      call generator%generate (prt_strings_in, prt_strings_out)
      write (u, "(A)") "* Additional radiation: "
      write (u, "(A)") "* Incoming: "
      call write_particle_string (prt_strings_in, u)
      write (u, "(A)") "* Outgoing: "
      call write_particle_string (prt_strings_out, u)
      call write_separator(u)
      call generator%reset_reshuffle_list ()
    end subroutine test_process

  end subroutine radiation_generator_3
```

⟨*radiation generator: test declarations*⟩+≡

```
    public :: radiation_generator_4
```

⟨*radiation generator: tests*⟩+≡

```
    subroutine radiation_generator_4 (u)
      integer, intent(in) :: u
      type(radiation_generator_t) :: generator
      type(pdg_array_t), dimension(:), allocatable :: pdg_in, pdg_out
      type(pdg_array_t), dimension(:), allocatable :: pdg_excluded
      type(os_data_t) :: os_data
      type(model_list_t) :: model_list
      type(model_t), pointer :: model => null ()
      integer, parameter :: max_multiplicity = 10
      type(string_t), dimension(:), allocatable :: prt_last

      write (u, "(A)") "* Test output: radiation_generator_4"
      write (u, "(A)") "* Purpose: Test the repeated application of &
           &a radiation generator splitting in QED"
      write (u, "(A)") "* Only Final state emissions! "
      write (u, "(A)")
      write (u, "(A)") "* Loading radiation model: SM.mdl"

      call syntax_model_file_init ()
      call os_data_init (os_data)
      call model_list%read_model &
         (var_str ("SM"), var_str ("SM.mdl"), &
          os_data, model)
      write (u, "(A)") "* Success"

      allocate (pdg_in (2))
      pdg_in(1) = 2; pdg_in(2) = -2
      allocate (pdg_out(2))
      pdg_out(1) = 11; pdg_out(2) = -11
      allocate ( pdg_excluded (14))
      pdg_excluded = [1, -1, 2, -2, 3, -3, 4, -4, 5, -5, 6, -6, 15, -15]

      write (u, "(A)") "* Leading order"
      write (u, "(A)", advance = 'no') "* Incoming: "
      call write_pdg_array (pdg_in, u)
      write (u, "(A)", advance = 'no') "* Outgoing: "
      call write_pdg_array (pdg_out, u)

      call generator%init (pdg_in, pdg_out, &
           pdg_excluded_gauge_splittings = pdg_excluded, qcd = .false., qed = .true.)
      call generator%set_n (2, 2, 0)
      call generator%set_constraints (.false., .false., .true., .true.)

      call write_separator (u)
      write (u, "(A)") "Generate higher-multiplicity states"
      write (u, "(A,I0)") "Desired multiplicity: ", max_multiplicity
      call generator%generate_multiple (max_multiplicity, model)
      call generator%prt_queue%write (u)
      call write_separator (u)
      write (u, "(A,I0)") "Number of higher-multiplicity states: ", generator%prt_queue%n_lists

      write (u, "(A)") "Check that no particle state occurs twice or more"
```

```
  if (.not. generator%prt_queue%check_for_same_prt_strings()) then
     write (u, "(A)") "SUCCESS"
  else
     write (u, "(A)") "FAIL"
  end if
  call write_separator (u)
  write (u, "(A,I0,A)") "Check that there are ", max_multiplicity, " particles in the last entry
  call generator%prt_queue%get_last (prt_last)
  if (size (prt_last) == max_multiplicity) then
     write (u, "(A)") "SUCCESS"
  else
     write (u, "(A)") "FAIL"
  end if
end subroutine radiation_generator_4
```

28.3 Sindarin Expression Implementation

This module defines expressions of all kinds, represented in a tree structure, for repeated evaluation. This provides an implementation of the `expr_base` abstract type.

We have two flavors of expressions: one with particles and one without particles. The latter version is used for defining cut/selection criteria and for online analysis.

⟨`eval_trees.f90`⟩≡

```
⟨File header⟩

module eval_trees

  use, intrinsic :: iso_c_binding !NODEP!
⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use constants, only: DEGREE, IMAGO, PI
  use format_defs, only: FMT_19
  use diagnostics
  use lorentz
  use md5
  use formats
  use sorting
  use ifiles
  use lexers
  use syntax_rules
  use parser
  use analysis
  use jets
  use pdg_arrays
  use subevents
  use user_code_interface
  use var_base
  use expr_base
  use variables
  use observables

⟨Standard module head⟩

⟨Eval trees: public⟩

⟨Eval trees: types⟩

⟨Eval trees: interfaces⟩

⟨Eval trees: variables⟩

contains

⟨Eval trees: procedures⟩

end module eval_trees
```

28.3.1 Tree nodes

The evaluation tree consists of branch nodes (unary and binary) and of leaf nodes, originating from a common root. The node object should be polymorphic. For the time being, polymorphism is emulated here. This means that we have to maintain all possibilities that the node may hold, including associated procedures as pointers.

The following parameter values characterize the node. Unary and binary operators have sub-nodes. The other are leaf nodes. Possible leafs are literal constants or named-parameter references.

⟨*Eval trees: types*⟩≡

```
integer, parameter :: EN_UNKNOWN = 0, EN_UNARY = 1, EN_BINARY = 2
integer, parameter :: EN_CONSTANT = 3, EN_VARIABLE = 4
integer, parameter :: EN_CONDITIONAL = 5, EN_BLOCK = 6
integer, parameter :: EN_RECORD_CMD = 7
integer, parameter :: EN_OBS1_INT = 11, EN_OBS2_INT = 12
integer, parameter :: EN_OBS1_REAL = 21, EN_OBS2_REAL = 22
integer, parameter :: EN_UOBS1_INT = 31, EN_UOBS2_INT = 32
integer, parameter :: EN_UOBS1_REAL = 41, EN_UOBS2_REAL = 42
integer, parameter :: EN_PRT_FUN_UNARY = 101, EN_PRT_FUN_BINARY = 102
integer, parameter :: EN_EVAL_FUN_UNARY = 111, EN_EVAL_FUN_BINARY = 112
integer, parameter :: EN_LOG_FUN_UNARY = 121, EN_LOG_FUN_BINARY = 122
integer, parameter :: EN_INT_FUN_UNARY = 131, EN_INT_FUN_BINARY = 132
integer, parameter :: EN_REAL_FUN_UNARY = 141, EN_REAL_FUN_BINARY = 142
integer, parameter :: EN_FORMAT_STR = 161
```

This is exported only for use within unit tests.

⟨*Eval trees: public*⟩≡

```
public :: eval_node_t
```

⟨*Eval trees: types*⟩+≡

```
type :: eval_node_t
   private
   type(string_t) :: tag
   integer :: type = EN_UNKNOWN
   integer :: result_type = V_NONE
   type(var_list_t), pointer :: var_list => null ()
   type(string_t) :: var_name
   logical, pointer :: value_is_known => null ()
   logical,           pointer :: lval => null ()
   integer,           pointer :: ival => null ()
   real(default),     pointer :: rval => null ()
   complex(default),  pointer :: cval => null ()
   type(subevt_t),  pointer :: pval => null ()
   type(pdg_array_t), pointer :: aval => null ()
   type(string_t),    pointer :: sval => null ()
   type(eval_node_t), pointer :: arg0 => null ()
   type(eval_node_t), pointer :: arg1 => null ()
   type(eval_node_t), pointer :: arg2 => null ()
   type(eval_node_t), pointer :: arg3 => null ()
   type(eval_node_t), pointer :: arg4 => null ()
   procedure(obs_unary_int),   nopass, pointer :: obs1_int  => null ()
   procedure(obs_unary_real),  nopass, pointer :: obs1_real => null ()
   procedure(obs_binary_int),  nopass, pointer :: obs2_int  => null ()
```

```
    procedure(obs_binary_real), nopass, pointer :: obs2_real => null ()
    integer, pointer :: prt_type => null ()
    integer, pointer :: index => null ()
    real(default), pointer :: tolerance => null ()
    integer, pointer :: jet_algorithm => null ()
    real(default), pointer :: jet_r => null ()
    real(default), pointer :: jet_p => null ()
    real(default), pointer :: jet_ycut => null ()
    type(prt_t), pointer :: prt1 => null ()
    type(prt_t), pointer :: prt2 => null ()
    procedure(unary_log),  nopass, pointer :: op1_log  => null ()
    procedure(unary_int),  nopass, pointer :: op1_int  => null ()
    procedure(unary_real), nopass, pointer :: op1_real => null ()
    procedure(unary_cmplx), nopass, pointer :: op1_cmplx => null ()
    procedure(unary_pdg),  nopass, pointer :: op1_pdg  => null ()
    procedure(unary_sev),  nopass, pointer :: op1_sev  => null ()
    procedure(unary_str),  nopass, pointer :: op1_str  => null ()
    procedure(unary_cut),  nopass, pointer :: op1_cut  => null ()
    procedure(unary_evi),  nopass, pointer :: op1_evi  => null ()
    procedure(unary_evr),  nopass, pointer :: op1_evr  => null ()
    procedure(binary_log),  nopass, pointer :: op2_log  => null ()
    procedure(binary_int),  nopass, pointer :: op2_int  => null ()
    procedure(binary_real), nopass, pointer :: op2_real => null ()
    procedure(binary_cmplx), nopass, pointer :: op2_cmplx => null ()
    procedure(binary_pdg),  nopass, pointer :: op2_pdg  => null ()
    procedure(binary_sev),  nopass, pointer :: op2_sev  => null ()
    procedure(binary_str),  nopass, pointer :: op2_str  => null ()
    procedure(binary_cut),  nopass, pointer :: op2_cut  => null ()
    procedure(binary_evi),  nopass, pointer :: op2_evi  => null ()
    procedure(binary_evr),  nopass, pointer :: op2_evr  => null ()
  contains
  ⟨Eval trees: eval node: TBP⟩
end type eval_node_t
```

Finalize a node recursively. Allocated constants are deleted, pointers are ignored.

⟨*Eval trees: eval node: TBP*⟩≡

```
procedure :: final_rec => eval_node_final_rec
```

⟨*Eval trees: procedures*⟩≡

```
recursive subroutine eval_node_final_rec (node)
  class(eval_node_t), intent(inout) :: node
  select case (node%type)
  case (EN_UNARY)
     call eval_node_final_rec (node%arg1)
  case (EN_BINARY)
     call eval_node_final_rec (node%arg1)
     call eval_node_final_rec (node%arg2)
  case (EN_CONDITIONAL)
     call eval_node_final_rec (node%arg0)
     call eval_node_final_rec (node%arg1)
     call eval_node_final_rec (node%arg2)
  case (EN_BLOCK)
     call eval_node_final_rec (node%arg0)
```

```
      call eval_node_final_rec (node%arg1)
   case (EN_PRT_FUN_UNARY, EN_EVAL_FUN_UNARY, &
         EN_LOG_FUN_UNARY, EN_INT_FUN_UNARY, EN_REAL_FUN_UNARY)
      if (associated (node%arg0))  call eval_node_final_rec (node%arg0)
      call eval_node_final_rec (node%arg1)
      deallocate (node%index)
      deallocate (node%prt1)
   case (EN_PRT_FUN_BINARY, EN_EVAL_FUN_BINARY, &
         EN_LOG_FUN_BINARY, EN_INT_FUN_BINARY, EN_REAL_FUN_BINARY)
      if (associated (node%arg0))  call eval_node_final_rec (node%arg0)
      call eval_node_final_rec (node%arg1)
      call eval_node_final_rec (node%arg2)
      deallocate (node%index)
      deallocate (node%prt1)
      deallocate (node%prt2)
   case (EN_FORMAT_STR)
      if (associated (node%arg0))  call eval_node_final_rec (node%arg0)
      if (associated (node%arg1))  call eval_node_final_rec (node%arg1)
      deallocate (node%ival)
   case (EN_RECORD_CMD)
      if (associated (node%arg0))  call eval_node_final_rec (node%arg0)
      if (associated (node%arg1))  call eval_node_final_rec (node%arg1)
      if (associated (node%arg2))  call eval_node_final_rec (node%arg2)
      if (associated (node%arg3))  call eval_node_final_rec (node%arg3)
      if (associated (node%arg4))  call eval_node_final_rec (node%arg4)
   end select
   select case (node%type)
   case (EN_UNARY, EN_BINARY, EN_CONDITIONAL, EN_CONSTANT, EN_BLOCK, &
         EN_PRT_FUN_UNARY, EN_PRT_FUN_BINARY, &
         EN_EVAL_FUN_UNARY, EN_EVAL_FUN_BINARY, &
         EN_LOG_FUN_UNARY, EN_LOG_FUN_BINARY, &
         EN_INT_FUN_UNARY, EN_INT_FUN_BINARY, &
         EN_REAL_FUN_UNARY, EN_REAL_FUN_BINARY, &
         EN_FORMAT_STR, EN_RECORD_CMD)
      select case (node%result_type)
      case (V_LOG);  deallocate (node%lval)
      case (V_INT);  deallocate (node%ival)
      case (V_REAL); deallocate (node%rval)
      case (V_CMPLX); deallocate (node%cval)
      case (V_SEV);  deallocate (node%pval)
      case (V_PDG);  deallocate (node%aval)
      case (V_STR);  deallocate (node%sval)
      end select
      deallocate (node%value_is_known)
   end select
end subroutine eval_node_final_rec
```

Leaf nodes

Initialize a leaf node with a literal constant.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_init_log (node, lval)
```

```
  type(eval_node_t), intent(out) :: node
  logical, intent(in) :: lval
  node%type = EN_CONSTANT
  node%result_type = V_LOG
  allocate (node%lval, node%value_is_known)
  node%lval = lval
  node%value_is_known = .true.
end subroutine eval_node_init_log

subroutine eval_node_init_int (node, ival)
  type(eval_node_t), intent(out) :: node
  integer, intent(in) :: ival
  node%type = EN_CONSTANT
  node%result_type = V_INT
  allocate (node%ival, node%value_is_known)
  node%ival = ival
  node%value_is_known = .true.
end subroutine eval_node_init_int

subroutine eval_node_init_real (node, rval)
  type(eval_node_t), intent(out) :: node
  real(default), intent(in) :: rval
  node%type = EN_CONSTANT
  node%result_type = V_REAL
  allocate (node%rval, node%value_is_known)
  node%rval = rval
  node%value_is_known = .true.
end subroutine eval_node_init_real

subroutine eval_node_init_cmplx (node, cval)
  type(eval_node_t), intent(out) :: node
  complex(default), intent(in) :: cval
  node%type = EN_CONSTANT
  node%result_type = V_CMPLX
  allocate (node%cval, node%value_is_known)
  node%cval = cval
  node%value_is_known = .true.
end subroutine eval_node_init_cmplx

subroutine eval_node_init_subevt (node, pval)
  type(eval_node_t), intent(out) :: node
  type(subevt_t), intent(in) :: pval
  node%type = EN_CONSTANT
  node%result_type = V_SEV
  allocate (node%pval, node%value_is_known)
  node%pval = pval
  node%value_is_known = .true.
end subroutine eval_node_init_subevt

subroutine eval_node_init_pdg_array (node, aval)
  type(eval_node_t), intent(out) :: node
  type(pdg_array_t), intent(in) :: aval
  node%type = EN_CONSTANT
  node%result_type = V_PDG
```

```
    allocate (node%aval, node%value_is_known)
    node%aval = aval
    node%value_is_known = .true.
  end subroutine eval_node_init_pdg_array

  subroutine eval_node_init_string (node, sval)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: sval
    node%type = EN_CONSTANT
    node%result_type = V_STR
    allocate (node%sval, node%value_is_known)
    node%sval = sval
    node%value_is_known = .true.
  end subroutine eval_node_init_string
```

Initialize a leaf node with a pointer to a named parameter

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_node_init_log_ptr (node, name, lval, is_known)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: name
    logical, intent(in), target :: lval
    logical, intent(in), target :: is_known
    node%type = EN_VARIABLE
    node%tag = name
    node%result_type = V_LOG
    node%lval => lval
    node%value_is_known => is_known
  end subroutine eval_node_init_log_ptr

  subroutine eval_node_init_int_ptr (node, name, ival, is_known)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: name
    integer, intent(in), target :: ival
    logical, intent(in), target :: is_known
    node%type = EN_VARIABLE
    node%tag = name
    node%result_type = V_INT
    node%ival => ival
    node%value_is_known => is_known
  end subroutine eval_node_init_int_ptr

  subroutine eval_node_init_real_ptr (node, name, rval, is_known)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: name
    real(default), intent(in), target :: rval
    logical, intent(in), target :: is_known
    node%type = EN_VARIABLE
    node%tag = name
    node%result_type = V_REAL
    node%rval => rval
    node%value_is_known => is_known
  end subroutine eval_node_init_real_ptr

  subroutine eval_node_init_cmplx_ptr (node, name, cval, is_known)
```

```
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: name
    complex(default), intent(in), target :: cval
    logical, intent(in), target :: is_known
    node%type = EN_VARIABLE
    node%tag = name
    node%result_type = V_CMPLX
    node%cval => cval
    node%value_is_known => is_known
  end subroutine eval_node_init_cmplx_ptr

  subroutine eval_node_init_subevt_ptr (node, name, pval, is_known)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: name
    type(subevt_t), intent(in), target :: pval
    logical, intent(in), target :: is_known
    node%type = EN_VARIABLE
    node%tag = name
    node%result_type = V_SEV
    node%pval => pval
    node%value_is_known => is_known
  end subroutine eval_node_init_subevt_ptr

  subroutine eval_node_init_pdg_array_ptr (node, name, aval, is_known)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: name
    type(pdg_array_t), intent(in), target :: aval
    logical, intent(in), target :: is_known
    node%type = EN_VARIABLE
    node%tag = name
    node%result_type = V_PDG
    node%aval => aval
    node%value_is_known => is_known
  end subroutine eval_node_init_pdg_array_ptr

  subroutine eval_node_init_string_ptr (node, name, sval, is_known)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: name
    type(string_t), intent(in), target :: sval
    logical, intent(in), target :: is_known
    node%type = EN_VARIABLE
    node%tag = name
    node%result_type = V_STR
    node%sval => sval
    node%value_is_known => is_known
  end subroutine eval_node_init_string_ptr
```

The procedure-pointer cases:

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_node_init_obs1_int_ptr (node, name, obs1_iptr, p1)
    type(eval_node_t), intent(out) :: node
    type(string_t), intent(in) :: name
    procedure(obs_unary_int), intent(in), pointer :: obs1_iptr
    type(prt_t), intent(in), target :: p1
```

```
  node%type = EN_OBS1_INT
  node%tag = name
  node%result_type = V_INT
  node%obs1_int => obs1_iptr
  node%prt1 => p1
  allocate (node%ival, node%value_is_known)
  node%value_is_known = .false.
end subroutine eval_node_init_obs1_int_ptr

subroutine eval_node_init_obs2_int_ptr (node, name, obs2_iptr, p1, p2)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  procedure(obs_binary_int), intent(in), pointer :: obs2_iptr
  type(prt_t), intent(in), target :: p1, p2
  node%type = EN_OBS2_INT
  node%tag = name
  node%result_type = V_INT
  node%obs2_int => obs2_iptr
  node%prt1 => p1
  node%prt2 => p2
  allocate (node%ival, node%value_is_known)
  node%value_is_known = .false.
end subroutine eval_node_init_obs2_int_ptr

subroutine eval_node_init_obs1_real_ptr (node, name, obs1_rptr, p1)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  procedure(obs_unary_real), intent(in), pointer :: obs1_rptr
  type(prt_t), intent(in), target :: p1
  node%type = EN_OBS1_REAL
  node%tag = name
  node%result_type = V_REAL
  node%obs1_real => obs1_rptr
  node%prt1 => p1
  allocate (node%rval, node%value_is_known)
  node%value_is_known = .false.
end subroutine eval_node_init_obs1_real_ptr

subroutine eval_node_init_obs2_real_ptr (node, name, obs2_rptr, p1, p2)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  procedure(obs_binary_real), intent(in), pointer :: obs2_rptr
  type(prt_t), intent(in), target :: p1, p2
  node%type = EN_OBS2_REAL
  node%tag = name
  node%result_type = V_REAL
  node%obs2_real => obs2_rptr
  node%prt1 => p1
  node%prt2 => p2
  allocate (node%rval, node%value_is_known)
  node%value_is_known = .false.
end subroutine eval_node_init_obs2_real_ptr
```

These nodes refer to user-defined procedures.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_init_uobs1_int (node, name, arg)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  type(eval_node_t), intent(in), target :: arg
  node%type = EN_UOBS1_INT
  node%tag = name
  node%result_type = V_INT
  allocate (node%ival, node%value_is_known)
  node%value_is_known = .false.
  node%arg0 => arg
end subroutine eval_node_init_uobs1_int

subroutine eval_node_init_uobs2_int (node, name, arg)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  type(eval_node_t), intent(in), target :: arg
  node%type = EN_UOBS2_INT
  node%tag = name
  node%result_type = V_INT
  allocate (node%ival, node%value_is_known)
  node%value_is_known = .false.
  node%arg0 => arg
end subroutine eval_node_init_uobs2_int

subroutine eval_node_init_uobs1_real (node, name, arg)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  type(eval_node_t), intent(in), target :: arg
  node%type = EN_UOBS1_REAL
  node%tag = name
  node%result_type = V_REAL
  allocate (node%rval, node%value_is_known)
  node%value_is_known = .false.
  node%arg0 => arg
end subroutine eval_node_init_uobs1_real

subroutine eval_node_init_uobs2_real (node, name, arg)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: name
  type(eval_node_t), intent(in), target :: arg
  node%type = EN_UOBS2_REAL
  node%tag = name
  node%result_type = V_REAL
  allocate (node%rval, node%value_is_known)
  node%value_is_known = .false.
  node%arg0 => arg
end subroutine eval_node_init_uobs2_real
```

Branch nodes

Initialize a branch node, sub-nodes are given.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_init_branch (node, tag, result_type, arg1, arg2)
  type(eval_node_t), intent(out) :: node
  type(string_t), intent(in) :: tag
  integer, intent(in) :: result_type
  type(eval_node_t), intent(in), target :: arg1
  type(eval_node_t), intent(in), target, optional :: arg2
  if (present (arg2)) then
     node%type = EN_BINARY
  else
     node%type = EN_UNARY
  end if
  node%tag = tag
  node%result_type = result_type
  call eval_node_allocate_value (node)
  node%arg1 => arg1
  if (present (arg2))  node%arg2 => arg2
end subroutine eval_node_init_branch
```

Allocate the node value according to the result type.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_allocate_value (node)
  type(eval_node_t), intent(inout) :: node
  select case (node%result_type)
  case (V_LOG);  allocate (node%lval)
  case (V_INT);  allocate (node%ival)
  case (V_REAL); allocate (node%rval)
  case (V_CMPLX); allocate (node%cval)
  case (V_PDG);  allocate (node%aval)
  case (V_SEV);  allocate (node%pval)
     call subevt_init (node%pval)
  case (V_STR);  allocate (node%sval)
  end select
  allocate (node%value_is_known)
  node%value_is_known = .false.
end subroutine eval_node_allocate_value
```

Initialize a block node which contains, in addition to the expression to be evaluated, a variable definition. The result type is not yet assigned, because we can compile the enclosed expression only after the var list is set up.

Note that the node always allocates a new variable list and appends it to the current one. Thus, if the variable redefines an existing one, it only shadows it but does not reset it. Any side-effects are therefore absent and need not be undone outside the block.

If the flag `new` is set, a variable is (re)declared. This must not be done for intrinsic variables. Vice versa, if the variable is not existent, the `new` flag is required.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_init_block (node, name, type, var_def, var_list)
  type(eval_node_t), intent(out), target :: node
  type(string_t), intent(in) :: name
  integer, intent(in) :: type
  type(eval_node_t), intent(in), target :: var_def
```

```
    type(var_list_t), intent(in), target :: var_list
    node%type = EN_BLOCK
    node%tag = "var_def"
    node%var_name = name
    node%arg1 => var_def
    allocate (node%var_list)
    call node%var_list%link (var_list)
    if (var_def%type == EN_CONSTANT) then
       select case (type)
       case (V_LOG)
          call var_list_append_log  (node%var_list, name, var_def%lval)
       case (V_INT)
          call var_list_append_int  (node%var_list, name, var_def%ival)
       case (V_REAL)
          call var_list_append_real (node%var_list, name, var_def%rval)
       case (V_CMPLX)
          call var_list_append_cmplx (node%var_list, name, var_def%cval)
       case (V_PDG)
          call var_list_append_pdg_array &
               (node%var_list, name, var_def%aval)
       case (V_SEV)
          call var_list_append_subevt &
               (node%var_list, name, var_def%pval)
       case (V_STR)
          call var_list_append_string (node%var_list, name, var_def%sval)
       end select
    else
       select case (type)
       case (V_LOG);  call var_list_append_log_ptr &
            (node%var_list, name, var_def%lval, var_def%value_is_known)
       case (V_INT);  call var_list_append_int_ptr &
            (node%var_list, name, var_def%ival, var_def%value_is_known)
       case (V_REAL); call var_list_append_real_ptr &
            (node%var_list, name, var_def%rval, var_def%value_is_known)
       case (V_CMPLX); call var_list_append_cmplx_ptr &
            (node%var_list, name, var_def%cval, var_def%value_is_known)
       case (V_PDG);  call var_list_append_pdg_array_ptr &
            (node%var_list, name, var_def%aval, var_def%value_is_known)
       case (V_SEV); call var_list_append_subevt_ptr &
            (node%var_list, name, var_def%pval, var_def%value_is_known)
       case (V_STR); call var_list_append_string_ptr &
            (node%var_list, name, var_def%sval, var_def%value_is_known)
       end select
    end if
  end subroutine eval_node_init_block
```

Complete block initialization by assigning the expression to evaluate to `arg0`.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_node_set_expr (node, arg, result_type)
    type(eval_node_t), intent(inout) :: node
    type(eval_node_t), intent(in), target :: arg
    integer, intent(in), optional :: result_type
    if (present (result_type)) then
       node%result_type = result_type
```

```
    else
       node%result_type = arg%result_type
    end if
    call eval_node_allocate_value (node)
    node%arg0 => arg
  end subroutine eval_node_set_expr
```

Initialize a conditional. There are three branches: the condition (evaluates to logical) and the two alternatives (evaluate both to the same arbitrary type).

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_node_init_conditional (node, result_type, cond, arg1, arg2)
    type(eval_node_t), intent(out) :: node
    integer, intent(in) :: result_type
    type(eval_node_t), intent(in), target :: cond, arg1, arg2
    node%type = EN_CONDITIONAL
    node%tag = "cond"
    node%result_type = result_type
    call eval_node_allocate_value (node)
    node%arg0 => cond
    node%arg1 => arg1
    node%arg2 => arg2
  end subroutine eval_node_init_conditional
```

Initialize a recording command (which evaluates to a logical constant). The first branch is the ID of the analysis object to be filled, the optional branches 1 to 4 are the values to be recorded.

If the event-weight pointer is null, we record values with unit weight. Otherwise, we use the value pointed to as event weight.

There can be up to four arguments which represent x, y, Δy, Δx. Therefore, this is the only node type that may fill four sub-nodes.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_node_init_record_cmd &
      (node, event_weight, id, arg1, arg2, arg3, arg4)
    type(eval_node_t), intent(out) :: node
    real(default), pointer :: event_weight
    type(eval_node_t), intent(in), target :: id
    type(eval_node_t), intent(in), optional, target :: arg1, arg2, arg3, arg4
    call eval_node_init_log (node, .true.)
    node%type = EN_RECORD_CMD
    node%rval => event_weight
    node%tag = "record_cmd"
    node%arg0 => id
    if (present (arg1)) then
       node%arg1 => arg1
       if (present (arg2)) then
          node%arg2 => arg2
          if (present (arg3)) then
             node%arg3 => arg3
             if (present (arg4)) then
                node%arg4 => arg4
             end if
          end if
```

```
      end if
    end if
  end subroutine eval_node_init_record_cmd
```

Initialize a node for operations on subevents. The particle lists (one or two) are inserted as `arg1` and `arg2`. We allocated particle pointers as temporaries for iterating over particle lists. The procedure pointer which holds the function to evaluate for the subevents (e.g., combine, select) is also initialized.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_node_init_prt_fun_unary (node, arg1, name, proc)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1
    type(string_t), intent(in) :: name
    procedure(unary_sev) :: proc
    node%type = EN_PRT_FUN_UNARY
    node%tag = name
    node%result_type = V_SEV
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    allocate (node%index, source = 0)
    allocate (node%prt1)
    node%op1_sev => proc
  end subroutine eval_node_init_prt_fun_unary

  subroutine eval_node_init_prt_fun_binary (node, arg1, arg2, name, proc)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1, arg2
    type(string_t), intent(in) :: name
    procedure(binary_sev) :: proc
    node%type = EN_PRT_FUN_BINARY
    node%tag = name
    node%result_type = V_SEV
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    node%arg2 => arg2
    allocate (node%index, source = 0)
    allocate (node%prt1)
    allocate (node%prt2)
    node%op2_sev => proc
  end subroutine eval_node_init_prt_fun_binary
```

Similar, but for particle-list functions that evaluate to a real value.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_node_init_eval_fun_unary (node, arg1, name)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), intent(in), target :: arg1
    type(string_t), intent(in) :: name
    node%type = EN_EVAL_FUN_UNARY
    node%tag = name
    node%result_type = V_REAL
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    allocate (node%index, source = 0)
```

```
  allocate (node%prt1)
end subroutine eval_node_init_eval_fun_unary

subroutine eval_node_init_eval_fun_binary (node, arg1, arg2, name)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), intent(in), target :: arg1, arg2
  type(string_t), intent(in) :: name
  node%type = EN_EVAL_FUN_BINARY
  node%tag = name
  node%result_type = V_REAL
  call eval_node_allocate_value (node)
  node%arg1 => arg1
  node%arg2 => arg2
  allocate (node%index, source = 0)
  allocate (node%prt1)
  allocate (node%prt2)
end subroutine eval_node_init_eval_fun_binary
```

These are for particle-list functions that evaluate to a logical value.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_init_log_fun_unary (node, arg1, name, proc)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), intent(in), target :: arg1
  type(string_t), intent(in) :: name
  procedure(unary_cut) :: proc
  node%type = EN_LOG_FUN_UNARY
  node%tag = name
  node%result_type = V_LOG
  call eval_node_allocate_value (node)
  node%arg1 => arg1
  allocate (node%index, source = 0)
  allocate (node%prt1)
  node%op1_cut => proc
end subroutine eval_node_init_log_fun_unary

subroutine eval_node_init_log_fun_binary (node, arg1, arg2, name, proc)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), intent(in), target :: arg1, arg2
  type(string_t), intent(in) :: name
  procedure(binary_cut) :: proc
  node%type = EN_LOG_FUN_BINARY
  node%tag = name
  node%result_type = V_LOG
  call eval_node_allocate_value (node)
  node%arg1 => arg1
  node%arg2 => arg2
  allocate (node%index, source = 0)
  allocate (node%prt1)
  allocate (node%prt2)
  node%op2_cut => proc
end subroutine eval_node_init_log_fun_binary
```

These are for particle-list functions that evaluate to an integer value.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_init_int_fun_unary (node, arg1, name, proc)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), intent(in), target :: arg1
  type(string_t), intent(in) :: name
  procedure(unary_evi) :: proc
  node%type = EN_INT_FUN_UNARY
  node%tag = name
  node%result_type = V_INT
  call eval_node_allocate_value (node)
  node%arg1 => arg1
  allocate (node%index, source = 0)
  allocate (node%prt1)
  node%op1_evi => proc
end subroutine eval_node_init_int_fun_unary

subroutine eval_node_init_int_fun_binary (node, arg1, arg2, name, proc)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), intent(in), target :: arg1, arg2
  type(string_t), intent(in) :: name
  procedure(binary_evi) :: proc
  node%type = EN_INT_FUN_BINARY
  node%tag = name
  node%result_type = V_INT
  call eval_node_allocate_value (node)
  node%arg1 => arg1
  node%arg2 => arg2
  allocate (node%index, source = 0)
  allocate (node%prt1)
  allocate (node%prt2)
  node%op2_evi => proc
end subroutine eval_node_init_int_fun_binary
```

These are for particle-list functions that evaluate to a real value.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_init_real_fun_unary (node, arg1, name, proc)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), intent(in), target :: arg1
  type(string_t), intent(in) :: name
  procedure(unary_evr) :: proc
  node%type = EN_REAL_FUN_UNARY
  node%tag = name
  node%result_type = V_INT
  call eval_node_allocate_value (node)
  node%arg1 => arg1
  allocate (node%index, source = 0)
  allocate (node%prt1)
  node%op1_evr => proc
end subroutine eval_node_init_real_fun_unary

subroutine eval_node_init_real_fun_binary (node, arg1, arg2, name, proc)
  type(eval_node_t), intent(out) :: node
  type(eval_node_t), intent(in), target :: arg1, arg2
  type(string_t), intent(in) :: name
```

```
    procedure(binary_evr) :: proc
    node%type = EN_REAL_FUN_BINARY
    node%tag = name
    node%result_type = V_INT
    call eval_node_allocate_value (node)
    node%arg1 => arg1
    node%arg2 => arg2
    allocate (node%index, source = 0)
    allocate (node%prt1)
    allocate (node%prt2)
    node%op2_evr => proc
  end subroutine eval_node_init_real_fun_binary
```

Initialize a node for a string formatting function (sprintf).

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_node_init_format_string (node, fmt, arg, name, n_args)
    type(eval_node_t), intent(out) :: node
    type(eval_node_t), pointer :: fmt, arg
    type(string_t), intent(in) :: name
    integer, intent(in) :: n_args
    node%type = EN_FORMAT_STR
    node%tag = name
    node%result_type = V_STR
    call eval_node_allocate_value (node)
    node%arg0 => fmt
    node%arg1 => arg
    allocate (node%ival)
    node%ival = n_args
  end subroutine eval_node_init_format_string
```

If particle functions depend upon a condition (or an expression is evaluated), the observables that can be evaluated for the given particles have to be thrown on the local variable stack. This is done here. Each observable is initialized with the particle pointers which have been allocated for the node.

The integer variable that is referred to by the `Index` pseudo-observable is always known when it is referred to.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_node_set_observables (node, var_list)
    type(eval_node_t), intent(inout) :: node
    type(var_list_t), intent(in), target :: var_list
    logical, save, target :: known = .true.
    allocate (node%var_list)
    call node%var_list%link (var_list)
    allocate (node%index, source = 0)
    call var_list_append_int_ptr &
         (node%var_list, var_str ("Index"), node%index, known, intrinsic=.true.)
    if (.not. associated (node%prt2)) then
       call var_list_set_observables_unary &
            (node%var_list, node%prt1)
    else
       call var_list_set_observables_binary &
            (node%var_list, node%prt1, node%prt2)
    end if
```

```
end subroutine eval_node_set_observables
```

Output

⟨*Eval trees: eval node: TBP*⟩+≡

```
procedure :: write => eval_node_write
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_write (node, unit, indent)
  class(eval_node_t), intent(in) :: node
  integer, intent(in), optional :: unit
  integer, intent(in), optional :: indent
  integer :: u, ind
  u = given_output_unit (unit);  if (u < 0)  return
  ind = 0;  if (present (indent)) ind = indent
  write (u, "(A)", advance="no")  repeat ("|  ", ind) // "o "
  select case (node%type)
  case (EN_UNARY, EN_BINARY, EN_CONDITIONAL, &
        EN_PRT_FUN_UNARY, EN_PRT_FUN_BINARY, &
        EN_EVAL_FUN_UNARY, EN_EVAL_FUN_BINARY, &
        EN_LOG_FUN_UNARY, EN_LOG_FUN_BINARY, &
        EN_INT_FUN_UNARY, EN_INT_FUN_BINARY, &
        EN_REAL_FUN_UNARY, EN_REAL_FUN_BINARY)
     write (u, "(A)", advance="no")  "[" // char (node%tag) // "] ="
  case (EN_CONSTANT)
     write (u, "(A)", advance="no")  "[const] ="
  case (EN_VARIABLE)
     write (u, "(A)", advance="no")  char (node%tag) // " =>"
  case (EN_OBS1_INT, EN_OBS2_INT, EN_OBS1_REAL, EN_OBS2_REAL, &
       EN_UOBS1_INT, EN_UOBS2_INT, EN_UOBS1_REAL, EN_UOBS2_REAL)
     write (u, "(A)", advance="no")  char (node%tag) // " ="
  case (EN_BLOCK)
     write (u, "(A)", advance="no")  "[" // char (node%tag) // "]" // &
          char (node%var_name) // " [expr] = "
  case default
     write (u, "(A)", advance="no")  "[???] ="
  end select
  select case (node%result_type)
  case (V_LOG)
     if (node%value_is_known) then
        if (node%lval) then
           write (u, "(1x,A)") "true"
        else
           write (u, "(1x,A)") "false"
        end if
     else
        write (u, "(1x,A)") "[unknown logical]"
     end if
  case (V_INT)
     if (node%value_is_known) then
        write (u, "(1x,I0)")  node%ival
     else
        write (u, "(1x,A)") "[unknown integer]"
     end if
```

```
  case (V_REAL)
     if (node%value_is_known) then
        write (u, "(1x," // FMT_19 // ")") node%rval
     else
        write (u, "(1x,A)") "[unknown real]"
     end if
 case (V_CMPLX)
     if (node%value_is_known) then
        write (u, "(1x,'(',," // FMT_19 // ",',',," // &
             FMT_19 // ",')')") node%cval
     else
        write (u, "(1x,A)") "[unknown complex]"
     end if
  case (V_SEV)
     if (char (node%tag) == "@evt") then
        write (u, "(1x,A)") "[event subevent]"
     else if (node%value_is_known) then
        call subevt_write &
             (node%pval, unit, prefix = repeat ("|  ", ind + 1))
     else
        write (u, "(1x,A)") "[unknown subevent]"
     end if
  case (V_PDG)
     write (u, "(1x)", advance="no")
     call pdg_array_write (node%aval, u);  write (u, *)
  case (V_STR)
     if (node%value_is_known) then
        write (u, "(A)")  '"' // char (node%sval) // '"'
     else
        write (u, "(1x,A)") "[unknown string]"
     end if
  case default
     write (u, "(1x,A)") "[empty]"
  end select
  select case (node%type)
  case (EN_OBS1_INT, EN_OBS1_REAL, EN_UOBS1_INT, EN_UOBS1_REAL)
     write (u, "(A,6x,A)", advance="no")  repeat ("|  ", ind), "prt1 ="
     call prt_write (node%prt1, unit)
  case (EN_OBS2_INT, EN_OBS2_REAL, EN_UOBS2_INT, EN_UOBS2_REAL)
     write (u, "(A,6x,A)", advance="no")  repeat ("|  ", ind), "prt1 ="
     call prt_write (node%prt1, unit)
     write (u, "(A,6x,A)", advance="no")  repeat ("|  ", ind), "prt2 ="
     call prt_write (node%prt2, unit)
  end select
end subroutine eval_node_write

recursive subroutine eval_node_write_rec (node, unit, indent)
  type(eval_node_t), intent(in) :: node
  integer, intent(in), optional :: unit
  integer, intent(in), optional :: indent
  integer :: u, ind
  u = given_output_unit (unit);  if (u < 0)  return
  ind = 0;  if (present (indent))  ind = indent
  call eval_node_write (node, unit, indent)
```

```
  select case (node%type)
  case (EN_UNARY)
     if (associated (node%arg0)) &
          call eval_node_write_rec (node%arg0, unit, ind+1)
     call eval_node_write_rec (node%arg1, unit, ind+1)
  case (EN_BINARY)
     if (associated (node%arg0)) &
          call eval_node_write_rec (node%arg0, unit, ind+1)
     call eval_node_write_rec (node%arg1, unit, ind+1)
     call eval_node_write_rec (node%arg2, unit, ind+1)
  case (EN_BLOCK)
     call eval_node_write_rec (node%arg1, unit, ind+1)
     call eval_node_write_rec (node%arg0, unit, ind+1)
  case (EN_CONDITIONAL)
     call eval_node_write_rec (node%arg0, unit, ind+1)
     call eval_node_write_rec (node%arg1, unit, ind+1)
     call eval_node_write_rec (node%arg2, unit, ind+1)
  case (EN_PRT_FUN_UNARY, EN_EVAL_FUN_UNARY, &
        EN_LOG_FUN_UNARY, EN_INT_FUN_UNARY, EN_REAL_FUN_UNARY)
     if (associated (node%arg0)) &
          call eval_node_write_rec (node%arg0, unit, ind+1)
     call eval_node_write_rec (node%arg1, unit, ind+1)
  case (EN_PRT_FUN_BINARY, EN_EVAL_FUN_BINARY, &
        EN_LOG_FUN_BINARY, EN_INT_FUN_BINARY, EN_REAL_FUN_BINARY)
     if (associated (node%arg0)) &
          call eval_node_write_rec (node%arg0, unit, ind+1)
     call eval_node_write_rec (node%arg1, unit, ind+1)
     call eval_node_write_rec (node%arg2, unit, ind+1)
  case (EN_RECORD_CMD)
     if (associated (node%arg1)) then
        call eval_node_write_rec (node%arg1, unit, ind+1)
        if (associated (node%arg2)) then
           call eval_node_write_rec (node%arg2, unit, ind+1)
           if (associated (node%arg3)) then
              call eval_node_write_rec (node%arg3, unit, ind+1)
              if (associated (node%arg4)) then
                 call eval_node_write_rec (node%arg4, unit, ind+1)
              end if
           end if
        end if
     end if
  end select
end subroutine eval_node_write_rec
```

28.3.2 Operation types

For the operations associated to evaluation tree nodes, we define abstract interfaces for all cases.

Particles/subevents are transferred by-reference, to avoid unnecessary copying. Therefore, subroutines instead of functions. (Furthermore, the function version of `unary_prt` triggers an obscure bug in nagfor 5.2(649) [invalid C code].)

⟨*Eval trees: interfaces*⟩≡

```
abstract interface
   logical function unary_log (arg)
     import eval_node_t
     type(eval_node_t), intent(in) :: arg
   end function unary_log
end interface
abstract interface
   integer function unary_int (arg)
     import eval_node_t
     type(eval_node_t), intent(in) :: arg
   end function unary_int
end interface
abstract interface
   real(default) function unary_real (arg)
     import default
     import eval_node_t
     type(eval_node_t), intent(in) :: arg
   end function unary_real
end interface
abstract interface
   complex(default) function unary_cmplx (arg)
     import default
     import eval_node_t
     type(eval_node_t), intent(in) :: arg
   end function unary_cmplx
end interface
abstract interface
   subroutine unary_pdg (pdg_array, arg)
     import pdg_array_t
     import eval_node_t
     type(pdg_array_t), intent(out) :: pdg_array
     type(eval_node_t), intent(in) :: arg
   end subroutine unary_pdg
end interface
abstract interface
   subroutine unary_sev (subevt, arg, arg0)
     import subevt_t
     import eval_node_t
     type(subevt_t), intent(inout) :: subevt
     type(eval_node_t), intent(in) :: arg
     type(eval_node_t), intent(inout), optional :: arg0
   end subroutine unary_sev
end interface
abstract interface
   subroutine unary_str (string, arg)
     import string_t
     import eval_node_t
     type(string_t), intent(out) :: string
     type(eval_node_t), intent(in) :: arg
   end subroutine unary_str
end interface
abstract interface
   logical function unary_cut (arg1, arg0)
     import eval_node_t
```

```
     type(eval_node_t), intent(in) :: arg1
     type(eval_node_t), intent(inout) :: arg0
   end function unary_cut
end interface
abstract interface
   subroutine unary_evi (ival, arg1, arg0)
     import eval_node_t
     integer, intent(out) :: ival
     type(eval_node_t), intent(in) :: arg1
     type(eval_node_t), intent(inout), optional :: arg0
   end subroutine unary_evi
end interface
abstract interface
   subroutine unary_evr (rval, arg1, arg0)
     import eval_node_t, default
     real(default), intent(out) :: rval
     type(eval_node_t), intent(in) :: arg1
     type(eval_node_t), intent(inout), optional :: arg0
   end subroutine unary_evr
end interface
abstract interface
   logical function binary_log (arg1, arg2)
     import eval_node_t
     type(eval_node_t), intent(in) :: arg1, arg2
   end function binary_log
end interface
abstract interface
   integer function binary_int (arg1, arg2)
     import eval_node_t
     type(eval_node_t), intent(in) :: arg1, arg2
   end function binary_int
end interface
abstract interface
   real(default) function binary_real (arg1, arg2)
     import default
     import eval_node_t
     type(eval_node_t), intent(in) :: arg1, arg2
   end function binary_real
end interface
abstract interface
   complex(default) function binary_cmplx (arg1, arg2)
     import default
     import eval_node_t
     type(eval_node_t), intent(in) :: arg1, arg2
   end function binary_cmplx
end interface
abstract interface
   subroutine binary_pdg (pdg_array, arg1, arg2)
     import pdg_array_t
     import eval_node_t
     type(pdg_array_t), intent(out) :: pdg_array
     type(eval_node_t), intent(in) :: arg1, arg2
   end subroutine binary_pdg
end interface
```

```
abstract interface
   subroutine binary_sev (subevt, arg1, arg2, arg0)
     import subevt_t
     import eval_node_t
     type(subevt_t), intent(inout) :: subevt
     type(eval_node_t), intent(in) :: arg1, arg2
     type(eval_node_t), intent(inout), optional :: arg0
   end subroutine binary_sev
end interface
abstract interface
   subroutine binary_str (string, arg1, arg2)
     import string_t
     import eval_node_t
     type(string_t), intent(out) :: string
     type(eval_node_t), intent(in) :: arg1, arg2
   end subroutine binary_str
end interface
abstract interface
   logical function binary_cut (arg1, arg2, arg0)
     import eval_node_t
     type(eval_node_t), intent(in) :: arg1, arg2
     type(eval_node_t), intent(inout) :: arg0
   end function binary_cut
end interface
abstract interface
   subroutine binary_evi (ival, arg1, arg2, arg0)
     import eval_node_t
     integer, intent(out) :: ival
     type(eval_node_t), intent(in) :: arg1, arg2
     type(eval_node_t), intent(inout), optional :: arg0
   end subroutine binary_evi
end interface
abstract interface
   subroutine binary_evr (rval, arg1, arg2, arg0)
     import eval_node_t, default
     real(default), intent(out) :: rval
     type(eval_node_t), intent(in) :: arg1, arg2
     type(eval_node_t), intent(inout), optional :: arg0
   end subroutine binary_evr
end interface
```

The following subroutines set the procedure pointer:

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_set_op1_log (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(unary_log) :: op
  en%op1_log => op
  end subroutine eval_node_set_op1_log

subroutine eval_node_set_op1_int (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(unary_int) :: op
  en%op1_int => op
end subroutine eval_node_set_op1_int
```

```
subroutine eval_node_set_op1_real (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(unary_real) :: op
  en%op1_real => op
end subroutine eval_node_set_op1_real

subroutine eval_node_set_op1_cmplx (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(unary_cmplx) :: op
  en%op1_cmplx => op
end subroutine eval_node_set_op1_cmplx

subroutine eval_node_set_op1_pdg (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(unary_pdg) :: op
  en%op1_pdg => op
end subroutine eval_node_set_op1_pdg

subroutine eval_node_set_op1_sev (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(unary_sev) :: op
  en%op1_sev => op
end subroutine eval_node_set_op1_sev

subroutine eval_node_set_op1_str (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(unary_str) :: op
  en%op1_str => op
end subroutine eval_node_set_op1_str

subroutine eval_node_set_op2_log (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_log) :: op
  en%op2_log => op
end subroutine eval_node_set_op2_log

subroutine eval_node_set_op2_int (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_int) :: op
  en%op2_int => op
end subroutine eval_node_set_op2_int

subroutine eval_node_set_op2_real (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_real) :: op
  en%op2_real => op
end subroutine eval_node_set_op2_real

subroutine eval_node_set_op2_cmplx (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_cmplx) :: op
  en%op2_cmplx => op
end subroutine eval_node_set_op2_cmplx
```

```
subroutine eval_node_set_op2_pdg (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_pdg) :: op
  en%op2_pdg => op
end subroutine eval_node_set_op2_pdg

subroutine eval_node_set_op2_sev (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_sev) :: op
  en%op2_sev => op
end subroutine eval_node_set_op2_sev

subroutine eval_node_set_op2_str (en, op)
  type(eval_node_t), intent(inout) :: en
  procedure(binary_str) :: op
  en%op2_str => op
end subroutine eval_node_set_op2_str
```

28.3.3 Specific operators

Our expression syntax contains all Fortran functions that make sense. These functions have to be provided in a form that they can be used in procedures pointers, and have the abstract interfaces above. For some intrinsic functions, we could use specific versions provided by Fortran directly. However, this has two drawbacks: (i) We should work with the values instead of the eval-nodes as argument, which complicates the interface; (ii) more importantly, the `default` real type need not be equivalent to double precision. This would, at least, introduce system dependencies. Finally, for operators there are no specific versions.

Therefore, we write wrappers for all possible functions, at the expense of some overhead.

Binary numerical functions

⟨*Eval trees: procedures*⟩+≡

```
integer function add_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival + en2%ival
end function add_ii
real(default) function add_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival + en2%rval
end function add_ir
complex(default) function add_ic (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival + en2%cval
end function add_ic
real(default) function add_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval + en2%ival
end function add_ri
complex(default) function add_ci (en1, en2) result (y)
```

```
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval + en2%ival
end function add_ci
complex(default) function add_cr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval + en2%rval
end function add_cr
complex(default) function add_rc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval + en2%cval
end function add_rc
real(default) function add_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval + en2%rval
end function add_rr
complex(default) function add_cc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval + en2%cval
end function add_cc

integer function sub_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival - en2%ival
end function sub_ii
real(default) function sub_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival - en2%rval
end function sub_ir
real(default) function sub_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval - en2%ival
end function sub_ri
complex(default) function sub_ic (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival - en2%cval
end function sub_ic
complex(default) function sub_ci (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval - en2%ival
end function sub_ci
complex(default) function sub_cr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval - en2%rval
end function sub_cr
complex(default) function sub_rc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval - en2%cval
end function sub_rc
real(default) function sub_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval - en2%rval
end function sub_rr
complex(default) function sub_cc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
```

```
  y = en1%cval - en2%cval
end function sub_cc

integer function mul_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival * en2%ival
end function mul_ii
real(default) function mul_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival * en2%rval
end function mul_ir
real(default) function mul_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval * en2%ival
end function mul_ri
complex(default) function mul_ic (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival * en2%cval
end function mul_ic
complex(default) function mul_ci (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval * en2%ival
end function mul_ci
complex(default) function mul_rc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval * en2%cval
end function mul_rc
complex(default) function mul_cr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval * en2%rval
end function mul_cr
real(default) function mul_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval * en2%rval
end function mul_rr
complex(default) function mul_cc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval * en2%cval
end function mul_cc

integer function div_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (en2%ival == 0) then
     if (en1%ival >= 0) then
        call msg_warning ("division by zero: " // int2char (en1%ival) // &
           " / 0 ; result set to 0")
     else
        call msg_warning ("division by zero: (" // int2char (en1%ival) // &
           ") / 0 ; result set to 0")
     end if
     y = 0
     return
  end if
  y = en1%ival / en2%ival
```

```
end function div_ii
real(default) function div_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival / en2%rval
end function div_ir
real(default) function div_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval / en2%ival
end function div_ri
complex(default) function div_ic (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival / en2%cval
end function div_ic
complex(default) function div_ci (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval / en2%ival
end function div_ci
complex(default) function div_rc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval / en2%cval
end function div_rc
complex(default) function div_cr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval / en2%rval
end function div_cr
real(default) function div_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval / en2%rval
end function div_rr
complex(default) function div_cc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval / en2%cval
end function div_cc

integer function pow_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  integer :: a, b
  real(default) :: rres
  a = en1%ival
  b = en2%ival
  if ((a == 0) .and. (b < 0)) then
     call msg_warning ("division by zero: " // int2char (a) // &
        " ^ (" // int2char (b) // ") ; result set to 0")
     y = 0
     return
  end if
  rres = real(a, default) ** b
  y = rres
  if (real(y, default) /= rres) then
     if (b < 0) then
        call msg_warning ("result of all-integer operation " // &
              int2char (a) // " ^ (" // int2char (b) // &
              ") has been trucated to "// int2char (y), &
           [  var_str ("Chances are that you want to use " // &
```

```
              "reals instead of integers at this point.") ])
     else
        call msg_warning ("integer overflow in " // int2char (a) // &
              " ^ " // int2char (b) // " ; result is " // int2char (y), &
           [  var_str ("Using reals instead of integers might help.")])
     end if
  end if
end function pow_ii
real(default) function pow_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval ** en2%ival
end function pow_ri
complex(default) function pow_ci (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval ** en2%ival
end function pow_ci
real(default) function pow_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival ** en2%rval
end function pow_ir
real(default) function pow_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval ** en2%rval
end function pow_rr
complex(default) function pow_cr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval ** en2%rval
end function pow_cr
complex(default) function pow_ic (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival ** en2%cval
end function pow_ic
complex(default) function pow_rc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval ** en2%cval
end function pow_rc
complex(default) function pow_cc (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%cval ** en2%cval
end function pow_cc

integer function max_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = max (en1%ival, en2%ival)
end function max_ii
real(default) function max_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = max (real (en1%ival, default), en2%rval)
end function max_ir
real(default) function max_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = max (en1%rval, real (en2%ival, default))
end function max_ri
real(default) function max_rr (en1, en2) result (y)
```

```
  type(eval_node_t), intent(in) :: en1, en2
  y = max (en1%rval, en2%rval)
end function max_rr
integer function min_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = min (en1%ival, en2%ival)
end function min_ii
real(default) function min_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = min (real (en1%ival, default), en2%rval)
end function min_ir
real(default) function min_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = min (en1%rval, real (en2%ival, default))
end function min_ri
real(default) function min_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = min (en1%rval, en2%rval)
end function min_rr

integer function mod_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = mod (en1%ival, en2%ival)
end function mod_ii
real(default) function mod_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = mod (real (en1%ival, default), en2%rval)
end function mod_ir
real(default) function mod_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = mod (en1%rval, real (en2%ival, default))
end function mod_ri
real(default) function mod_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = mod (en1%rval, en2%rval)
end function mod_rr
integer function modulo_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = modulo (en1%ival, en2%ival)
end function modulo_ii
real(default) function modulo_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = modulo (real (en1%ival, default), en2%rval)
end function modulo_ir
real(default) function modulo_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = modulo (en1%rval, real (en2%ival, default))
end function modulo_ri
real(default) function modulo_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = modulo (en1%rval, en2%rval)
end function modulo_rr
```

Unary numeric functions

⟨*Eval trees: procedures*⟩+≡

```
real(default) function real_i (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = en%ival
end function real_i
real(default) function real_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = en%cval
end function real_c
integer function int_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = en%rval
end function int_r
complex(default) function cmplx_i (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = en%ival
end function cmplx_i
integer function int_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = en%cval
end function int_c
complex(default) function cmplx_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = en%rval
end function cmplx_r
integer function nint_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = nint (en%rval)
end function nint_r
integer function floor_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = floor (en%rval)
end function floor_r
integer function ceiling_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = ceiling (en%rval)
end function ceiling_r

integer function neg_i (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = - en%ival
end function neg_i
real(default) function neg_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = - en%rval
end function neg_r
complex(default) function neg_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = - en%cval
end function neg_c
integer function abs_i (en) result (y)
  type(eval_node_t), intent(in) :: en
```

```
  y = abs (en%ival)
end function abs_i
real(default) function abs_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = abs (en%rval)
end function abs_r
real(default) function abs_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = abs (en%cval)
end function abs_c
integer function sgn_i (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sign (1, en%ival)
end function sgn_i
real(default) function sgn_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sign (1._default, en%rval)
end function sgn_r

real(default) function sqrt_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sqrt (en%rval)
end function sqrt_r
real(default) function exp_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = exp (en%rval)
end function exp_r
real(default) function log_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = log (en%rval)
end function log_r
real(default) function log10_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = log10 (en%rval)
end function log10_r

complex(default) function sqrt_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sqrt (en%cval)
end function sqrt_c
complex(default) function exp_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = exp (en%cval)
end function exp_c
complex(default) function log_c (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = log (en%cval)
end function log_c

real(default) function sin_r (en) result (y)
  type(eval_node_t), intent(in) :: en
  y = sin (en%rval)
end function sin_r
real(default) function cos_r (en) result (y)
```

```
    type(eval_node_t), intent(in) :: en
    y = cos (en%rval)
  end function cos_r
  real(default) function tan_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = tan (en%rval)
  end function tan_r
  real(default) function asin_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = asin (en%rval)
  end function asin_r
  real(default) function acos_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = acos (en%rval)
  end function acos_r
  real(default) function atan_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = atan (en%rval)
  end function atan_r

  complex(default) function sin_c (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = sin (en%cval)
  end function sin_c
  complex(default) function cos_c (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = cos (en%cval)
  end function cos_c

  real(default) function sinh_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = sinh (en%rval)
  end function sinh_r
  real(default) function cosh_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = cosh (en%rval)
  end function cosh_r
  real(default) function tanh_r (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = tanh (en%rval)
  end function tanh_r
!!! These are F2008 additions but accepted by nagfor 5.3 and gfortran 4.6+
!!! Currently not used.
!   real(default) function asinh_r (en) result (y)
!     type(eval_node_t), intent(in) :: en
!     y = asinh (en%rval)
!   end function asinh_r
!   real(default) function acosh_r (en) result (y)
!     type(eval_node_t), intent(in) :: en
!     y = acosh (en%rval)
!   end function acosh_r
!   real(default) function atanh_r (en) result (y)
!     type(eval_node_t), intent(in) :: en
!     y = atanh (en%rval)
```

```
!   end function atanh_r
```

Binary logical functions

Logical expressions:

⟨*Eval trees: procedures*⟩+≡

```
logical function ignore_first_ll (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en2%lval
end function ignore_first_ll
logical function or_ll (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%lval .or. en2%lval
end function or_ll
logical function and_ll (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%lval .and. en2%lval
end function and_ll
```

Comparisons:

⟨*Eval trees: procedures*⟩+≡

```
logical function comp_lt_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival < en2%ival
end function comp_lt_ii
logical function comp_lt_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival < en2%rval
end function comp_lt_ir
logical function comp_lt_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval < en2%ival
end function comp_lt_ri
logical function comp_lt_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval < en2%rval
end function comp_lt_rr

logical function comp_gt_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival > en2%ival
end function comp_gt_ii
logical function comp_gt_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival > en2%rval
end function comp_gt_ir
logical function comp_gt_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval > en2%ival
end function comp_gt_ri
logical function comp_gt_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
```

```
  y = en1%rval > en2%rval
end function comp_gt_rr

logical function comp_le_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival <= en2%ival
end function comp_le_ii
logical function comp_le_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival <= en2%rval
end function comp_le_ir
logical function comp_le_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval <= en2%ival
end function comp_le_ri
logical function comp_le_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval <= en2%rval
end function comp_le_rr

logical function comp_ge_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival >= en2%ival
end function comp_ge_ii
logical function comp_ge_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival >= en2%rval
end function comp_ge_ir
logical function comp_ge_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval >= en2%ival
end function comp_ge_ri
logical function comp_ge_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval >= en2%rval
end function comp_ge_rr

logical function comp_eq_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival == en2%ival
end function comp_eq_ii
logical function comp_eq_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival == en2%rval
end function comp_eq_ir
logical function comp_eq_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval == en2%ival
end function comp_eq_ri
logical function comp_eq_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval == en2%rval
end function comp_eq_rr
logical function comp_eq_ss (en1, en2) result (y)
```

```
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%sval == en2%sval
end function comp_eq_ss

logical function comp_ne_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival /= en2%ival
end function comp_ne_ii
logical function comp_ne_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%ival /= en2%rval
end function comp_ne_ir
logical function comp_ne_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval /= en2%ival
end function comp_ne_ri
logical function comp_ne_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%rval /= en2%rval
end function comp_ne_rr
logical function comp_ne_ss (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  y = en1%sval /= en2%sval
end function comp_ne_ss
```

Comparisons with tolerance:

⟨*Eval trees: procedures*⟩+≡

```
logical function comp_se_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = abs (en1%ival - en2%ival) <= en1%tolerance
  else
     y = en1%ival == en2%ival
  end if
end function comp_se_ii
logical function comp_se_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = abs (en1%rval - en2%ival) <= en1%tolerance
  else
     y = en1%rval == en2%ival
  end if
end function comp_se_ri
logical function comp_se_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = abs (en1%ival - en2%rval) <= en1%tolerance
  else
     y = en1%ival == en2%rval
  end if
end function comp_se_ir
logical function comp_se_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
```

```
      y = abs (en1%rval - en2%rval) <= en1%tolerance
   else
      y = en1%rval == en2%rval
   end if
end function comp_se_rr
logical function comp_ns_ii (en1, en2) result (y)
   type(eval_node_t), intent(in) :: en1, en2
   if (associated (en1%tolerance)) then
      y = abs (en1%ival - en2%ival) > en1%tolerance
   else
      y = en1%ival /= en2%ival
   end if
end function comp_ns_ii
logical function comp_ns_ri (en1, en2) result (y)
   type(eval_node_t), intent(in) :: en1, en2
   if (associated (en1%tolerance)) then
      y = abs (en1%rval - en2%ival) > en1%tolerance
   else
      y = en1%rval /= en2%ival
   end if
end function comp_ns_ri
logical function comp_ns_ir (en1, en2) result (y)
   type(eval_node_t), intent(in) :: en1, en2
   if (associated (en1%tolerance)) then
      y = abs (en1%ival - en2%rval) > en1%tolerance
   else
      y = en1%ival /= en2%rval
   end if
end function comp_ns_ir
logical function comp_ns_rr (en1, en2) result (y)
   type(eval_node_t), intent(in) :: en1, en2
   if (associated (en1%tolerance)) then
      y = abs (en1%rval - en2%rval) > en1%tolerance
   else
      y = en1%rval /= en2%rval
   end if
end function comp_ns_rr

logical function comp_ls_ii (en1, en2) result (y)
   type(eval_node_t), intent(in) :: en1, en2
   if (associated (en1%tolerance)) then
      y = en1%ival <= en2%ival + en1%tolerance
   else
      y = en1%ival <= en2%ival
   end if
end function comp_ls_ii
logical function comp_ls_ri (en1, en2) result (y)
   type(eval_node_t), intent(in) :: en1, en2
   if (associated (en1%tolerance)) then
      y = en1%rval <= en2%ival + en1%tolerance
   else
      y = en1%rval <= en2%ival
   end if
end function comp_ls_ri
```

```
logical function comp_ls_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%ival <= en2%rval + en1%tolerance
  else
     y = en1%ival <= en2%rval
  end if
end function comp_ls_ir
logical function comp_ls_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%rval <= en2%rval + en1%tolerance
  else
     y = en1%rval <= en2%rval
  end if
end function comp_ls_rr

logical function comp_ll_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%ival < en2%ival - en1%tolerance
  else
     y = en1%ival < en2%ival
  end if
end function comp_ll_ii
logical function comp_ll_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%rval < en2%ival - en1%tolerance
  else
     y = en1%rval < en2%ival
  end if
end function comp_ll_ri
logical function comp_ll_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%ival < en2%rval - en1%tolerance
  else
     y = en1%ival < en2%rval
  end if
end function comp_ll_ir
logical function comp_ll_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%rval < en2%rval - en1%tolerance
  else
     y = en1%rval < en2%rval
  end if
end function comp_ll_rr

logical function comp_gs_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%ival >= en2%ival - en1%tolerance
```

```
  else
     y = en1%ival >= en2%ival
  end if
end function comp_gs_ii
logical function comp_gs_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%rval >= en2%ival - en1%tolerance
  else
     y = en1%rval >= en2%ival
  end if
end function comp_gs_ri
logical function comp_gs_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%ival >= en2%rval - en1%tolerance
  else
     y = en1%ival >= en2%rval
  end if
end function comp_gs_ir
logical function comp_gs_rr (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%rval >= en2%rval - en1%tolerance
  else
     y = en1%rval >= en2%rval
  end if
end function comp_gs_rr

logical function comp_gg_ii (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%ival > en2%ival + en1%tolerance
  else
     y = en1%ival > en2%ival
  end if
end function comp_gg_ii
logical function comp_gg_ri (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%rval > en2%ival + en1%tolerance
  else
     y = en1%rval > en2%ival
  end if
end function comp_gg_ri
logical function comp_gg_ir (en1, en2) result (y)
  type(eval_node_t), intent(in) :: en1, en2
  if (associated (en1%tolerance)) then
     y = en1%ival > en2%rval + en1%tolerance
  else
     y = en1%ival > en2%rval
  end if
end function comp_gg_ir
logical function comp_gg_rr (en1, en2) result (y)
```

```
    type(eval_node_t), intent(in) :: en1, en2
    if (associated (en1%tolerance)) then
       y = en1%rval > en2%rval + en1%tolerance
    else
       y = en1%rval > en2%rval
    end if
  end function comp_gg_rr
```

Unary logical functions

⟨*Eval trees: procedures*⟩+≡

```
  logical function not_l (en) result (y)
    type(eval_node_t), intent(in) :: en
    y = .not. en%lval
  end function not_l
```

Unary PDG-array functions

Make a PDG-array object from an integer.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine pdg_i (pdg_array, en)
    type(pdg_array_t), intent(out) :: pdg_array
    type(eval_node_t), intent(in) :: en
    pdg_array = en%ival
  end subroutine pdg_i
```

Binary PDG-array functions

Concatenate two PDG-array objects.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine concat_cc (pdg_array, en1, en2)
    type(pdg_array_t), intent(out) :: pdg_array
    type(eval_node_t), intent(in) :: en1, en2
    pdg_array = en1%aval // en2%aval
  end subroutine concat_cc
```

Unary particle-list functions

Combine all particles of the first argument. If `en0` is present, create a mask which is true only for those particles that pass the test.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine collect_p (subevt, en1, en0)
    type(subevt_t), intent(inout) :: subevt
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout), optional :: en0
    logical, dimension(:), allocatable :: mask1
    integer :: n, i
    n = subevt_get_length (en1%pval)
```

```
    allocate (mask1 (n))
    if (present (en0)) then
       do i = 1, n
          en0%index = i
          en0%prt1 = subevt_get_prt (en1%pval, i)
          call eval_node_evaluate (en0)
          mask1(i) = en0%lval
       end do
    else
       mask1 = .true.
    end if
    call subevt_collect (subevt, en1%pval, mask1)
  end subroutine collect_p
```

Cluster the particles of the first argument. If en0 is present, create a mask which is true only for those particles that pass the test.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine cluster_p (subevt, en1, en0)
    type(subevt_t), intent(inout) :: subevt
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout), optional :: en0
    logical, dimension(:), allocatable :: mask1
    integer :: n, i
    !!! Should not be initialized for every event
    type(jet_definition_t) :: jet_def
    logical :: keep_jets
    call jet_def%init (en1%jet_algorithm, en1%jet_r, en1%jet_p, en1%jet_ycut)
    n = subevt_get_length (en1%pval)
    allocate (mask1 (n))
    if (present (en0)) then
       do i = 1, n
          en0%index = i
          en0%prt1 = subevt_get_prt (en1%pval, i)
          call eval_node_evaluate (en0)
          mask1(i) = en0%lval
       end do
    else
       mask1 = .true.
    end if
    if (associated (en1%var_list)) then
       keep_jets = en1%var_list%get_lval (var_str("?keep_flavors_when_clustering"))
    else
       keep_jets = .false.
    end if
    call subevt_cluster (subevt, en1%pval, mask1, jet_def, keep_jets)
    call jet_def%final ()
  end subroutine cluster_p
```

Select all particles of the first argument. If en0 is present, create a mask which is true only for those particles that pass the test.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine select_p (subevt, en1, en0)
    type(subevt_t), intent(inout) :: subevt
```

```
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout), optional :: en0
    logical, dimension(:), allocatable :: mask1
    integer :: n, i
    n = subevt_get_length (en1%pval)
    allocate (mask1 (n))
    if (present (en0)) then
       do i = 1, subevt_get_length (en1%pval)
          en0%index = i
          en0%prt1 = subevt_get_prt (en1%pval, i)
          call eval_node_evaluate (en0)
          mask1(i) = en0%lval
       end do
    else
       mask1 = .true.
    end if
    call subevt_select (subevt, en1%pval, mask1)
  end subroutine select_p
```

Extract the particle with index given by `en0` from the argument list. Negative indices count from the end. If `en0` is absent, extract the first particle. The result is a list with a single entry, or no entries if the original list was empty or if the index is out of range.

This function has no counterpart with two arguments.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine extract_p (subevt, en1, en0)
    type(subevt_t), intent(inout) :: subevt
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout), optional :: en0
    integer :: index
    if (present (en0)) then
       call eval_node_evaluate (en0)
       select case (en0%result_type)
       case (V_INT);  index = en0%ival
       case default
          call eval_node_write (en0)
          call msg_fatal (" Index parameter of 'extract' must be integer.")
       end select
    else
       index = 1
    end if
    call subevt_extract (subevt, en1%pval, index)
  end subroutine extract_p
```

Sort the subevent according to the result of evaluating `en0`. If `en0` is absent, sort by default method (PDG code, particles before antiparticles).

⟨*Eval trees: procedures*⟩+≡

```
  subroutine sort_p (subevt, en1, en0)
    type(subevt_t), intent(inout) :: subevt
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout), optional :: en0
    integer, dimension(:), allocatable :: ival
    real(default), dimension(:), allocatable :: rval
```

```
  integer :: i, n
  n = subevt_get_length (en1%pval)
  if (present (en0)) then
     select case (en0%result_type)
     case (V_INT);  allocate (ival (n))
     case (V_REAL); allocate (rval (n))
     end select
     do i = 1, n
        en0%index = i
        en0%prt1 = subevt_get_prt (en1%pval, i)
        call eval_node_evaluate (en0)
        select case (en0%result_type)
        case (V_INT);  ival(i) = en0%ival
        case (V_REAL); rval(i) = en0%rval
        end select
     end do
     select case (en0%result_type)
     case (V_INT);  call subevt_sort (subevt, en1%pval, ival)
     case (V_REAL); call subevt_sort (subevt, en1%pval, rval)
     end select
  else
     call subevt_sort (subevt, en1%pval)
  end if
end subroutine sort_p
```

The following functions return a logical value. `all` evaluates to true if the condition `en0` is true for all elements of the subevent. `any` and `no` are analogous.

⟨*Eval trees: procedures*⟩+≡

```
function all_p (en1, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1
  type(eval_node_t), intent(inout) :: en0
  integer :: i, n
  n = subevt_get_length (en1%pval)
  lval = .true.
  do i = 1, n
     en0%index = i
     en0%prt1 = subevt_get_prt (en1%pval, i)
     call eval_node_evaluate (en0)
     lval = en0%lval
     if (.not. lval)  exit
  end do
end function all_p

function any_p (en1, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1
  type(eval_node_t), intent(inout) :: en0
  integer :: i, n
  n = subevt_get_length (en1%pval)
  lval = .false.
  do i = 1, n
     en0%index = i
```

```
      en0%prt1 = subevt_get_prt (en1%pval, i)
      call eval_node_evaluate (en0)
      lval = en0%lval
      if (lval)  exit
   end do
end function any_p

function no_p (en1, en0) result (lval)
   logical :: lval
   type(eval_node_t), intent(in) :: en1
   type(eval_node_t), intent(inout) :: en0
   integer :: i, n
   n = subevt_get_length (en1%pval)
   lval = .true.
   do i = 1, n
      en0%index = i
      en0%prt1 = subevt_get_prt (en1%pval, i)
      call eval_node_evaluate (en0)
      lval = .not. en0%lval
      if (lval)  exit
   end do
end function no_p
```

This is the interface to user-supplied observables. The node `en0` evaluates to a string that indicates the procedure name. We search for the procedure in the dynamic library and load it into the procedure pointer which is then called. `en1` is the subevent on which the external code operates. The external function returns a `c_int`, which we translate into a real value.

⟨*Eval trees: procedures*⟩+≡

```
function user_obs_int_p (en0, prt1) result (ival)
   integer :: ival
   type(eval_node_t), intent(inout) :: en0
   type(prt_t), intent(in) :: prt1
   type(string_t) :: name
   procedure(user_obs_int_unary), pointer :: user_obs
   call eval_node_evaluate (en0)
   if (en0%value_is_known) then
      select case (en0%result_type)
      case (V_STR);  name = en0%sval
      case default
         call msg_bug ("user_obs: procedure name must be a string")
         name = ""
      end select
      call c_f_procpointer (user_code_find_proc (name), user_obs)
      ival = user_obs (c_prt (prt1))
   else
      call eval_node_write_rec (en0)
      call msg_fatal ("User observable name is undefined")
   end if
end function user_obs_int_p

function user_obs_real_p (en0, prt1) result (rval)
   real(default) :: rval
```

```
    type(eval_node_t), intent(inout) :: en0
    type(prt_t), intent(in) :: prt1
    type(string_t) :: name
    procedure(user_obs_real_unary), pointer :: user_obs
    call eval_node_evaluate (en0)
    if (en0%value_is_known) then
       select case (en0%result_type)
       case (V_STR);  name = en0%sval
       case default
          call msg_bug ("user_obs: procedure name must be a string")
          name = ""
       end select
       call c_f_procpointer (user_code_find_proc (name), user_obs)
       rval = user_obs (c_prt (prt1))
    else
       call eval_node_write_rec (en0)
       call msg_fatal ("User observable name is undefined")
    end if
  end function user_obs_real_p
```

This is the interface to user-supplied cut code. The node `en0` evaluates to a string that indicates the procedure name.

⟨*Eval trees: procedures*⟩+≡

```
  function user_cut_p (en1, en0) result (lval)
    logical :: lval
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout) :: en0
    type(string_t) :: name
    procedure(user_cut_fun), pointer :: user_cut
    call eval_node_evaluate (en0)
    select case (en0%result_type)
    case (V_STR);  name = en0%sval
    case default
       call msg_bug ("user_cut: procedure name must be a string")
       name = ""
    end select
    call c_f_procpointer (user_code_find_proc (name), user_cut)
    lval = user_cut (c_prt (en1%pval), &
                     int (subevt_get_length (en1%pval), kind=c_int)) &
           /= 0
  end function user_cut_p
```

The following function returns an integer value, namely the number of particles for which the condition is true. If there is no condition, it returns simply the length of the subevent.

A function would be more natural. Making it a subroutine avoids another compiler bug (internal error in nagfor 5.2 (649)). (See the interface `unary_evi`.)

⟨*Eval trees: procedures*⟩+≡

```
  subroutine count_a (ival, en1, en0)
    integer, intent(out) :: ival
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout), optional :: en0
    integer :: i, n, count
```

```
    n = subevt_get_length (en1%pval)
    if (present (en0)) then
       count = 0
       do i = 1, n
          en0%index = i
          en0%prt1 = subevt_get_prt (en1%pval, i)
          call eval_node_evaluate (en0)
          if (en0%lval)  count = count + 1
       end do
       ival = count
    else
       ival = n
    end if
  end subroutine count_a
```

This evaluates a user-defined event-shape observable for the current subevent.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine user_event_shape_a (rval, en1, en0)
    real(default), intent(out) :: rval
    type(eval_node_t), intent(in) :: en1
    type(eval_node_t), intent(inout), optional :: en0
    type(string_t) :: name
    procedure(user_event_shape_fun), pointer :: user_event_shape
    if (.not. present (en0))  call msg_bug &
         ("user_event_shape called without procedure name")
    call eval_node_evaluate (en0)
    select case (en0%result_type)
    case (V_STR);  name = en0%sval
    case default
       call msg_bug ("user_event_shape: procedure name must be a string")
       name = ""
    end select
    call c_f_procpointer (user_code_find_proc (name), user_event_shape)
    rval = user_event_shape (c_prt (en1%pval), &
                             int (subevt_get_length (en1%pval), kind=c_int))
  end subroutine user_event_shape_a
```

Binary particle-list functions

This joins two subevents, stored in the evaluation nodes `en1` and `en2`. If `en0` is also present, it amounts to a logical test returning true or false for every pair of particles. A particle of the second list gets a mask entry only if it passes the test for all particles of the first list.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine join_pp (subevt, en1, en2, en0)
    type(subevt_t), intent(inout) :: subevt
    type(eval_node_t), intent(in) :: en1, en2
    type(eval_node_t), intent(inout), optional :: en0
    logical, dimension(:), allocatable :: mask2
    integer :: i, j, n1, n2
    n1 = subevt_get_length (en1%pval)
    n2 = subevt_get_length (en2%pval)
```

```
  allocate (mask2 (n2))
  mask2 = .true.
  if (present (en0)) then
     do i = 1, n1
        en0%index = i
        en0%prt1 = subevt_get_prt (en1%pval, i)
        do j = 1, n2
           en0%prt2 = subevt_get_prt (en2%pval, j)
           call eval_node_evaluate (en0)
           mask2(j) = mask2(j) .and. en0%lval
        end do
     end do
  end if
  call subevt_join (subevt, en1%pval, en2%pval, mask2)
end subroutine join_pp
```

Combine two subevents, i.e., make a list of composite particles built from all possible particle pairs from the two lists. If `en0` is present, create a mask which is true only for those pairs that pass the test.

⟨*Eval trees: procedures*⟩+≡

```
subroutine combine_pp (subevt, en1, en2, en0)
  type(subevt_t), intent(inout) :: subevt
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout), optional :: en0
  logical, dimension(:,:), allocatable :: mask12
  integer :: i, j, n1, n2
  n1 = subevt_get_length (en1%pval)
  n2 = subevt_get_length (en2%pval)
  if (present (en0)) then
     allocate (mask12 (n1, n2))
     do i = 1, n1
        en0%index = i
        en0%prt1 = subevt_get_prt (en1%pval, i)
        do j = 1, n2
           en0%prt2 = subevt_get_prt (en2%pval, j)
           call eval_node_evaluate (en0)
           mask12(i,j) = en0%lval
        end do
     end do
     call subevt_combine (subevt, en1%pval, en2%pval, mask12)
  else
     call subevt_combine (subevt, en1%pval, en2%pval)
  end if
end subroutine combine_pp
```

Combine all particles of the first argument. If `en0` is present, create a mask which is true only for those particles that pass the test w.r.t. all particles in the second argument. If `en0` is absent, the second argument is ignored.

⟨*Eval trees: procedures*⟩+≡

```
subroutine collect_pp (subevt, en1, en2, en0)
  type(subevt_t), intent(inout) :: subevt
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout), optional :: en0
```

```
    logical, dimension(:), allocatable :: mask1
    integer :: i, j, n1, n2
    n1 = subevt_get_length (en1%pval)
    n2 = subevt_get_length (en2%pval)
    allocate (mask1 (n1))
    mask1 = .true.
    if (present (en0)) then
       do i = 1, n1
          en0%index = i
          en0%prt1 = subevt_get_prt (en1%pval, i)
          do j = 1, n2
             en0%prt2 = subevt_get_prt (en2%pval, j)
             call eval_node_evaluate (en0)
             mask1(i) = mask1(i) .and. en0%lval
          end do
       end do
    end if
    call subevt_collect (subevt, en1%pval, mask1)
  end subroutine collect_pp
```

Select all particles of the first argument. If `en0` is present, create a mask which is true only for those particles that pass the test w.r.t. all particles in the second argument. If `en0` is absent, the second argument is ignored, and the first argument is transferred unchanged. (This case is not very useful, of course.)

⟨*Eval trees: procedures*⟩+≡

```
  subroutine select_pp (subevt, en1, en2, en0)
    type(subevt_t), intent(inout) :: subevt
    type(eval_node_t), intent(in) :: en1, en2
    type(eval_node_t), intent(inout), optional :: en0
    logical, dimension(:), allocatable :: mask1
    integer :: i, j, n1, n2
    n1 = subevt_get_length (en1%pval)
    n2 = subevt_get_length (en2%pval)
    allocate (mask1 (n1))
    mask1 = .true.
    if (present (en0)) then
       do i = 1, n1
          en0%index = i
          en0%prt1 = subevt_get_prt (en1%pval, i)
          do j = 1, n2
             en0%prt2 = subevt_get_prt (en2%pval, j)
             call eval_node_evaluate (en0)
             mask1(i) = mask1(i) .and. en0%lval
          end do
       end do
    end if
    call subevt_select (subevt, en1%pval, mask1)
  end subroutine select_pp
```

Sort the first subevent according to the result of evaluating `en0`. From the second subevent, only the first element is taken as reference. If `en0` is absent, we sort by default method (PDG code, particles before antiparticles).

⟨*Eval trees: procedures*⟩+≡

```
subroutine sort_pp (subevt, en1, en2, en0)
  type(subevt_t), intent(inout) :: subevt
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout), optional :: en0
  integer, dimension(:), allocatable :: ival
  real(default), dimension(:), allocatable :: rval
  integer :: i, n1
  n1 = subevt_get_length (en1%pval)
  if (present (en0)) then
     select case (en0%result_type)
     case (V_INT);  allocate (ival (n1))
     case (V_REAL); allocate (rval (n1))
     end select
     do i = 1, n1
        en0%index = i
        en0%prt1 = subevt_get_prt (en1%pval, i)
        en0%prt2 = subevt_get_prt (en2%pval, 1)
        call eval_node_evaluate (en0)
        select case (en0%result_type)
        case (V_INT);  ival(i) = en0%ival
        case (V_REAL); rval(i) = en0%rval
        end select
     end do
     select case (en0%result_type)
     case (V_INT);  call subevt_sort (subevt, en1%pval, ival)
     case (V_REAL); call subevt_sort (subevt, en1%pval, rval)
     end select
  else
     call subevt_sort (subevt, en1%pval)
  end if
end subroutine sort_pp
```

The following functions return a logical value. `all` evaluates to true if the condition `en0` is true for all valid element pairs of both subevents. Invalid pairs (with common `src` entry) are ignored.

`any` and `no` are analogous.

⟨*Eval trees: procedures*⟩+≡

```
function all_pp (en1, en2, en0) result (lval)
  logical :: lval
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout) :: en0
  integer :: i, j, n1, n2
  n1 = subevt_get_length (en1%pval)
  n2 = subevt_get_length (en2%pval)
  lval = .true.
  LOOP1: do i = 1, n1
     en0%index = i
     en0%prt1 = subevt_get_prt (en1%pval, i)
     do j = 1, n2
        en0%prt2 = subevt_get_prt (en2%pval, j)
        if (are_disjoint (en0%prt1, en0%prt2)) then
           call eval_node_evaluate (en0)
           lval = en0%lval
```

```
            if (.not. lval)  exit LOOP1
         end if
      end do
   end do LOOP1
end function all_pp

function any_pp (en1, en2, en0) result (lval)
   logical :: lval
   type(eval_node_t), intent(in) :: en1, en2
   type(eval_node_t), intent(inout) :: en0
   integer :: i, j, n1, n2
   n1 = subevt_get_length (en1%pval)
   n2 = subevt_get_length (en2%pval)
   lval = .false.
   LOOP1: do i = 1, n1
      en0%index = i
      en0%prt1 = subevt_get_prt (en1%pval, i)
      do j = 1, n2
         en0%prt2 = subevt_get_prt (en2%pval, j)
         if (are_disjoint (en0%prt1, en0%prt2)) then
            call eval_node_evaluate (en0)
            lval = en0%lval
            if (lval)  exit LOOP1
         end if
      end do
   end do LOOP1
end function any_pp

function no_pp (en1, en2, en0) result (lval)
   logical :: lval
   type(eval_node_t), intent(in) :: en1, en2
   type(eval_node_t), intent(inout) :: en0
   integer :: i, j, n1, n2
   n1 = subevt_get_length (en1%pval)
   n2 = subevt_get_length (en2%pval)
   lval = .true.
   LOOP1: do i = 1, n1
      en0%index = i
      en0%prt1 = subevt_get_prt (en1%pval, i)
      do j = 1, n2
         en0%prt2 = subevt_get_prt (en2%pval, j)
         if (are_disjoint (en0%prt1, en0%prt2)) then
            call eval_node_evaluate (en0)
            lval = .not. en0%lval
            if (lval)  exit LOOP1
         end if
      end do
   end do LOOP1
end function no_pp
```

This function evaluates an observable for a pair of particles. From the two particle lists, we take the first pair without `src` overlap. If there is no valid pair, we revert the status of the value to unknown.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_pp (en1, en2, en0, rval, is_known)
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout) :: en0
  real(default), intent(out) :: rval
  logical, intent(out) :: is_known
  integer :: i, j, n1, n2
  n1 = subevt_get_length (en1%pval)
  n2 = subevt_get_length (en2%pval)
  rval = 0
  is_known = .false.
  LOOP1: do i = 1, n1
     en0%index = i
     en0%prt1 = subevt_get_prt (en1%pval, i)
     do j = 1, n2
        en0%prt2 = subevt_get_prt (en2%pval, j)
        if (are_disjoint (en0%prt1, en0%prt2)) then
           call eval_node_evaluate (en0)
           rval = en0%rval
           is_known = .true.
           exit LOOP1
        end if
     end do
  end do LOOP1
end subroutine eval_pp
```

This is the interface to user-supplied observables. The node `en0` evaluates to a string that indicates the procedure name. We search for the procedure in the dynamic library and load it into the procedure pointer which is then called. `en1` is the subevent on which the external code operates. The external function returns a `c_int`, which we translate into a real value.

⟨*Eval trees: procedures*⟩+≡

```
function user_obs_int_pp (en0, prt1, prt2) result (ival)
  integer :: ival
  type(eval_node_t), intent(inout) :: en0
  type(prt_t), intent(in) :: prt1, prt2
  type(string_t) :: name
  procedure(user_obs_int_binary), pointer :: user_obs
  call eval_node_evaluate (en0)
  if (en0%value_is_known) then
     select case (en0%result_type)
     case (V_STR);  name = en0%sval
     case default
        call msg_bug ("user_obs: procedure name must be a string")
        name = ""
     end select
     call c_f_procpointer (user_code_find_proc (name), user_obs)
     ival = user_obs (c_prt (prt1), c_prt (prt2))
  else
     call eval_node_write_rec (en0)
     call msg_fatal ("User observable name is undefined")
  end if
end function user_obs_int_pp
```

```
function user_obs_real_pp (en0, prt1, prt2) result (rval)
  real(default) :: rval
  type(eval_node_t), intent(inout) :: en0
  type(prt_t), intent(in) :: prt1, prt2
  type(string_t) :: name
  procedure(user_obs_real_binary), pointer :: user_obs
  call eval_node_evaluate (en0)
  if (en0%value_is_known) then
     select case (en0%result_type)
     case (V_STR);  name = en0%sval
     case default
        call msg_bug ("user_obs: procedure name must be a string")
        name = ""
     end select
     call c_f_procpointer (user_code_find_proc (name), user_obs)
     rval = user_obs (c_prt (prt1), c_prt (prt2))
  else
     call eval_node_write_rec (en0)
     call msg_fatal ("User observable name is undefined")
  end if
end function user_obs_real_pp
```

The following function returns an integer value, namely the number of valid particle-pairs from both lists for which the condition is true. Invalid pairs (with common `src` entry) are ignored. If there is no condition, it returns the number of valid particle pairs.

A function would be more natural. Making it a subroutine avoids another compiler bug (internal error in nagfor 5.2 (649)). (See the interface `binary_num`.)

⟨*Eval trees: procedures*⟩+≡

```
subroutine count_pp (ival, en1, en2, en0)
  integer, intent(out) :: ival
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout), optional :: en0
  integer :: i, j, n1, n2, count
  n1 = subevt_get_length (en1%pval)
  n2 = subevt_get_length (en2%pval)
  if (present (en0)) then
     count = 0
     do i = 1, n1
        en0%index = i
        en0%prt1 = subevt_get_prt (en1%pval, i)
        do j = 1, n2
           en0%prt2 = subevt_get_prt (en2%pval, j)
           if (are_disjoint (en0%prt1, en0%prt2)) then
              call eval_node_evaluate (en0)
              if (en0%lval)  count = count + 1
           end if
        end do
     end do
  else
     count = 0
     do i = 1, n1
```

```
         do j = 1, n2
            if (are_disjoint (subevt_get_prt (en1%pval, i), &
                              subevt_get_prt (en2%pval, j))) then
               count = count + 1
            end if
         end do
      end do
   end if
   ival = count
end subroutine count_pp
```

This function makes up a subevent from the second argument which consists only of particles which match the PDG code array (first argument).

⟨*Eval trees: procedures*⟩+≡

```
subroutine select_pdg_ca (subevt, en1, en2, en0)
  type(subevt_t), intent(inout) :: subevt
  type(eval_node_t), intent(in) :: en1, en2
  type(eval_node_t), intent(inout), optional :: en0
  if (present (en0)) then
     call subevt_select_pdg_code (subevt, en1%aval, en2%pval, en0%ival)
  else
     call subevt_select_pdg_code (subevt, en1%aval, en2%pval)
  end if
end subroutine select_pdg_ca
```

Binary string functions

Currently, the only string operation is concatenation.

⟨*Eval trees: procedures*⟩+≡

```
subroutine concat_ss (string, en1, en2)
  type(string_t), intent(out) :: string
  type(eval_node_t), intent(in) :: en1, en2
  string = en1%sval // en2%sval
end subroutine concat_ss
```

28.3.4 Compiling the parse tree

The evaluation tree is built recursively by following a parse tree. Evaluate an expression. The requested type is given as an optional argument; default is numeric (integer or real).

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_genexpr &
     (en, pn, var_list, result_type)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  integer, intent(in), optional :: result_type
  if (debug_active (D_MODEL_F)) then
     print *, "read genexpr";  call parse_node_write (pn)
  end if
```

```
    if (present (result_type)) then
       select case (result_type)
       case (V_INT, V_REAL, V_CMPLX)
          call eval_node_compile_expr  (en, pn, var_list)
       case (V_LOG)
          call eval_node_compile_lexpr (en, pn, var_list)
       case (V_SEV)
          call eval_node_compile_pexpr (en, pn, var_list)
       case (V_PDG)
          call eval_node_compile_cexpr (en, pn, var_list)
       case (V_STR)
          call eval_node_compile_sexpr (en, pn, var_list)
       end select
    else
       call eval_node_compile_expr  (en, pn, var_list)
    end if
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done genexpr"
    end if
  end subroutine eval_node_compile_genexpr
```

Numeric expressions

This procedure compiles a numerical expression. This is a single term or a sum or difference of terms. We have to account for all combinations of integer and real arguments. If both are constant, we immediately do the calculation and allocate a constant node.

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_expr (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_term, pn_addition, pn_op, pn_arg
    type(eval_node_t), pointer :: en1, en2
    type(string_t) :: key
    integer :: t1, t2, t
    if (debug_active (D_MODEL_F)) then
       print *, "read expr";  call parse_node_write (pn)
    end if
    pn_term => parse_node_get_sub_ptr (pn)
    select case (char (parse_node_get_rule_key (pn_term)))
    case ("term")
       call eval_node_compile_term (en, pn_term, var_list)
       pn_addition => parse_node_get_next_ptr (pn_term, tag="addition")
    case ("addition")
       en => null ()
       pn_addition => pn_term
    case default
       call parse_node_mismatch ("term|addition", pn)
    end select
    do while (associated (pn_addition))
```

```
pn_op => parse_node_get_sub_ptr (pn_addition)
pn_arg => parse_node_get_next_ptr (pn_op, tag="term")
call eval_node_compile_term (en2, pn_arg, var_list)
t2 = en2%result_type
if (associated (en)) then
   en1 => en
   t1 = en1%result_type
else
   allocate (en1)
   select case (t2)
   case (V_INT);  call eval_node_init_int  (en1, 0)
   case (V_REAL); call eval_node_init_real (en1, 0._default)
   case (V_CMPLX); call eval_node_init_cmplx (en1, cmplx &
                  (0._default, 0._default, kind=default))
   end select
   t1 = t2
end if
t = numeric_result_type (t1, t2)
allocate (en)
key = parse_node_get_key (pn_op)
if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
   select case (char (key))
   case ("+")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_int  (en, add_ii (en1, en2))
         case (V_REAL); call eval_node_init_real (en, add_ir (en1, en2))
         case (V_CMPLX); call eval_node_init_cmplx (en, add_ic (en1, en2))
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_real (en, add_ri (en1, en2))
         case (V_REAL); call eval_node_init_real (en, add_rr (en1, en2))
         case (V_CMPLX); call eval_node_init_cmplx (en, add_rc (en1, en2))
         end select
       case (V_CMPLX)
         select case (t2)
         case (V_INT);  call eval_node_init_cmplx (en, add_ci (en1, en2))
         case (V_REAL); call eval_node_init_cmplx (en, add_cr (en1, en2))
         case (V_CMPLX); call eval_node_init_cmplx (en, add_cc (en1, en2))
         end select
      end select
   case ("-")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_int  (en, sub_ii (en1, en2))
         case (V_REAL); call eval_node_init_real (en, sub_ir (en1, en2))
         case (V_CMPLX); call eval_node_init_cmplx (en, sub_ic (en1, en2))
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_real (en, sub_ri (en1, en2))
```

```
         case (V_REAL); call eval_node_init_real (en, sub_rr (en1, en2))
         case (V_CMPLX); call eval_node_init_cmplx (en, sub_rc (en1, en2))
         end select
      case (V_CMPLX)
         select case (t2)
         case (V_INT);  call eval_node_init_cmplx (en, sub_ci (en1, en2))
         case (V_REAL); call eval_node_init_cmplx (en, sub_cr (en1, en2))
         case (V_CMPLX); call eval_node_init_cmplx (en, sub_cc (en1, en2))
         end select
      end select
   end select
   call eval_node_final_rec (en1)
   call eval_node_final_rec (en2)
   deallocate (en1, en2)
else
   call eval_node_init_branch (en, key, t, en1, en2)
   select case (char (key))
   case ("+")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);   call eval_node_set_op2_int  (en, add_ii)
         case (V_REAL);  call eval_node_set_op2_real (en, add_ir)
         case (V_CMPLX);  call eval_node_set_op2_cmplx (en, add_ic)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);   call eval_node_set_op2_real (en, add_ri)
         case (V_REAL);  call eval_node_set_op2_real (en, add_rr)
         case (V_CMPLX);  call eval_node_set_op2_cmplx (en, add_rc)
         end select
      case (V_CMPLX)
         select case (t2)
         case (V_INT);   call eval_node_set_op2_cmplx (en, add_ci)
         case (V_REAL);  call eval_node_set_op2_cmplx (en, add_cr)
         case (V_CMPLX);  call eval_node_set_op2_cmplx (en, add_cc)
         end select
      end select
   case ("-")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);   call eval_node_set_op2_int  (en, sub_ii)
         case (V_REAL);  call eval_node_set_op2_real (en, sub_ir)
         case (V_CMPLX);  call eval_node_set_op2_cmplx (en, sub_ic)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);   call eval_node_set_op2_real (en, sub_ri)
         case (V_REAL);  call eval_node_set_op2_real (en, sub_rr)
         case (V_CMPLX);  call eval_node_set_op2_cmplx (en, sub_rc)
         end select
      case (V_CMPLX)
         select case (t2)
```

```
            case (V_INT);   call eval_node_set_op2_cmplx (en, sub_ci)
            case (V_REAL);  call eval_node_set_op2_cmplx (en, sub_cr)
            case (V_CMPLX);  call eval_node_set_op2_cmplx (en, sub_cc)
            end select
         end select
      end select
   end if
   pn_addition => parse_node_get_next_ptr (pn_addition)
end do
if (debug_active (D_MODEL_F)) then
   call eval_node_write (en)
   print *, "done expr"
end if
end subroutine eval_node_compile_expr
```

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_term (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_factor, pn_multiplication, pn_op, pn_arg
  type(eval_node_t), pointer :: en1, en2
  type(string_t) :: key
  integer :: t1, t2, t
  if (debug_active (D_MODEL_F)) then
     print *, "read term";  call parse_node_write (pn)
  end if
  pn_factor => parse_node_get_sub_ptr (pn, tag="factor")
  call eval_node_compile_factor (en, pn_factor, var_list)
  pn_multiplication => &
       parse_node_get_next_ptr (pn_factor, tag="multiplication")
  do while (associated (pn_multiplication))
     pn_op => parse_node_get_sub_ptr (pn_multiplication)
     pn_arg => parse_node_get_next_ptr (pn_op, tag="factor")
     en1 => en
     call eval_node_compile_factor (en2, pn_arg, var_list)
     t1 = en1%result_type
     t2 = en2%result_type
     t = numeric_result_type (t1, t2)
     allocate (en)
     key = parse_node_get_key (pn_op)
     if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
        select case (char (key))
        case ("*")
           select case (t1)
           case (V_INT)
              select case (t2)
              case (V_INT);  call eval_node_init_int  (en, mul_ii (en1, en2))
              case (V_REAL); call eval_node_init_real (en, mul_ir (en1, en2))
              case (V_CMPLX); call eval_node_init_cmplx (en, mul_ic (en1, en2))
              end select
           case (V_REAL)
              select case (t2)
              case (V_INT);  call eval_node_init_real (en, mul_ri (en1, en2))
```

```
            case (V_REAL); call eval_node_init_real (en, mul_rr (en1, en2))
            case (V_CMPLX); call eval_node_init_cmplx (en, mul_rc (en1, en2))
            end select
         case (V_CMPLX)
            select case (t2)
            case (V_INT);  call eval_node_init_cmplx (en, mul_ci (en1, en2))
            case (V_REAL); call eval_node_init_cmplx (en, mul_cr (en1, en2))
            case (V_CMPLX); call eval_node_init_cmplx (en, mul_cc (en1, en2))
            end select
         end select
      case ("/")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_init_int  (en, div_ii (en1, en2))
            case (V_REAL); call eval_node_init_real (en, div_ir (en1, en2))
            case (V_CMPLX); call eval_node_init_real (en, div_ir (en1, en2))
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_init_real (en, div_ri (en1, en2))
            case (V_REAL); call eval_node_init_real (en, div_rr (en1, en2))
            case (V_CMPLX); call eval_node_init_cmplx (en, div_rc (en1, en2))
            end select
         case (V_CMPLX)
            select case (t2)
            case (V_INT);  call eval_node_init_cmplx (en, div_ci (en1, en2))
            case (V_REAL); call eval_node_init_cmplx (en, div_cr (en1, en2))
            case (V_CMPLX); call eval_node_init_cmplx (en, div_cc (en1, en2))
            end select
         end select
      end select
      call eval_node_final_rec (en1)
      call eval_node_final_rec (en2)
      deallocate (en1, en2)
   else
      call eval_node_init_branch (en, key, t, en1, en2)
      select case (char (key))
      case ("*")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);   call eval_node_set_op2_int  (en, mul_ii)
            case (V_REAL);  call eval_node_set_op2_real (en, mul_ir)
            case (V_CMPLX);  call eval_node_set_op2_cmplx (en, mul_ic)
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);   call eval_node_set_op2_real (en, mul_ri)
            case (V_REAL);  call eval_node_set_op2_real (en, mul_rr)
            case (V_CMPLX);  call eval_node_set_op2_cmplx (en, mul_rc)
            end select
         case (V_CMPLX)
            select case (t2)
```

```
            case (V_INT);   call eval_node_set_op2_cmplx (en, mul_ci)
            case (V_REAL);  call eval_node_set_op2_cmplx (en, mul_cr)
            case (V_CMPLX);  call eval_node_set_op2_cmplx (en, mul_cc)
            end select
         end select
      case ("/")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);   call eval_node_set_op2_int  (en, div_ii)
            case (V_REAL);  call eval_node_set_op2_real (en, div_ir)
            case (V_CMPLX);  call eval_node_set_op2_cmplx (en, div_ic)
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);   call eval_node_set_op2_real (en, div_ri)
            case (V_REAL);  call eval_node_set_op2_real (en, div_rr)
            case (V_CMPLX);  call eval_node_set_op2_cmplx (en, div_rc)
            end select
         case (V_CMPLX)
            select case (t2)
            case (V_INT);   call eval_node_set_op2_cmplx (en, div_ci)
            case (V_REAL);  call eval_node_set_op2_cmplx (en, div_cr)
            case (V_CMPLX);  call eval_node_set_op2_cmplx (en, div_cc)
            end select
         end select
      end select
   end if
   pn_multiplication => parse_node_get_next_ptr (pn_multiplication)
end do
if (debug_active (D_MODEL_F)) then
   call eval_node_write (en)
   print *, "done term"
end if
end subroutine eval_node_compile_term
```

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_factor (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_value, pn_exponentiation, pn_op, pn_arg
  type(eval_node_t), pointer :: en1, en2
  type(string_t) :: key
  integer :: t1, t2, t
  if (debug_active (D_MODEL_F)) then
     print *, "read factor";  call parse_node_write (pn)
  end if
  pn_value => parse_node_get_sub_ptr (pn)
  call eval_node_compile_signed_value (en, pn_value, var_list)
  pn_exponentiation => &
       parse_node_get_next_ptr (pn_value, tag="exponentiation")
  if (associated (pn_exponentiation)) then
     pn_op => parse_node_get_sub_ptr (pn_exponentiation)
```

```
pn_arg => parse_node_get_next_ptr (pn_op)
en1 => en
call eval_node_compile_signed_value (en2, pn_arg, var_list)
t1 = en1%result_type
t2 = en2%result_type
t = numeric_result_type (t1, t2)
allocate (en)
key = parse_node_get_key (pn_op)
if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);   call eval_node_init_int   (en, pow_ii (en1, en2))
      case (V_REAL);  call eval_node_init_real  (en, pow_ir (en1, en2))
      case (V_CMPLX); call eval_node_init_cmplx (en, pow_ic (en1, en2))
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);   call eval_node_init_real  (en, pow_ri (en1, en2))
      case (V_REAL);  call eval_node_init_real  (en, pow_rr (en1, en2))
      case (V_CMPLX); call eval_node_init_cmplx (en, pow_rc (en1, en2))
      end select
   case (V_CMPLX)
      select case (t2)
      case (V_INT);   call eval_node_init_cmplx (en, pow_ci (en1, en2))
      case (V_REAL);  call eval_node_init_cmplx (en, pow_cr (en1, en2))
      case (V_CMPLX); call eval_node_init_cmplx (en, pow_cc (en1, en2))
      end select
   end select
   call eval_node_final_rec (en1)
   call eval_node_final_rec (en2)
   deallocate (en1, en2)
else
   call eval_node_init_branch (en, key, t, en1, en2)
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);   call eval_node_set_op2_int  (en, pow_ii)
      case (V_REAL,V_CMPLX);  call eval_type_error (pn, "exponentiation", t1)
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);   call eval_node_set_op2_real (en, pow_ri)
      case (V_REAL);  call eval_node_set_op2_real (en, pow_rr)
      case (V_CMPLX);  call eval_type_error (pn, "exponentiation", t1)
      end select
   case (V_CMPLX)
      select case (t2)
      case (V_INT);   call eval_node_set_op2_cmplx (en, pow_ci)
      case (V_REAL);  call eval_node_set_op2_cmplx (en, pow_cr)
      case (V_CMPLX);  call eval_node_set_op2_cmplx (en, pow_cc)
      end select
   end select
end if
```

```
    end if
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done factor"
    end if
  end subroutine eval_node_compile_factor
```

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_signed_value (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_arg
    type(eval_node_t), pointer :: en1
    integer :: t
    if (debug_active (D_MODEL_F)) then
       print *, "read signed value";  call parse_node_write (pn)
    end if
    select case (char (parse_node_get_rule_key (pn)))
    case ("signed_value")
       pn_arg => parse_node_get_sub_ptr (pn, 2)
       call eval_node_compile_value (en1, pn_arg, var_list)
       t = en1%result_type
       allocate (en)
       if (en1%type == EN_CONSTANT) then
          select case (t)
          case (V_INT);  call eval_node_init_int  (en, neg_i (en1))
          case (V_REAL); call eval_node_init_real (en, neg_r (en1))
          case (V_CMPLX); call eval_node_init_cmplx (en, neg_c (en1))
          end select
          call eval_node_final_rec (en1)
          deallocate (en1)
       else
          call eval_node_init_branch (en, var_str ("-"), t, en1)
          select case (t)
          case (V_INT);  call eval_node_set_op1_int  (en, neg_i)
          case (V_REAL); call eval_node_set_op1_real (en, neg_r)
          case (V_CMPLX); call eval_node_set_op1_cmplx (en, neg_c)
          end select
       end if
    case default
       call eval_node_compile_value (en, pn, var_list)
    end select
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done signed value"
    end if
  end subroutine eval_node_compile_signed_value
```

Integer, real and complex values have an optional unit. The unit is extracted and applied immediately. An integer with unit evaluates to a real constant.

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_value (en, pn, var_list)
```

```
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    if (debug_active (D_MODEL_F)) then
       print *, "read value";  call parse_node_write (pn)
    end if
    select case (char (parse_node_get_rule_key (pn)))
    case ("integer_value", "real_value", "complex_value")
       call eval_node_compile_numeric_value (en, pn)
    case ("pi")
       call eval_node_compile_constant (en, pn)
    case ("I")
       call eval_node_compile_constant (en, pn)
    case ("variable")
       call eval_node_compile_variable (en, pn, var_list)
    case ("result")
       call eval_node_compile_result (en, pn, var_list)
    case ("user_observable")
       call eval_node_compile_user_observable (en, pn, var_list)
    case ("expr")
       call eval_node_compile_expr (en, pn, var_list)
    case ("block_expr")
       call eval_node_compile_block_expr (en, pn, var_list)
    case ("conditional_expr")
       call eval_node_compile_conditional (en, pn, var_list)
    case ("unary_function")
       call eval_node_compile_unary_function (en, pn, var_list)
    case ("binary_function")
       call eval_node_compile_binary_function (en, pn, var_list)
    case ("eval_fun")
       call eval_node_compile_eval_function (en, pn, var_list)
    case ("count_fun", "user_event_fun")
       call eval_node_compile_numeric_function (en, pn, var_list)
    case default
       call parse_node_mismatch &
            ("integer|real|complex|constant|variable|" // &
             "expr|block_expr|conditional_expr|" // &
             "unary_function|binary_function|numeric_pexpr", pn)
    end select
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done value"
    end if
  end subroutine eval_node_compile_value
```

Real, complex and integer values are numeric literals with an optional unit attached. In case of an integer, the unit actually makes it a real value in disguise. The signed version of real values is not possible in generic expressions; it is a special case for numeric constants in model files (see below). We do not introduce signed versions of complex values.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_node_compile_numeric_value (en, pn)
    type(eval_node_t), pointer :: en
```

```
    type(parse_node_t), intent(in), target :: pn
    type(parse_node_t), pointer :: pn_val, pn_unit
    allocate (en)
    pn_val => parse_node_get_sub_ptr (pn)
    pn_unit => parse_node_get_next_ptr (pn_val)
    select case (char (parse_node_get_rule_key (pn)))
    case ("integer_value")
       if (associated (pn_unit)) then
          call eval_node_init_real (en, &
               parse_node_get_integer (pn_val) * parse_node_get_unit (pn_unit))
       else
          call eval_node_init_int (en, parse_node_get_integer (pn_val))
       end if
    case ("real_value")
       if (associated (pn_unit)) then
          call eval_node_init_real (en, &
               parse_node_get_real (pn_val) * parse_node_get_unit (pn_unit))
       else
          call eval_node_init_real (en, parse_node_get_real (pn_val))
       end if
    case ("complex_value")
       if (associated (pn_unit)) then
          call eval_node_init_cmplx (en, &
               parse_node_get_cmplx (pn_val) * parse_node_get_unit (pn_unit))
       else
          call eval_node_init_cmplx (en, parse_node_get_cmplx (pn_val))
       end if
    case ("neg_real_value")
       pn_val => parse_node_get_sub_ptr (parse_node_get_sub_ptr (pn, 2))
       pn_unit => parse_node_get_next_ptr (pn_val)
       if (associated (pn_unit)) then
          call eval_node_init_real (en, &
               - parse_node_get_real (pn_val) * parse_node_get_unit (pn_unit))
       else
          call eval_node_init_real (en, - parse_node_get_real (pn_val))
       end if
    case ("pos_real_value")
       pn_val => parse_node_get_sub_ptr (parse_node_get_sub_ptr (pn, 2))
       pn_unit => parse_node_get_next_ptr (pn_val)
       if (associated (pn_unit)) then
          call eval_node_init_real (en, &
               parse_node_get_real (pn_val) * parse_node_get_unit (pn_unit))
       else
          call eval_node_init_real (en, parse_node_get_real (pn_val))
       end if
    case default
       call parse_node_mismatch &
       ("integer_value|real_value|complex_value|neg_real_value|pos_real_value", pn)
    end select
  end subroutine eval_node_compile_numeric_value
```

These are the units, predefined and hardcoded. The default energy unit is GeV, the default angular unit is radians. We include units for observables of dimension energy squared. Luminosities are normalized in inverse femtobarns.

⟨*Eval trees: procedures*⟩+≡

```
function parse_node_get_unit (pn) result (factor)
  real(default) :: factor
  real(default) :: unit
  type(parse_node_t), intent(in) :: pn
  type(parse_node_t), pointer :: pn_unit, pn_unit_power
  type(parse_node_t), pointer :: pn_frac, pn_num, pn_int, pn_div, pn_den
  integer :: num, den
  pn_unit => parse_node_get_sub_ptr (pn)
  select case (char (parse_node_get_key (pn_unit)))
  case ("TeV");  unit = 1.e3_default
  case ("GeV");  unit = 1
  case ("MeV");  unit = 1.e-3_default
  case ("keV");  unit = 1.e-6_default
  case ("eV");   unit = 1.e-9_default
  case ("meV");   unit = 1.e-12_default
  case ("nbarn");  unit = 1.e6_default
  case ("pbarn");  unit = 1.e3_default
  case ("fbarn");  unit = 1
  case ("abarn");  unit = 1.e-3_default
  case ("rad");     unit = 1
  case ("mrad");    unit = 1.e-3_default
  case ("degree");  unit = degree
  case ("%");  unit = 1.e-2_default
  case default
     call msg_bug (" Unit '" // &
          char (parse_node_get_key (pn)) // "' is undefined.")
  end select
  pn_unit_power => parse_node_get_next_ptr (pn_unit)
  if (associated (pn_unit_power)) then
     pn_frac => parse_node_get_sub_ptr (pn_unit_power, 2)
     pn_num => parse_node_get_sub_ptr (pn_frac)
     select case (char (parse_node_get_rule_key (pn_num)))
     case ("neg_int")
        pn_int => parse_node_get_sub_ptr (pn_num, 2)
        num = - parse_node_get_integer (pn_int)
     case ("pos_int")
        pn_int => parse_node_get_sub_ptr (pn_num, 2)
        num = parse_node_get_integer (pn_int)
     case ("integer_literal")
        num = parse_node_get_integer (pn_num)
     case default
        call parse_node_mismatch ("neg_int|pos_int|integer_literal", pn_num)
     end select
     pn_div => parse_node_get_next_ptr (pn_num)
     if (associated (pn_div)) then
        pn_den => parse_node_get_sub_ptr (pn_div, 2)
        den = parse_node_get_integer (pn_den)
     else
        den = 1
     end if
  else
     num = 1
     den = 1
```

```
    end if
    factor = unit ** (real (num, default) / den)
  end function parse_node_get_unit
```

There are only two predefined constants, but more can be added easily.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_node_compile_constant (en, pn)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    if (debug_active (D_MODEL_F)) then
       print *, "read constant";  call parse_node_write (pn)
    end if
    allocate (en)
    select case (char (parse_node_get_key (pn)))
    case ("pi");     call eval_node_init_real (en, pi)
    case ("I");      call eval_node_init_cmplx (en, imago)
    case default
       call parse_node_mismatch ("pi or I", pn)
    end select
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done constant"
    end if
  end subroutine eval_node_compile_constant
```

Compile a variable, with or without a specified type. Take the list of variables, look for the name and make a node with a pointer to the value. If no type is provided, the variable is numeric, and the stored value determines whether it is real or integer.

We explicitly demand that the variable is defined, so we do not accidentally point to variables that are declared only later in the script but have come into existence in a previous compilation pass.

Variables may actually be anonymous, these are expressions in disguise. In that case, the expression replaces the variable name in the parse tree, and we allocate an ordinary expression node in the eval tree.

Variables of type `V_PDG` (pdg-code array) are not treated here. They are handled by `eval_node_compile_cvariable`.

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_variable (en, pn, var_list, var_type)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in), target :: pn
    type(var_list_t), intent(in), target :: var_list
    integer, intent(in), optional :: var_type
    type(parse_node_t), pointer :: pn_name
    type(string_t) :: var_name
    logical, target, save :: no_lval
    real(default), target, save :: no_rval
    type(subevt_t), target, save :: no_pval
    type(string_t), target, save :: no_sval
    logical, target, save :: unknown = .false.
    integer :: type
    logical :: defined
```

```
logical, pointer :: known
logical, pointer :: lptr
integer, pointer :: iptr
real(default), pointer :: rptr
complex(default), pointer :: cptr
type(subevt_t), pointer :: pptr
type(string_t), pointer :: sptr
procedure(obs_unary_int), pointer :: obs1_iptr
procedure(obs_unary_real), pointer :: obs1_rptr
procedure(obs_binary_int), pointer :: obs2_iptr
procedure(obs_binary_real), pointer :: obs2_rptr
type(prt_t), pointer :: p1, p2
if (debug_active (D_MODEL_F)) then
   print *, "read variable";  call parse_node_write (pn)
end if
if (present (var_type)) then
   select case (var_type)
   case (V_REAL, V_OBS1_REAL, V_OBS2_REAL, V_INT, V_OBS1_INT, &
            V_OBS2_INT, V_CMPLX)
      pn_name => pn
   case default
      pn_name => parse_node_get_sub_ptr (pn, 2)
   end select
else
   pn_name => pn
end if
select case (char (parse_node_get_rule_key (pn_name)))
case ("expr")
   call eval_node_compile_expr (en, pn_name, var_list)
case ("lexpr")
   call eval_node_compile_lexpr (en, pn_name, var_list)
case ("sexpr")
   call eval_node_compile_sexpr (en, pn_name, var_list)
case ("pexpr")
   call eval_node_compile_pexpr (en, pn_name, var_list)
case ("variable")
   var_name = parse_node_get_string (pn_name)
   if (present (var_type)) then
      select case (var_type)
      case (V_LOG);  var_name = "?" // var_name
      case (V_SEV);  var_name = "@" // var_name
      case (V_STR);  var_name = "$" // var_name   ! $ sign
      end select
   end if
   call var_list%get_var_properties &
        (var_name, req_type=var_type, type=type, is_defined=defined)
   allocate (en)
   if (defined) then
      select case (type)
      case (V_LOG)
         call var_list%get_lptr (var_name, lptr, known)
         call eval_node_init_log_ptr (en, var_name, lptr, known)
      case (V_INT)
         call var_list%get_iptr (var_name, iptr, known)
```

```
      call eval_node_init_int_ptr (en, var_name, iptr, known)
   case (V_REAL)
      call var_list%get_rptr (var_name, rptr, known)
      call eval_node_init_real_ptr (en, var_name, rptr, known)
   case (V_CMPLX)
      call var_list%get_cptr (var_name, cptr, known)
      call eval_node_init_cmplx_ptr (en, var_name, cptr, known)
   case (V_SEV)
      call var_list%get_pptr (var_name, pptr, known)
      call eval_node_init_subevt_ptr (en, var_name, pptr, known)
   case (V_STR)
      call var_list%get_sptr (var_name, sptr, known)
      call eval_node_init_string_ptr (en, var_name, sptr, known)
   case (V_OBS1_INT)
      call var_list%get_obs1_iptr (var_name, obs1_iptr, p1)
      call eval_node_init_obs1_int_ptr (en, var_name, obs1_iptr, p1)
   case (V_OBS2_INT)
      call var_list%get_obs2_iptr (var_name, obs2_iptr, p1, p2)
      call eval_node_init_obs2_int_ptr (en, var_name, obs2_iptr, p1, p2)
   case (V_OBS1_REAL)
      call var_list%get_obs1_rptr (var_name, obs1_rptr, p1)
      call eval_node_init_obs1_real_ptr (en, var_name, obs1_rptr, p1)
   case (V_OBS2_REAL)
      call var_list%get_obs2_rptr (var_name, obs2_rptr, p1, p2)
      call eval_node_init_obs2_real_ptr (en, var_name, obs2_rptr, p1, p2)
   case default
      call parse_node_write (pn)
      call msg_fatal ("Variable of this type " // &
           "is not allowed in the present context")
      if (present (var_type)) then
         select case (var_type)
         case (V_LOG)
            call eval_node_init_log_ptr (en, var_name, no_lval, unknown)
         case (V_SEV)
            call eval_node_init_subevt_ptr &
                 (en, var_name, no_pval, unknown)
         case (V_STR)
            call eval_node_init_string_ptr &
                 (en, var_name, no_sval, unknown)
         end select
      else
         call eval_node_init_real_ptr (en, var_name, no_rval, unknown)
      end if
   end select
else
   call parse_node_write (pn)
   call msg_error ("This variable is undefined at this point")
   if (present (var_type)) then
      select case (var_type)
      case (V_LOG)
         call eval_node_init_log_ptr (en, var_name, no_lval, unknown)
      case (V_SEV)
         call eval_node_init_subevt_ptr &
              (en, var_name, no_pval, unknown)
```

```
            case (V_STR)
               call eval_node_init_string_ptr (en, var_name, no_sval, unknown)
            end select
         else
            call eval_node_init_real_ptr (en, var_name, no_rval, unknown)
         end if
      end if
   end select
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done variable"
   end if
end subroutine eval_node_compile_variable
```

In a given context, a variable has to have a certain type.

⟨*Eval trees: procedures*⟩+≡

```
subroutine check_var_type (pn, ok, type_actual, type_requested)
  type(parse_node_t), intent(in) :: pn
  logical, intent(out) :: ok
  integer, intent(in) :: type_actual
  integer, intent(in), optional :: type_requested
  if (present (type_requested)) then
     select case (type_requested)
     case (V_LOG)
        select case (type_actual)
        case (V_LOG)
        case default
           call parse_node_write (pn)
           call msg_fatal ("Variable type is invalid (should be logical)")
           ok = .false.
        end select
     case (V_SEV)
        select case (type_actual)
        case (V_SEV)
        case default
           call parse_node_write (pn)
           call msg_fatal &
                ("Variable type is invalid (should be particle set)")
           ok = .false.
        end select
     case (V_PDG)
        select case (type_actual)
        case (V_PDG)
        case default
           call parse_node_write (pn)
           call msg_fatal &
                ("Variable type is invalid (should be PDG array)")
           ok = .false.
        end select
     case (V_STR)
        select case (type_actual)
        case (V_STR)
        case default
           call parse_node_write (pn)
```

```
            call msg_fatal &
                 ("Variable type is invalid (should be string)")
            ok = .false.
         end select
      case default
         call parse_node_write (pn)
         call msg_bug ("Variable type is unknown")
      end select
   else
      select case (type_actual)
      case (V_REAL, V_OBS1_REAL, V_OBS2_REAL, V_INT, V_OBS1_INT, &
               V_OBS2_INT, V_CMPLX)
      case default
         call parse_node_write (pn)
         call msg_fatal ("Variable type is invalid (should be numeric)")
         ok = .false.
      end select
   end if
   ok = .true.
end subroutine check_var_type
```

Retrieve the result of an integration. If the requested process has been integrated, the results are available as special variables. (The variables cannot be accessed in the usual way since they contain brackets in their names.)

Since this compilation step may occur before the processes have been loaded, we have to initialize the required variables before they are used.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_compile_result (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in), target :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_key, pn_prc_id
  type(string_t) :: key, prc_id, var_name
  integer, pointer :: iptr
  real(default), pointer :: rptr
  logical, pointer :: known
  if (debug_active (D_MODEL_F)) then
     print *, "read result";  call parse_node_write (pn)
  end if
  pn_key => parse_node_get_sub_ptr (pn)
  pn_prc_id => parse_node_get_next_ptr (pn_key)
  key = parse_node_get_key (pn_key)
  prc_id = parse_node_get_string (pn_prc_id)
  var_name = key // "(" // prc_id // ")"
  if (var_list%contains (var_name)) then
     allocate (en)
     select case (char(key))
     case ("num_id", "n_calls")
        call var_list%get_iptr (var_name, iptr, known)
        call eval_node_init_int_ptr (en, var_name, iptr, known)
     case ("integral", "error")
        call var_list%get_rptr (var_name, rptr, known)
        call eval_node_init_real_ptr (en, var_name, rptr, known)
```

```
      end select
   else
      call msg_fatal ("Result variable '" // char (var_name) &
           // "' is undefined (call 'integrate' before use)")
   end if
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done result"
   end if
end subroutine eval_node_compile_result
```

This user observable behaves like a variable. We link the node to the generic user-observable entry in the variable list. The syntax element has an argument which provides the name of the user variable, this is stored as an eval-node alongside with the variable. When the variable value is used, the user-supplied external function is called and provides the (real) result value.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_compile_user_observable (en, pn, var_list)
   type(eval_node_t), pointer :: en
   type(parse_node_t), intent(in), target :: pn
   type(var_list_t), intent(in), target :: var_list
   type(parse_node_t), pointer :: pn_key, pn_arg, pn_obs
   type(eval_node_t), pointer :: en0
   integer :: res_type
   type(string_t) :: var_name
   integer :: type
   logical :: defined
   if (debug_active (D_MODEL_F)) then
      print *, "read user observable";  call parse_node_write (pn)
   end if
   pn_key => parse_node_get_sub_ptr (pn)
   select case (char (parse_node_get_key (pn_key)))
   case ("user_obs")
      res_type = V_REAL
   case default
      call parse_node_write (pn_key)
      call msg_bug ("user_observable: wrong keyword")
   end select
   pn_arg => parse_node_get_next_ptr (pn_key)
   pn_obs => parse_node_get_sub_ptr (pn_arg)
   call eval_node_compile_sexpr (en0, pn_obs, var_list)
   select case (res_type)
   case (V_INT);  var_name = "_User_obs_int"
   case (V_REAL); var_name = "_User_obs_real"
   end select
   call var_list%get_var_properties (var_name, type=type, is_defined=defined)
   allocate (en)
   if (defined) then
      select case (type)
      case (V_UOBS1_INT)
         call eval_node_init_uobs1_int (en, var_name, en0)
      case (V_UOBS2_INT)
         call eval_node_init_uobs2_int (en, var_name, en0)
```

```
      case (V_UOBS1_REAL)
         call eval_node_init_uobs1_real (en, var_name, en0)
      case (V_UOBS2_REAL)
         call eval_node_init_uobs2_real (en, var_name, en0)
      end select
   else
      call parse_node_write (pn)
      call msg_error ("This variable is undefined at this point")
   end if
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done user observable"
   end if
end subroutine eval_node_compile_user_observable
```

Functions with a single argument. For non-constant arguments, watch for functions which convert their argument to a different type.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_unary_function (en, pn, var_list)
   type(eval_node_t), pointer :: en
   type(parse_node_t), intent(in) :: pn
   type(var_list_t), intent(in), target :: var_list
   type(parse_node_t), pointer :: pn_fname, pn_arg
   type(eval_node_t), pointer :: en1
   type(string_t) :: key
   integer :: t
   if (debug_active (D_MODEL_F)) then
      print *, "read unary function";  call parse_node_write (pn)
   end if
   pn_fname => parse_node_get_sub_ptr (pn)
   pn_arg => parse_node_get_next_ptr (pn_fname, tag="function_arg1")
   call eval_node_compile_expr &
        (en1, parse_node_get_sub_ptr (pn_arg, tag="expr"), var_list)
   t = en1%result_type
   allocate (en)
   key = parse_node_get_key (pn_fname)
   if (en1%type == EN_CONSTANT) then
      select case (char (key))
      case ("complex")
         select case (t)
         case (V_INT);  call eval_node_init_cmplx (en, cmplx_i (en1))
         case (V_REAL); call eval_node_init_cmplx (en, cmplx_r (en1))
         case (V_CMPLX); deallocate (en);  en => en1;  en1 => null ()
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("real")
         select case (t)
         case (V_INT);  call eval_node_init_real (en, real_i (en1))
         case (V_REAL); deallocate (en);  en => en1;  en1 => null ()
         case (V_CMPLX); call eval_node_init_real (en, real_c (en1))
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("int")
```

```
   select case (t)
   case (V_INT);  deallocate (en);  en => en1;  en1 => null ()
   case (V_REAL); call eval_node_init_int  (en, int_r (en1))
   case (V_CMPLX); call eval_node_init_int  (en, int_c (en1))
   end select
case ("nint")
   select case (t)
   case (V_INT);  deallocate (en);  en => en1;  en1 => null ()
   case (V_REAL); call eval_node_init_int  (en, nint_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("floor")
   select case (t)
   case (V_INT);  deallocate (en);  en => en1;  en1 => null ()
   case (V_REAL); call eval_node_init_int  (en, floor_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("ceiling")
   select case (t)
   case (V_INT);  deallocate (en);  en => en1;  en1 => null ()
   case (V_REAL); call eval_node_init_int  (en, ceiling_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("abs")
   select case (t)
   case (V_INT);  call eval_node_init_int  (en, abs_i (en1))
   case (V_REAL); call eval_node_init_real (en, abs_r (en1))
   case (V_CMPLX); call eval_node_init_real (en, abs_c (en1))
   end select
case ("sgn")
   select case (t)
   case (V_INT);  call eval_node_init_int  (en, sgn_i (en1))
   case (V_REAL); call eval_node_init_real (en, sgn_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("sqrt")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, sqrt_r (en1))
   case (V_CMPLX); call eval_node_init_cmplx (en, sqrt_c (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("exp")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, exp_r (en1))
   case (V_CMPLX); call eval_node_init_cmplx (en, exp_c (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("log")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, log_r (en1))
   case (V_CMPLX); call eval_node_init_cmplx (en, log_c (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("log10")
```

```
   select case (t)
   case (V_REAL); call eval_node_init_real (en, log10_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("sin")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, sin_r (en1))
   case (V_CMPLX); call eval_node_init_cmplx (en, sin_c (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("cos")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, cos_r (en1))
   case (V_CMPLX); call eval_node_init_cmplx (en, cos_c (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("tan")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, tan_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("asin")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, asin_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("acos")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, acos_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("atan")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, atan_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("sinh")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, sinh_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("cosh")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, cosh_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("tanh")
   select case (t)
   case (V_REAL); call eval_node_init_real (en, tanh_r (en1))
   case default;  call eval_type_error (pn, char (key), t)
   end select
case default
   call parse_node_mismatch ("function name", pn_fname)
end select
```

```
   if (associated (en1)) then
      call eval_node_final_rec (en1)
      deallocate (en1)
   end if
else
   select case (char (key))
   case ("complex")
      call eval_node_init_branch (en, key, V_CMPLX, en1)
   case ("real")
      call eval_node_init_branch (en, key, V_REAL, en1)
   case ("int", "nint", "floor", "ceiling")
      call eval_node_init_branch (en, key, V_INT, en1)
   case default
      call eval_node_init_branch (en, key, t, en1)
   end select
   select case (char (key))
   case ("complex")
      select case (t)
      case (V_INT);  call eval_node_set_op1_cmplx (en, cmplx_i)
      case (V_REAL); call eval_node_set_op1_cmplx (en, cmplx_r)
      case (V_CMPLX); deallocate (en);  en => en1
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("real")
      select case (t)
      case (V_INT);  call eval_node_set_op1_real (en, real_i)
      case (V_REAL); deallocate (en);  en => en1
      case (V_CMPLX); call eval_node_set_op1_real (en, real_c)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("int")
      select case (t)
      case (V_INT);  deallocate (en);  en => en1
      case (V_REAL); call eval_node_set_op1_int (en, int_r)
      case (V_CMPLX); call eval_node_set_op1_int (en, int_c)
      end select
   case ("nint")
      select case (t)
      case (V_INT);  deallocate (en);  en => en1
      case (V_REAL); call eval_node_set_op1_int (en, nint_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("floor")
      select case (t)
      case (V_INT);  deallocate (en);  en => en1
      case (V_REAL); call eval_node_set_op1_int (en, floor_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
   case ("ceiling")
      select case (t)
      case (V_INT);  deallocate (en);  en => en1
      case (V_REAL); call eval_node_set_op1_int (en, ceiling_r)
      case default;  call eval_type_error (pn, char (key), t)
      end select
```

```
case ("abs")
   select case (t)
   case (V_INT);  call eval_node_set_op1_int  (en, abs_i)
   case (V_REAL); call eval_node_set_op1_real (en, abs_r)
   case (V_CMPLX);
      call eval_node_init_branch (en, key, V_REAL, en1)
      call eval_node_set_op1_real (en, abs_c)
   end select
case ("sgn")
   select case (t)
   case (V_INT);  call eval_node_set_op1_int  (en, sgn_i)
   case (V_REAL); call eval_node_set_op1_real (en, sgn_r)
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("sqrt")
   select case (t)
   case (V_REAL); call eval_node_set_op1_real (en, sqrt_r)
   case (V_CMPLX); call eval_node_set_op1_cmplx (en, sqrt_c)
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("exp")
   select case (t)
   case (V_REAL); call eval_node_set_op1_real (en, exp_r)
   case (V_CMPLX); call eval_node_set_op1_cmplx (en, exp_c)
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("log")
   select case (t)
   case (V_REAL); call eval_node_set_op1_real (en, log_r)
   case (V_CMPLX); call eval_node_set_op1_cmplx (en, log_c)
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("log10")
   select case (t)
   case (V_REAL); call eval_node_set_op1_real (en, log10_r)
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("sin")
   select case (t)
   case (V_REAL); call eval_node_set_op1_real (en, sin_r)
   case (V_CMPLX); call eval_node_set_op1_cmplx (en, sin_c)
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("cos")
   select case (t)
   case (V_REAL); call eval_node_set_op1_real (en, cos_r)
   case (V_CMPLX); call eval_node_set_op1_cmplx (en, cos_c)
   case default;  call eval_type_error (pn, char (key), t)
   end select
case ("tan")
   select case (t)
   case (V_REAL); call eval_node_set_op1_real (en, tan_r)
   case default;  call eval_type_error (pn, char (key), t)
   end select
```

```
      case ("asin")
         select case (t)
         case (V_REAL); call eval_node_set_op1_real (en, asin_r)
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("acos")
         select case (t)
         case (V_REAL); call eval_node_set_op1_real (en, acos_r)
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("atan")
         select case (t)
         case (V_REAL); call eval_node_set_op1_real (en, atan_r)
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("sinh")
         select case (t)
         case (V_REAL); call eval_node_set_op1_real (en, sinh_r)
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("cosh")
         select case (t)
         case (V_REAL); call eval_node_set_op1_real (en, cosh_r)
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case ("tanh")
         select case (t)
         case (V_REAL); call eval_node_set_op1_real (en, tanh_r)
         case default;  call eval_type_error (pn, char (key), t)
         end select
      case default
         call parse_node_mismatch ("function name", pn_fname)
      end select
   end if
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done function"
   end if
end subroutine eval_node_compile_unary_function
```

Functions with two arguments.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_binary_function (en, pn, var_list)
   type(eval_node_t), pointer :: en
   type(parse_node_t), intent(in) :: pn
   type(var_list_t), intent(in), target :: var_list
   type(parse_node_t), pointer :: pn_fname, pn_arg, pn_arg1, pn_arg2
   type(eval_node_t), pointer :: en1, en2
   type(string_t) :: key
   integer :: t1, t2
   if (debug_active (D_MODEL_F)) then
      print *, "read binary function";  call parse_node_write (pn)
   end if
   pn_fname => parse_node_get_sub_ptr (pn)
```

```
pn_arg => parse_node_get_next_ptr (pn_fname, tag="function_arg2")
pn_arg1 => parse_node_get_sub_ptr (pn_arg, tag="expr")
pn_arg2 => parse_node_get_next_ptr (pn_arg1, tag="expr")
call eval_node_compile_expr (en1, pn_arg1, var_list)
call eval_node_compile_expr (en2, pn_arg2, var_list)
t1 = en1%result_type
t2 = en2%result_type
allocate (en)
key = parse_node_get_key (pn_fname)
if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
   select case (char (key))
   case ("max")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_int  (en, max_ii (en1, en2))
         case (V_REAL); call eval_node_init_real (en, max_ir (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_real (en, max_ri (en1, en2))
         case (V_REAL); call eval_node_init_real (en, max_rr (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
         end select
       case default;  call eval_type_error (pn, char (key), t1)
     end select
   case ("min")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_int  (en, min_ii (en1, en2))
         case (V_REAL); call eval_node_init_real (en, min_ir (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_real (en, min_ri (en1, en2))
         case (V_REAL); call eval_node_init_real (en, min_rr (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
         end select
       case default;  call eval_type_error (pn, char (key), t1)
     end select
   case ("mod")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_int  (en, mod_ii (en1, en2))
         case (V_REAL); call eval_node_init_real (en, mod_ir (en1, en2))
         case default;  call eval_type_error (pn, char (key), t2)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_real (en, mod_ri (en1, en2))
```

```
            case (V_REAL); call eval_node_init_real (en, mod_rr (en1, en2))
            case default;  call eval_type_error (pn, char (key), t2)
            end select
         case default;  call eval_type_error (pn, char (key), t1)
         end select
      case ("modulo")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_init_int  (en, modulo_ii (en1, en2))
            case (V_REAL); call eval_node_init_real (en, modulo_ir (en1, en2))
            case default;  call eval_type_error (pn, char (key), t2)
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_init_real (en, modulo_ri (en1, en2))
            case (V_REAL); call eval_node_init_real (en, modulo_rr (en1, en2))
            case default;  call eval_type_error (pn, char (key), t2)
            end select
         case default;  call eval_type_error (pn, char (key), t2)
        end select
      case default
         call parse_node_mismatch ("function name", pn_fname)
      end select
      call eval_node_final_rec (en1)
      deallocate (en1)
   else
      call eval_node_init_branch (en, key, t1, en1, en2)
      select case (char (key))
      case ("max")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_int  (en, max_ii)
            case (V_REAL); call eval_node_set_op2_real (en, max_ir)
            case default;  call eval_type_error (pn, char (key), t2)
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_real (en, max_ri)
            case (V_REAL); call eval_node_set_op2_real (en, max_rr)
            case default;  call eval_type_error (pn, char (key), t2)
            end select
         case default;  call eval_type_error (pn, char (key), t2)
        end select
      case ("min")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_int  (en, min_ii)
            case (V_REAL); call eval_node_set_op2_real (en, min_ir)
            case default;  call eval_type_error (pn, char (key), t2)
            end select
         case (V_REAL)
```

```
            select case (t2)
            case (V_INT);  call eval_node_set_op2_real (en, min_ri)
            case (V_REAL); call eval_node_set_op2_real (en, min_rr)
            case default;  call eval_type_error (pn, char (key), t2)
            end select
         case default;  call eval_type_error (pn, char (key), t2)
      end select
      case ("mod")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_int  (en, mod_ii)
            case (V_REAL); call eval_node_set_op2_real (en, mod_ir)
            case default;  call eval_type_error (pn, char (key), t2)
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_real (en, mod_ri)
            case (V_REAL); call eval_node_set_op2_real (en, mod_rr)
            case default;  call eval_type_error (pn, char (key), t2)
            end select
         case default;  call eval_type_error (pn, char (key), t2)
      end select
      case ("modulo")
         select case (t1)
         case (V_INT)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_int  (en, modulo_ii)
            case (V_REAL); call eval_node_set_op2_real (en, modulo_ir)
            case default;  call eval_type_error (pn, char (key), t2)
            end select
         case (V_REAL)
            select case (t2)
            case (V_INT);  call eval_node_set_op2_real (en, modulo_ri)
            case (V_REAL); call eval_node_set_op2_real (en, modulo_rr)
            case default;  call eval_type_error (pn, char (key), t2)
            end select
         case default;  call eval_type_error (pn, char (key), t2)
      end select
      case default
         call parse_node_mismatch ("function name", pn_fname)
      end select
   end if
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done function"
   end if
end subroutine eval_node_compile_binary_function
```

Variable definition

A block expression contains a variable definition (first argument) and an expression where the definition can be used (second argument). The `result_type`

decides which type of expression is expected for the second argument. For numeric variables, if there is a mismatch between real and integer type, insert an extra node for type conversion.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_block_expr &
     (en, pn, var_list, result_type)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  integer, intent(in), optional :: result_type
  type(parse_node_t), pointer :: pn_var_spec, pn_var_subspec
  type(parse_node_t), pointer :: pn_var_type, pn_var_name, pn_var_expr
  type(parse_node_t), pointer :: pn_expr
  type(string_t) :: var_name
  type(eval_node_t), pointer :: en1, en2
  integer :: var_type
  logical :: new
  if (debug_active (D_MODEL_F)) then
     print *, "read block expr";  call parse_node_write (pn)
  end if
  new = .false.
  pn_var_spec => parse_node_get_sub_ptr (pn, 2)
  select case (char (parse_node_get_rule_key (pn_var_spec)))
  case ("var_num");      var_type = V_NONE
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec)
  case ("var_int");      var_type = V_INT
     new = .true.
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
  case ("var_real");     var_type = V_REAL
     new = .true.
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
  case ("var_cmplx");     var_type = V_CMPLX
     new = .true.
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
  case ("var_logical_new");  var_type = V_LOG
     new = .true.
     pn_var_subspec => parse_node_get_sub_ptr (pn_var_spec, 2)
     pn_var_name => parse_node_get_sub_ptr (pn_var_subspec, 2)
  case ("var_logical_spec");  var_type = V_LOG
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
  case ("var_plist_new");    var_type = V_SEV
     new = .true.
     pn_var_subspec => parse_node_get_sub_ptr (pn_var_spec, 2)
     pn_var_name => parse_node_get_sub_ptr (pn_var_subspec, 2)
  case ("var_plist_spec");    var_type = V_SEV
     new = .true.
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
  case ("var_alias");    var_type = V_PDG
     new = .true.
     pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
  case ("var_string_new");   var_type = V_STR
     new = .true.
     pn_var_subspec => parse_node_get_sub_ptr (pn_var_spec, 2)
     pn_var_name => parse_node_get_sub_ptr (pn_var_subspec, 2)
```

```
    case ("var_string_spec");   var_type = V_STR
       pn_var_name => parse_node_get_sub_ptr (pn_var_spec, 2)
    case default
       call parse_node_mismatch &
            ("logical|int|real|plist|alias", pn_var_type)
    end select
    pn_var_expr => parse_node_get_next_ptr (pn_var_name, 2)
    pn_expr => parse_node_get_next_ptr (pn_var_spec, 2)
    var_name = parse_node_get_string (pn_var_name)
    select case (var_type)
    case (V_LOG);  var_name = "?" // var_name
    case (V_SEV);  var_name = "@" // var_name
    case (V_STR);  var_name = "$" // var_name    ! $ sign
    end select
    call var_list_check_user_var (var_list, var_name, var_type, new)
    call eval_node_compile_genexpr (en1, pn_var_expr, var_list, var_type)
    call insert_conversion_node (en1, var_type)
    allocate (en)
    call eval_node_init_block (en, var_name, var_type, en1, var_list)
    call eval_node_compile_genexpr (en2, pn_expr, en%var_list, result_type)
    call eval_node_set_expr (en, en2)
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done block expr"
    end if
  end subroutine eval_node_compile_block_expr
```

Insert a conversion node for integer/real/complex transformation if necessary. What shall we do for the complex to integer/real conversion?

⟨*Eval trees: procedures*⟩+≡

```
  subroutine insert_conversion_node (en, result_type)
    type(eval_node_t), pointer :: en
    integer, intent(in) :: result_type
    type(eval_node_t), pointer :: en_conv
    select case (en%result_type)
    case (V_INT)
       select case (result_type)
       case (V_REAL)
          allocate (en_conv)
          call eval_node_init_branch (en_conv, var_str ("real"), V_REAL, en)
          call eval_node_set_op1_real (en_conv, real_i)
          en => en_conv
       case (V_CMPLX)
          allocate (en_conv)
          call eval_node_init_branch (en_conv, var_str ("complex"), V_CMPLX, en)
          call eval_node_set_op1_cmplx (en_conv, cmplx_i)
          en => en_conv
       end select
    case (V_REAL)
       select case (result_type)
       case (V_INT)
          allocate (en_conv)
          call eval_node_init_branch (en_conv, var_str ("int"), V_INT, en)
```

```
            call eval_node_set_op1_int (en_conv, int_r)
            en => en_conv
         case (V_CMPLX)
            allocate (en_conv)
            call eval_node_init_branch (en_conv, var_str ("complex"), V_CMPLX, en)
            call eval_node_set_op1_cmplx (en_conv, cmplx_r)
            en => en_conv
         end select
      case (V_CMPLX)
         select case (result_type)
         case (V_INT)
            allocate (en_conv)
            call eval_node_init_branch (en_conv, var_str ("int"), V_INT, en)
            call eval_node_set_op1_int (en_conv, int_c)
            en => en_conv
         case (V_REAL)
            allocate (en_conv)
            call eval_node_init_branch (en_conv, var_str ("real"), V_REAL, en)
            call eval_node_set_op1_real (en_conv, real_c)
            en => en_conv
         end select
       case default
       end select
   end subroutine insert_conversion_node
```

Conditionals

A conditional has the structure if lexpr then expr else expr. So we first evaluate the logical expression, then depending on the result the first or second expression. Note that the second expression is mandatory.

The `result_type`, if present, defines the requested type of the `then` and `else` clauses. Default is numeric (int/real). If there is a mismatch between real and integer result types, insert conversion nodes.

⟨*Eval trees: procedures*⟩+≡

```
   recursive subroutine eval_node_compile_conditional &
        (en, pn, var_list, result_type)
      type(eval_node_t), pointer :: en
      type(parse_node_t), intent(in) :: pn
      type(var_list_t), intent(in), target :: var_list
      integer, intent(in), optional :: result_type
      type(parse_node_t), pointer :: pn_condition, pn_expr
      type(parse_node_t), pointer :: pn_maybe_elsif, pn_elsif_branch
      type(parse_node_t), pointer :: pn_maybe_else, pn_else_branch, pn_else_expr
      type(eval_node_t), pointer :: en0, en1, en2
      integer :: restype
      if (debug_active (D_MODEL_F)) then
         print *, "read conditional";  call parse_node_write (pn)
      end if
      pn_condition => parse_node_get_sub_ptr (pn, 2, tag="lexpr")
      pn_expr => parse_node_get_next_ptr (pn_condition, 2)
      call eval_node_compile_lexpr (en0, pn_condition, var_list)
      call eval_node_compile_genexpr (en1, pn_expr, var_list, result_type)
```

```
   if (present (result_type)) then
      restype = major_result_type (result_type, en1%result_type)
   else
      restype = en1%result_type
   end if
   pn_maybe_elsif => parse_node_get_next_ptr (pn_expr)
   select case (char (parse_node_get_rule_key (pn_maybe_elsif)))
   case ("maybe_elsif_expr", &
         "maybe_elsif_lexpr", &
         "maybe_elsif_pexpr", &
         "maybe_elsif_cexpr", &
         "maybe_elsif_sexpr")
      pn_elsif_branch => parse_node_get_sub_ptr (pn_maybe_elsif)
      pn_maybe_else => parse_node_get_next_ptr (pn_maybe_elsif)
      select case (char (parse_node_get_rule_key (pn_maybe_else)))
      case ("maybe_else_expr", &
         "maybe_else_lexpr", &
         "maybe_else_pexpr", &
         "maybe_else_cexpr", &
         "maybe_else_sexpr")
         pn_else_branch => parse_node_get_sub_ptr (pn_maybe_else)
         pn_else_expr => parse_node_get_sub_ptr (pn_else_branch, 2)
      case default
         pn_else_expr => null ()
      end select
      call eval_node_compile_elsif &
           (en2, pn_elsif_branch, pn_else_expr, var_list, restype)
   case ("maybe_else_expr", &
         "maybe_else_lexpr", &
         "maybe_else_pexpr", &
         "maybe_else_cexpr", &
         "maybe_else_sexpr")
      pn_maybe_else => pn_maybe_elsif
      pn_maybe_elsif => null ()
      pn_else_branch => parse_node_get_sub_ptr (pn_maybe_else)
      pn_else_expr => parse_node_get_sub_ptr (pn_else_branch, 2)
      call eval_node_compile_genexpr &
           (en2, pn_else_expr, var_list, restype)
   case ("endif")
      call eval_node_compile_default_else (en2, restype)
   case default
      call msg_bug ("Broken conditional: unexpected " &
           // char (parse_node_get_rule_key (pn_maybe_elsif)))
   end select
   call eval_node_create_conditional (en, en0, en1, en2, restype)
   call conditional_insert_conversion_nodes (en, restype)
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done conditional"
   end if
end subroutine eval_node_compile_conditional
```

This recursively generates 'elsif' conditionals as a chain of sub-nodes of the main conditional.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_elsif &
     (en, pn, pn_else_expr, var_list, result_type)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in), target :: pn
  type(parse_node_t), pointer :: pn_else_expr
  type(var_list_t), intent(in), target :: var_list
  integer, intent(inout) :: result_type
  type(parse_node_t), pointer :: pn_next, pn_condition, pn_expr
  type(eval_node_t), pointer :: en0, en1, en2
  pn_condition => parse_node_get_sub_ptr (pn, 2, tag="lexpr")
  pn_expr => parse_node_get_next_ptr (pn_condition, 2)
  call eval_node_compile_lexpr (en0, pn_condition, var_list)
  call eval_node_compile_genexpr (en1, pn_expr, var_list, result_type)
  result_type = major_result_type (result_type, en1%result_type)
  pn_next => parse_node_get_next_ptr (pn)
  if (associated (pn_next)) then
     call eval_node_compile_elsif &
          (en2, pn_next, pn_else_expr, var_list, result_type)
     result_type = major_result_type (result_type, en2%result_type)
  else if (associated (pn_else_expr)) then
     call eval_node_compile_genexpr &
          (en2, pn_else_expr, var_list, result_type)
     result_type = major_result_type (result_type, en2%result_type)
  else
     call eval_node_compile_default_else (en2, result_type)
  end if
  call eval_node_create_conditional (en, en0, en1, en2, result_type)
end subroutine eval_node_compile_elsif
```

This makes a default 'else' branch in case it was omitted. The default value just depends on the expected type.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_compile_default_else (en, result_type)
  type(eval_node_t), pointer :: en
  integer, intent(in) :: result_type
  type(subevt_t) :: pval_empty
  type(pdg_array_t) :: aval_undefined
  allocate (en)
  select case (result_type)
  case (V_LOG);  call eval_node_init_log (en, .false.)
  case (V_INT);  call eval_node_init_int (en, 0)
  case (V_REAL);  call eval_node_init_real (en, 0._default)
  case (V_CMPLX)
       call eval_node_init_cmplx (en, (0._default, 0._default))
  case (V_SEV)
     call subevt_init (pval_empty)
     call eval_node_init_subevt (en, pval_empty)
  case (V_PDG)
     call eval_node_init_pdg_array  (en, aval_undefined)
  case (V_STR)
     call eval_node_init_string (en, var_str (""))
  case default
```

```
      call msg_bug ("Undefined type for 'else' branch in conditional")
    end select
  end subroutine eval_node_compile_default_else
```

If the logical expression is constant, we can simplify the conditional node by replacing it with the selected branch. Otherwise, we initialize a true branching.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_node_create_conditional (en, en0, en1, en2, result_type)
    type(eval_node_t), pointer :: en, en0, en1, en2
    integer, intent(in) :: result_type
    if (en0%type == EN_CONSTANT) then
      if (en0%lval) then
        en => en1
        call eval_node_final_rec (en2)
        deallocate (en2)
      else
        en => en2
        call eval_node_final_rec (en1)
        deallocate (en1)
      end if
    else
      allocate (en)
      call eval_node_init_conditional (en, result_type, en0, en1, en2)
    end if
  end subroutine eval_node_create_conditional
```

Return the numerical result type which should be used for the combination of the two result types.

⟨*Eval trees: procedures*⟩+≡

```
  function major_result_type (t1, t2) result (t)
    integer :: t
    integer, intent(in) :: t1, t2
    select case (t1)
    case (V_INT)
      select case (t2)
      case (V_INT, V_REAL, V_CMPLX)
        t = t2
      case default
        call type_mismatch ()
      end select
    case (V_REAL)
      select case (t2)
      case (V_INT)
        t = t1
      case (V_REAL, V_CMPLX)
        t = t2
      case default
        call type_mismatch ()
      end select
    case (V_CMPLX)
      select case (t2)
      case (V_INT, V_REAL, V_CMPLX)
        t = t1
```

```
      case default
         call type_mismatch ()
      end select
   case default
      if (t1 == t2) then
         t = t1
      else
         call type_mismatch ()
      end if
   end select
contains
   subroutine type_mismatch ()
     call msg_bug ("Type mismatch in branches of a conditional expression")
   end subroutine type_mismatch
end function major_result_type
```

Recursively insert conversion nodes where necessary.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine conditional_insert_conversion_nodes (en, result_type)
  type(eval_node_t), intent(inout), target :: en
  integer, intent(in) :: result_type
  select case (result_type)
  case (V_INT, V_REAL, V_CMPLX)
     call insert_conversion_node (en%arg1, result_type)
     if (en%arg2%type == EN_CONDITIONAL) then
        call conditional_insert_conversion_nodes (en%arg2, result_type)
     else
        call insert_conversion_node (en%arg2, result_type)
     end if
  end select
end subroutine conditional_insert_conversion_nodes
```

Logical expressions

A logical expression consists of one or more singlet logical expressions concatenated by `;`. This is for allowing side-effects, only the last value is used.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_lexpr (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_term, pn_sequel, pn_arg
  type(eval_node_t), pointer :: en1, en2
  if (debug_active (D_MODEL_F)) then
     print *, "read lexpr";  call parse_node_write (pn)
  end if
  pn_term => parse_node_get_sub_ptr (pn, tag="lsinglet")
  call eval_node_compile_lsinglet (en, pn_term, var_list)
  pn_sequel => parse_node_get_next_ptr (pn_term, tag="lsequel")
  do while (associated (pn_sequel))
     pn_arg => parse_node_get_sub_ptr (pn_sequel, 2, tag="lsinglet")
     en1 => en
```

```
      call eval_node_compile_lsinglet (en2, pn_arg, var_list)
      allocate (en)
      if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
         call eval_node_init_log (en, ignore_first_ll (en1, en2))
         call eval_node_final_rec (en1)
         call eval_node_final_rec (en2)
         deallocate (en1, en2)
      else
         call eval_node_init_branch &
              (en, var_str ("lsequel"), V_LOG, en1, en2)
         call eval_node_set_op2_log (en, ignore_first_ll)
      end if
      pn_sequel => parse_node_get_next_ptr (pn_sequel)
   end do
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done lexpr"
   end if
end subroutine eval_node_compile_lexpr
```

A logical singlet expression consists of one or more logical terms concatenated by `or`.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_lsinglet (en, pn, var_list)
   type(eval_node_t), pointer :: en
   type(parse_node_t), intent(in) :: pn
   type(var_list_t), intent(in), target :: var_list
   type(parse_node_t), pointer :: pn_term, pn_alternative, pn_arg
   type(eval_node_t), pointer :: en1, en2
   if (debug_active (D_MODEL_F)) then
      print *, "read lsinglet";  call parse_node_write (pn)
   end if
   pn_term => parse_node_get_sub_ptr (pn, tag="lterm")
   call eval_node_compile_lterm (en, pn_term, var_list)
   pn_alternative => parse_node_get_next_ptr (pn_term, tag="alternative")
   do while (associated (pn_alternative))
      pn_arg => parse_node_get_sub_ptr (pn_alternative, 2, tag="lterm")
      en1 => en
      call eval_node_compile_lterm (en2, pn_arg, var_list)
      allocate (en)
      if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
         call eval_node_init_log (en, or_ll (en1, en2))
         call eval_node_final_rec (en1)
         call eval_node_final_rec (en2)
         deallocate (en1, en2)
      else
         call eval_node_init_branch &
              (en, var_str ("alternative"), V_LOG, en1, en2)
         call eval_node_set_op2_log (en, or_ll)
      end if
      pn_alternative => parse_node_get_next_ptr (pn_alternative)
   end do
   if (debug_active (D_MODEL_F)) then
```

```
      call eval_node_write (en)
      print *, "done lsinglet"
    end if
  end subroutine eval_node_compile_lsinglet
```

A logical term consists of one or more logical values concatenated by and.

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_lterm (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_term, pn_coincidence, pn_arg
    type(eval_node_t), pointer :: en1, en2
    if (debug_active (D_MODEL_F)) then
       print *, "read lterm";  call parse_node_write (pn)
    end if
    pn_term => parse_node_get_sub_ptr (pn)
    call eval_node_compile_lvalue (en, pn_term, var_list)
    pn_coincidence => parse_node_get_next_ptr (pn_term, tag="coincidence")
    do while (associated (pn_coincidence))
       pn_arg => parse_node_get_sub_ptr (pn_coincidence, 2)
       en1 => en
       call eval_node_compile_lvalue (en2, pn_arg, var_list)
       allocate (en)
       if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
          call eval_node_init_log (en, and_ll (en1, en2))
          call eval_node_final_rec (en1)
          call eval_node_final_rec (en2)
          deallocate (en1, en2)
       else
          call eval_node_init_branch &
               (en, var_str ("coincidence"), V_LOG, en1, en2)
          call eval_node_set_op2_log (en, and_ll)
       end if
       pn_coincidence => parse_node_get_next_ptr (pn_coincidence)
    end do
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done lterm"
    end if
  end subroutine eval_node_compile_lterm
```

Logical variables are disabled, because they are confused with the l.h.s. of compared expressions.

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_lvalue (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    if (debug_active (D_MODEL_F)) then
       print *, "read lvalue";  call parse_node_write (pn)
    end if
    select case (char (parse_node_get_rule_key (pn)))
```

```
    case ("true")
       allocate (en)
       call eval_node_init_log (en, .true.)
    case ("false")
       allocate (en)
       call eval_node_init_log (en, .false.)
    case ("negation")
       call eval_node_compile_negation (en, pn, var_list)
    case ("lvariable")
       call eval_node_compile_variable (en, pn, var_list, V_LOG)
    case ("lexpr")
       call eval_node_compile_lexpr (en, pn, var_list)
    case ("block_lexpr")
       call eval_node_compile_block_expr (en, pn, var_list, V_LOG)
    case ("conditional_lexpr")
       call eval_node_compile_conditional (en, pn, var_list, V_LOG)
    case ("compared_expr")
       call eval_node_compile_compared_expr (en, pn, var_list, V_REAL)
    case ("compared_sexpr")
       call eval_node_compile_compared_expr (en, pn, var_list, V_STR)
    case ("all_fun", "any_fun", "no_fun", "user_cut_fun")
       call eval_node_compile_log_function (en, pn, var_list)
    case ("record_cmd")
       call eval_node_compile_record_cmd (en, pn, var_list)
    case default
       call parse_node_mismatch &
            ("true|false|negation|lvariable|" // &
             "lexpr|block_lexpr|conditional_lexpr|" // &
             "compared_expr|compared_sexpr|logical_pexpr", pn)
    end select
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done lvalue"
    end if
  end subroutine eval_node_compile_lvalue
```

A negation consists of the keyword `not` and a logical value.

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_negation (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_arg
    type(eval_node_t), pointer :: en1
    if (debug_active (D_MODEL_F)) then
       print *, "read negation";  call parse_node_write (pn)
    end if
    pn_arg => parse_node_get_sub_ptr (pn, 2)
    call eval_node_compile_lvalue (en1, pn_arg, var_list)
    allocate (en)
    if (en1%type == EN_CONSTANT) then
       call eval_node_init_log (en, not_l (en1))
       call eval_node_final_rec (en1)
       deallocate (en1)
```

```
    else
       call eval_node_init_branch (en, var_str ("not"), V_LOG, en1)
       call eval_node_set_op1_log (en, not_l)
    end if
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done negation"
    end if
  end subroutine eval_node_compile_negation
```

Comparisons

Up to the loop, this is easy. There is always at least one comparison. This is evaluated, and the result is the logical node `en`. If it is constant, we keep its second sub-node as `en2`. (Thus, at the very end `en2` has to be deleted if `en` is (still) constant.)

If there is another comparison, we first check if the first comparison was constant. In that case, there are two possibilities: (i) it was true. Then, its right-hand side is compared with the new right-hand side, and the result replaces the previous one which is deleted. (ii) it was false. In this case, the result of the whole comparison is false, and we can exit the loop without evaluating anything else.

Now assume that the first comparison results in a valid branch, its second sub-node kept as `en2`. We first need a copy of this, which becomes the new left-hand side. If `en2` is constant, we make an identical constant node `en1`. Otherwise, we make `en1` an appropriate pointer node. Next, the first branch is saved as `en0` and we evaluate the comparison between `en1` and the a right-hand side. If this turns out to be constant, there are again two possibilities: (i) true, then we revert to the previous result. (ii) false, then the wh

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_compared_expr (en, pn, var_list, type)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    integer, intent(in) :: type
    type(parse_node_t), pointer :: pn_comparison, pn_expr1
    type(eval_node_t), pointer :: en0, en1, en2
    if (debug_active (D_MODEL_F)) then
       print *, "read comparison";  call parse_node_write (pn)
    end if
    select case (type)
    case (V_INT, V_REAL)
       pn_expr1 => parse_node_get_sub_ptr (pn, tag="expr")
       call eval_node_compile_expr (en1, pn_expr1, var_list)
       pn_comparison => parse_node_get_next_ptr (pn_expr1, tag="comparison")
    case (V_STR)
       pn_expr1 => parse_node_get_sub_ptr (pn, tag="sexpr")
       call eval_node_compile_sexpr (en1, pn_expr1, var_list)
       pn_comparison => parse_node_get_next_ptr (pn_expr1, tag="str_comparison")
    end select
    call eval_node_compile_comparison &
```

```
     (en, en1, en2, pn_comparison, var_list, type)
pn_comparison => parse_node_get_next_ptr (pn_comparison)
SCAN_FURTHER: do while (associated (pn_comparison))
   if (en%type == EN_CONSTANT) then
      if (en%lval) then
         en1 => en2
         call eval_node_final_rec (en);  deallocate (en)
         call eval_node_compile_comparison &
              (en, en1, en2, pn_comparison, var_list, type)
      else
         exit SCAN_FURTHER
      end if
   else
      allocate (en1)
      if (en2%type == EN_CONSTANT) then
         select case (en2%result_type)
         case (V_INT);  call eval_node_init_int    (en1, en2%ival)
         case (V_REAL); call eval_node_init_real   (en1, en2%rval)
         case (V_STR);  call eval_node_init_string (en1, en2%sval)
         end select
      else
         select case (en2%result_type)
         case (V_INT);  call eval_node_init_int_ptr &
              (en1, var_str ("(previous)"), en2%ival, en2%value_is_known)
         case (V_REAL); call eval_node_init_real_ptr &
              (en1, var_str ("(previous)"), en2%rval, en2%value_is_known)
         case (V_STR);  call eval_node_init_string_ptr &
              (en1, var_str ("(previous)"), en2%sval, en2%value_is_known)
         end select
      end if
      en0 => en
      call eval_node_compile_comparison &
           (en, en1, en2, pn_comparison, var_list, type)
      if (en%type == EN_CONSTANT) then
         if (en%lval) then
            call eval_node_final_rec (en);  deallocate (en)
            en => en0
         else
            call eval_node_final_rec (en0);  deallocate (en0)
            exit SCAN_FURTHER
         end if
      else
         en1 => en
         allocate (en)
         call eval_node_init_branch (en, var_str ("and"), V_LOG, en0, en1)
         call eval_node_set_op2_log (en, and_ll)
      end if
   end if
   pn_comparison => parse_node_get_next_ptr (pn_comparison)
end do SCAN_FURTHER
if (en%type == EN_CONSTANT .and. associated (en2)) then
   call eval_node_final_rec (en2);  deallocate (en2)
end if
if (debug_active (D_MODEL_F)) then
```

```
      call eval_node_write (en)
      print *, "done compared_expr"
   end if
end subroutine eval_node_compile_compared_expr
```

This takes two extra arguments: `en1`, the left-hand-side of the comparison, is already allocated and evaluated. `en2` (the right-hand side) and `en` (the result) are allocated by the routine. `pn` is the parse node which contains the operator and the right-hand side as subnodes.

If the result of the comparison is constant, `en1` is deleted but `en2` is kept, because it may be used in a subsequent comparison. `en` then becomes a constant. If the result is variable, `en` becomes a branch node which refers to `en1` and `en2`.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_comparison &
     (en, en1, en2, pn, var_list, type)
  type(eval_node_t), pointer :: en, en1, en2
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  integer, intent(in) :: type
  type(parse_node_t), pointer :: pn_op, pn_arg
  type(string_t) :: key
  integer :: t1, t2
  real(default), pointer :: tolerance_ptr
  pn_op => parse_node_get_sub_ptr (pn)
  key = parse_node_get_key (pn_op)
  select case (type)
  case (V_INT, V_REAL)
     pn_arg => parse_node_get_next_ptr (pn_op, tag="expr")
     call eval_node_compile_expr (en2, pn_arg, var_list)
  case (V_STR)
     pn_arg => parse_node_get_next_ptr (pn_op, tag="sexpr")
     call eval_node_compile_sexpr (en2, pn_arg, var_list)
  end select
  t1 = en1%result_type
  t2 = en2%result_type
  allocate (en)
  if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
     call var_list%get_rptr (var_str ("tolerance"), tolerance_ptr)
     en1%tolerance => tolerance_ptr
     select case (char (key))
     case ("<")
        select case (t1)
        case (V_INT)
           select case (t2)
           case (V_INT);  call eval_node_init_log (en, comp_lt_ii (en1, en2))
           case (V_REAL); call eval_node_init_log (en, comp_ll_ir (en1, en2))
           end select
        case (V_REAL)
           select case (t2)
           case (V_INT);  call eval_node_init_log (en, comp_ll_ri (en1, en2))
           case (V_REAL); call eval_node_init_log (en, comp_ll_rr (en1, en2))
           end select
        end select
```

```
case (">")
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);  call eval_node_init_log (en, comp_gt_ii (en1, en2))
      case (V_REAL); call eval_node_init_log (en, comp_gg_ir (en1, en2))
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);  call eval_node_init_log (en, comp_gg_ri (en1, en2))
      case (V_REAL); call eval_node_init_log (en, comp_gg_rr (en1, en2))
      end select
   end select
case ("<=")
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);  call eval_node_init_log (en, comp_le_ii (en1, en2))
      case (V_REAL); call eval_node_init_log (en, comp_ls_ir (en1, en2))
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);  call eval_node_init_log (en, comp_ls_ri (en1, en2))
      case (V_REAL); call eval_node_init_log (en, comp_ls_rr (en1, en2))
      end select
   end select
case (">=")
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);  call eval_node_init_log (en, comp_ge_ii (en1, en2))
      case (V_REAL); call eval_node_init_log (en, comp_gs_ir (en1, en2))
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);  call eval_node_init_log (en, comp_gs_ri (en1, en2))
      case (V_REAL); call eval_node_init_log (en, comp_gs_rr (en1, en2))
      end select
   end select
case ("==")
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);  call eval_node_init_log (en, comp_eq_ii (en1, en2))
      case (V_REAL); call eval_node_init_log (en, comp_se_ir (en1, en2))
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);  call eval_node_init_log (en, comp_se_ri (en1, en2))
      case (V_REAL); call eval_node_init_log (en, comp_se_rr (en1, en2))
      end select
   case (V_STR)
      select case (t2)
      case (V_STR);  call eval_node_init_log (en, comp_eq_ss (en1, en2))
```

```
         end select
      end select
   case ("<>")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_ne_ii (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_ns_ir (en1, en2))
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_init_log (en, comp_ns_ri (en1, en2))
         case (V_REAL); call eval_node_init_log (en, comp_ns_rr (en1, en2))
         end select
      case (V_STR)
         select case (t2)
         case (V_STR);  call eval_node_init_log (en, comp_ne_ss (en1, en2))
         end select
      end select
   end select
   call eval_node_final_rec (en1)
   deallocate (en1)
else
   call eval_node_init_branch (en, key, V_LOG, en1, en2)
   select case (char (key))
   case ("<")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_set_op2_log (en, comp_lt_ii)
         case (V_REAL); call eval_node_set_op2_log (en, comp_ll_ir)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_set_op2_log (en, comp_ll_ri)
         case (V_REAL); call eval_node_set_op2_log (en, comp_ll_rr)
         end select
      end select
   case (">")
      select case (t1)
      case (V_INT)
         select case (t2)
         case (V_INT);  call eval_node_set_op2_log (en, comp_gt_ii)
         case (V_REAL); call eval_node_set_op2_log (en, comp_gg_ir)
         end select
      case (V_REAL)
         select case (t2)
         case (V_INT);  call eval_node_set_op2_log (en, comp_gg_ri)
         case (V_REAL); call eval_node_set_op2_log (en, comp_gg_rr)
         end select
      end select
   case ("<=")
      select case (t1)
      case (V_INT)
```

```
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_le_ii)
      case (V_REAL); call eval_node_set_op2_log (en, comp_ls_ir)
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_ls_ri)
      case (V_REAL); call eval_node_set_op2_log (en, comp_ls_rr)
      end select
   end select
case (">=")
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_ge_ii)
      case (V_REAL); call eval_node_set_op2_log (en, comp_gs_ir)
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_gs_ri)
      case (V_REAL); call eval_node_set_op2_log (en, comp_gs_rr)
      end select
   end select
case ("==")
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_eq_ii)
      case (V_REAL); call eval_node_set_op2_log (en, comp_se_ir)
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_se_ri)
      case (V_REAL); call eval_node_set_op2_log (en, comp_se_rr)
      end select
   case (V_STR)
      select case (t2)
      case (V_STR);  call eval_node_set_op2_log (en, comp_eq_ss)
      end select
   end select
case ("<>")
   select case (t1)
   case (V_INT)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_ne_ii)
      case (V_REAL); call eval_node_set_op2_log (en, comp_ns_ir)
      end select
   case (V_REAL)
      select case (t2)
      case (V_INT);  call eval_node_set_op2_log (en, comp_ns_ri)
      case (V_REAL); call eval_node_set_op2_log (en, comp_ns_rr)
      end select
   case (V_STR)
      select case (t2)
```

```
            case (V_STR);  call eval_node_set_op2_log (en, comp_ne_ss)
            end select
         end select
      end select
      call var_list%get_rptr (var_str ("tolerance"), tolerance_ptr)
      en1%tolerance => tolerance_ptr
   end if
end subroutine eval_node_compile_comparison
```

Recording analysis data

The `record` command is actually a logical expression which always evaluates `true`.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_record_cmd (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_key, pn_tag, pn_arg
  type(parse_node_t), pointer :: pn_arg1, pn_arg2, pn_arg3, pn_arg4
  type(eval_node_t), pointer :: en0, en1, en2, en3, en4
  real(default), pointer :: event_weight
  if (debug_active (D_MODEL_F)) then
     print *, "read record_cmd";  call parse_node_write (pn)
  end if
  pn_key => parse_node_get_sub_ptr (pn)
  pn_tag => parse_node_get_next_ptr (pn_key)
  pn_arg => parse_node_get_next_ptr (pn_tag)
  select case (char (parse_node_get_key (pn_key)))
  case ("record")
     call var_list%get_rptr (var_str ("event_weight"), event_weight)
  case ("record_unweighted")
     event_weight => null ()
  case ("record_excess")
     call var_list%get_rptr (var_str ("event_excess"), event_weight)
  end select
  select case (char (parse_node_get_rule_key (pn_tag)))
  case ("analysis_id")
     allocate (en0)
     call eval_node_init_string (en0, parse_node_get_string (pn_tag))
  case default
     call eval_node_compile_sexpr (en0, pn_tag, var_list)
  end select
  allocate (en)
  if (associated (pn_arg)) then
     pn_arg1 => parse_node_get_sub_ptr (pn_arg)
     call eval_node_compile_expr (en1, pn_arg1, var_list)
     if (en1%result_type == V_INT) &
          call insert_conversion_node (en1, V_REAL)
     pn_arg2 => parse_node_get_next_ptr (pn_arg1)
     if (associated (pn_arg2)) then
        call eval_node_compile_expr (en2, pn_arg2, var_list)
```

```
         if (en2%result_type == V_INT) &
              call insert_conversion_node (en2, V_REAL)
         pn_arg3 => parse_node_get_next_ptr (pn_arg2)
         if (associated (pn_arg3)) then
            call eval_node_compile_expr (en3, pn_arg3, var_list)
            if (en3%result_type == V_INT) &
                 call insert_conversion_node (en3, V_REAL)
            pn_arg4 => parse_node_get_next_ptr (pn_arg3)
            if (associated (pn_arg4)) then
               call eval_node_compile_expr (en4, pn_arg4, var_list)
               if (en4%result_type == V_INT) &
                    call insert_conversion_node (en4, V_REAL)
               call eval_node_init_record_cmd &
                    (en, event_weight, en0, en1, en2, en3, en4)
            else
               call eval_node_init_record_cmd &
                    (en, event_weight, en0, en1, en2, en3)
            end if
         else
            call eval_node_init_record_cmd (en, event_weight, en0, en1, en2)
         end if
      else
         call eval_node_init_record_cmd (en, event_weight, en0, en1)
      end if
   else
      call eval_node_init_record_cmd (en, event_weight, en0)
   end if
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done record_cmd"
   end if
end subroutine eval_node_compile_record_cmd
```

Particle-list expressions

A particle expression is a subevent or a concatenation of particle-list terms (using `join`).

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_pexpr (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_pterm, pn_concatenation, pn_op, pn_arg
  type(eval_node_t), pointer :: en1, en2
  type(subevt_t) :: subevt
  if (debug_active (D_MODEL_F)) then
     print *, "read pexpr";  call parse_node_write (pn)
  end if
  pn_pterm => parse_node_get_sub_ptr (pn)
  call eval_node_compile_pterm (en, pn_pterm, var_list)
  pn_concatenation => &
       parse_node_get_next_ptr (pn_pterm, tag="pconcatenation")
```

```
    do while (associated (pn_concatenation))
       pn_op => parse_node_get_sub_ptr (pn_concatenation)
       pn_arg => parse_node_get_next_ptr (pn_op)
       en1 => en
       call eval_node_compile_pterm (en2, pn_arg, var_list)
       allocate (en)
       if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
          call subevt_join (subevt, en1%pval, en2%pval)
          call eval_node_init_subevt (en, subevt)
          call eval_node_final_rec (en1)
          call eval_node_final_rec (en2)
          deallocate (en1, en2)
       else
          call eval_node_init_branch &
               (en, var_str ("join"), V_SEV, en1, en2)
          call eval_node_set_op2_sev (en, join_pp)
       end if
       pn_concatenation => parse_node_get_next_ptr (pn_concatenation)
    end do
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done pexpr"
    end if
  end subroutine eval_node_compile_pexpr
```

A particle term is a subevent or a combination of particle-list values (using `combine`).

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_pterm (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_pvalue, pn_combination, pn_op, pn_arg
    type(eval_node_t), pointer :: en1, en2
    type(subevt_t) :: subevt
    if (debug_active (D_MODEL_F)) then
       print *, "read pterm";  call parse_node_write (pn)
    end if
    pn_pvalue => parse_node_get_sub_ptr (pn)
    call eval_node_compile_pvalue (en, pn_pvalue, var_list)
    pn_combination => &
         parse_node_get_next_ptr (pn_pvalue, tag="pcombination")
    do while (associated (pn_combination))
       pn_op => parse_node_get_sub_ptr (pn_combination)
       pn_arg => parse_node_get_next_ptr (pn_op)
       en1 => en
       call eval_node_compile_pvalue (en2, pn_arg, var_list)
       allocate (en)
       if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
          call subevt_combine (subevt, en1%pval, en2%pval)
          call eval_node_init_subevt (en, subevt)
          call eval_node_final_rec (en1)
          call eval_node_final_rec (en2)
```

```
         deallocate (en1, en2)
      else
         call eval_node_init_branch &
              (en, var_str ("combine"), V_SEV, en1, en2)
         call eval_node_set_op2_sev (en, combine_pp)
      end if
      pn_combination => parse_node_get_next_ptr (pn_combination)
   end do
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done pterm"
   end if
end subroutine eval_node_compile_pterm
```

A particle-list value is a PDG-code array, a particle identifier, a variable, a (grouped) pexpr, a block pexpr, a conditional, or a particle-list function.

The `cexpr` node is responsible for transforming a constant PDG-code array into a subevent. It takes the code array as its first argument, the event subevent as its second argument, and the requested particle type (incoming/outgoing) as its zero-th argument. The result is the list of particles in the event that match the code array.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_pvalue (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_prefix_cexpr
  type(eval_node_t), pointer :: en1, en2, en0
  type(string_t) :: key
  type(subevt_t), pointer :: evt_ptr
  logical, pointer :: known
  if (debug_active (D_MODEL_F)) then
     print *, "read pvalue";  call parse_node_write (pn)
  end if
  select case (char (parse_node_get_rule_key (pn)))
  case ("pexpr_src")
     call eval_node_compile_prefix_cexpr (en1, pn, var_list)
     allocate (en2)
     if (var_list%contains (var_str ("@evt"))) then
        call var_list%get_pptr (var_str ("@evt"), evt_ptr, known)
        call eval_node_init_subevt_ptr (en2, var_str ("@evt"), evt_ptr, known)
        allocate (en)
        call eval_node_init_branch &
             (en, var_str ("prt_selection"), V_SEV, en1, en2)
        call eval_node_set_op2_sev (en, select_pdg_ca)
        allocate (en0)
        pn_prefix_cexpr => parse_node_get_sub_ptr (pn)
        key = parse_node_get_rule_key (pn_prefix_cexpr)
        select case (char (key))
        case ("incoming_prt")
           call eval_node_init_int (en0, PRT_INCOMING)
           en%arg0 => en0
        case ("outgoing_prt")
```

```
            call eval_node_init_int (en0, PRT_OUTGOING)
            en%arg0 => en0
         end select
      else
         call parse_node_write (pn)
         call msg_bug (" Missing event data while compiling pvalue")
      end if
   case ("pvariable")
      call eval_node_compile_variable (en, pn, var_list, V_SEV)
   case ("pexpr")
      call eval_node_compile_pexpr (en, pn, var_list)
   case ("block_pexpr")
      call eval_node_compile_block_expr (en, pn, var_list, V_SEV)
   case ("conditional_pexpr")
      call eval_node_compile_conditional (en, pn, var_list, V_SEV)
   case ("join_fun", "combine_fun", "collect_fun", "cluster_fun", &
         "select_fun", "extract_fun", "sort_fun")
      call eval_node_compile_prt_function (en, pn, var_list)
   case default
      call parse_node_mismatch &
           ("prefix_cexpr|pvariable|" // &
            "grouped_pexpr|block_pexpr|conditional_pexpr|" // &
            "prt_function", pn)
   end select
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done pvalue"
   end if
end subroutine eval_node_compile_pvalue
```

Particle functions

This combines the treatment of 'join', 'combine', 'collect', 'cluster', 'select', and 'extract' which all have the same syntax. The one or two argument nodes are allocated. If there is a condition, the condition node is also allocated as a logical expression, for which the variable list is augmented by the appropriate (unary/binary) observables.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_prt_function (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_clause, pn_key, pn_cond, pn_args
  type(parse_node_t), pointer :: pn_arg0, pn_arg1, pn_arg2
  type(eval_node_t), pointer :: en0, en1, en2
  type(string_t) :: key
  if (debug_active (D_MODEL_F)) then
     print *, "read prt_function";  call parse_node_write (pn)
  end if
  pn_clause => parse_node_get_sub_ptr (pn)
  pn_key  => parse_node_get_sub_ptr (pn_clause)
  pn_cond => parse_node_get_next_ptr (pn_key)
```

```
if (associated (pn_cond)) &
     pn_arg0 => parse_node_get_sub_ptr (pn_cond, 2)
pn_args => parse_node_get_next_ptr (pn_clause)
pn_arg1 => parse_node_get_sub_ptr (pn_args)
pn_arg2 => parse_node_get_next_ptr (pn_arg1)
key = parse_node_get_key (pn_key)
call eval_node_compile_pexpr (en1, pn_arg1, var_list)
allocate (en)
if (.not. associated (pn_arg2)) then
   select case (char (key))
   case ("collect")
      call eval_node_init_prt_fun_unary (en, en1, key, collect_p)
   case ("cluster")
      if (fastjet_available ()) then
         call fastjet_init ()
      else
         call msg_fatal &
           ("'cluster' function requires FastJet, which is not enabled")
      end if
      en1%var_list => var_list
      call eval_node_init_prt_fun_unary (en, en1, key, cluster_p)
      call var_list%get_iptr (var_str ("jet_algorithm"), en1%jet_algorithm)
      call var_list%get_rptr (var_str ("jet_r"), en1%jet_r)
      call var_list%get_rptr (var_str ("jet_p"), en1%jet_p)
      call var_list%get_rptr (var_str ("jet_ycut"), en1%jet_ycut)
   case ("select")
      call eval_node_init_prt_fun_unary (en, en1, key, select_p)
   case ("extract")
      call eval_node_init_prt_fun_unary (en, en1, key, extract_p)
   case ("sort")
      call eval_node_init_prt_fun_unary (en, en1, key, sort_p)
   case default
      call msg_bug (" Unary particle function '" // char (key) // &
           "' undefined")
   end select
else
   call eval_node_compile_pexpr (en2, pn_arg2, var_list)
   select case (char (key))
   case ("join")
      call eval_node_init_prt_fun_binary (en, en1, en2, key, join_pp)
   case ("combine")
      call eval_node_init_prt_fun_binary (en, en1, en2, key, combine_pp)
   case ("collect")
      call eval_node_init_prt_fun_binary (en, en1, en2, key, collect_pp)
   case ("select")
      call eval_node_init_prt_fun_binary (en, en1, en2, key, select_pp)
   case ("sort")
      call eval_node_init_prt_fun_binary (en, en1, en2, key, sort_pp)
   case default
      call msg_bug (" Binary particle function '" // char (key) // &
           "' undefined")
   end select
end if
if (associated (pn_cond)) then
```

```
      call eval_node_set_observables (en, var_list)
      select case (char (key))
      case ("extract", "sort")
         call eval_node_compile_expr (en0, pn_arg0, en%var_list)
      case default
         call eval_node_compile_lexpr (en0, pn_arg0, en%var_list)
      end select
      en%arg0 => en0
   end if
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done prt_function"
   end if
end subroutine eval_node_compile_prt_function
```

The `eval` expression is similar, but here the expression `arg0` is mandatory, and the whole thing evaluates to a numeric value.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_eval_function (en, pn, var_list)
   type(eval_node_t), pointer :: en
   type(parse_node_t), intent(in) :: pn
   type(var_list_t), intent(in), target :: var_list
   type(parse_node_t), pointer :: pn_key, pn_arg0, pn_args, pn_arg1, pn_arg2
   type(eval_node_t), pointer :: en0, en1, en2
   type(string_t) :: key
   if (debug_active (D_MODEL_F)) then
      print *, "read eval_function";  call parse_node_write (pn)
   end if
   pn_key => parse_node_get_sub_ptr (pn)
   pn_arg0 => parse_node_get_next_ptr (pn_key)
   pn_args => parse_node_get_next_ptr (pn_arg0)
   pn_arg1 => parse_node_get_sub_ptr (pn_args)
   pn_arg2 => parse_node_get_next_ptr (pn_arg1)
   key = parse_node_get_key (pn_key)
   call eval_node_compile_pexpr (en1, pn_arg1, var_list)
   allocate (en)
   if (.not. associated (pn_arg2)) then
      call eval_node_init_eval_fun_unary (en, en1, key)
   else
      call eval_node_compile_pexpr (en2, pn_arg2, var_list)
      call eval_node_init_eval_fun_binary (en, en1, en2, key)
   end if
   call eval_node_set_observables (en, var_list)
   call eval_node_compile_expr (en0, pn_arg0, en%var_list)
   if (en0%result_type /= V_REAL) &
        call msg_fatal (" 'eval' function does not result in real value")
   call eval_node_set_expr (en, en0)
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done eval_function"
   end if
end subroutine eval_node_compile_eval_function
```

Logical functions of subevents.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_log_function (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in) :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_key, pn_str
  type(parse_node_t), pointer :: pn_arg0, pn_args, pn_arg1, pn_arg2
  type(eval_node_t), pointer :: en0, en1, en2
  type(string_t) :: key
  if (debug_active (D_MODEL_F)) then
     print *, "read log_function";  call parse_node_write (pn)
  end if
  select case (char (parse_node_get_rule_key (pn)))
  case ("all_fun", "any_fun", "no_fun")
     pn_key => parse_node_get_sub_ptr (pn)
     pn_arg0 => parse_node_get_next_ptr (pn_key)
     pn_args => parse_node_get_next_ptr (pn_arg0)
  case ("user_cut_fun")
     pn_key => parse_node_get_sub_ptr (pn)
     pn_str => parse_node_get_next_ptr (pn_key)
     pn_arg0 => parse_node_get_sub_ptr (pn_str)
     pn_args => parse_node_get_next_ptr (pn_str)
  case default
     call parse_node_mismatch &
          ("all_fun|any_fun|no_fun|user_cut_fun", &
          pn)
  end select
  pn_arg1 => parse_node_get_sub_ptr (pn_args)
  pn_arg2 => parse_node_get_next_ptr (pn_arg1)
  key = parse_node_get_key (pn_key)
  call eval_node_compile_pexpr (en1, pn_arg1, var_list)
  allocate (en)
  if (.not. associated (pn_arg2)) then
     select case (char (key))
     case ("all")
        call eval_node_init_log_fun_unary (en, en1, key, all_p)
     case ("any")
        call eval_node_init_log_fun_unary (en, en1, key, any_p)
     case ("no")
        call eval_node_init_log_fun_unary (en, en1, key, no_p)
     case ("user_cut")
        call eval_node_init_log_fun_unary (en, en1, key, user_cut_p)
     case default
        call msg_bug ("Unary logical particle function '" // char (key) // &
             "' undefined")
     end select
  else
     call eval_node_compile_pexpr (en2, pn_arg2, var_list)
     select case (char (key))
     case ("all")
        call eval_node_init_log_fun_binary (en, en1, en2, key, all_pp)
     case ("any")
        call eval_node_init_log_fun_binary (en, en1, en2, key, any_pp)
```

```
      case ("no")
         call eval_node_init_log_fun_binary (en, en1, en2, key, no_pp)
      case default
         call msg_bug ("Binary logical particle function '" // char (key) // &
              "' undefined")
      end select
   end if
   if (associated (pn_arg0)) then
      call eval_node_set_observables (en, var_list)
      select case (char (key))
      case ("all", "any", "no")
         call eval_node_compile_lexpr (en0, pn_arg0, en%var_list)
      case ("user_cut")
         call eval_node_compile_sexpr (en0, pn_arg0, en%var_list)
      case default
         call msg_bug ("Compiling logical particle function: missing mode")
      end select
      call eval_node_set_expr (en, en0, V_LOG)
   end if
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done log_function"
   end if
end subroutine eval_node_compile_log_function
```

Numeric functions of subevents.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_numeric_function (en, pn, var_list)
   type(eval_node_t), pointer :: en
   type(parse_node_t), intent(in) :: pn
   type(var_list_t), intent(in), target :: var_list
   type(parse_node_t), pointer :: pn_clause, pn_key, pn_cond, pn_args
   type(parse_node_t), pointer :: pn_arg0, pn_arg1, pn_arg2
   type(eval_node_t), pointer :: en0, en1, en2
   type(string_t) :: key
   if (debug_active (D_MODEL_F)) then
      print *, "read numeric_function";  call parse_node_write (pn)
   end if
   select case (char (parse_node_get_rule_key (pn)))
   case ("count_fun")
      pn_clause => parse_node_get_sub_ptr (pn)
      pn_key  => parse_node_get_sub_ptr (pn_clause)
      pn_cond => parse_node_get_next_ptr (pn_key)
      if (associated (pn_cond)) then
         pn_arg0 => parse_node_get_sub_ptr (pn_cond, 2)
      else
         pn_arg0 => null ()
      end if
      pn_args => parse_node_get_next_ptr (pn_clause)
   case ("user_event_fun")
      pn_key => parse_node_get_sub_ptr (pn)
      pn_cond => parse_node_get_next_ptr (pn_key)
      pn_arg0 => parse_node_get_sub_ptr (pn_cond)
      pn_args => parse_node_get_next_ptr (pn_cond)
```

```
    end select
    pn_arg1 => parse_node_get_sub_ptr (pn_args)
    pn_arg2 => parse_node_get_next_ptr (pn_arg1)
    key = parse_node_get_key (pn_key)
    call eval_node_compile_pexpr (en1, pn_arg1, var_list)
    allocate (en)
    if (.not. associated (pn_arg2)) then
       select case (char (key))
       case ("count")
          call eval_node_init_int_fun_unary (en, en1, key, count_a)
       case ("user_event_shape")
          call eval_node_init_real_fun_unary (en, en1, key, user_event_shape_a)
       case default
          call msg_bug ("Unary subevent function '" // char (key) // &
               "' undefined")
       end select
    else
       call eval_node_compile_pexpr (en2, pn_arg2, var_list)
       select case (char (key))
       case ("count")
          call eval_node_init_int_fun_binary (en, en1, en2, key, count_pp)
       case default
          call msg_bug ("Binary subevent function '" // char (key) // &
               "' undefined")
       end select
    end if
    if (associated (pn_arg0)) then
       call eval_node_set_observables (en, var_list)
       select case (char (key))
       case ("count")
          call eval_node_compile_lexpr (en0, pn_arg0, en%var_list)
          call eval_node_set_expr (en, en0, V_INT)
       case ("user_event_shape")
          call eval_node_compile_sexpr (en0, pn_arg0, en%var_list)
          call eval_node_set_expr (en, en0, V_REAL)
       end select
    end if
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done numeric_function"
    end if
  end subroutine eval_node_compile_numeric_function
```

PDG-code arrays

A PDG-code expression is either prefixed by `incoming` or `outgoing`, a block, or a conditional. In any case, it evaluates to a constant.

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_prefix_cexpr (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
```

```
    type(parse_node_t), pointer :: pn_avalue, pn_prt
    type(string_t) :: key
    if (debug_active (D_MODEL_F)) then
       print *, "read prefix_cexpr";  call parse_node_write (pn)
    end if
    pn_avalue => parse_node_get_sub_ptr (pn)
    key = parse_node_get_rule_key (pn_avalue)
    select case (char (key))
    case ("incoming_prt")
       pn_prt => parse_node_get_sub_ptr (pn_avalue, 2)
       call eval_node_compile_cexpr (en, pn_prt, var_list)
    case ("outgoing_prt")
       pn_prt => parse_node_get_sub_ptr (pn_avalue, 1)
       call eval_node_compile_cexpr (en, pn_prt, var_list)
    case default
       call parse_node_mismatch &
            ("incoming_prt|outgoing_prt", &
             pn_avalue)
    end select
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done prefix_cexpr"
    end if
  end subroutine eval_node_compile_prefix_cexpr
```

A PDG array is a string of PDG code definitions (or aliases), concatenated by ':'. The code definitions may be variables which are not defined at compile time, so we have to allocate sub-nodes. This analogous to `eval_node_compile_term`.

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_cexpr (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_prt, pn_concatenation
    type(eval_node_t), pointer :: en1, en2
    type(pdg_array_t) :: aval
    if (debug_active (D_MODEL_F)) then
       print *, "read cexpr";  call parse_node_write (pn)
    end if
    pn_prt => parse_node_get_sub_ptr (pn)
    call eval_node_compile_avalue (en, pn_prt, var_list)
    pn_concatenation => parse_node_get_next_ptr (pn_prt)
    do while (associated (pn_concatenation))
       pn_prt => parse_node_get_sub_ptr (pn_concatenation, 2)
       en1 => en
       call eval_node_compile_avalue (en2, pn_prt, var_list)
       allocate (en)
       if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
          call concat_cc (aval, en1, en2)
          call eval_node_init_pdg_array (en, aval)
          call eval_node_final_rec (en1)
          call eval_node_final_rec (en2)
          deallocate (en1, en2)
```

```
      else
         call eval_node_init_branch (en, var_str (":"), V_PDG, en1, en2)
         call eval_node_set_op2_pdg (en, concat_cc)
      end if
      pn_concatenation => parse_node_get_next_ptr (pn_concatenation)
   end do
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done cexpr"
   end if
end subroutine eval_node_compile_cexpr
```

Compile a PDG-code type value. It may be either an integer expression or a variable of type PDG array, optionally quoted.

⟨*Eval trees: procedures*⟩+≡

```
recursive subroutine eval_node_compile_avalue (en, pn, var_list)
   type(eval_node_t), pointer :: en
   type(parse_node_t), intent(in) :: pn
   type(var_list_t), intent(in), target :: var_list
   if (debug_active (D_MODEL_F)) then
      print *, "read avalue";  call parse_node_write (pn)
   end if
   select case (char (parse_node_get_rule_key (pn)))
   case ("pdg_code")
      call eval_node_compile_pdg_code (en, pn, var_list)
   case ("cvariable", "variable", "prt_name")
      call eval_node_compile_cvariable (en, pn, var_list)
   case ("cexpr")
      call eval_node_compile_cexpr (en, pn, var_list)
   case ("block_cexpr")
      call eval_node_compile_block_expr (en, pn, var_list, V_PDG)
   case ("conditional_cexpr")
      call eval_node_compile_conditional (en, pn, var_list, V_PDG)
   case default
      call parse_node_mismatch &
           ("grouped_cexpr|block_cexpr|conditional_cexpr|" // &
            "pdg_code|cvariable|prt_name", pn)
   end select
   if (debug_active (D_MODEL_F)) then
      call eval_node_write (en)
      print *, "done avalue"
   end if
end subroutine eval_node_compile_avalue
```

Compile a PDG-code expression, which is the key `PDG` with an integer expression as argument. The procedure is analogous to `eval_node_compile_unary_function`.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_compile_pdg_code (en, pn, var_list)
   type(eval_node_t), pointer :: en
   type(parse_node_t), intent(in), target :: pn
   type(var_list_t), intent(in), target :: var_list
   type(parse_node_t), pointer :: pn_arg
   type(eval_node_t), pointer :: en1
```

```
  type(string_t) :: key
  type(pdg_array_t) :: aval
  integer :: t
  if (debug_active (D_MODEL_F)) then
     print *, "read PDG code";  call parse_node_write (pn)
  end if
  pn_arg => parse_node_get_sub_ptr (pn, 2)
  call eval_node_compile_expr &
       (en1, parse_node_get_sub_ptr (pn_arg, tag="expr"), var_list)
  t = en1%result_type
  allocate (en)
  key = "PDG"
  if (en1%type == EN_CONSTANT) then
     select case (t)
     case (V_INT)
        call pdg_i (aval, en1)
        call eval_node_init_pdg_array (en, aval)
     case default;  call eval_type_error (pn, char (key), t)
     end select
     call eval_node_final_rec (en1)
     deallocate (en1)
  else
     select case (t)
     case (V_INT);  call eval_node_set_op1_pdg (en, pdg_i)
     case default;  call eval_type_error (pn, char (key), t)
     end select
  end if
  if (debug_active (D_MODEL_F)) then
     call eval_node_write (en)
     print *, "done function"
  end if
end subroutine eval_node_compile_pdg_code
```

This is entirely analogous to `eval_node_compile_variable`. However, PDG-array variables occur in different contexts.

To avoid name clashes between PDG-array variables and ordinary variables, we prepend a character (*). This is not visible to the user.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_compile_cvariable (en, pn, var_list)
  type(eval_node_t), pointer :: en
  type(parse_node_t), intent(in), target :: pn
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_name
  type(string_t) :: var_name
  type(pdg_array_t), pointer :: aptr
  type(pdg_array_t), target, save :: no_aval
  logical, pointer :: known
  logical, target, save :: unknown = .false.
  if (debug_active (D_MODEL_F)) then
     print *, "read cvariable";  call parse_node_write (pn)
  end if
  pn_name => pn
  var_name = parse_node_get_string (pn_name)
```

```
    allocate (en)
    if (var_list%contains (var_name)) then
       call var_list%get_aptr (var_name, aptr, known)
       call eval_node_init_pdg_array_ptr (en, var_name, aptr, known)
    else
       call parse_node_write (pn)
       call msg_error ("This PDG-array variable is undefined at this point")
       call eval_node_init_pdg_array_ptr (en, var_name, no_aval, unknown)
    end if
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done cvariable"
    end if
  end subroutine eval_node_compile_cvariable
```

String expressions

A string expression is either a string value or a concatenation of string values.

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_sexpr (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_svalue, pn_concatenation, pn_op, pn_arg
    type(eval_node_t), pointer :: en1, en2
    type(string_t) :: string
    if (debug_active (D_MODEL_F)) then
       print *, "read sexpr";  call parse_node_write (pn)
    end if
    pn_svalue => parse_node_get_sub_ptr (pn)
    call eval_node_compile_svalue (en, pn_svalue, var_list)
    pn_concatenation => &
         parse_node_get_next_ptr (pn_svalue, tag="str_concatenation")
    do while (associated (pn_concatenation))
       pn_op => parse_node_get_sub_ptr (pn_concatenation)
       pn_arg => parse_node_get_next_ptr (pn_op)
       en1 => en
       call eval_node_compile_svalue (en2, pn_arg, var_list)
       allocate (en)
       if (en1%type == EN_CONSTANT .and. en2%type == EN_CONSTANT) then
          call concat_ss (string, en1, en2)
          call eval_node_init_string (en, string)
          call eval_node_final_rec (en1)
          call eval_node_final_rec (en2)
          deallocate (en1, en2)
       else
          call eval_node_init_branch &
               (en, var_str ("concat"), V_STR, en1, en2)
          call eval_node_set_op2_str (en, concat_ss)
       end if
       pn_concatenation => parse_node_get_next_ptr (pn_concatenation)
    end do
    if (debug_active (D_MODEL_F)) then
```

```
      call eval_node_write (en)
      print *, "done sexpr"
    end if
  end subroutine eval_node_compile_sexpr
```

A string value is a string literal, a variable, a (grouped) sexpr, a block sexpr, or a conditional.

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_svalue (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    if (debug_active (D_MODEL_F)) then
       print *, "read svalue";  call parse_node_write (pn)
    end if
    select case (char (parse_node_get_rule_key (pn)))
    case ("svariable")
       call eval_node_compile_variable (en, pn, var_list, V_STR)
    case ("sexpr")
       call eval_node_compile_sexpr (en, pn, var_list)
    case ("block_sexpr")
       call eval_node_compile_block_expr (en, pn, var_list, V_STR)
    case ("conditional_sexpr")
       call eval_node_compile_conditional (en, pn, var_list, V_STR)
    case ("sprintf_fun")
       call eval_node_compile_sprintf (en, pn, var_list)
    case ("string_literal")
       allocate (en)
       call eval_node_init_string (en, parse_node_get_string (pn))
    case default
       call parse_node_mismatch &
            ("svariable|" // &
             "grouped_sexpr|block_sexpr|conditional_sexpr|" // &
             "string_function|string_literal", pn)
    end select
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done svalue"
    end if
  end subroutine eval_node_compile_svalue
```

There is currently one string function, `sprintf`. For `sprintf`, the first argument (no brackets) is the format string, the optional arguments in brackets are the expressions or variables to be formatted.

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_compile_sprintf (en, pn, var_list)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    type(parse_node_t), pointer :: pn_clause, pn_key, pn_args
    type(parse_node_t), pointer :: pn_arg0
    type(eval_node_t), pointer :: en0, en1
    integer :: n_args
```

```
    type(string_t) :: key
    if (debug_active (D_MODEL_F)) then
       print *, "read sprintf_fun";  call parse_node_write (pn)
    end if
    pn_clause => parse_node_get_sub_ptr (pn)
    pn_key  => parse_node_get_sub_ptr (pn_clause)
    pn_arg0 => parse_node_get_next_ptr (pn_key)
    pn_args => parse_node_get_next_ptr (pn_clause)
    call eval_node_compile_sexpr (en0, pn_arg0, var_list)
    if (associated (pn_args)) then
       call eval_node_compile_sprintf_args (en1, pn_args, var_list, n_args)
    else
       n_args = 0
       en1 => null ()
    end if
    allocate (en)
    key = parse_node_get_key (pn_key)
    call eval_node_init_format_string (en, en0, en1, key, n_args)
    if (debug_active (D_MODEL_F)) then
       call eval_node_write (en)
       print *, "done sprintf_fun"
    end if
  end subroutine eval_node_compile_sprintf
```

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_node_compile_sprintf_args (en, pn, var_list, n_args)
    type(eval_node_t), pointer :: en
    type(parse_node_t), intent(in) :: pn
    type(var_list_t), intent(in), target :: var_list
    integer, intent(out) :: n_args
    type(parse_node_t), pointer :: pn_arg
    integer :: i
    type(eval_node_t), pointer :: en1, en2
    n_args = parse_node_get_n_sub (pn)
    en => null ()
    do i = n_args, 1, -1
       pn_arg => parse_node_get_sub_ptr (pn, i)
       select case (char (parse_node_get_rule_key (pn_arg)))
       case ("lvariable")
          call eval_node_compile_variable (en1, pn_arg, var_list, V_LOG)
       case ("svariable")
          call eval_node_compile_variable (en1, pn_arg, var_list, V_STR)
       case ("expr")
          call eval_node_compile_expr (en1, pn_arg, var_list)
       case default
          call parse_node_mismatch ("variable|svariable|lvariable|expr", pn_arg)
       end select
       if (associated (en)) then
          en2 => en
          allocate (en)
          call eval_node_init_branch &
               (en, var_str ("sprintf_arg"), V_NONE, en1, en2)
       else
          allocate (en)
```

```
            call eval_node_init_branch &
                 (en, var_str ("sprintf_arg"), V_NONE, en1)
         end if
      end do
   end subroutine eval_node_compile_sprintf_args
```

Evaluation. We allocate the argument list and apply the Fortran wrapper for the `sprintf` function.

⟨*Eval trees: procedures*⟩+≡

```
   subroutine evaluate_sprintf (string, n_args, en_fmt, en_arg)
     type(string_t), intent(out) :: string
     integer, intent(in) :: n_args
     type(eval_node_t), pointer :: en_fmt
     type(eval_node_t), intent(in), optional, target :: en_arg
     type(eval_node_t), pointer :: en_branch, en_var
     type(sprintf_arg_t), dimension(:), allocatable :: arg
     type(string_t) :: fmt
     logical :: autoformat
     integer :: i, j, sprintf_argc
     autoformat = .not. associated (en_fmt)
     if (autoformat) fmt = ""
     if (present (en_arg)) then
        sprintf_argc = 0
        en_branch => en_arg
        do i = 1, n_args
           select case (en_branch%arg1%result_type)
              case (V_CMPLX); sprintf_argc = sprintf_argc + 2
              case default  ; sprintf_argc = sprintf_argc + 1
           end select
           en_branch => en_branch%arg2
        end do
        allocate (arg (sprintf_argc))
        j = 1
        en_branch => en_arg
        do i = 1, n_args
           en_var => en_branch%arg1
           select case (en_var%result_type)
           case (V_LOG)
              call sprintf_arg_init (arg(j), en_var%lval)
              if (autoformat) fmt = fmt // "%s "
           case (V_INT);
              call sprintf_arg_init (arg(j), en_var%ival)
              if (autoformat) fmt = fmt // "%i "
           case (V_REAL);
              call sprintf_arg_init (arg(j), en_var%rval)
              if (autoformat) fmt = fmt // "%g "
           case (V_STR)
              call sprintf_arg_init (arg(j), en_var%sval)
              if (autoformat) fmt = fmt // "%s "
           case (V_CMPLX)
              call sprintf_arg_init (arg(j), real (en_var%cval, default))
              j = j + 1
              call sprintf_arg_init (arg(j), aimag (en_var%cval))
```

```
            if (autoformat) fmt = fmt // "(%g + %g * I) "
         case default
            call eval_node_write (en_var)
            call msg_error ("sprintf is implemented " &
                 // "for logical, integer, real, and string values only")
         end select
         j = j + 1
         en_branch => en_branch%arg2
      end do
   else
      allocate (arg(0))
   end if
   if (autoformat) then
      string = sprintf (trim (fmt), arg)
   else
      string = sprintf (en_fmt%sval, arg)
   end if
end subroutine evaluate_sprintf
```

28.3.5 Auxiliary functions for the compiler

Issue an error that the current node could not be compiled because of type mismatch:

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_type_error (pn, string, t)
  type(parse_node_t), intent(in) :: pn
  character(*), intent(in) :: string
  integer, intent(in) :: t
  type(string_t) :: type
  select case (t)
  case (V_NONE); type = "(none)"
  case (V_LOG);  type = "'logical'"
  case (V_INT);  type = "'integer'"
  case (V_REAL); type = "'real'"
  case (V_CMPLX); type = "'complex'"
  case default;  type = "(unknown)"
  end select
  call parse_node_write (pn)
  call msg_fatal (" The " // string // &
       " operation is not defined for the given argument type " // &
       char (type))
end subroutine eval_type_error
```

If two numerics are combined, the result is integer if both arguments are integer, if one is integer and the other real or both are real, than its argument is real, otherwise complex.

⟨*Eval trees: procedures*⟩+≡

```
function numeric_result_type (t1, t2) result (t)
  integer, intent(in) :: t1, t2
  integer :: t
  if (t1 == V_INT .and. t2 == V_INT) then
     t = V_INT
```

```
    else if (t1 == V_INT .and. t2 == V_REAL) then
       t = V_REAL
    else if (t1 == V_REAL .and. t2 == V_INT) then
       t = V_REAL
    else if (t1 == V_REAL .and. t2 == V_REAL) then
       t = V_REAL
    else
       t = V_CMPLX
    end if
  end function numeric_result_type
```

28.3.6 Evaluation

Evaluation is done recursively. For leaf nodes nothing is to be done.

Evaluating particle-list functions: First, we evaluate the particle lists. If a condition is present, we assign the particle pointers of the condition node to the allocated particle entries in the parent node, keeping in mind that the observables in the variable stack used for the evaluation of the condition also contain pointers to these entries. Then, the assigned procedure is evaluated, which sets the subevent in the parent node. If required, the procedure evaluates the condition node once for each (pair of) particles to determine the result.

⟨*Eval trees: procedures*⟩+≡

```
  recursive subroutine eval_node_evaluate (en)
    type(eval_node_t), intent(inout) :: en
    logical :: exist
    select case (en%type)
    case (EN_UNARY)
       if (associated (en%arg1)) then
          call eval_node_evaluate (en%arg1)
          en%value_is_known = en%arg1%value_is_known
       else
          en%value_is_known = .false.
       end if
       if (en%value_is_known) then
          select case (en%result_type)
          case (V_LOG);  en%lval = en%op1_log  (en%arg1)
          case (V_INT);  en%ival = en%op1_int  (en%arg1)
          case (V_REAL); en%rval = en%op1_real (en%arg1)
          case (V_CMPLX); en%cval = en%op1_cmplx (en%arg1)
          case (V_PDG);
             call en%op1_pdg  (en%aval, en%arg1)
          case (V_SEV)
             if (associated (en%arg0)) then
                call en%op1_sev (en%pval, en%arg1, en%arg0)
             else
                call en%op1_sev (en%pval, en%arg1)
             end if
          case (V_STR)
             call en%op1_str (en%sval, en%arg1)
          end select
       end if
    case (EN_BINARY)
```

```
   if (associated (en%arg1) .and. associated (en%arg2)) then
      call eval_node_evaluate (en%arg1)
      call eval_node_evaluate (en%arg2)
      en%value_is_known = &
           en%arg1%value_is_known .and. en%arg2%value_is_known
   else
      en%value_is_known = .false.
   end if
   if (en%value_is_known) then
      select case (en%result_type)
      case (V_LOG);  en%lval = en%op2_log  (en%arg1, en%arg2)
      case (V_INT);  en%ival = en%op2_int  (en%arg1, en%arg2)
      case (V_REAL); en%rval = en%op2_real (en%arg1, en%arg2)
      case (V_CMPLX); en%cval = en%op2_cmplx (en%arg1, en%arg2)
      case (V_PDG)
         call en%op2_pdg  (en%aval, en%arg1, en%arg2)
      case (V_SEV)
         if (associated (en%arg0)) then
            call en%op2_sev (en%pval, en%arg1, en%arg2, en%arg0)
         else
            call en%op2_sev (en%pval, en%arg1, en%arg2)
         end if
      case (V_STR)
         call en%op2_str (en%sval, en%arg1, en%arg2)
      end select
   end if
case (EN_BLOCK)
   if (associated (en%arg1) .and. associated (en%arg0)) then
      call eval_node_evaluate (en%arg1)
      call eval_node_evaluate (en%arg0)
      en%value_is_known = en%arg0%value_is_known
   else
      en%value_is_known = .false.
   end if
   if (en%value_is_known) then
      select case (en%result_type)
      case (V_LOG);  en%lval = en%arg0%lval
      case (V_INT);  en%ival = en%arg0%ival
      case (V_REAL); en%rval = en%arg0%rval
      case (V_CMPLX); en%cval = en%arg0%cval
      case (V_PDG);  en%aval = en%arg0%aval
      case (V_SEV);  en%pval = en%arg0%pval
      case (V_STR);  en%sval = en%arg0%sval
      end select
   end if
case (EN_CONDITIONAL)
   if (associated (en%arg0)) then
      call eval_node_evaluate (en%arg0)
      en%value_is_known = en%arg0%value_is_known
   else
      en%value_is_known = .false.
   end if
   if (en%arg0%value_is_known) then
      if (en%arg0%lval) then
```

```
         call eval_node_evaluate (en%arg1)
         en%value_is_known = en%arg1%value_is_known
         if (en%value_is_known) then
            select case (en%result_type)
            case (V_LOG);  en%lval = en%arg1%lval
            case (V_INT);  en%ival = en%arg1%ival
            case (V_REAL); en%rval = en%arg1%rval
            case (V_CMPLX); en%cval = en%arg1%cval
            case (V_PDG);  en%aval = en%arg1%aval
            case (V_SEV);  en%pval = en%arg1%pval
            case (V_STR);  en%sval = en%arg1%sval
            end select
         end if
      else
         call eval_node_evaluate (en%arg2)
         en%value_is_known = en%arg2%value_is_known
         if (en%value_is_known) then
            select case (en%result_type)
            case (V_LOG);  en%lval = en%arg2%lval
            case (V_INT);  en%ival = en%arg2%ival
            case (V_REAL); en%rval = en%arg2%rval
            case (V_CMPLX); en%cval = en%arg2%cval
            case (V_PDG);  en%aval = en%arg2%aval
            case (V_SEV);  en%pval = en%arg2%pval
            case (V_STR);  en%sval = en%arg2%sval
            end select
         end if
      end if
   end if
case (EN_RECORD_CMD)
   exist = .true.
   en%lval = .false.
   call eval_node_evaluate (en%arg0)
   if (en%arg0%value_is_known) then
      if (associated (en%arg1)) then
         call eval_node_evaluate (en%arg1)
         if (en%arg1%value_is_known) then
            if (associated (en%arg2)) then
               call eval_node_evaluate (en%arg2)
               if (en%arg2%value_is_known) then
                  if (associated (en%arg3)) then
                     call eval_node_evaluate (en%arg3)
                     if (en%arg3%value_is_known) then
                        if (associated (en%arg4)) then
                           call eval_node_evaluate (en%arg4)
                           if (en%arg4%value_is_known) then
                              if (associated (en%rval)) then
                                 call analysis_record_data (en%arg0%sval, &
                                      en%arg1%rval, en%arg2%rval, &
                                      en%arg3%rval, en%arg4%rval, &
                                      weight=en%rval, exist=exist, &
                                      success=en%lval)
                              else
                                 call analysis_record_data (en%arg0%sval, &
```

```
                                    en%arg1%rval, en%arg2%rval, &
                                    en%arg3%rval, en%arg4%rval, &
                                    exist=exist, success=en%lval)
                            end if
                         end if
                      else
                         if (associated (en%rval)) then
                            call analysis_record_data (en%arg0%sval, &
                                 en%arg1%rval, en%arg2%rval, &
                                 en%arg3%rval, &
                                 weight=en%rval, exist=exist, &
                                 success=en%lval)
                         else
                            call analysis_record_data (en%arg0%sval, &
                                 en%arg1%rval, en%arg2%rval, &
                                 en%arg3%rval, &
                                 exist=exist, success=en%lval)
                         end if
                      end if
                   end if
                else
                   if (associated (en%rval)) then
                      call analysis_record_data (en%arg0%sval, &
                           en%arg1%rval, en%arg2%rval, &
                           weight=en%rval, exist=exist, &
                           success=en%lval)
                   else
                      call analysis_record_data (en%arg0%sval, &
                           en%arg1%rval, en%arg2%rval, &
                           exist=exist, success=en%lval)
                   end if
                end if
             end if
          else
             if (associated (en%rval)) then
                call analysis_record_data (en%arg0%sval, &
                     en%arg1%rval, &
                     weight=en%rval, exist=exist, success=en%lval)
             else
                call analysis_record_data (en%arg0%sval, &
                     en%arg1%rval, &
                     exist=exist, success=en%lval)
             end if
          end if
       end if
    else
       if (associated (en%rval)) then
          call analysis_record_data (en%arg0%sval, 1._default, &
               weight=en%rval, exist=exist, success=en%lval)
       else
          call analysis_record_data (en%arg0%sval, 1._default, &
               exist=exist, success=en%lval)
       end if
    end if
```

```
      if (.not. exist) then
         call msg_error ("Analysis object '" // char (en%arg0%sval) &
              // "' is undefined")
         en%arg0%value_is_known = .false.
      end if
   end if
case (EN_OBS1_INT)
   en%ival = en%obs1_int (en%prt1)
   en%value_is_known = .true.
case (EN_OBS2_INT)
   en%ival = en%obs2_int (en%prt1, en%prt2)
   en%value_is_known = .true.
case (EN_OBS1_REAL)
   en%rval = en%obs1_real (en%prt1)
   en%value_is_known = .true.
case (EN_OBS2_REAL)
   en%rval = en%obs2_real (en%prt1, en%prt2)
   en%value_is_known = .true.
case (EN_UOBS1_INT)
   en%ival = user_obs_int_p (en%arg0, en%prt1)
   en%value_is_known = .true.
case (EN_UOBS2_INT)
   en%ival = user_obs_int_pp (en%arg0, en%prt1, en%prt2)
   en%value_is_known = .true.
case (EN_UOBS1_REAL)
   en%rval = user_obs_real_p (en%arg0, en%prt1)
   en%value_is_known = .true.
case (EN_UOBS2_REAL)
   en%rval = user_obs_real_pp (en%arg0, en%prt1, en%prt2)
   en%value_is_known = .true.
case (EN_PRT_FUN_UNARY)
   call eval_node_evaluate (en%arg1)
   en%value_is_known = en%arg1%value_is_known
   if (en%value_is_known) then
      if (associated (en%arg0)) then
         en%arg0%index => en%index
         en%arg0%prt1 => en%prt1
         call en%op1_sev (en%pval, en%arg1, en%arg0)
      else
         call en%op1_sev (en%pval, en%arg1)
      end if
   end if
case (EN_PRT_FUN_BINARY)
   call eval_node_evaluate (en%arg1)
   call eval_node_evaluate (en%arg2)
   en%value_is_known = &
        en%arg1%value_is_known .and. en%arg2%value_is_known
   if (en%value_is_known) then
      if (associated (en%arg0)) then
         en%arg0%index => en%index
         en%arg0%prt1 => en%prt1
         en%arg0%prt2 => en%prt2
         call en%op2_sev (en%pval, en%arg1, en%arg2, en%arg0)
      else
```

```
         call en%op2_sev (en%pval, en%arg1, en%arg2)
      end if
   end if
case (EN_EVAL_FUN_UNARY)
   call eval_node_evaluate (en%arg1)
   en%value_is_known = subevt_is_nonempty (en%arg1%pval)
   if (en%value_is_known) then
      en%arg0%index => en%index
      en%index = 1
      en%arg0%prt1 => en%prt1
      en%prt1 = subevt_get_prt (en%arg1%pval, 1)
      call eval_node_evaluate (en%arg0)
      en%rval = en%arg0%rval
   end if
case (EN_EVAL_FUN_BINARY)
   call eval_node_evaluate (en%arg1)
   call eval_node_evaluate (en%arg2)
   en%value_is_known = &
        subevt_is_nonempty (en%arg1%pval) .and. &
        subevt_is_nonempty (en%arg2%pval)
   if (en%value_is_known) then
      en%arg0%index => en%index
      en%arg0%prt1 => en%prt1
      en%arg0%prt2 => en%prt2
      en%index = 1
      call eval_pp (en%arg1, en%arg2, en%arg0, en%rval, en%value_is_known)
   end if
case (EN_LOG_FUN_UNARY)
   call eval_node_evaluate (en%arg1)
   en%value_is_known = .true.
   if (en%value_is_known) then
      en%arg0%index => en%index
      en%arg0%prt1 => en%prt1
      en%lval = en%op1_cut (en%arg1, en%arg0)
   end if
case (EN_LOG_FUN_BINARY)
   call eval_node_evaluate (en%arg1)
   call eval_node_evaluate (en%arg2)
   en%value_is_known = .true.
   if (en%value_is_known) then
      en%arg0%index => en%index
      en%arg0%prt1 => en%prt1
      en%arg0%prt2 => en%prt2
      en%lval = en%op2_cut (en%arg1, en%arg2, en%arg0)
   end if
case (EN_INT_FUN_UNARY)
   call eval_node_evaluate (en%arg1)
   en%value_is_known = en%arg1%value_is_known
   if (en%value_is_known) then
      if (associated (en%arg0)) then
         en%arg0%index => en%index
         en%arg0%prt1 => en%prt1
         call en%op1_evi (en%ival, en%arg1, en%arg0)
      else
```

```
         call en%op1_evi (en%ival, en%arg1)
      end if
   end if
case (EN_INT_FUN_BINARY)
   call eval_node_evaluate (en%arg1)
   call eval_node_evaluate (en%arg2)
   en%value_is_known = &
        en%arg1%value_is_known .and. &
        en%arg2%value_is_known
   if (en%value_is_known) then
      if (associated (en%arg0)) then
         en%arg0%index => en%index
         en%arg0%prt1 => en%prt1
         en%arg0%prt2 => en%prt2
         call en%op2_evi (en%ival, en%arg1, en%arg2, en%arg0)
      else
         call en%op2_evi (en%ival, en%arg1, en%arg2)
      end if
   end if
case (EN_REAL_FUN_UNARY)
   call eval_node_evaluate (en%arg1)
   en%value_is_known = en%arg1%value_is_known
   if (en%value_is_known) then
      if (associated (en%arg0)) then
         en%arg0%index => en%index
         en%arg0%prt1 => en%prt1
         call en%op1_evr (en%rval, en%arg1, en%arg0)
      else
         call en%op1_evr (en%rval, en%arg1)
      end if
   end if
case (EN_REAL_FUN_BINARY)
   call eval_node_evaluate (en%arg1)
   call eval_node_evaluate (en%arg2)
   en%value_is_known = &
        en%arg1%value_is_known .and. &
        en%arg2%value_is_known
   if (en%value_is_known) then
      if (associated (en%arg0)) then
         en%arg0%index => en%index
         en%arg0%prt1 => en%prt1
         en%arg0%prt2 => en%prt2
         call en%op2_evr (en%rval, en%arg1, en%arg2, en%arg0)
      else
         call en%op2_evr (en%rval, en%arg1, en%arg2)
      end if
   end if
case (EN_FORMAT_STR)
   if (associated (en%arg0)) then
      call eval_node_evaluate (en%arg0)
      en%value_is_known = en%arg0%value_is_known
   else
      en%value_is_known = .true.
   end if
```

```
      if (associated (en%arg1)) then
         call eval_node_evaluate (en%arg1)
         en%value_is_known = &
              en%value_is_known .and. en%arg1%value_is_known
         if (en%value_is_known) then
            call evaluate_sprintf (en%sval, en%ival, en%arg0, en%arg1)
         end if
      else
         if (en%value_is_known) then
            call evaluate_sprintf (en%sval, en%ival, en%arg0)
         end if
      end if
   end select
   if (debug2_active (D_MODEL_F)) then
      print *, "eval_node_evaluate"
      call eval_node_write (en)
   end if
end subroutine eval_node_evaluate
```

Test method

This is called from a unit test: initialize a particular observable.

⟨*Eval trees: eval node: TBP*⟩+≡

```
procedure :: test_obs => eval_node_test_obs
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_node_test_obs (node, var_list, var_name)
  class(eval_node_t), intent(inout) :: node
  type(var_list_t), intent(in) :: var_list
  type(string_t), intent(in) :: var_name
  procedure(obs_unary_int), pointer :: obs1_iptr
  type(prt_t), pointer :: p1
  call var_list%get_obs1_iptr (var_name, obs1_iptr, p1)
  call eval_node_init_obs1_int_ptr (node, var_name, obs1_iptr, p1)
end subroutine eval_node_test_obs
```

28.3.7 Evaluation syntax

We have two different flavors of the syntax: with and without particles.

⟨*Eval trees: public*⟩+≡

```
public :: syntax_expr
public :: syntax_pexpr
```

⟨*Eval trees: variables*⟩≡

```
type(syntax_t), target, save :: syntax_expr
type(syntax_t), target, save :: syntax_pexpr
```

These are for testing only and may be removed:

⟨*Eval trees: public*⟩+≡

```
public :: syntax_expr_init
public :: syntax_pexpr_init
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine syntax_expr_init ()
  type(ifile_t) :: ifile
  call define_expr_syntax (ifile, particles=.false., analysis=.false.)
  call syntax_init (syntax_expr, ifile)
  call ifile_final (ifile)
end subroutine syntax_expr_init

subroutine syntax_pexpr_init ()
  type(ifile_t) :: ifile
  call define_expr_syntax (ifile, particles=.true., analysis=.false.)
  call syntax_init (syntax_pexpr, ifile)
  call ifile_final (ifile)
end subroutine syntax_pexpr_init
```

⟨*Eval trees: public*⟩+≡

```
public :: syntax_expr_final
public :: syntax_pexpr_final
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine syntax_expr_final ()
  call syntax_final (syntax_expr)
end subroutine syntax_expr_final

subroutine syntax_pexpr_final ()
  call syntax_final (syntax_pexpr)
end subroutine syntax_pexpr_final
```

⟨*Eval trees: public*⟩+≡

```
public :: syntax_pexpr_write
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine syntax_pexpr_write (unit)
  integer, intent(in), optional :: unit
  call syntax_write (syntax_pexpr, unit)
end subroutine syntax_pexpr_write
```

⟨*Eval trees: public*⟩+≡

```
public :: define_expr_syntax
```

Numeric expressions.

⟨*Eval trees: procedures*⟩+≡

```
subroutine define_expr_syntax (ifile, particles, analysis)
  type(ifile_t), intent(inout) :: ifile
  logical, intent(in) :: particles, analysis
  type(string_t) :: numeric_pexpr
  type(string_t) :: var_plist, var_alias
  if (particles) then
     numeric_pexpr = " | numeric_pexpr"
     var_plist = " | var_plist"
     var_alias = " | var_alias"
  else
     numeric_pexpr = ""
     var_plist = ""
```

```
   var_alias = ""
end if
call ifile_append (ifile, "SEQ expr = subexpr addition*")
call ifile_append (ifile, "ALT subexpr = addition | term")
call ifile_append (ifile, "SEQ addition = plus_or_minus term")
call ifile_append (ifile, "SEQ term = factor multiplication*")
call ifile_append (ifile, "SEQ multiplication = times_or_over factor")
call ifile_append (ifile, "SEQ factor = value exponentiation?")
call ifile_append (ifile, "SEQ exponentiation = to_the value")
call ifile_append (ifile, "ALT plus_or_minus = '+' | '-'")
call ifile_append (ifile, "ALT times_or_over = '*' | '/'")
call ifile_append (ifile, "ALT to_the = '^' | '**'")
call ifile_append (ifile, "KEY '+'")
call ifile_append (ifile, "KEY '-'")
call ifile_append (ifile, "KEY '*'")
call ifile_append (ifile, "KEY '/'")
call ifile_append (ifile, "KEY '^'")
call ifile_append (ifile, "KEY '**'")
call ifile_append (ifile, "ALT value = signed_value | unsigned_value")
call ifile_append (ifile, "SEQ signed_value = '-' unsigned_value")
call ifile_append (ifile, "ALT unsigned_value = " // &
     "numeric_value | constant | variable | " // &
     "result | user_observable | " // &
     "grouped_expr | block_expr | conditional_expr | " // &
     "unary_function | binary_function" // &
     numeric_pexpr)
call ifile_append (ifile, "ALT numeric_value = integer_value | " &
     // "real_value | complex_value")
call ifile_append (ifile, "SEQ integer_value = integer_literal unit_expr?")
call ifile_append (ifile, "SEQ real_value = real_literal unit_expr?")
call ifile_append (ifile, "SEQ complex_value = complex_literal unit_expr?")
call ifile_append (ifile, "INT integer_literal")
call ifile_append (ifile, "REA real_literal")
call ifile_append (ifile, "COM complex_literal")
call ifile_append (ifile, "SEQ unit_expr = unit unit_power?")
call ifile_append (ifile, "ALT unit = " // &
     "TeV | GeV | MeV | keV | eV | meV | " // &
     "nbarn | pbarn | fbarn | abarn | " // &
     "rad | mrad | degree | '%'")
call ifile_append (ifile, "KEY TeV")
call ifile_append (ifile, "KEY GeV")
call ifile_append (ifile, "KEY MeV")
call ifile_append (ifile, "KEY keV")
call ifile_append (ifile, "KEY eV")
call ifile_append (ifile, "KEY meV")
call ifile_append (ifile, "KEY nbarn")
call ifile_append (ifile, "KEY pbarn")
call ifile_append (ifile, "KEY fbarn")
call ifile_append (ifile, "KEY abarn")
call ifile_append (ifile, "KEY rad")
call ifile_append (ifile, "KEY mrad")
call ifile_append (ifile, "KEY degree")
call ifile_append (ifile, "KEY '%'")
call ifile_append (ifile, "SEQ unit_power = '^' frac_expr")
```

```
call ifile_append (ifile, "ALT frac_expr = frac | grouped_frac")
call ifile_append (ifile, "GRO grouped_frac = ( frac_expr )")
call ifile_append (ifile, "SEQ frac = signed_int div?")
call ifile_append (ifile, "ALT signed_int = " &
     // "neg_int | pos_int | integer_literal")
call ifile_append (ifile, "SEQ neg_int = '-' integer_literal")
call ifile_append (ifile, "SEQ pos_int = '+' integer_literal")
call ifile_append (ifile, "SEQ div = '/' integer_literal")
call ifile_append (ifile, "ALT constant = pi | I")
call ifile_append (ifile, "KEY pi")
call ifile_append (ifile, "KEY I")
call ifile_append (ifile, "IDE variable")
call ifile_append (ifile, "SEQ result = result_key result_arg")
call ifile_append (ifile, "ALT result_key = " // &
     "num_id | integral | error")
call ifile_append (ifile, "SEQ user_observable = user_obs user_arg")
call ifile_append (ifile, "KEY user_obs")
call ifile_append (ifile, "ARG user_arg = ( sexpr )")
call ifile_append (ifile, "KEY num_id")
call ifile_append (ifile, "KEY integral")
call ifile_append (ifile, "KEY error")
call ifile_append (ifile, "GRO result_arg = ( process_id )")
call ifile_append (ifile, "IDE process_id")
call ifile_append (ifile, "SEQ unary_function = fun_unary function_arg1")
call ifile_append (ifile, "SEQ binary_function = fun_binary function_arg2")
call ifile_append (ifile, "ALT fun_unary = " // &
     "complex | real | int | nint | floor | ceiling | abs | sgn | " // &
     "sqrt | exp | log | log10 | " // &
     "sin | cos | tan | asin | acos | atan | " // &
     "sinh | cosh | tanh")
call ifile_append (ifile, "KEY complex")
call ifile_append (ifile, "KEY real")
call ifile_append (ifile, "KEY int")
call ifile_append (ifile, "KEY nint")
call ifile_append (ifile, "KEY floor")
call ifile_append (ifile, "KEY ceiling")
call ifile_append (ifile, "KEY abs")
call ifile_append (ifile, "KEY sgn")
call ifile_append (ifile, "KEY sqrt")
call ifile_append (ifile, "KEY exp")
call ifile_append (ifile, "KEY log")
call ifile_append (ifile, "KEY log10")
call ifile_append (ifile, "KEY sin")
call ifile_append (ifile, "KEY cos")
call ifile_append (ifile, "KEY tan")
call ifile_append (ifile, "KEY asin")
call ifile_append (ifile, "KEY acos")
call ifile_append (ifile, "KEY atan")
call ifile_append (ifile, "KEY sinh")
call ifile_append (ifile, "KEY cosh")
call ifile_append (ifile, "KEY tanh")
call ifile_append (ifile, "ALT fun_binary = max | min | mod | modulo")
call ifile_append (ifile, "KEY max")
call ifile_append (ifile, "KEY min")
```

```
    call ifile_append (ifile, "KEY mod")
    call ifile_append (ifile, "KEY modulo")
    call ifile_append (ifile, "ARG function_arg1 = ( expr )")
    call ifile_append (ifile, "ARG function_arg2 = ( expr, expr )")
    call ifile_append (ifile, "GRO grouped_expr = ( expr )")
    call ifile_append (ifile, "SEQ block_expr = let var_spec in expr")
    call ifile_append (ifile, "KEY let")
    call ifile_append (ifile, "ALT var_spec = " // &
         "var_num | var_int | var_real | var_complex | " // &
         "var_logical" // var_plist // var_alias // " | var_string")
    call ifile_append (ifile, "SEQ var_num = var_name '=' expr")
    call ifile_append (ifile, "SEQ var_int = int var_name '=' expr")
    call ifile_append (ifile, "SEQ var_real = real var_name '=' expr")
    call ifile_append (ifile, "SEQ var_complex = complex var_name '=' complex_expr")
    call ifile_append (ifile, "ALT complex_expr = " // &
         "cexpr_real | cexpr_complex")
    call ifile_append (ifile, "ARG cexpr_complex = ( expr, expr )")
    call ifile_append (ifile, "SEQ cexpr_real = expr")
    call ifile_append (ifile, "IDE var_name")
    call ifile_append (ifile, "KEY '='")
    call ifile_append (ifile, "KEY in")
    call ifile_append (ifile, "SEQ conditional_expr = " // &
         "if lexpr then expr maybe_elsif_expr maybe_else_expr endif")
    call ifile_append (ifile, "SEQ maybe_elsif_expr = elsif_expr*")
    call ifile_append (ifile, "SEQ maybe_else_expr = else_expr?")
    call ifile_append (ifile, "SEQ elsif_expr = elsif lexpr then expr")
    call ifile_append (ifile, "SEQ else_expr = else expr")
    call ifile_append (ifile, "KEY if")
    call ifile_append (ifile, "KEY then")
    call ifile_append (ifile, "KEY elsif")
    call ifile_append (ifile, "KEY else")
    call ifile_append (ifile, "KEY endif")
    call define_lexpr_syntax (ifile, particles, analysis)
    call define_sexpr_syntax (ifile)
    if (particles) then
       call define_pexpr_syntax (ifile)
       call define_cexpr_syntax (ifile)
       call define_var_plist_syntax (ifile)
       call define_var_alias_syntax (ifile)
       call define_numeric_pexpr_syntax (ifile)
       call define_logical_pexpr_syntax (ifile)
    end if

  end subroutine define_expr_syntax
```

Logical expressions.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine define_lexpr_syntax (ifile, particles, analysis)
    type(ifile_t), intent(inout) :: ifile
    logical, intent(in) :: particles, analysis
    type(string_t) :: logical_pexpr, record_cmd
    if (particles) then
       logical_pexpr = " | logical_pexpr"
    else
```

```
   logical_pexpr = ""
end if
if (analysis) then
   record_cmd = " | record_cmd"
else
   record_cmd = ""
end if
call ifile_append (ifile, "SEQ lexpr = lsinglet lsequel*")
call ifile_append (ifile, "SEQ lsequel = ';' lsinglet")
call ifile_append (ifile, "SEQ lsinglet = lterm alternative*")
call ifile_append (ifile, "SEQ alternative = or lterm")
call ifile_append (ifile, "SEQ lterm = lvalue coincidence*")
call ifile_append (ifile, "SEQ coincidence = and lvalue")
call ifile_append (ifile, "KEY ';'")
call ifile_append (ifile, "KEY or")
call ifile_append (ifile, "KEY and")
call ifile_append (ifile, "ALT lvalue = " // &
     "true | false | lvariable | negation | " // &
     "grouped_lexpr | block_lexpr | conditional_lexpr | " // &
     "compared_expr | compared_sexpr" // &
     logical_pexpr //  record_cmd)
call ifile_append (ifile, "KEY true")
call ifile_append (ifile, "KEY false")
call ifile_append (ifile, "SEQ lvariable = '?' alt_lvariable")
call ifile_append (ifile, "KEY '?'")
call ifile_append (ifile, "ALT alt_lvariable = variable | grouped_lexpr")
call ifile_append (ifile, "SEQ negation = not lvalue")
call ifile_append (ifile, "KEY not")
call ifile_append (ifile, "GRO grouped_lexpr = ( lexpr )")
call ifile_append (ifile, "SEQ block_lexpr = let var_spec in lexpr")
call ifile_append (ifile, "ALT var_logical = " // &
     "var_logical_new | var_logical_spec")
call ifile_append (ifile, "SEQ var_logical_new = logical var_logical_spec")
call ifile_append (ifile, "KEY logical")
call ifile_append (ifile, "SEQ var_logical_spec = '?' var_name = lexpr")
call ifile_append (ifile, "SEQ conditional_lexpr = " // &
     "if lexpr then lexpr maybe_elsif_lexpr maybe_else_lexpr endif")
call ifile_append (ifile, "SEQ maybe_elsif_lexpr = elsif_lexpr*")
call ifile_append (ifile, "SEQ maybe_else_lexpr = else_lexpr?")
call ifile_append (ifile, "SEQ elsif_lexpr = elsif lexpr then lexpr")
call ifile_append (ifile, "SEQ else_lexpr = else lexpr")
call ifile_append (ifile, "SEQ compared_expr = expr comparison+")
call ifile_append (ifile, "SEQ comparison = compare expr")
call ifile_append (ifile, "ALT compare = " // &
     "'<' | '>' | '<=' | '>=' | '==' | '<>'")
call ifile_append (ifile, "KEY '<'")
call ifile_append (ifile, "KEY '>'")
call ifile_append (ifile, "KEY '<='")
call ifile_append (ifile, "KEY '>='")
call ifile_append (ifile, "KEY '=='")
call ifile_append (ifile, "KEY '<>'")
call ifile_append (ifile, "SEQ compared_sexpr = sexpr str_comparison+")
call ifile_append (ifile, "SEQ str_comparison = str_compare sexpr")
call ifile_append (ifile, "ALT str_compare = '==' | '<>'")
```

```
    if (analysis) then
       call ifile_append (ifile, "SEQ record_cmd = " // &
            "record_key analysis_tag record_arg?")
       call ifile_append (ifile, "ALT record_key = " // &
            "record | record_unweighted | record_excess")
       call ifile_append (ifile, "KEY record")
       call ifile_append (ifile, "KEY record_unweighted")
       call ifile_append (ifile, "KEY record_excess")
       call ifile_append (ifile, "ALT analysis_tag = analysis_id | sexpr")
       call ifile_append (ifile, "IDE analysis_id")
       call ifile_append (ifile, "ARG record_arg = ( expr+ )")
    end if
  end subroutine define_lexpr_syntax
```

String expressions.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine define_sexpr_syntax (ifile)
    type(ifile_t), intent(inout) :: ifile
    call ifile_append (ifile, "SEQ sexpr = svalue str_concatenation*")
    call ifile_append (ifile, "SEQ str_concatenation = '&' svalue")
    call ifile_append (ifile, "KEY '&'")
    call ifile_append (ifile, "ALT svalue = " // &
         "grouped_sexpr | block_sexpr | conditional_sexpr | " // &
         "svariable | string_function | string_literal")
    call ifile_append (ifile, "GRO grouped_sexpr = ( sexpr )")
    call ifile_append (ifile, "SEQ block_sexpr = let var_spec in sexpr")
    call ifile_append (ifile, "SEQ conditional_sexpr = " // &
         "if lexpr then sexpr maybe_elsif_sexpr maybe_else_sexpr endif")
    call ifile_append (ifile, "SEQ maybe_elsif_sexpr = elsif_sexpr*")
    call ifile_append (ifile, "SEQ maybe_else_sexpr = else_sexpr?")
    call ifile_append (ifile, "SEQ elsif_sexpr = elsif lexpr then sexpr")
    call ifile_append (ifile, "SEQ else_sexpr = else sexpr")
    call ifile_append (ifile, "SEQ svariable = '$' alt_svariable")
    call ifile_append (ifile, "KEY '$'")
    call ifile_append (ifile, "ALT alt_svariable = variable | grouped_sexpr")
    call ifile_append (ifile, "ALT var_string = " // &
         "var_string_new | var_string_spec")
    call ifile_append (ifile, "SEQ var_string_new = string var_string_spec")
    call ifile_append (ifile, "KEY string")
    call ifile_append (ifile, "SEQ var_string_spec = '$' var_name = sexpr") ! $
    call ifile_append (ifile, "ALT string_function = sprintf_fun")
    call ifile_append (ifile, "SEQ sprintf_fun = sprintf_clause sprintf_args?")
    call ifile_append (ifile, "SEQ sprintf_clause = sprintf sexpr")
    call ifile_append (ifile, "KEY sprintf")
    call ifile_append (ifile, "ARG sprintf_args = ( sprintf_arg* )")
    call ifile_append (ifile, "ALT sprintf_arg = " &
         // "lvariable | svariable | expr")
    call ifile_append (ifile, "QUO string_literal = '""'...'""'")
  end subroutine define_sexpr_syntax
```

Eval trees that evaluate to subevents.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine define_pexpr_syntax (ifile)
```

```
type(ifile_t), intent(inout) :: ifile
call ifile_append (ifile, "SEQ pexpr = pterm pconcatenation*")
call ifile_append (ifile, "SEQ pconcatenation = '&' pterm")
! call ifile_append (ifile, "KEY '&'")   !!! (Key exists already)
call ifile_append (ifile, "SEQ pterm = pvalue pcombination*")
call ifile_append (ifile, "SEQ pcombination = '+' pvalue")
! call ifile_append (ifile, "KEY '+'")   !!! (Key exists already)
call ifile_append (ifile, "ALT pvalue = " // &
     "pexpr_src | pvariable | " // &
     "grouped_pexpr | block_pexpr | conditional_pexpr | " // &
     "prt_function")
call ifile_append (ifile, "SEQ pexpr_src = prefix_cexpr")
call ifile_append (ifile, "ALT prefix_cexpr = " // &
     "incoming_prt | outgoing_prt")
call ifile_append (ifile, "SEQ incoming_prt = incoming cexpr")
call ifile_append (ifile, "KEY incoming")
call ifile_append (ifile, "SEQ outgoing_prt = cexpr")
call ifile_append (ifile, "SEQ pvariable = '@' alt_pvariable")
call ifile_append (ifile, "KEY '@'")
call ifile_append (ifile, "ALT alt_pvariable = variable | grouped_pexpr")
call ifile_append (ifile, "GRO grouped_pexpr = '[' pexpr ']'")
call ifile_append (ifile, "SEQ block_pexpr = let var_spec in pexpr")
call ifile_append (ifile, "SEQ conditional_pexpr = " // &
     "if lexpr then pexpr maybe_elsif_pexpr maybe_else_pexpr endif")
call ifile_append (ifile, "SEQ maybe_elsif_pexpr = elsif_pexpr*")
call ifile_append (ifile, "SEQ maybe_else_pexpr = else_pexpr?")
call ifile_append (ifile, "SEQ elsif_pexpr = elsif lexpr then pexpr")
call ifile_append (ifile, "SEQ else_pexpr = else pexpr")
call ifile_append (ifile, "ALT prt_function = " // &
     "join_fun | combine_fun | collect_fun | cluster_fun | " // &
     "select_fun | extract_fun | sort_fun")
call ifile_append (ifile, "SEQ join_fun = join_clause pargs2")
call ifile_append (ifile, "SEQ combine_fun = combine_clause pargs2")
call ifile_append (ifile, "SEQ collect_fun = collect_clause pargs1")
call ifile_append (ifile, "SEQ cluster_fun = cluster_clause pargs1")
call ifile_append (ifile, "SEQ select_fun = select_clause pargs1")
call ifile_append (ifile, "SEQ extract_fun = extract_clause pargs1")
call ifile_append (ifile, "SEQ sort_fun = sort_clause pargs1")
call ifile_append (ifile, "SEQ join_clause = join condition?")
call ifile_append (ifile, "SEQ combine_clause = combine condition?")
call ifile_append (ifile, "SEQ collect_clause = collect condition?")
call ifile_append (ifile, "SEQ cluster_clause = cluster condition?")
call ifile_append (ifile, "SEQ select_clause = select condition?")
call ifile_append (ifile, "SEQ extract_clause = extract position?")
call ifile_append (ifile, "SEQ sort_clause = sort criterion?")
call ifile_append (ifile, "KEY join")
call ifile_append (ifile, "KEY combine")
call ifile_append (ifile, "KEY collect")
call ifile_append (ifile, "KEY cluster")
call ifile_append (ifile, "KEY select")
call ifile_append (ifile, "SEQ condition = if lexpr")
call ifile_append (ifile, "KEY extract")
call ifile_append (ifile, "SEQ position = index expr")
call ifile_append (ifile, "KEY sort")
```

```
    call ifile_append (ifile, "SEQ criterion = by expr")
    call ifile_append (ifile, "KEY index")
    call ifile_append (ifile, "KEY by")
    call ifile_append (ifile, "ARG pargs2 = '[' pexpr, pexpr ']'")
    call ifile_append (ifile, "ARG pargs1 = '[' pexpr, pexpr? ']'")
  end subroutine define_pexpr_syntax
```

Eval trees that evaluate to PDG-code arrays.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine define_cexpr_syntax (ifile)
    type(ifile_t), intent(inout) :: ifile
    call ifile_append (ifile, "SEQ cexpr = avalue concatenation*")
    call ifile_append (ifile, "SEQ concatenation = ':' avalue")
    call ifile_append (ifile, "KEY ':'")
    call ifile_append (ifile, "ALT avalue = " // &
         "grouped_cexpr | block_cexpr | conditional_cexpr | " // &
         "variable | pdg_code | prt_name")
    call ifile_append (ifile, "GRO grouped_cexpr = ( cexpr )")
    call ifile_append (ifile, "SEQ block_cexpr = let var_spec in cexpr")
    call ifile_append (ifile, "SEQ conditional_cexpr = " // &
         "if lexpr then cexpr maybe_elsif_cexpr maybe_else_cexpr endif")
    call ifile_append (ifile, "SEQ maybe_elsif_cexpr = elsif_cexpr*")
    call ifile_append (ifile, "SEQ maybe_else_cexpr = else_cexpr?")
    call ifile_append (ifile, "SEQ elsif_cexpr = elsif lexpr then cexpr")
    call ifile_append (ifile, "SEQ else_cexpr = else cexpr")
    call ifile_append (ifile, "SEQ pdg_code = pdg pdg_arg")
    call ifile_append (ifile, "KEY pdg")
    call ifile_append (ifile, "ARG pdg_arg = ( expr )")
    call ifile_append (ifile, "QUO prt_name = '""'...'""'")
  end subroutine define_cexpr_syntax
```

Extra variable types.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine define_var_plist_syntax (ifile)
    type(ifile_t), intent(inout) :: ifile
    call ifile_append (ifile, "ALT var_plist = var_plist_new | var_plist_spec")
    call ifile_append (ifile, "SEQ var_plist_new = subevt var_plist_spec")
    call ifile_append (ifile, "KEY subevt")
    call ifile_append (ifile, "SEQ var_plist_spec = '@' var_name '=' pexpr")
  end subroutine define_var_plist_syntax

  subroutine define_var_alias_syntax (ifile)
    type(ifile_t), intent(inout) :: ifile
    call ifile_append (ifile, "SEQ var_alias = alias var_name '=' cexpr")
    call ifile_append (ifile, "KEY alias")
  end subroutine define_var_alias_syntax
```

Particle-list expressions that evaluate to numeric values

⟨*Eval trees: procedures*⟩+≡

```
  subroutine define_numeric_pexpr_syntax (ifile)
    type(ifile_t), intent(inout) :: ifile
    call ifile_append (ifile, "ALT numeric_pexpr = " &
```

```
        // "eval_fun | count_fun | event_shape_fun")
    call ifile_append (ifile, "SEQ eval_fun = eval expr pargs1")
    call ifile_append (ifile, "SEQ count_fun = count_clause pargs1")
    call ifile_append (ifile, "SEQ count_clause = count condition?")
    call ifile_append (ifile, "KEY eval")
    call ifile_append (ifile, "KEY count")
    call ifile_append (ifile, "ALT event_shape_fun = user_event_fun")
    call ifile_append (ifile, "SEQ user_event_fun = " &
        // "user_event_shape user_arg pargs1")
    call ifile_append (ifile, "KEY user_event_shape")
  end subroutine define_numeric_pexpr_syntax
```

Particle-list functions that evaluate to logical values.

⟨*Eval trees: procedures*⟩+≡

```
  subroutine define_logical_pexpr_syntax (ifile)
    type(ifile_t), intent(inout) :: ifile
    call ifile_append (ifile, "ALT logical_pexpr = " // &
        "all_fun | any_fun | no_fun | user_cut_fun")
    call ifile_append (ifile, "SEQ all_fun = all lexpr pargs1")
    call ifile_append (ifile, "SEQ any_fun = any lexpr pargs1")
    call ifile_append (ifile, "SEQ no_fun = no lexpr pargs1")
    call ifile_append (ifile, "KEY all")
    call ifile_append (ifile, "KEY any")
    call ifile_append (ifile, "KEY no")
    call ifile_append (ifile, "SEQ user_cut_fun = user_cut user_arg pargs1")
    call ifile_append (ifile, "KEY user_cut")
  end subroutine define_logical_pexpr_syntax
```

All characters that can occur in expressions (apart from alphanumeric).

⟨*Eval trees: procedures*⟩+≡

```
  subroutine lexer_init_eval_tree (lexer, particles)
    type(lexer_t), intent(out) :: lexer
    logical, intent(in) :: particles
    type(keyword_list_t), pointer :: keyword_list
    if (particles) then
       keyword_list => syntax_get_keyword_list_ptr (syntax_pexpr)
    else
       keyword_list => syntax_get_keyword_list_ptr (syntax_expr)
    end if
    call lexer_init (lexer, &
        comment_chars = "#!", &
        quote_chars = '"', &
        quote_match = '"', &
        single_chars = "()[],;:&%?$@", &
        special_class = [ "+-*/^", "<>=~ " ] , &
        keyword_list = keyword_list)
  end subroutine lexer_init_eval_tree
```

28.3.8 Set up appropriate parse trees

Parse an input stream as a specific flavor of expression. The appropriate expression syntax has to be available.

⟨*Eval trees: public*⟩+≡

```
public :: parse_tree_init_expr
public :: parse_tree_init_lexpr
public :: parse_tree_init_pexpr
public :: parse_tree_init_cexpr
public :: parse_tree_init_sexpr
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine parse_tree_init_expr (parse_tree, stream, particles)
  type(parse_tree_t), intent(out) :: parse_tree
  type(stream_t), intent(inout), target :: stream
  logical, intent(in) :: particles
  type(lexer_t) :: lexer
  call lexer_init_eval_tree (lexer, particles)
  call lexer_assign_stream (lexer, stream)
  if (particles) then
     call parse_tree_init &
          (parse_tree, syntax_pexpr, lexer, var_str ("expr"))
  else
     call parse_tree_init &
          (parse_tree, syntax_expr, lexer, var_str ("expr"))
  end if
  call lexer_final (lexer)
end subroutine parse_tree_init_expr

subroutine parse_tree_init_lexpr (parse_tree, stream, particles)
  type(parse_tree_t), intent(out) :: parse_tree
  type(stream_t), intent(inout), target :: stream
  logical, intent(in) :: particles
  type(lexer_t) :: lexer
  call lexer_init_eval_tree (lexer, particles)
  call lexer_assign_stream (lexer, stream)
  if (particles) then
     call parse_tree_init &
          (parse_tree, syntax_pexpr, lexer, var_str ("lexpr"))
  else
     call parse_tree_init &
          (parse_tree, syntax_expr, lexer, var_str ("lexpr"))
  end if
  call lexer_final (lexer)
end subroutine parse_tree_init_lexpr

subroutine parse_tree_init_pexpr (parse_tree, stream)
  type(parse_tree_t), intent(out) :: parse_tree
  type(stream_t), intent(inout), target :: stream
  type(lexer_t) :: lexer
  call lexer_init_eval_tree (lexer, .true.)
  call lexer_assign_stream (lexer, stream)
  call parse_tree_init &
       (parse_tree, syntax_pexpr, lexer, var_str ("pexpr"))
  call lexer_final (lexer)
end subroutine parse_tree_init_pexpr

subroutine parse_tree_init_cexpr (parse_tree, stream)
  type(parse_tree_t), intent(out) :: parse_tree
```

```
  type(stream_t), intent(inout), target :: stream
  type(lexer_t) :: lexer
  call lexer_init_eval_tree (lexer, .true.)
  call lexer_assign_stream (lexer, stream)
  call parse_tree_init &
       (parse_tree, syntax_pexpr, lexer, var_str ("cexpr"))
  call lexer_final (lexer)
end subroutine parse_tree_init_cexpr

subroutine parse_tree_init_sexpr (parse_tree, stream, particles)
  type(parse_tree_t), intent(out) :: parse_tree
  type(stream_t), intent(inout), target :: stream
  logical, intent(in) :: particles
  type(lexer_t) :: lexer
  call lexer_init_eval_tree (lexer, particles)
  call lexer_assign_stream (lexer, stream)
  if (particles) then
     call parse_tree_init &
          (parse_tree, syntax_pexpr, lexer, var_str ("sexpr"))
  else
     call parse_tree_init &
          (parse_tree, syntax_expr, lexer, var_str ("sexpr"))
  end if
  call lexer_final (lexer)
end subroutine parse_tree_init_sexpr
```

28.3.9 The evaluation tree

The evaluation tree contains the initial variable list and the root node.

⟨*Eval trees: public*⟩+≡

```
public :: eval_tree_t
```

⟨*Eval trees: types*⟩+≡

```
type, extends (expr_t) :: eval_tree_t
   private
   type(parse_node_t), pointer :: pn => null ()
   type(var_list_t) :: var_list
   type(eval_node_t), pointer :: root => null ()
 contains
 ⟨Eval trees: eval tree: TBP⟩
end type eval_tree_t
```

Init from stream, using a temporary parse tree.

⟨*Eval trees: eval tree: TBP*⟩≡

```
procedure :: init_stream => eval_tree_init_stream
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_tree_init_stream &
     (eval_tree, stream, var_list, subevt, result_type)
  class(eval_tree_t), intent(out), target :: eval_tree
  type(stream_t), intent(inout), target :: stream
  type(var_list_t), intent(in), target :: var_list
  type(subevt_t), intent(in), target, optional :: subevt
```

```
    integer, intent(in), optional :: result_type
    type(parse_tree_t) :: parse_tree
    type(parse_node_t), pointer :: nd_root
    integer :: type
    type = V_REAL;  if (present (result_type))  type = result_type
    select case (type)
    case (V_INT, V_REAL, V_CMPLX)
       call parse_tree_init_expr (parse_tree, stream, present (subevt))
    case (V_LOG)
       call parse_tree_init_lexpr (parse_tree, stream, present (subevt))
    case (V_SEV)
       call parse_tree_init_pexpr (parse_tree, stream)
    case (V_PDG)
       call parse_tree_init_cexpr (parse_tree, stream)
    case (V_STR)
       call parse_tree_init_sexpr (parse_tree, stream, present (subevt))
    end select
    nd_root => parse_tree%get_root_ptr ()
    if (associated (nd_root)) then
       select case (type)
       case (V_INT, V_REAL, V_CMPLX)
          call eval_tree_init_expr (eval_tree, nd_root, var_list, subevt)
       case (V_LOG)
          call eval_tree_init_lexpr (eval_tree, nd_root, var_list, subevt)
       case (V_SEV)
          call eval_tree_init_pexpr (eval_tree, nd_root, var_list, subevt)
       case (V_PDG)
          call eval_tree_init_cexpr (eval_tree, nd_root, var_list, subevt)
       case (V_STR)
          call eval_tree_init_sexpr (eval_tree, nd_root, var_list, subevt)
       end select
    end if
    call parse_tree_final (parse_tree)
  end subroutine eval_tree_init_stream
```

API (to be superseded by the methods below): Init from a given parse-tree node. If we evaluate an expression that contains particle-list references, the original subevent has to be supplied. The initial variable list is optional.

⟨*Eval trees: eval tree: TBP*⟩+≡

```
  procedure :: init_expr  => eval_tree_init_expr
  procedure :: init_lexpr => eval_tree_init_lexpr
  procedure :: init_pexpr => eval_tree_init_pexpr
  procedure :: init_cexpr => eval_tree_init_cexpr
  procedure :: init_sexpr => eval_tree_init_sexpr
```

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_tree_init_expr &
      (expr, parse_node, var_list, subevt)
    class(eval_tree_t), intent(out), target :: expr
    type(parse_node_t), intent(in), target :: parse_node
    type(var_list_t), intent(in), target :: var_list
    type(subevt_t), intent(in), optional, target :: subevt
    call eval_tree_link_var_list (expr, var_list)
    if (present (subevt))  call eval_tree_set_subevt (expr, subevt)
```

```
  call eval_node_compile_expr &
       (expr%root, parse_node, expr%var_list)
end subroutine eval_tree_init_expr

subroutine eval_tree_init_lexpr &
    (expr, parse_node, var_list, subevt)
  class(eval_tree_t), intent(out), target :: expr
  type(parse_node_t), intent(in), target :: parse_node
  type(var_list_t), intent(in), target :: var_list
  type(subevt_t), intent(in), optional, target :: subevt
  call eval_tree_link_var_list (expr, var_list)
  if (present (subevt))  call eval_tree_set_subevt (expr, subevt)
  call eval_node_compile_lexpr &
       (expr%root, parse_node, expr%var_list)
end subroutine eval_tree_init_lexpr

subroutine eval_tree_init_pexpr &
    (expr, parse_node, var_list, subevt)
  class(eval_tree_t), intent(out), target :: expr
  type(parse_node_t), intent(in), target :: parse_node
  type(var_list_t), intent(in), target :: var_list
  type(subevt_t), intent(in), optional, target :: subevt
  call eval_tree_link_var_list (expr, var_list)
  if (present (subevt))  call eval_tree_set_subevt (expr, subevt)
  call eval_node_compile_pexpr &
       (expr%root, parse_node, expr%var_list)
end subroutine eval_tree_init_pexpr

subroutine eval_tree_init_cexpr &
    (expr, parse_node, var_list, subevt)
  class(eval_tree_t), intent(out), target :: expr
  type(parse_node_t), intent(in), target :: parse_node
  type(var_list_t), intent(in), target :: var_list
  type(subevt_t), intent(in), optional, target :: subevt
  call eval_tree_link_var_list (expr, var_list)
  if (present (subevt))  call eval_tree_set_subevt (expr, subevt)
  call eval_node_compile_cexpr &
       (expr%root, parse_node, expr%var_list)
end subroutine eval_tree_init_cexpr

subroutine eval_tree_init_sexpr &
    (expr, parse_node, var_list, subevt)
  class(eval_tree_t), intent(out), target :: expr
  type(parse_node_t), intent(in), target :: parse_node
  type(var_list_t), intent(in), target :: var_list
  type(subevt_t), intent(in), optional, target :: subevt
  call eval_tree_link_var_list (expr, var_list)
  if (present (subevt))  call eval_tree_set_subevt (expr, subevt)
  call eval_node_compile_sexpr &
       (expr%root, parse_node, expr%var_list)
end subroutine eval_tree_init_sexpr
```

Alternative: set up the expression using the parse node that has already been stored. We assume that the `subevt` or any other variable that may be referred

to has already been added to the local variable list.

⟨*Eval trees: eval tree: TBP*⟩+≡

```
procedure :: setup_expr  => eval_tree_setup_expr
procedure :: setup_lexpr => eval_tree_setup_lexpr
procedure :: setup_pexpr => eval_tree_setup_pexpr
procedure :: setup_cexpr => eval_tree_setup_cexpr
procedure :: setup_sexpr => eval_tree_setup_sexpr
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_tree_setup_expr (expr, vars)
  class(eval_tree_t), intent(inout), target :: expr
  class(vars_t), intent(in), target :: vars
  call eval_tree_link_var_list (expr, vars)
  call eval_node_compile_expr (expr%root, expr%pn, expr%var_list)
end subroutine eval_tree_setup_expr

subroutine eval_tree_setup_lexpr (expr, vars)
  class(eval_tree_t), intent(inout), target :: expr
  class(vars_t), intent(in), target :: vars
  call eval_tree_link_var_list (expr, vars)
  call eval_node_compile_lexpr (expr%root, expr%pn, expr%var_list)
end subroutine eval_tree_setup_lexpr

subroutine eval_tree_setup_pexpr (expr, vars)
  class(eval_tree_t), intent(inout), target :: expr
  class(vars_t), intent(in), target :: vars
  call eval_tree_link_var_list (expr, vars)
  call eval_node_compile_pexpr (expr%root, expr%pn, expr%var_list)
end subroutine eval_tree_setup_pexpr

subroutine eval_tree_setup_cexpr (expr, vars)
  class(eval_tree_t), intent(inout), target :: expr
  class(vars_t), intent(in), target :: vars
  call eval_tree_link_var_list (expr, vars)
  call eval_node_compile_cexpr (expr%root, expr%pn, expr%var_list)
end subroutine eval_tree_setup_cexpr

subroutine eval_tree_setup_sexpr (expr, vars)
  class(eval_tree_t), intent(inout), target :: expr
  class(vars_t), intent(in), target :: vars
  call eval_tree_link_var_list (expr, vars)
  call eval_node_compile_sexpr (expr%root, expr%pn, expr%var_list)
end subroutine eval_tree_setup_sexpr
```

This extra API function handles numerical constant expressions only. The only nontrivial part is the optional unit.

⟨*Eval trees: eval tree: TBP*⟩+≡

```
procedure :: init_numeric_value => eval_tree_init_numeric_value
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_tree_init_numeric_value (eval_tree, parse_node)
  class(eval_tree_t), intent(out), target :: eval_tree
  type(parse_node_t), intent(in), target :: parse_node
  call eval_node_compile_numeric_value (eval_tree%root, parse_node)
```

```
end subroutine eval_tree_init_numeric_value
```

Initialize the variable list, linking it to a context variable list.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_tree_link_var_list (eval_tree, vars)
  type(eval_tree_t), intent(inout), target :: eval_tree
  class(vars_t), intent(in), target :: vars
  call eval_tree%var_list%link (vars)
end subroutine eval_tree_link_var_list
```

Include a subevent object in the initialization. We add a pointer to this as variable `@evt` in the local variable list.

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_tree_set_subevt (eval_tree, subevt)
  type(eval_tree_t), intent(inout), target :: eval_tree
  type(subevt_t), intent(in), target :: subevt
  logical, save, target :: known = .true.
  call var_list_append_subevt_ptr &
       (eval_tree%var_list, var_str ("@evt"), subevt, known, &
       intrinsic=.true.)
end subroutine eval_tree_set_subevt
```

Finalizer.

⟨*Eval trees: eval tree: TBP*⟩+≡

```
procedure :: final => eval_tree_final
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_tree_final (expr)
  class(eval_tree_t), intent(inout) :: expr
  call expr%var_list%final ()
  if (associated (expr%root)) then
     call eval_node_final_rec (expr%root)
     deallocate (expr%root)
  end if
end subroutine eval_tree_final
```

⟨*Eval trees: eval tree: TBP*⟩+≡

```
procedure :: evaluate => eval_tree_evaluate
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_tree_evaluate (expr)
  class(eval_tree_t), intent(inout) :: expr
  if (associated (expr%root)) then
     call eval_node_evaluate (expr%root)
  end if
end subroutine eval_tree_evaluate
```

Check if the eval tree is allocated.

⟨*Eval trees: procedures*⟩+≡

```
function eval_tree_is_defined (eval_tree) result (flag)
  logical :: flag
  type(eval_tree_t), intent(in) :: eval_tree
```

```
    flag = associated (eval_tree%root)
  end function eval_tree_is_defined
```

Check if the eval tree result is constant.

⟨*Eval trees: procedures*⟩+≡

```
  function eval_tree_is_constant (eval_tree) result (flag)
    logical :: flag
    type(eval_tree_t), intent(in) :: eval_tree
    if (associated (eval_tree%root)) then
       flag = eval_tree%root%type == EN_CONSTANT
    else
       flag = .false.
    end if
  end function eval_tree_is_constant
```

Insert a conversion node at the root, if necessary (only for real/int conversion)

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_tree_convert_result (eval_tree, result_type)
    type(eval_tree_t), intent(inout) :: eval_tree
    integer, intent(in) :: result_type
    if (associated (eval_tree%root)) then
       call insert_conversion_node (eval_tree%root, result_type)
    end if
  end subroutine eval_tree_convert_result
```

Return the value of the top node, after evaluation. If the tree is empty, return the type of `V_NONE`. When extracting the value, no check for existence is done. For numeric values, the functions are safe against real/integer mismatch.

⟨*Eval trees: eval tree: TBP*⟩+≡

```
  procedure :: is_known => eval_tree_result_is_known
  procedure :: get_log => eval_tree_get_log
  procedure :: get_int => eval_tree_get_int
  procedure :: get_real => eval_tree_get_real
  procedure :: get_cmplx => eval_tree_get_cmplx
  procedure :: get_pdg_array => eval_tree_get_pdg_array
  procedure :: get_subevt => eval_tree_get_subevt
  procedure :: get_string => eval_tree_get_string
```

⟨*Eval trees: procedures*⟩+≡

```
  function eval_tree_get_result_type (expr) result (type)
    integer :: type
    class(eval_tree_t), intent(in) :: expr
    if (associated (expr%root)) then
       type = expr%root%result_type
    else
       type = V_NONE
    end if
  end function eval_tree_get_result_type

  function eval_tree_result_is_known (expr) result (flag)
    logical :: flag
    class(eval_tree_t), intent(in) :: expr
    if (associated (expr%root)) then
```

```
      select case (expr%root%result_type)
      case (V_LOG, V_INT, V_REAL)
         flag = expr%root%value_is_known
      case default
         flag = .true.
      end select
   else
      flag = .false.
   end if
end function eval_tree_result_is_known

function eval_tree_result_is_known_ptr (expr) result (ptr)
   logical, pointer :: ptr
   class(eval_tree_t), intent(in) :: expr
   logical, target, save :: known = .true.
   if (associated (expr%root)) then
      select case (expr%root%result_type)
      case (V_LOG, V_INT, V_REAL)
         ptr => expr%root%value_is_known
      case default
         ptr => known
      end select
   else
      ptr => null ()
   end if
end function eval_tree_result_is_known_ptr

function eval_tree_get_log (expr) result (lval)
   logical :: lval
   class(eval_tree_t), intent(in) :: expr
   if (associated (expr%root))  lval = expr%root%lval
end function eval_tree_get_log

function eval_tree_get_int (expr) result (ival)
   integer :: ival
   class(eval_tree_t), intent(in) :: expr
   if (associated (expr%root)) then
      select case (expr%root%result_type)
      case (V_INT);  ival = expr%root%ival
      case (V_REAL); ival = expr%root%rval
      case (V_CMPLX); ival = expr%root%cval
      end select
   end if
end function eval_tree_get_int

function eval_tree_get_real (expr) result (rval)
   real(default) :: rval
   class(eval_tree_t), intent(in) :: expr
   if (associated (expr%root)) then
      select case (expr%root%result_type)
      case (V_REAL); rval = expr%root%rval
      case (V_INT);  rval = expr%root%ival
      case (V_CMPLX);  rval = expr%root%cval
      end select
```

```
  end if
end function eval_tree_get_real

function eval_tree_get_cmplx (expr) result (cval)
  complex(default) :: cval
  class(eval_tree_t), intent(in) :: expr
  if (associated (expr%root)) then
     select case (expr%root%result_type)
     case (V_CMPLX); cval = expr%root%cval
     case (V_REAL); cval = expr%root%rval
     case (V_INT);  cval = expr%root%ival
     end select
  end if
end function eval_tree_get_cmplx

function eval_tree_get_pdg_array (expr) result (aval)
  type(pdg_array_t) :: aval
  class(eval_tree_t), intent(in) :: expr
  if (associated (expr%root)) then
     aval = expr%root%aval
  end if
end function eval_tree_get_pdg_array

function eval_tree_get_subevt (expr) result (pval)
  type(subevt_t) :: pval
  class(eval_tree_t), intent(in) :: expr
  if (associated (expr%root)) then
     pval = expr%root%pval
  end if
end function eval_tree_get_subevt

function eval_tree_get_string (expr) result (sval)
  type(string_t) :: sval
  class(eval_tree_t), intent(in) :: expr
  if (associated (expr%root)) then
     sval = expr%root%sval
  end if
end function eval_tree_get_string
```

Return a pointer to the value of the top node.

⟨*Eval trees: procedures*⟩+≡

```
function eval_tree_get_log_ptr (eval_tree) result (lval)
  logical, pointer :: lval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     lval => eval_tree%root%lval
  else
     lval => null ()
  end if
end function eval_tree_get_log_ptr

function eval_tree_get_int_ptr (eval_tree) result (ival)
  integer, pointer :: ival
```

```
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     ival => eval_tree%root%ival
  else
     ival => null ()
  end if
end function eval_tree_get_int_ptr

function eval_tree_get_real_ptr (eval_tree) result (rval)
  real(default), pointer :: rval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     rval => eval_tree%root%rval
  else
     rval => null ()
  end if
end function eval_tree_get_real_ptr

function eval_tree_get_cmplx_ptr (eval_tree) result (cval)
  complex(default), pointer :: cval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     cval => eval_tree%root%cval
  else
     cval => null ()
  end if
end function eval_tree_get_cmplx_ptr

function eval_tree_get_subevt_ptr (eval_tree) result (pval)
  type(subevt_t), pointer :: pval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     pval => eval_tree%root%pval
  else
     pval => null ()
  end if
end function eval_tree_get_subevt_ptr

function eval_tree_get_pdg_array_ptr (eval_tree) result (aval)
  type(pdg_array_t), pointer :: aval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     aval => eval_tree%root%aval
  else
     aval => null ()
  end if
end function eval_tree_get_pdg_array_ptr

function eval_tree_get_string_ptr (eval_tree) result (sval)
  type(string_t), pointer :: sval
  type(eval_tree_t), intent(in) :: eval_tree
  if (associated (eval_tree%root)) then
     sval => eval_tree%root%sval
  else
```

```
      sval => null ()
    end if
  end function eval_tree_get_string_ptr
```

⟨*Eval trees: eval tree: TBP*⟩+≡

```
  procedure :: write => eval_tree_write
```

⟨*Eval trees: procedures*⟩+≡

```
  subroutine eval_tree_write (expr, unit, write_vars)
    class(eval_tree_t), intent(in) :: expr
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: write_vars
    integer :: u
    logical :: vl
    u = given_output_unit (unit);  if (u < 0)  return
    vl = .false.;  if (present (write_vars))  vl = write_vars
    write (u, "(1x,A)") "Evaluation tree:"
    if (associated (expr%root)) then
       call eval_node_write_rec (expr%root, unit)
    else
       write (u, "(3x,A)") "[empty]"
    end if
    if (vl)  call var_list_write (expr%var_list, unit)
  end subroutine eval_tree_write
```

Use the written representation for generating an MD5 sum:

⟨*Eval trees: procedures*⟩+≡

```
  function eval_tree_get_md5sum (eval_tree) result (md5sum_et)
    character(32) :: md5sum_et
    type(eval_tree_t), intent(in) :: eval_tree
    integer :: u
    u = free_unit ()
    open (unit = u, status = "scratch", action = "readwrite")
    call eval_tree_write (eval_tree, unit=u)
    rewind (u)
    md5sum_et = md5sum (u)
    close (u)
  end function eval_tree_get_md5sum
```

28.3.10 Direct evaluation

These procedures create an eval tree and evaluate it on-the-fly, returning only the final value. The evaluation must yield a well-defined value, unless the `is_known` flag is present, which will be set accordingly.

⟨*Eval trees: public*⟩+≡

```
  public :: eval_log
  public :: eval_int
  public :: eval_real
  public :: eval_cmplx
  public :: eval_subevt
  public :: eval_pdg_array
  public :: eval_string
```

⟨*Eval trees: procedures*⟩+≡

```
function eval_log &
     (parse_node, var_list, subevt, is_known) result (lval)
  logical :: lval
  type(parse_node_t), intent(in), target :: parse_node
  type(var_list_t), intent(in), target :: var_list
  type(subevt_t), intent(in), optional, target :: subevt
  logical, intent(out), optional :: is_known
  type(eval_tree_t), target :: eval_tree
  call eval_tree_init_lexpr &
       (eval_tree, parse_node, var_list, subevt)
  call eval_tree_evaluate (eval_tree)
  if (eval_tree_result_is_known (eval_tree)) then
     if (present (is_known))  is_known = .true.
     lval = eval_tree_get_log (eval_tree)
  else if (present (is_known)) then
     is_known = .false.
  else
     call eval_tree_unknown (eval_tree, parse_node)
     lval = .false.
  end if
  call eval_tree_final (eval_tree)
end function eval_log

function eval_int &
     (parse_node, var_list, subevt, is_known) result (ival)
  integer :: ival
  type(parse_node_t), intent(in), target :: parse_node
  type(var_list_t), intent(in), target :: var_list
  type(subevt_t), intent(in), optional, target :: subevt
  logical, intent(out), optional :: is_known
  type(eval_tree_t), target :: eval_tree
  call eval_tree_init_expr &
       (eval_tree, parse_node, var_list, subevt)
  call eval_tree_evaluate (eval_tree)
  if (eval_tree_result_is_known (eval_tree)) then
     if (present (is_known))  is_known = .true.
     ival = eval_tree_get_int (eval_tree)
  else if (present (is_known)) then
     is_known = .false.
  else
     call eval_tree_unknown (eval_tree, parse_node)
     ival = 0
  end if
  call eval_tree_final (eval_tree)
end function eval_int

function eval_real &
     (parse_node, var_list, subevt, is_known) result (rval)
  real(default) :: rval
  type(parse_node_t), intent(in), target :: parse_node
  type(var_list_t), intent(in), target :: var_list
  type(subevt_t), intent(in), optional, target :: subevt
  logical, intent(out), optional :: is_known
```

```
  type(eval_tree_t), target :: eval_tree
  call eval_tree_init_expr &
       (eval_tree, parse_node, var_list, subevt)
  call eval_tree_evaluate (eval_tree)
  if (eval_tree_result_is_known (eval_tree)) then
     if (present (is_known))  is_known = .true.
     rval = eval_tree_get_real (eval_tree)
  else if (present (is_known)) then
     is_known = .false.
  else
     call eval_tree_unknown (eval_tree, parse_node)
     rval = 0
  end if
  call eval_tree_final (eval_tree)
end function eval_real

function eval_cmplx &
     (parse_node, var_list, subevt, is_known) result (cval)
  complex(default) :: cval
  type(parse_node_t), intent(in), target :: parse_node
  type(var_list_t), intent(in), target :: var_list
  type(subevt_t), intent(in), optional, target :: subevt
  logical, intent(out), optional :: is_known
  type(eval_tree_t), target :: eval_tree
  call eval_tree_init_expr &
       (eval_tree, parse_node, var_list, subevt)
  call eval_tree_evaluate (eval_tree)
  if (eval_tree_result_is_known (eval_tree)) then
     if (present (is_known))  is_known = .true.
     cval = eval_tree_get_cmplx (eval_tree)
  else if (present (is_known)) then
     is_known = .false.
  else
     call eval_tree_unknown (eval_tree, parse_node)
     cval = 0
  end if
  call eval_tree_final (eval_tree)
end function eval_cmplx

function eval_subevt &
     (parse_node, var_list, subevt, is_known) result (pval)
  type(subevt_t) :: pval
  type(parse_node_t), intent(in), target :: parse_node
  type(var_list_t), intent(in), target :: var_list
  type(subevt_t), intent(in), optional, target :: subevt
  logical, intent(out), optional :: is_known
  type(eval_tree_t), target :: eval_tree
  call eval_tree_init_pexpr &
       (eval_tree, parse_node, var_list, subevt)
  call eval_tree_evaluate (eval_tree)
  if (eval_tree_result_is_known (eval_tree)) then
     if (present (is_known))  is_known = .true.
     pval = eval_tree_get_subevt (eval_tree)
  else if (present (is_known)) then
```

```
      is_known = .false.
   else
      call eval_tree_unknown (eval_tree, parse_node)
   end if
   call eval_tree_final (eval_tree)
end function eval_subevt

function eval_pdg_array &
      (parse_node, var_list, subevt, is_known) result (aval)
   type(pdg_array_t) :: aval
   type(parse_node_t), intent(in), target :: parse_node
   type(var_list_t), intent(in), target :: var_list
   type(subevt_t), intent(in), optional, target :: subevt
   logical, intent(out), optional :: is_known
   type(eval_tree_t), target :: eval_tree
   call eval_tree_init_cexpr &
         (eval_tree, parse_node, var_list, subevt)
   call eval_tree_evaluate (eval_tree)
   if (eval_tree_result_is_known (eval_tree)) then
      if (present (is_known))  is_known = .true.
      aval = eval_tree_get_pdg_array (eval_tree)
   else if (present (is_known)) then
      is_known = .false.
   else
      call eval_tree_unknown (eval_tree, parse_node)
   end if
   call eval_tree_final (eval_tree)
end function eval_pdg_array

function eval_string &
      (parse_node, var_list, subevt, is_known) result (sval)
   type(string_t) :: sval
   type(parse_node_t), intent(in), target :: parse_node
   type(var_list_t), intent(in), target :: var_list
   type(subevt_t), intent(in), optional, target :: subevt
   logical, intent(out), optional :: is_known
   type(eval_tree_t), target :: eval_tree
   call eval_tree_init_sexpr &
         (eval_tree, parse_node, var_list, subevt)
   call eval_tree_evaluate (eval_tree)
   if (eval_tree_result_is_known (eval_tree)) then
      if (present (is_known))  is_known = .true.
      sval = eval_tree_get_string (eval_tree)
   else if (present (is_known)) then
      is_known = .false.
   else
      call eval_tree_unknown (eval_tree, parse_node)
      sval = ""
   end if
   call eval_tree_final (eval_tree)
end function eval_string
```

Here is a variant that returns numeric values of all possible kinds, the appropriate kind to be selected later:

⟨*Eval trees: public*⟩+≡

```
public :: eval_numeric
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_numeric &
     (parse_node, var_list, subevt, ival, rval, cval, &
      is_known, result_type)
  type(parse_node_t), intent(in), target :: parse_node
  type(var_list_t), intent(in), target :: var_list
  type(subevt_t), intent(in), optional, target :: subevt
  integer, intent(out), optional :: ival
  real(default), intent(out), optional :: rval
  complex(default), intent(out), optional :: cval
  logical, intent(out), optional :: is_known
  integer, intent(out), optional :: result_type
  type(eval_tree_t), target :: eval_tree
  call eval_tree_init_expr &
       (eval_tree, parse_node, var_list, subevt)
  call eval_tree_evaluate (eval_tree)
  if (eval_tree_result_is_known (eval_tree)) then
     if (present (ival))  ival = eval_tree_get_int (eval_tree)
     if (present (rval))  rval = eval_tree_get_real (eval_tree)
     if (present (cval))  cval = eval_tree_get_cmplx (eval_tree)
     if (present (is_known))  is_known = .true.
  else
     call eval_tree_unknown (eval_tree, parse_node)
     if (present (ival))  ival = 0
     if (present (rval))  rval = 0
     if (present (cval))  cval = 0
     if (present (is_known))  is_known = .false.
  end if
  if (present (result_type))  &
       result_type = eval_tree_get_result_type (eval_tree)
  call eval_tree_final (eval_tree)
end subroutine eval_numeric
```

Error message with debugging info:

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_tree_unknown (eval_tree, parse_node)
  type(eval_tree_t), intent(in) :: eval_tree
  type(parse_node_t), intent(in) :: parse_node
  call parse_node_write_rec (parse_node)
  call eval_tree_write (eval_tree)
  call msg_error ("Evaluation yields an undefined result, inserting default")
end subroutine eval_tree_unknown
```

28.3.11 Factory Type

Since `eval_tree_t` is an implementation of `expr_t`, we also need a matching factory type and build method.

⟨*Eval trees: public*⟩+≡

```
public :: eval_tree_factory_t
```

⟨*Eval trees: types*⟩+≡

```
type, extends (expr_factory_t) :: eval_tree_factory_t
   private
   type(parse_node_t), pointer :: pn => null ()
 contains
⟨Eval trees: eval tree factory: TBP⟩
end type eval_tree_factory_t
```

Output: delegate to the output of the embedded parse node.

⟨*Eval trees: eval tree factory: TBP*⟩≡

```
procedure :: write => eval_tree_factory_write
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_tree_factory_write (expr_factory, unit)
  class(eval_tree_factory_t), intent(in) :: expr_factory
  integer, intent(in), optional :: unit
  if (associated (expr_factory%pn)) then
     call parse_node_write_rec (expr_factory%pn, unit)
  end if
end subroutine eval_tree_factory_write
```

Initializer: take a parse node and hide it thus from the environment.

⟨*Eval trees: eval tree factory: TBP*⟩+≡

```
procedure :: init => eval_tree_factory_init
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_tree_factory_init (expr_factory, pn)
  class(eval_tree_factory_t), intent(out) :: expr_factory
  type(parse_node_t), intent(in), pointer :: pn
  expr_factory%pn => pn
end subroutine eval_tree_factory_init
```

Factory method: allocate expression with correct eval tree type. If the stored parse node is not associate, don't allocate.

⟨*Eval trees: eval tree factory: TBP*⟩+≡

```
procedure :: build => eval_tree_factory_build
```

⟨*Eval trees: procedures*⟩+≡

```
subroutine eval_tree_factory_build (expr_factory, expr)
  class(eval_tree_factory_t), intent(in) :: expr_factory
  class(expr_t), intent(out), allocatable :: expr
  if (associated (expr_factory%pn)) then
     allocate (eval_tree_t :: expr)
     select type (expr)
     type is (eval_tree_t)
        expr%pn => expr_factory%pn
     end select
  end if
end subroutine eval_tree_factory_build
```

28.3.12 Unit tests

Test module, followed by the corresponding implementation module.

⟨eval_trees_ut.f90⟩≡

```
⟨File header⟩

module eval_trees_ut
  use unit_tests
  use eval_trees_uti

⟨Standard module head⟩

⟨Eval trees: public test⟩

contains

⟨Eval trees: test driver⟩

end module eval_trees_ut
```

⟨eval_trees_uti.f90⟩≡

```
⟨File header⟩

module eval_trees_uti

⟨Use kinds⟩
⟨Use strings⟩

  use ifiles
  use lexers
  use lorentz
  use syntax_rules, only: syntax_write
  use pdg_arrays
  use subevents
  use variables
  use observables

  use eval_trees

⟨Standard module head⟩

⟨Eval trees: test declarations⟩

contains

⟨Eval trees: tests⟩

end module eval_trees_uti
```

API: driver for the unit tests below.

⟨*Eval trees: public test*⟩≡

```
  public :: expressions_test
```

⟨*Eval trees: test driver*⟩≡

```
  subroutine expressions_test (u, results)
    integer, intent(in) :: u
```

```
  type (test_results_t), intent(inout) :: results
⟨Eval trees: execute tests⟩
end subroutine expressions_test
```

Testing the routines of the expressions module. First a simple unary observable and the node evaluation.

⟨*Eval trees: execute tests*⟩≡

```
call test (expressions_1, "expressions_1", &
     "check simple observable", &
     u, results)
```

⟨*Eval trees: test declarations*⟩≡

```
public :: expressions_1
```

⟨*Eval trees: tests*⟩≡

```
subroutine expressions_1 (u)
  integer, intent(in) :: u
  type(var_list_t), pointer :: var_list => null ()
  type(eval_node_t), pointer :: node => null ()
  type(prt_t), pointer :: prt => null ()
  type(string_t) :: var_name

  write (u, "(A)")  "* Test output: Expressions"
  write (u, "(A)")  "*   Purpose: test simple observable and node evaluation"
  write (u, "(A)")

  write (u, "(A)")  "* Setting a unary observable:"
  write (u, "(A)")

  allocate (var_list)
  allocate (prt)
  call var_list_set_observables_unary (var_list, prt)
  call var_list%write (u)

  write (u, "(A)")  "* Evaluating the observable node:"
  write (u, "(A)")

  var_name = "PDG"

  allocate (node)
  call node%test_obs (var_list, var_name)
  call node%write (u)

  write (u, "(A)")  "* Cleanup"
  write (u, "(A)")

  call node%final_rec ()
  deallocate (node)
  call var_list%final ()
  deallocate (var_list)
  deallocate (prt)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: expressions_1"
```

```
end subroutine expressions_1
```

Parse a complicated expression, transfer it to a parse tree and evaluate.

⟨*Eval trees: execute tests*⟩+≡

```
call test (expressions_2, "expressions_2", &
     "check expression transfer to parse tree", &
     u, results)
```

⟨*Eval trees: test declarations*⟩+≡

```
public :: expressions_2
```

⟨*Eval trees: tests*⟩+≡

```
subroutine expressions_2 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(stream_t) :: stream
  type(eval_tree_t) :: eval_tree
  type(string_t) :: expr_text
  type(var_list_t), pointer :: var_list => null ()

  write (u, "(A)")  "* Test output: Expressions"
  write (u, "(A)")  "*   Purpose: test parse routines"
  write (u, "(A)")

  call syntax_expr_init ()
  call syntax_write (syntax_expr, u)
  allocate (var_list)
  call var_list_append_real (var_list, var_str ("tolerance"), 0._default)
  call var_list_append_real (var_list, var_str ("x"), -5._default)
  call var_list_append_int  (var_list, var_str ("foo"), -27)
  call var_list_append_real (var_list, var_str ("mb"), 4._default)
  expr_text = &
       "let real twopi = 2 * pi in" // &
       "  twopi * sqrt (25.d0 - mb^2)" // &
       "  / (let int mb_or_0 = max (mb, 0) in" // &
       "       1 + (if -1 TeV <= x < mb_or_0 then abs(x) else x endif))"
  call ifile_append (ifile, expr_text)
  call stream_init (stream, ifile)
  call var_list%write (u)
  call eval_tree%init_stream (stream, var_list=var_list)
  call eval_tree%evaluate ()
  call eval_tree%write (u)

  write (u, "(A)")  "* Input string:"
  write (u, "(A,A)")  "     ", char (expr_text)
  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call stream_final (stream)
  call ifile_final (ifile)
  call eval_tree%final ()
  call var_list%final ()
  deallocate (var_list)
  call syntax_expr_final ()
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: expressions_2"

  end subroutine expressions_2
```

Test a subevent expression.

⟨*Eval trees: execute tests*⟩+≡

```
  call test (expressions_3, "expressions_3", &
       "check subevent expressions", &
       u, results)
```

⟨*Eval trees: test declarations*⟩+≡

```
  public :: expressions_3
```

⟨*Eval trees: tests*⟩+≡

```
  subroutine expressions_3 (u)
    integer, intent(in) :: u
    type(subevt_t) :: subevt

    write (u, "(A)")  "* Test output: Expressions"
    write (u, "(A)")  "*   Purpose: test subevent expressions"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize subevent:"
    write (u, "(A)")

    call subevt_init (subevt)
    call subevt_reset (subevt, 1)
    call subevt_set_incoming (subevt, 1, &
         22, vector4_moving (1.e3_default, 1.e3_default, 1), &
         0._default, [2])
    call subevt_write (subevt, u)
    call subevt_reset (subevt, 4)
    call subevt_reset (subevt, 3)
    call subevt_set_incoming (subevt, 1, &
         21, vector4_moving (1.e3_default, 1.e3_default, 3), &
         0._default, [1])
    call subevt_polarize (subevt, 1, -1)
    call subevt_set_outgoing (subevt, 2, &
         1, vector4_moving (0._default, 1.e3_default, 3), &
         -1.e6_default, [7])
    call subevt_set_composite (subevt, 3, &
         vector4_moving (-1.e3_default, 0._default, 3), &
         [2, 7])
    call subevt_write (subevt, u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: expressions_3"

  end subroutine expressions_3
```

Test expressions from a PDG array.

⟨*Eval trees: execute tests*⟩+≡

```
  call test (expressions_4, "expressions_4", &
```

```
        "check pdg array expressions", &
        u, results)
```

⟨*Eval trees: test declarations*⟩+≡

```
    public :: expressions_4
```

⟨*Eval trees: tests*⟩+≡

```
    subroutine expressions_4 (u)
      integer, intent(in) :: u
      type(subevt_t), target :: subevt
      type(string_t) :: expr_text
      type(ifile_t) :: ifile
      type(stream_t) :: stream
      type(eval_tree_t) :: eval_tree
      type(var_list_t), pointer :: var_list => null ()
      type(pdg_array_t) :: aval

      write (u, "(A)")  "* Test output: Expressions"
      write (u, "(A)")  "*   Purpose: test pdg array expressions"
      write (u, "(A)")

      write (u, "(A)")  "* Initialization:"
      write (u, "(A)")

      call syntax_pexpr_init ()
      call syntax_write (syntax_pexpr, u)
      allocate (var_list)
      call var_list_append_real (var_list, var_str ("tolerance"), 0._default)
      aval = 0
      call var_list_append_pdg_array (var_list, var_str ("particle"), aval)
      aval = [11,-11]
      call var_list_append_pdg_array (var_list, var_str ("lepton"), aval)
      aval = 22
      call var_list_append_pdg_array (var_list, var_str ("photon"), aval)
      aval = 1
      call var_list_append_pdg_array (var_list, var_str ("u"), aval)
      call subevt_init (subevt)
      call subevt_reset (subevt, 6)
      call subevt_set_incoming (subevt, 1, &
           1, vector4_moving (1._default, 1._default, 1), 0._default)
      call subevt_set_incoming (subevt, 2, &
           -1, vector4_moving (2._default, 2._default, 1), 0._default)
      call subevt_set_outgoing (subevt, 3, &
           22, vector4_moving (3._default, 3._default, 1), 0._default)
      call subevt_set_outgoing (subevt, 4, &
           22, vector4_moving (4._default, 4._default, 1), 0._default)
      call subevt_set_outgoing (subevt, 5, &
           11, vector4_moving (5._default, 5._default, 1), 0._default)
      call subevt_set_outgoing (subevt, 6, &
           -11, vector4_moving (6._default, 6._default, 1), 0._default)
      write (u, "(A)")
      write (u, "(A)")  "* Expression:"
      expr_text = &
           "let alias quark = pdg(1):pdg(2):pdg(3) in" // &
           "  any E > 3 GeV " // &
```

```
      "    [sort by - Pt " // &
      "       [select if Index < 6 " // &
      "          [photon:pdg(-11):pdg(3):quark " // &
      "           & incoming particle]]]" // &
      "  and" // &
      "  eval Theta [extract index -1 [photon]] > 45 degree" // &
      "  and" // &
      "  count [incoming photon] * 3 > 0"
  write (u, "(A,A)")  "     ", char (expr_text)
  write (u, "(A)")

  write (u, "(A)")
  write (u, "(A)")  "* Extract the evaluation tree:"
  write (u, "(A)")

  call ifile_append (ifile, expr_text)
  call stream_init (stream, ifile)
  call eval_tree%init_stream (stream, var_list, subevt, V_LOG)
  call eval_tree%write (u)
  call eval_tree%evaluate ()

  write (u, "(A)")
  write (u, "(A)")  "* Evaluate the tree:"
  write (u, "(A)")

  call eval_tree%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"
  write (u, "(A)")

  call stream_final (stream)
  call ifile_final (ifile)
  call eval_tree%final ()
  call var_list%final ()
  deallocate (var_list)
  call syntax_pexpr_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: expressions_4"

end subroutine expressions_4
```

28.4 Physics Models

A model object represents a physics model. It contains a table of particle data, a list of parameters, and a vertex table. The list of parameters is a variable list which includes the real parameters (which are pointers to the particle data table) and PDG array variables for the particles themselves. The vertex list is used for phase-space generation, not for calculating the matrix element.

The actual numeric model data are in the base type `model_data_t`, as part of the `qft` section. We implement the `model_t` as an extension of this, for convenient direct access to the base-type methods via inheritance. (Alternatively, we could delegate these calls explicitly.) The extension contains administrative additions, such as the methods for recalculating derived data and keeping the parameter set consistent. It thus acts as a proxy of the actual model-data object towards the `WHIZARD` package. There are further proxy objects, such as the `parameter_t` array which provides the interface to the actual numeric parameters.

Model definitions are read from model files. Therefore, this module contains a parser for model files. The parameter definitions (derived parameters) are Sindarin expressions.

The models, as read from file, are stored in a model library which is a simple list of model definitions. For setting up a process object we should make a copy (an instance) of a model, which gets the current parameter values from the global variable list.

28.4.1 Module

⟨`models.f90`⟩≡
  ⟨*File header*⟩

```
  module models

    use, intrinsic :: iso_c_binding !NODEP!

  ⟨Use kinds⟩
    use kinds, only: c_default_float
  ⟨Use strings⟩
    use io_units
    use diagnostics
    use md5
    use os_interface
    use physics_defs, only: UNDEFINED
    use model_data

    use ifiles
    use syntax_rules
    use lexers
    use parser
    use pdg_arrays
    use variables
    use expr_base
    use eval_trees
```

```
  use ttv_formfactors, only: init_parameters
```

⟨*Standard module head*⟩

⟨*Models: public*⟩

⟨*Models: parameters*⟩

⟨*Models: types*⟩

⟨*Models: interfaces*⟩

⟨*Models: variables*⟩

```
contains
```

⟨*Models: procedures*⟩

```
end module models
```

28.4.2 Physics Parameters

A parameter has a name, a value. Derived parameters also have a definition in terms of other parameters, which is stored as an `eval_tree`. External parameters are set by an external program.

This parameter object should be considered as a proxy object. The parameter name and value are stored in a corresponding `modelpar_data_t` object which is located in a `model_data_t` object. The latter is a component of the `model_t` handler. Methods of `parameter_t` can be delegated to the `par_data_t` component.

The `pn` component is a pointer to the parameter definition inside the model parse tree. It allows us to recreate the `eval_tree` when making copies (instances) of the parameter object.

⟨*Models: parameters*⟩≡

```
    integer, parameter :: PAR_NONE = 0, PAR_UNUSED = -1
    integer, parameter :: PAR_INDEPENDENT = 1, PAR_DERIVED = 2
    integer, parameter :: PAR_EXTERNAL = 3
```

⟨*Models: types*⟩≡

```
    type :: parameter_t
       private
       integer :: type  = PAR_NONE
       class(modelpar_data_t), pointer :: data => null ()
       type(parse_node_t), pointer :: pn => null ()
       class(expr_t), allocatable :: expr
     contains
     ⟨Models: parameter: TBP⟩
    end type parameter_t
```

Initialization depends on parameter type. Independent parameters are initialized by a constant value or a constant numerical expression (which may contain

a unit). Derived parameters are initialized by an arbitrary numerical expression, which makes use of the current variable list. The expression is evaluated by the function `parameter_reset`.

This implementation supports only real parameters and real values.

⟨*Models: parameter: TBP*⟩≡

```
procedure :: init_independent_value => parameter_init_independent_value
procedure :: init_independent => parameter_init_independent
procedure :: init_derived => parameter_init_derived
procedure :: init_external => parameter_init_external
procedure :: init_unused => parameter_init_unused
```

⟨*Models: procedures*⟩≡

```
subroutine parameter_init_independent_value (par, par_data, name, value)
  class(parameter_t), intent(out) :: par
  class(modelpar_data_t), intent(in), target :: par_data
  type(string_t), intent(in) :: name
  real(default), intent(in) :: value
  par%type = PAR_INDEPENDENT
  par%data => par_data
  call par%data%init (name, value)
end subroutine parameter_init_independent_value

subroutine parameter_init_independent (par, par_data, name, pn)
  class(parameter_t), intent(out) :: par
  class(modelpar_data_t), intent(in), target :: par_data
  type(string_t), intent(in) :: name
  type(parse_node_t), intent(in), target :: pn
  par%type = PAR_INDEPENDENT
  par%pn => pn
  allocate (eval_tree_t :: par%expr)
  select type (expr => par%expr)
  type is (eval_tree_t)
     call expr%init_numeric_value (pn)
  end select
  par%data => par_data
  call par%data%init (name, par%expr%get_real ())
end subroutine parameter_init_independent

subroutine parameter_init_derived (par, par_data, name, pn, var_list)
  class(parameter_t), intent(out) :: par
  class(modelpar_data_t), intent(in), target :: par_data
  type(string_t), intent(in) :: name
  type(parse_node_t), intent(in), target :: pn
  type(var_list_t), intent(in), target :: var_list
  par%type = PAR_DERIVED
  par%pn => pn
  allocate (eval_tree_t :: par%expr)
  select type (expr => par%expr)
  type is (eval_tree_t)
     call expr%init_expr (pn, var_list=var_list)
  end select
  par%data => par_data
!    call par%expr%evaluate ()
  call par%data%init (name, 0._default)
```

```
end subroutine parameter_init_derived

subroutine parameter_init_external (par, par_data, name)
  class(parameter_t), intent(out) :: par
  class(modelpar_data_t), intent(in), target :: par_data
  type(string_t), intent(in) :: name
  par%type = PAR_EXTERNAL
  par%data => par_data
  call par%data%init (name, 0._default)
end subroutine parameter_init_external

subroutine parameter_init_unused (par, par_data, name)
  class(parameter_t), intent(out) :: par
  class(modelpar_data_t), intent(in), target :: par_data
  type(string_t), intent(in) :: name
  par%type = PAR_UNUSED
  par%data => par_data
  call par%data%init (name, 0._default)
end subroutine parameter_init_unused
```

The finalizer is needed for the evaluation tree in the definition.

⟨*Models: parameter: TBP*⟩+≡

```
procedure :: final => parameter_final
```

⟨*Models: procedures*⟩+≡

```
subroutine parameter_final (par)
  class(parameter_t), intent(inout) :: par
  if (allocated (par%expr)) then
     call par%expr%final ()
  end if
end subroutine parameter_final
```

All derived parameters should be recalculated if some independent parameters have changed:

⟨*Models: parameter: TBP*⟩+≡

```
procedure :: reset_derived => parameter_reset_derived
```

⟨*Models: procedures*⟩+≡

```
subroutine parameter_reset_derived (par)
  class(parameter_t), intent(inout) :: par
  select case (par%type)
  case (PAR_DERIVED)
     call par%expr%evaluate ()
     par%data = par%expr%get_real ()
  end select
end subroutine parameter_reset_derived
```

Output. [We should have a formula format for the eval tree, suitable for input and output!]

⟨*Models: parameter: TBP*⟩+≡

```
procedure :: write => parameter_write
```

⟨*Models: procedures*⟩+≡

```
subroutine parameter_write (par, unit, write_defs)
  class(parameter_t), intent(in) :: par
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: write_defs
  logical :: defs
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  defs = .false.;  if (present (write_defs))  defs = write_defs
  select case (par%type)
  case (PAR_INDEPENDENT)
     write (u, "(3x,A)", advance="no")  "parameter"
     call par%data%write (u)
  case (PAR_DERIVED)
     write (u, "(3x,A)", advance="no")  "derived"
     call par%data%write (u)
  case (PAR_EXTERNAL)
     write (u, "(3x,A)", advance="no")  "external"
     call par%data%write (u)
  case (PAR_UNUSED)
     write (u, "(3x,A)", advance="no")  "unused"
     write (u, "(1x,A)", advance="no")  char (par%data%get_name ())
  end select
  select case (par%type)
  case (PAR_DERIVED)
     if (defs) then
        call par%expr%write (unit)
     else
        write (u, *)
     end if
  case default
     write (u, *)
  end select
end subroutine parameter_write
```

Screen output variant. Restrict output to the given parameter type.

⟨*Models: parameter: TBP*⟩+≡

```
procedure :: show => parameter_show
```

⟨*Models: procedures*⟩+≡

```
subroutine parameter_show (par, l, u, partype)
  class(parameter_t), intent(in) :: par
  integer, intent(in) :: l, u
  integer, intent(in) :: partype
  if (par%type == partype) then
     call par%data%show (l, u)
  end if
end subroutine parameter_show
```

28.4.3 Model Object

A model object holds all information about parameters, particles, and vertices. For models that require an external program for parameter calculation, there is

the pointer to a function that does this calculation, given the set of independent and derived parameters.

As explained above, the type inherits from `model_data_t`, which is the actual storage for the model data.

When reading a model, we create a parse tree. Parameter definitions are available via parse nodes. Since we may need those later when making model instances, we keep the whole parse tree in the model definition (but not in the instances).

⟨*Models: public*⟩≡
```
public :: model_t
```

⟨*Models: types*⟩+≡
```
type, extends (model_data_t) :: model_t
   private
   character(32) :: md5sum = ""
   type(string_t), dimension(:), allocatable :: schemes
   type(string_t), allocatable :: selected_scheme
   type(parameter_t), dimension(:), allocatable :: par
   integer :: max_par_name_length = 0
   integer :: max_field_name_length = 0
   type(var_list_t) :: var_list
   type(string_t) :: dlname
   procedure(model_init_external_parameters), nopass, pointer :: &
        init_external_parameters => null ()
   type(dlaccess_t) :: dlaccess
   type(parse_tree_t) :: parse_tree
 contains
 ⟨Models: model: TBP⟩
end type model_t
```

This is the interface for a procedure that initializes the calculation of external parameters, given the array of all parameters.

⟨*Models: interfaces*⟩≡
```
abstract interface
   subroutine model_init_external_parameters (par) bind (C)
     import
     real(c_default_float), dimension(*), intent(inout) :: par
   end subroutine model_init_external_parameters
end interface
```

Initialization: Specify the number of parameters, particles, vertices and allocate memory. If an associated DL library is specified, load this library.

The model may already carry scheme information, so we have to save and restore the scheme number when actually initializing the `model_data_t` base.

⟨*Models: model: TBP*⟩≡
```
generic :: init => model_init
procedure, private :: model_init
```

⟨*Models: procedures*⟩+≡
```
subroutine model_init &
     (model, name, libname, os_data, n_par, n_prt, n_vtx)
  class(model_t), intent(inout) :: model
  type(string_t), intent(in) :: name, libname
```

```
    type(os_data_t), intent(in) :: os_data
    integer, intent(in) :: n_par, n_prt, n_vtx
    type(c_funptr) :: c_fptr
    type(string_t) :: libpath
    integer :: scheme_num
    scheme_num = model%get_scheme_num ()
    call model%basic_init (name, n_par, n_prt, n_vtx)
    call model%set_scheme_num (scheme_num)
    if (libname /= "") then
       if (.not. os_data%use_testfiles) then
          libpath = os_data%whizard_models_libpath_local
          model%dlname = os_get_dlname ( &
            libpath // "/" // libname, os_data, ignore=.true.)
       end if
       if (model%dlname == "") then
          libpath = os_data%whizard_models_libpath
          model%dlname = os_get_dlname (libpath // "/" // libname, os_data)
       end if
    else
       model%dlname = ""
    end if
    if (model%dlname /= "") then
       if (.not. dlaccess_is_open (model%dlaccess)) then
          if (logging) &
               call msg_message ("Loading model auxiliary library '" &
               // char (libpath) // "/" // char (model%dlname) // "'")
          call dlaccess_init (model%dlaccess, os_data%whizard_models_libpath, &
               model%dlname, os_data)
          if (dlaccess_has_error (model%dlaccess)) then
             call msg_message (char (dlaccess_get_error (model%dlaccess)))
             call msg_fatal ("Loading model auxiliary library '" &
                  // char (model%dlname) // "' failed")
             return
          end if
          c_fptr = dlaccess_get_c_funptr (model%dlaccess, &
               var_str ("init_external_parameters"))
          if (dlaccess_has_error (model%dlaccess)) then
             call msg_message (char (dlaccess_get_error (model%dlaccess)))
             call msg_fatal ("Loading function from auxiliary library '" &
                  // char (model%dlname) // "' failed")
             return
          end if
          call c_f_procpointer (c_fptr, model% init_external_parameters)
       end if
    end if
  end subroutine model_init
```

For a model instance, we do not attempt to load a DL library. This is the core of the full initializer above.

⟨*Models: model: TBP*⟩+≡

```
  procedure, private :: basic_init => model_basic_init
```

⟨*Models: procedures*⟩+≡

```
  subroutine model_basic_init (model, name, n_par, n_prt, n_vtx)
```

```
    class(model_t), intent(inout) :: model
    type(string_t), intent(in) :: name
    integer, intent(in) :: n_par, n_prt, n_vtx
    allocate (model%par (n_par))
    call model%model_data_t%init (name, n_par, 0, n_prt, n_vtx)
  end subroutine model_basic_init
```

Finalization: The variable list contains allocated pointers, also the parse tree. We also close the DL access object, if any, that enables external parameter calculation.

⟨*Models: model: TBP*⟩+≡

```
  procedure :: final => model_final
```

⟨*Models: procedures*⟩+≡

```
  subroutine model_final (model)
    class(model_t), intent(inout) :: model
    integer :: i
    if (allocated (model%par)) then
       do i = 1, size (model%par)
          call model%par(i)%final ()
       end do
    end if
    call model%var_list%final (follow_link=.false.)
    if (model%dlname /= "")  call dlaccess_final (model%dlaccess)
    call parse_tree_final (model%parse_tree)
    call model%model_data_t%final ()
  end subroutine model_final
```

Output. By default, the output is in the form of an input file. If `verbose` is true, for each derived parameter the definition (eval tree) is displayed, and the vertex hash table is shown.

⟨*Models: model: TBP*⟩+≡

```
  procedure :: write => model_write
```

⟨*Models: procedures*⟩+≡

```
  subroutine model_write (model, unit, verbose, &
       show_md5sum, show_variables, show_parameters, &
       show_particles, show_vertices, show_scheme)
    class(model_t), intent(in) :: model
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    logical, intent(in), optional :: show_md5sum
    logical, intent(in), optional :: show_variables
    logical, intent(in), optional :: show_parameters
    logical, intent(in), optional :: show_particles
    logical, intent(in), optional :: show_vertices
    logical, intent(in), optional :: show_scheme
    logical :: verb, show_md5, show_par, show_var
    integer :: u, i
    u = given_output_unit (unit);  if (u < 0)  return
    verb = .false.;  if (present (verbose))  verb = verbose
    show_md5 = .true.;  if (present (show_md5sum)) &
         show_md5 = show_md5sum
```

```
    show_par = .true.;  if (present (show_parameters)) &
         show_par = show_parameters
    show_var = verb;  if (present (show_variables)) &
         show_var = show_variables
    write (u, "(A,A,A)") 'model "', char (model%get_name ()), '"'
    if (show_md5 .and. model%md5sum /= "") &
         write (u, "(1x,A,A,A)") "! md5sum = '", model%md5sum, "'"
    if (model%has_schemes ()) then
       write (u, "(1x,A)", advance="no")  "! schemes ="
       do i = 1, size (model%schemes)
          if (i > 1)  write (u, "(',')", advance="no")
          write (u, "(1x,A,A,A)", advance="no") &
               "'", char (model%schemes(i)), "'"
       end do
       write (u, *)
       if (allocated (model%selected_scheme)) then
          write (u, "(1x,A,A,A,I0,A)")  &
               "! selected scheme = '", char (model%get_scheme ()), &
               "' (", model%get_scheme_num (), ")"
       end if
    end if
    if (show_par) then
       write (u, "(A)")
       do i = 1, size (model%par)
          call model%par(i)%write (u, write_defs=verbose)
       end do
    end if
    call model%model_data_t%write (unit, verbose, &
         show_md5sum, show_variables, &
         show_parameters=.false., &
         show_particles=show_particles, &
         show_vertices=show_vertices, &
         show_scheme=show_scheme)
    if (show_var) then
       write (u, "(A)")
       call var_list_write (model%var_list, unit, follow_link=.false.)
    end if
  end subroutine model_write
```

Screen output, condensed form.

⟨*Models: model: TBP*⟩+≡

```
  procedure :: show => model_show
```

⟨*Models: procedures*⟩+≡

```
  subroutine model_show (model, unit)
    class(model_t), intent(in) :: model
    integer, intent(in), optional :: unit
    integer :: i, u, l
    u = given_output_unit (unit)
    write (u, "(A,1x,A)")  "Model:", char (model%get_name ())
    if (model%has_schemes ()) then
       write (u, "(2x,A,A,A,I0,A)")  "Scheme: '", &
            char (model%get_scheme ()), "' (", model%get_scheme_num (), ")"
    end if
```

```
    l = model%max_field_name_length
    call model%show_fields (l, u)
    l = model%max_par_name_length
    if (any (model%par%type == PAR_INDEPENDENT)) then
       write (u, "(2x,A)")  "Independent parameters:"
       do i = 1, size (model%par)
          call model%par(i)%show (l, u, PAR_INDEPENDENT)
       end do
    end if
    if (any (model%par%type == PAR_DERIVED)) then
       write (u, "(2x,A)")  "Derived parameters:"
       do i = 1, size (model%par)
          call model%par(i)%show (l, u, PAR_DERIVED)
       end do
    end if
    if (any (model%par%type == PAR_EXTERNAL)) then
       write (u, "(2x,A)")  "External parameters:"
       do i = 1, size (model%par)
          call model%par(i)%show (l, u, PAR_EXTERNAL)
       end do
    end if
    if (any (model%par%type == PAR_UNUSED)) then
       write (u, "(2x,A)")  "Unused parameters:"
       do i = 1, size (model%par)
          call model%par(i)%show (l, u, PAR_UNUSED)
       end do
    end if
  end subroutine model_show
```

Show all fields/particles.

⟨*Models: model: TBP*⟩+≡

```
  procedure :: show_fields => model_show_fields
```

⟨*Models: procedures*⟩+≡

```
  subroutine model_show_fields (model, l, unit)
    class(model_t), intent(in), target :: model
    integer, intent(in) :: l
    integer, intent(in), optional :: unit
    type(field_data_t), pointer :: field
    integer :: u, i
    u = given_output_unit (unit)
    write (u, "(2x,A)")  "Particles:"
    do i = 1, model%get_n_field ()
       field => model%get_field_ptr_by_index (i)
       call field%show (l, u)
    end do
  end subroutine model_show_fields
```

Show the list of stable, unstable, polarized, or unpolarized particles, respectively.

⟨*Models: model: TBP*⟩+≡

```
  procedure :: show_stable => model_show_stable
  procedure :: show_unstable => model_show_unstable
  procedure :: show_polarized => model_show_polarized
  procedure :: show_unpolarized => model_show_unpolarized
```

⟨*Models: procedures*⟩+≡

```
subroutine model_show_stable (model, unit)
  class(model_t), intent(in), target :: model
  integer, intent(in), optional :: unit
  type(field_data_t), pointer :: field
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(A,1x)", advance="no")  "Stable particles:"
  do i = 1, model%get_n_field ()
     field => model%get_field_ptr_by_index (i)
     if (field%is_stable (.false.)) then
        write (u, "(1x,A)", advance="no")  char (field%get_name (.false.))
     end if
     if (field%has_antiparticle ()) then
        if (field%is_stable (.true.)) then
           write (u, "(1x,A)", advance="no")  char (field%get_name (.true.))
        end if
     end if
  end do
  write (u, *)
end subroutine model_show_stable

subroutine model_show_unstable (model, unit)
  class(model_t), intent(in), target :: model
  integer, intent(in), optional :: unit
  type(field_data_t), pointer :: field
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(A,1x)", advance="no")  "Unstable particles:"
  do i = 1, model%get_n_field ()
     field => model%get_field_ptr_by_index (i)
     if (.not. field%is_stable (.false.)) then
        write (u, "(1x,A)", advance="no")  char (field%get_name (.false.))
     end if
     if (field%has_antiparticle ()) then
        if (.not. field%is_stable (.true.)) then
           write (u, "(1x,A)", advance="no")  char (field%get_name (.true.))
        end if
     end if
  end do
  write (u, *)
end subroutine model_show_unstable

subroutine model_show_polarized (model, unit)
  class(model_t), intent(in), target :: model
  integer, intent(in), optional :: unit
  type(field_data_t), pointer :: field
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(A,1x)", advance="no")  "Polarized particles:"
  do i = 1, model%get_n_field ()
     field => model%get_field_ptr_by_index (i)
     if (field%is_polarized (.false.)) then
        write (u, "(1x,A)", advance="no") char (field%get_name (.false.))
```

```
      end if
      if (field%has_antiparticle ()) then
         if (field%is_polarized (.true.)) then
            write (u, "(1x,A)", advance="no") char (field%get_name (.true.))
         end if
      end if
   end do
   write (u, *)
end subroutine model_show_polarized

subroutine model_show_unpolarized (model, unit)
   class(model_t), intent(in), target :: model
   integer, intent(in), optional :: unit
   type(field_data_t), pointer :: field
   integer :: u, i
   u = given_output_unit (unit)
   write (u, "(A,1x)", advance="no")  "Unpolarized particles:"
   do i = 1, model%get_n_field ()
      field => model%get_field_ptr_by_index (i)
      if (.not. field%is_polarized (.false.)) then
         write (u, "(1x,A)", advance="no") &
              char (field%get_name (.false.))
      end if
      if (field%has_antiparticle ()) then
         if (.not. field%is_polarized (.true.)) then
            write (u, "(1x,A)", advance="no") char (field%get_name (.true.))
         end if
      end if
   end do
   write (u, *)
end subroutine model_show_unpolarized
```

Retrieve the MD5 sum of a model (actually, of the model file).

⟨*Models: model: TBP*⟩+≡

```
procedure :: get_md5sum => model_get_md5sum
```

⟨*Models: procedures*⟩+≡

```
function model_get_md5sum (model) result (md5sum)
   character(32) :: md5sum
   class(model_t), intent(in) :: model
   md5sum = model%md5sum
end function model_get_md5sum
```

Parameters are defined by an expression which may be constant or arbitrary.

Note: the auxiliary pointer `value_ptr` is a workaround for a bug in gfortran 4.8.1: the target of the function pointer is lost, if the pointer is provided directly as argument.

⟨*Models: model: TBP*⟩+≡

```
procedure :: &
     set_parameter_constant => model_set_parameter_constant
procedure, private :: &
     set_parameter_parse_node => model_set_parameter_parse_node
procedure :: &
```

```
      set_parameter_external => model_set_parameter_external
procedure :: &
      set_parameter_unused => model_set_parameter_unused
```

⟨*Models: procedures*⟩+≡

```
subroutine model_set_parameter_constant (model, i, name, value)
  class(model_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(string_t), intent(in) :: name
  real(default), intent(in) :: value
  logical, save, target :: known = .true.
  class(modelpar_data_t), pointer :: par_data
  real(default), pointer :: value_ptr
  par_data => model%get_par_real_ptr (i)
  call model%par(i)%init_independent_value (par_data, name, value)
  value_ptr => par_data%get_real_ptr ()
  call var_list_append_real_ptr (model%var_list, &
       name, value_ptr, &
       is_known=known, intrinsic=.true.)
  model%max_par_name_length = max (model%max_par_name_length, len (name))
end subroutine model_set_parameter_constant

subroutine model_set_parameter_parse_node (model, i, name, pn, constant)
  class(model_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(string_t), intent(in) :: name
  type(parse_node_t), intent(in), target :: pn
  logical, intent(in) :: constant
  logical, save, target :: known = .true.
  class(modelpar_data_t), pointer :: par_data
  real(default), pointer :: value_ptr
  par_data => model%get_par_real_ptr (i)
  if (constant) then
     call model%par(i)%init_independent (par_data, name, pn)
  else
     call model%par(i)%init_derived (par_data, name, pn, model%var_list)
  end if
  value_ptr => par_data%get_real_ptr ()
  call var_list_append_real_ptr (model%var_list, &
       name, value_ptr, &
       is_known=known, locked=.not.constant, intrinsic=.true.)
  model%max_par_name_length = max (model%max_par_name_length, len (name))
end subroutine model_set_parameter_parse_node

subroutine model_set_parameter_external (model, i, name)
  class(model_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(string_t), intent(in) :: name
  logical, save, target :: known = .true.
  class(modelpar_data_t), pointer :: par_data
  real(default), pointer :: value_ptr
  par_data => model%get_par_real_ptr (i)
  call model%par(i)%init_external (par_data, name)
  value_ptr => par_data%get_real_ptr ()
  call var_list_append_real_ptr (model%var_list, &
```

```
      name, value_ptr, &
      is_known=known, locked=.true., intrinsic=.true.)
  model%max_par_name_length = max (model%max_par_name_length, len (name))
end subroutine model_set_parameter_external

subroutine model_set_parameter_unused (model, i, name)
  class(model_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(string_t), intent(in) :: name
  class(modelpar_data_t), pointer :: par_data
  par_data => model%get_par_real_ptr (i)
  call model%par(i)%init_unused (par_data, name)
  call var_list_append_real (model%var_list, &
      name, locked=.true., intrinsic=.true.)
  model%max_par_name_length = max (model%max_par_name_length, len (name))
end subroutine model_set_parameter_unused
```

Make a copy of a parameter. We assume that the `model_data_t` parameter arrays have already been copied, so names and values are available in the current model, and can be used as targets. The eval tree should not be copied, since it should refer to the new variable list. The safe solution is to make use of the above initializers, which also take care of the building a new variable list.

⟨*Models: model: TBP*⟩+≡

```
procedure, private :: copy_parameter => model_copy_parameter
```

⟨*Models: procedures*⟩+≡

```
subroutine model_copy_parameter (model, i, par)
  class(model_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(parameter_t), intent(in) :: par
  type(string_t) :: name
  real(default) :: value
  name = par%data%get_name ()
  select case (par%type)
  case (PAR_INDEPENDENT)
     if (associated (par%pn)) then
        call model%set_parameter_parse_node (i, name, par%pn, &
             constant = .true.)
     else
        value = par%data%get_real ()
        call model%set_parameter_constant (i, name, value)
     end if
  case (PAR_DERIVED)
     call model%set_parameter_parse_node (i, name, par%pn, &
          constant = .false.)
  case (PAR_EXTERNAL)
     call model%set_parameter_external (i, name)
  case (PAR_UNUSED)
     call model%set_parameter_unused (i, name)
  end select
end subroutine model_copy_parameter
```

Recalculate all derived parameters.

⟨*Models: model: TBP*⟩+≡

```
procedure :: update_parameters => model_parameters_update
```

⟨*Models: procedures*⟩+≡

```
subroutine model_parameters_update (model)
  class(model_t), intent(inout) :: model
  integer :: i
  real(default), dimension(:), allocatable :: par
  do i = 1, size (model%par)
     call model%par(i)%reset_derived ()
  end do
  if (associated (model%init_external_parameters)) then
     allocate (par (model%get_n_real ()))
     call model%real_parameters_to_c_array (par)
     call model%init_external_parameters (par)
     call model%real_parameters_from_c_array (par)
     if (model%get_name() == var_str ("SM_tt_threshold")) &
          call set_threshold_parameters ()
  end if
contains
  subroutine set_threshold_parameters ()
    real(default) :: mpole, wtop
    !!! !!! !!! Workaround for OS-X and BSD which do not load
    !!! !!! !!! the global values created previously. Therefore
    !!! !!! !!! a second initialization for the threshold model,
    !!! !!! !!! where M1S is required to compute the top mass.
    call init_parameters (mpole, wtop, &
         par(20), par(21), par(22), &
         par(19), par(39), par(4), par(1), &
         par(2), par(10), par(24), par(25), &
         par(23), par(26), par(27), par(29), &
         par(30), par(31), par(32), par(33), &
         par(36) > 0._default, par(28))
  end subroutine set_threshold_parameters
end subroutine model_parameters_update
```

Initialize field data with PDG long name and PDG code.

⟨*Models: model: TBP*⟩+≡

```
procedure, private :: init_field => model_init_field
```

⟨*Models: procedures*⟩+≡

```
subroutine model_init_field (model, i, longname, pdg)
  class(model_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(string_t), intent(in) :: longname
  integer, intent(in) :: pdg
  type(field_data_t), pointer :: field
  field => model%get_field_ptr_by_index (i)
  call field%init (longname, pdg)
end subroutine model_init_field
```

Copy field data for index `i` from another particle which serves as a template. The name should be the unique long name.

⟨*Models: model: TBP*⟩+≡

```
procedure, private :: copy_field => model_copy_field
```

⟨*Models: procedures*⟩+≡

```
subroutine model_copy_field (model, i, name_src)
  class(model_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(string_t), intent(in) :: name_src
  type(field_data_t), pointer :: field_src, field
  field_src => model%get_field_ptr (name_src)
  field => model%get_field_ptr_by_index (i)
  call field%copy_from (field_src)
end subroutine model_copy_field
```

28.4.4 Model Access via Variables

Write the model variable list.

⟨*Models: model: TBP*⟩+≡

```
procedure :: write_var_list => model_write_var_list
```

⟨*Models: procedures*⟩+≡

```
subroutine model_write_var_list (model, unit, follow_link)
  class(model_t), intent(in) :: model
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: follow_link
  call var_list_write (model%var_list, unit, follow_link)
end subroutine model_write_var_list
```

Link a variable list to the model variables.

⟨*Models: model: TBP*⟩+≡

```
procedure :: link_var_list => model_link_var_list
```

⟨*Models: procedures*⟩+≡

```
subroutine model_link_var_list (model, var_list)
  class(model_t), intent(inout) :: model
  type(var_list_t), intent(in), target :: var_list
  call model%var_list%link (var_list)
end subroutine model_link_var_list
```

Check if the model contains a named variable.

⟨*Models: model: TBP*⟩+≡

```
procedure :: var_exists => model_var_exists
```

⟨*Models: procedures*⟩+≡

```
function model_var_exists (model, name) result (flag)
  class(model_t), intent(in) :: model
  type(string_t), intent(in) :: name
  logical :: flag
  flag = model%var_list%contains (name, follow_link=.false.)
end function model_var_exists
```

Check if the model variable is a derived parameter, i.e., locked.

⟨*Models: model: TBP*⟩+≡

```
procedure :: var_is_locked => model_var_is_locked
```

⟨*Models: procedures*⟩+≡

```
function model_var_is_locked (model, name) result (flag)
  class(model_t), intent(in) :: model
  type(string_t), intent(in) :: name
  logical :: flag
  flag = model%var_list%is_locked (name, follow_link=.false.)
end function model_var_is_locked
```

Set a model parameter via the named variable. We assume that the variable exists and is writable, i.e., non-locked. We update the model and variable list, so independent and derived parameters are always synchronized.

⟨*Models: model: TBP*⟩+≡

```
procedure :: set_real => model_var_set_real
```

⟨*Models: procedures*⟩+≡

```
subroutine model_var_set_real (model, name, rval, verbose, pacified)
  class(model_t), intent(inout) :: model
  type(string_t), intent(in) :: name
  real(default), intent(in) :: rval
  logical, intent(in), optional :: verbose, pacified
  call model%var_list%set_real (name, rval, &
       is_known=.true., ignore=.false., &
       verbose=verbose, model_name=model%get_name (), pacified=pacified)
  call model%update_parameters ()
end subroutine model_var_set_real
```

Retrieve a model parameter value.

⟨*Models: model: TBP*⟩+≡

```
procedure :: get_rval => model_var_get_rval
```

⟨*Models: procedures*⟩+≡

```
function model_var_get_rval (model, name) result (rval)
  class(model_t), intent(in) :: model
  type(string_t), intent(in) :: name
  real(default) :: rval
  rval = model%var_list%get_rval (name, follow_link=.false.)
end function model_var_get_rval
```

[To be deleted] Return a pointer to the variable list.

⟨*Models: model: TBP*⟩+≡

```
procedure :: get_var_list_ptr => model_get_var_list_ptr
```

⟨*Models: procedures*⟩+≡

```
function model_get_var_list_ptr (model) result (var_list)
  type(var_list_t), pointer :: var_list
  class(model_t), intent(in), target :: model
  var_list => model%var_list
end function model_get_var_list_ptr
```

28.4.5 Scheme handling

A model can specify a set of allowed schemes that steer the setup of model variables. The model file can contain scheme-specific declarations that are selected by a `select scheme` clause. Scheme support is optional.

If enabled, the model object contains a list of allowed schemes, and the model reader takes the active scheme as an argument. After the model has been read, the scheme is fixed and can no longer be modified.

The model supports schemes if the scheme array is allocated.

⟨*Models: model: TBP*⟩+≡

```
procedure :: has_schemes => model_has_schemes
```

⟨*Models: procedures*⟩+≡

```
function model_has_schemes (model) result (flag)
  logical :: flag
  class(model_t), intent(in) :: model
  flag = allocated (model%schemes)
end function model_has_schemes
```

Enable schemes: fix the list of allowed schemes.

⟨*Models: model: TBP*⟩+≡

```
procedure :: enable_schemes => model_enable_schemes
```

⟨*Models: procedures*⟩+≡

```
subroutine model_enable_schemes (model, scheme)
  class(model_t), intent(inout) :: model
  type(string_t), dimension(:), intent(in) :: scheme
  allocate (model%schemes (size (scheme)), source = scheme)
end subroutine model_enable_schemes
```

Find the scheme. Check if the scheme is allowed. The numeric index of the selected scheme is stored in the `model_data_t` base object.

If no argument is given, select the first scheme. The numeric scheme ID will then be 1, while a model without schemes retains 0.

⟨*Models: model: TBP*⟩+≡

```
procedure :: set_scheme => model_set_scheme
```

⟨*Models: procedures*⟩+≡

```
subroutine model_set_scheme (model, scheme)
  class(model_t), intent(inout) :: model
  type(string_t), intent(in), optional :: scheme
  logical :: ok
  integer :: i
  if (model%has_schemes ()) then
     if (present (scheme)) then
        ok = .false.
        CHECK_SCHEME: do i = 1, size (model%schemes)
           if (scheme == model%schemes(i)) then
              allocate (model%selected_scheme, source = scheme)
              call model%set_scheme_num (i)
              ok = .true.
              exit CHECK_SCHEME
           end if
        end do CHECK_SCHEME
```

```
         if (.not. ok) then
            call msg_fatal &
                 ("Model '" // char (model%get_name ()) &
                 // "': scheme '" // char (scheme) // "' not supported")
         end if
      else
         allocate (model%selected_scheme, source = model%schemes(1))
         call model%set_scheme_num (1)
      end if
   else
      if (present (scheme)) then
         call msg_error &
                 ("Model '" // char (model%get_name ()) &
                 // "' does not support schemes")
      end if
   end if
end subroutine model_set_scheme
```

Get the scheme. Note that the base `model_data_t` provides a `get_scheme_num` getter function.

⟨*Models: model: TBP*⟩+≡

```
procedure :: get_scheme => model_get_scheme
```

⟨*Models: procedures*⟩+≡

```
function model_get_scheme (model) result (scheme)
  class(model_t), intent(in) :: model
  type(string_t) :: scheme
  if (allocated (model%selected_scheme)) then
     scheme = model%selected_scheme
  else
     scheme = ""
  end if
end function model_get_scheme
```

Check if a model has been set up with a specific name and (if applicable) scheme. This helps in determining whether we need a new model object.

⟨*Models: model: TBP*⟩+≡

```
procedure :: matches => model_matches
```

⟨*Models: procedures*⟩+≡

```
function model_matches (model, name, scheme) result (flag)
  logical :: flag
  class(model_t), intent(in) :: model
  type(string_t), intent(in) :: name
  type(string_t), intent(in), optional :: scheme
  flag = model%get_name () == name
  if (flag) then
     if (model%has_schemes ()) then
        if (present (scheme)) then
           flag = model%get_scheme () == scheme
        else
           flag = model%get_scheme_num () == 1
        end if
     else
```

```
        flag = .not. present (scheme)
      end if
    end if
  end function model_matches
```

28.4.6 Reading models from file

This procedure defines the model-file syntax for the parser, returning an internal file (ifile).

Note that arithmetic operators are defined as keywords in the expression syntax, so we exclude them here.

⟨*Models: procedures*⟩+≡

```
  subroutine define_model_file_syntax (ifile)
    type(ifile_t), intent(inout) :: ifile
    call ifile_append (ifile, "SEQ model_def = model_name_def " // &
         "scheme_header parameters external_pars particles vertices")
    call ifile_append (ifile, "SEQ model_name_def = model model_name")
    call ifile_append (ifile, "KEY model")
    call ifile_append (ifile, "QUO model_name = '""'...'""'")
    call ifile_append (ifile, "SEQ scheme_header = scheme_decl?")
    call ifile_append (ifile, "SEQ scheme_decl = schemes '=' scheme_list")
    call ifile_append (ifile, "KEY schemes")
    call ifile_append (ifile, "LIS scheme_list = scheme_name+")
    call ifile_append (ifile, "QUO scheme_name = '""'...'""'")
    call ifile_append (ifile, "SEQ parameters = generic_par_def*")
    call ifile_append (ifile, "ALT generic_par_def = &
         &parameter_def | derived_def | unused_def | scheme_block")
    call ifile_append (ifile, "SEQ parameter_def = parameter par_name " // &
         "'=' any_real_value")
    call ifile_append (ifile, "ALT any_real_value = " &
         // "neg_real_value | pos_real_value | real_value")
    call ifile_append (ifile, "SEQ neg_real_value = '-' real_value")
    call ifile_append (ifile, "SEQ pos_real_value = '+' real_value")
    call ifile_append (ifile, "KEY parameter")
    call ifile_append (ifile, "IDE par_name")
    ! call ifile_append (ifile, "KEY '='")          !!! Key already exists
    call ifile_append (ifile, "SEQ derived_def = derived par_name " // &
         "'=' expr")
    call ifile_append (ifile, "KEY derived")
    call ifile_append (ifile, "SEQ unused_def = unused par_name")
    call ifile_append (ifile, "KEY unused")
    call ifile_append (ifile, "SEQ external_pars = external_def*")
    call ifile_append (ifile, "SEQ external_def = external par_name")
    call ifile_append (ifile, "KEY external")
    call ifile_append (ifile, "SEQ scheme_block = &
         &scheme_block_beg scheme_block_body scheme_block_end")
    call ifile_append (ifile, "SEQ scheme_block_beg = select scheme")
    call ifile_append (ifile, "SEQ scheme_block_body = scheme_block_case*")
    call ifile_append (ifile, "SEQ scheme_block_case = &
         &scheme scheme_id parameters")
    call ifile_append (ifile, "ALT scheme_id = scheme_list | other")
    call ifile_append (ifile, "SEQ scheme_block_end = end select")
```

```
  call ifile_append (ifile, "KEY select")
  call ifile_append (ifile, "KEY scheme")
  call ifile_append (ifile, "KEY other")
  call ifile_append (ifile, "KEY end")
  call ifile_append (ifile, "SEQ particles = particle_def*")
  call ifile_append (ifile, "SEQ particle_def = particle prt_longname " // &
       "prt_pdg prt_details")
  call ifile_append (ifile, "KEY particle")
  call ifile_append (ifile, "IDE prt_longname")
  call ifile_append (ifile, "INT prt_pdg")
  call ifile_append (ifile, "ALT prt_details = prt_src | prt_properties")
  call ifile_append (ifile, "SEQ prt_src = like prt_longname prt_properties")
  call ifile_append (ifile, "KEY like")
  call ifile_append (ifile, "SEQ prt_properties = prt_property*")
  call ifile_append (ifile, "ALT prt_property = " // &
       "parton | invisible | gauge | left | right | " // &
       "prt_name | prt_anti | prt_tex_name | prt_tex_anti | " // &
       "prt_spin | prt_isospin | prt_charge | " // &
       "prt_color | prt_mass | prt_width")
  call ifile_append (ifile, "KEY parton")
  call ifile_append (ifile, "KEY invisible")
  call ifile_append (ifile, "KEY gauge")
  call ifile_append (ifile, "KEY left")
  call ifile_append (ifile, "KEY right")
  call ifile_append (ifile, "SEQ prt_name = name name_def+")
  call ifile_append (ifile, "SEQ prt_anti = anti name_def+")
  call ifile_append (ifile, "SEQ prt_tex_name = tex_name name_def")
  call ifile_append (ifile, "SEQ prt_tex_anti = tex_anti name_def")
  call ifile_append (ifile, "KEY name")
  call ifile_append (ifile, "KEY anti")
  call ifile_append (ifile, "KEY tex_name")
  call ifile_append (ifile, "KEY tex_anti")
  call ifile_append (ifile, "ALT name_def = name_string | name_id")
  call ifile_append (ifile, "QUO name_string = '""'...'""'")
  call ifile_append (ifile, "IDE name_id")
  call ifile_append (ifile, "SEQ prt_spin = spin frac")
  call ifile_append (ifile, "KEY spin")
  call ifile_append (ifile, "SEQ prt_isospin = isospin frac")
  call ifile_append (ifile, "KEY isospin")
  call ifile_append (ifile, "SEQ prt_charge = charge frac")
  call ifile_append (ifile, "KEY charge")
  call ifile_append (ifile, "SEQ prt_color = color integer_literal")
  call ifile_append (ifile, "KEY color")
  call ifile_append (ifile, "SEQ prt_mass = mass par_name")
  call ifile_append (ifile, "KEY mass")
  call ifile_append (ifile, "SEQ prt_width = width par_name")
  call ifile_append (ifile, "KEY width")
  call ifile_append (ifile, "SEQ vertices = vertex_def*")
  call ifile_append (ifile, "SEQ vertex_def = vertex name_def+")
  call ifile_append (ifile, "KEY vertex")
  call define_expr_syntax (ifile, particles=.false., analysis=.false.)
end subroutine define_model_file_syntax
```

The model-file syntax and lexer are fixed, therefore stored as module variables:

⟨*Models: variables*⟩≡
```
type(syntax_t), target, save :: syntax_model_file
```

⟨*Models: public*⟩+≡
```
public :: syntax_model_file_init
```

⟨*Models: procedures*⟩+≡
```
subroutine syntax_model_file_init ()
  type(ifile_t) :: ifile
  call define_model_file_syntax (ifile)
  call syntax_init (syntax_model_file, ifile)
  call ifile_final (ifile)
end subroutine syntax_model_file_init
```

⟨*Models: procedures*⟩+≡
```
subroutine lexer_init_model_file (lexer)
  type(lexer_t), intent(out) :: lexer
  call lexer_init (lexer, &
       comment_chars = "#!", &
       quote_chars = '"{', &
       quote_match = '"}', &
       single_chars = ":(),", &
       special_class = [ "+-*/^", "<>=  " ] , &
       keyword_list = syntax_get_keyword_list_ptr (syntax_model_file))
end subroutine lexer_init_model_file
```

⟨*Models: public*⟩+≡
```
public :: syntax_model_file_final
```

⟨*Models: procedures*⟩+≡
```
subroutine syntax_model_file_final ()
  call syntax_final (syntax_model_file)
end subroutine syntax_model_file_final
```

⟨*Models: public*⟩+≡
```
public :: syntax_model_file_write
```

⟨*Models: procedures*⟩+≡
```
subroutine syntax_model_file_write (unit)
  integer, intent(in), optional :: unit
  call syntax_write (syntax_model_file, unit)
end subroutine syntax_model_file_write
```

⟨*Models: model: TBP*⟩+≡
```
procedure :: read => model_read
```

⟨*Models: procedures*⟩+≡
```
subroutine model_read (model, filename, os_data, exist, scheme)
  class(model_t), intent(out), target :: model
  type(string_t), intent(in) :: filename
  type(os_data_t), intent(in) :: os_data
  logical, intent(out), optional :: exist
  type(string_t), intent(in), optional :: scheme
  type(string_t) :: file
```

```
type(stream_t), target :: stream
type(lexer_t) :: lexer
integer :: unit
character(32) :: model_md5sum
type(parse_node_t), pointer :: nd_model_def, nd_model_name_def
type(parse_node_t), pointer :: nd_schemes, nd_scheme_decl
type(parse_node_t), pointer :: nd_parameters
type(parse_node_t), pointer :: nd_external_pars
type(parse_node_t), pointer :: nd_particles, nd_vertices
type(string_t) :: model_name, lib_name
integer :: n_parblock, n_par, i_par, n_ext, n_prt, n_vtx
type(parse_node_t), pointer :: nd_par_def
type(parse_node_t), pointer :: nd_ext_def
type(parse_node_t), pointer :: nd_prt
type(parse_node_t), pointer :: nd_vtx
logical :: model_exist
file = filename
inquire (file=char(file), exist=model_exist)
if ((.not. model_exist) .and. (.not. os_data%use_testfiles)) then
   file = os_data%whizard_modelpath_local // "/" // filename
   inquire (file = char (file), exist = model_exist)
end if
if (.not. model_exist) then
   file = os_data%whizard_modelpath // "/" // filename
   inquire (file = char (file), exist = model_exist)
end if
if (.not. model_exist) then
   call msg_fatal ("Model file '" // char (filename) // "' not found")
   if (present (exist))  exist = .false.
   return
end if
if (present (exist))  exist = .true.

if (logging) call msg_message ("Reading model file '" // char (file) // "'")

unit = free_unit ()
open (file=char(file), unit=unit, action="read", status="old")
model_md5sum = md5sum (unit)
close (unit)

call lexer_init_model_file (lexer)
call stream_init (stream, char (file))
call lexer_assign_stream (lexer, stream)
call parse_tree_init (model%parse_tree, syntax_model_file, lexer)
call stream_final (stream)
call lexer_final (lexer)

nd_model_def => model%parse_tree%get_root_ptr ()
nd_model_name_def => parse_node_get_sub_ptr (nd_model_def)
model_name = parse_node_get_string &
     (parse_node_get_sub_ptr (nd_model_name_def, 2))
nd_schemes => nd_model_name_def%get_next_ptr ()

call find_block &
```

```
      ("scheme_header", nd_schemes, nd_scheme_decl, nd_next=nd_parameters)
  call find_block &
      ("parameters", nd_parameters, nd_par_def, n_parblock, nd_external_pars)
  call find_block &
      ("external_pars", nd_external_pars, nd_ext_def, n_ext, nd_particles)
  call find_block &
      ("particles", nd_particles, nd_prt, n_prt, nd_vertices)
  call find_block &
      ("vertices", nd_vertices, nd_vtx, n_vtx)

  if (associated (nd_external_pars)) then
     lib_name = "external." // model_name
  else
     lib_name = ""
  end if

  if (associated (nd_scheme_decl)) then
     call handle_schemes (nd_scheme_decl, scheme)
  end if

  n_par = 0
  call count_parameters (nd_par_def, n_parblock, n_par)

  call model%init (model_name, lib_name, os_data, n_par + n_ext, n_prt, n_vtx)
  model%md5sum = model_md5sum

  if (associated (nd_par_def)) then
     i_par = 0
     call handle_parameters (nd_par_def, n_parblock, i_par)
  end if
  if (associated (nd_ext_def)) then
     call handle_external (nd_ext_def, n_par, n_ext)
  end if
  call model%update_parameters ()
  if (associated (nd_prt)) then
     call handle_fields (nd_prt, n_prt)
  end if
  if (associated (nd_vtx)) then
     call handle_vertices (nd_vtx, n_vtx)
  end if

  call model%freeze_vertices ()
  call model%append_field_vars ()

contains

  subroutine find_block (key, nd, nd_item, n_item, nd_next)
    character(*), intent(in) :: key
    type(parse_node_t), pointer, intent(inout) :: nd
    type(parse_node_t), pointer, intent(out) :: nd_item
    integer, intent(out), optional :: n_item
    type(parse_node_t), pointer, intent(out), optional :: nd_next
    if (associated (nd)) then
       if (nd%get_rule_key () == key) then
```

```
            nd_item => nd%get_sub_ptr ()
            if (present (n_item))  n_item = nd%get_n_sub ()
            if (present (nd_next))  nd_next => nd%get_next_ptr ()
         else
            nd_item => null ()
            if (present (n_item))  n_item = 0
            if (present (nd_next))  nd_next => nd
            nd => null ()
         end if
      else
         nd_item => null ()
         if (present (n_item))  n_item = 0
         if (present (nd_next))  nd_next => null ()
      end if
   end subroutine find_block

   subroutine handle_schemes (nd_scheme_decl, scheme)
      type(parse_node_t), pointer, intent(in) :: nd_scheme_decl
      type(string_t), intent(in), optional :: scheme
      type(parse_node_t), pointer :: nd_list, nd_entry
      type(string_t), dimension(:), allocatable :: schemes
      integer :: i, n_schemes
      nd_list => nd_scheme_decl%get_sub_ptr (3)
      nd_entry => nd_list%get_sub_ptr ()
      n_schemes = nd_list%get_n_sub ()
      allocate (schemes (n_schemes))
      do i = 1, n_schemes
         schemes(i) = nd_entry%get_string ()
         nd_entry => nd_entry%get_next_ptr ()
      end do
      if (present (scheme)) then
         do i = 1, n_schemes
            if (schemes(i) == scheme)  goto 10   ! block exit
         end do
         call msg_fatal ("Scheme '" // char (scheme) &
              // "' is not supported by model '" // char (model_name) // "'")
      end if
10    continue
      call model%enable_schemes (schemes)
      call model%set_scheme (scheme)
   end subroutine handle_schemes

   subroutine select_scheme (nd_scheme_block, n_parblock_sub, nd_par_def)
      type(parse_node_t), pointer, intent(in) :: nd_scheme_block
      integer, intent(out) :: n_parblock_sub
      type(parse_node_t), pointer, intent(out) :: nd_par_def
      type(parse_node_t), pointer :: nd_scheme_body
      type(parse_node_t), pointer :: nd_scheme_case, nd_scheme_id, nd_scheme
      type(string_t) :: scheme
      integer :: n_cases, i
      scheme = model%get_scheme ()
      nd_scheme_body => nd_scheme_block%get_sub_ptr (2)
      nd_parameters => null ()
      select case (char (nd_scheme_body%get_rule_key ()))
```

```
  case ("scheme_block_body")
     n_cases = nd_scheme_body%get_n_sub ()
     FIND_SCHEME: do i = 1, n_cases
        nd_scheme_case => nd_scheme_body%get_sub_ptr (i)
        nd_scheme_id => nd_scheme_case%get_sub_ptr (2)
        select case (char (nd_scheme_id%get_rule_key ()))
        case ("scheme_list")
           nd_scheme => nd_scheme_id%get_sub_ptr ()
           do while (associated (nd_scheme))
              if (scheme == nd_scheme%get_string ()) then
                 nd_parameters => nd_scheme_id%get_next_ptr ()
                 exit FIND_SCHEME
              end if
              nd_scheme => nd_scheme%get_next_ptr ()
           end do
        case ("other")
           nd_parameters => nd_scheme_id%get_next_ptr ()
           exit FIND_SCHEME
        case default
           print *, "'", char (nd_scheme_id%get_rule_key ()), "'"
           call msg_bug ("Model read: impossible scheme rule")
        end select
     end do FIND_SCHEME
  end select
  if (associated (nd_parameters)) then
     select case (char (nd_parameters%get_rule_key ()))
     case ("parameters")
        n_parblock_sub = nd_parameters%get_n_sub ()
        if (n_parblock_sub > 0) then
           nd_par_def => nd_parameters%get_sub_ptr ()
        else
           nd_par_def => null ()
        end if
     case default
        n_parblock_sub = 0
        nd_par_def => null ()
     end select
  else
     n_parblock_sub = 0
     nd_par_def => null ()
  end if
end subroutine select_scheme

recursive subroutine count_parameters (nd_par_def_in, n_parblock, n_par)
  type(parse_node_t), pointer, intent(in) :: nd_par_def_in
  integer, intent(in) :: n_parblock
  integer, intent(inout) :: n_par
  type(parse_node_t), pointer :: nd_par_def, nd_par_key
  type(parse_node_t), pointer :: nd_par_def_sub
  integer :: n_parblock_sub
  integer :: i
  nd_par_def => nd_par_def_in
  do i = 1, n_parblock
     nd_par_key => nd_par_def%get_sub_ptr ()
```

```
      select case (char (nd_par_key%get_rule_key ()))
      case ("parameter", "derived", "unused")
         n_par = n_par + 1
      case ("scheme_block_beg")
         call select_scheme (nd_par_def, n_parblock_sub, nd_par_def_sub)
         if (n_parblock_sub > 0) then
            call count_parameters (nd_par_def_sub, n_parblock_sub, n_par)
         end if
      case default
         print *, "'", char (nd_par_key%get_rule_key ()), "'"
         call msg_bug ("Model read: impossible parameter rule")
      end select
      nd_par_def => parse_node_get_next_ptr (nd_par_def)
   end do
end subroutine count_parameters

recursive subroutine handle_parameters (nd_par_def_in, n_parblock, i_par)
   type(parse_node_t), pointer, intent(in) :: nd_par_def_in
   integer, intent(in) :: n_parblock
   integer, intent(inout) :: i_par
   type(parse_node_t), pointer :: nd_par_def, nd_par_key
   type(parse_node_t), pointer :: nd_par_def_sub
   integer :: n_parblock_sub
   integer :: i
   nd_par_def => nd_par_def_in
   do i = 1, n_parblock
      nd_par_key => nd_par_def%get_sub_ptr ()
      select case (char (nd_par_key%get_rule_key ()))
      case ("parameter")
         i_par = i_par + 1
         call model%read_parameter (i_par, nd_par_def)
      case ("derived")
         i_par = i_par + 1
         call model%read_derived (i_par, nd_par_def)
      case ("unused")
         i_par = i_par + 1
         call model%read_unused (i_par, nd_par_def)
      case ("scheme_block_beg")
         call select_scheme (nd_par_def, n_parblock_sub, nd_par_def_sub)
         if (n_parblock_sub > 0) then
            call handle_parameters (nd_par_def_sub, n_parblock_sub, i_par)
         end if
      end select
      nd_par_def => parse_node_get_next_ptr (nd_par_def)
   end do
end subroutine handle_parameters

subroutine handle_external (nd_ext_def, n_par, n_ext)
   type(parse_node_t), pointer, intent(inout) :: nd_ext_def
   integer, intent(in) :: n_par, n_ext
   integer :: i
   do i = n_par + 1, n_par + n_ext
      call model%read_external (i, nd_ext_def)
      nd_ext_def => parse_node_get_next_ptr (nd_ext_def)
```

```
      end do
!     real(c_default_float), dimension(:), allocatable :: par
!       if (associated (model%init_external_parameters)) then
!          allocate (par (model%get_n_real ()))
!          call model%real_parameters_to_c_array (par)
!          call model%init_external_parameters (par)
!          call model%real_parameters_from_c_array (par)
!       end if
    end subroutine handle_external

    subroutine handle_fields (nd_prt, n_prt)
      type(parse_node_t), pointer, intent(inout) :: nd_prt
      integer, intent(in) :: n_prt
      integer :: i
      do i = 1, n_prt
         call model%read_field (i, nd_prt)
         nd_prt => parse_node_get_next_ptr (nd_prt)
      end do
    end subroutine handle_fields

    subroutine handle_vertices (nd_vtx, n_vtx)
      type(parse_node_t), pointer, intent(inout) :: nd_vtx
      integer, intent(in) :: n_vtx
      integer :: i
      do i = 1, n_vtx
         call model%read_vertex (i, nd_vtx)
         nd_vtx => parse_node_get_next_ptr (nd_vtx)
      end do
    end subroutine handle_vertices

  end subroutine model_read
```

Parameters are real values (literal) with an optional unit.

⟨*Models: model: TBP*⟩+≡

```
  procedure, private :: read_parameter => model_read_parameter
```

⟨*Models: procedures*⟩+≡

```
  subroutine model_read_parameter (model, i, node)
    class(model_t), intent(inout), target :: model
    integer, intent(in) :: i
    type(parse_node_t), intent(in), target :: node
    type(parse_node_t), pointer :: node_name, node_val
    type(string_t) :: name
    node_name => parse_node_get_sub_ptr (node, 2)
    name = parse_node_get_string (node_name)
    node_val => parse_node_get_next_ptr (node_name, 2)
    call model%set_parameter_parse_node (i, name, node_val, constant=.true.)
  end subroutine model_read_parameter
```

Derived parameters have any numeric expression as their definition. Don't evaluate the expression, yet.

⟨*Models: model: TBP*⟩+≡

```
  procedure, private :: read_derived => model_read_derived
```

⟨*Models: procedures*⟩+≡

```
subroutine model_read_derived (model, i, node)
  class(model_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(parse_node_t), intent(in), target :: node
  type(string_t) :: name
  type(parse_node_t), pointer :: pn_expr
  name = parse_node_get_string (parse_node_get_sub_ptr (node, 2))
  pn_expr => parse_node_get_sub_ptr (node, 4)
  call model%set_parameter_parse_node (i, name, pn_expr, constant=.false.)
end subroutine model_read_derived
```

External parameters have no definition; they are handled by an external library.

⟨*Models: model: TBP*⟩+≡

```
procedure, private :: read_external => model_read_external
```

⟨*Models: procedures*⟩+≡

```
subroutine model_read_external (model, i, node)
  class(model_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(parse_node_t), intent(in), target :: node
  type(string_t) :: name
  name = parse_node_get_string (parse_node_get_sub_ptr (node, 2))
  call model%set_parameter_external (i, name)
end subroutine model_read_external
```

Ditto for unused parameters, they are there just for reserving the name.

⟨*Models: model: TBP*⟩+≡

```
procedure, private :: read_unused => model_read_unused
```

⟨*Models: procedures*⟩+≡

```
subroutine model_read_unused (model, i, node)
  class(model_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(parse_node_t), intent(in), target :: node
  type(string_t) :: name
  name = parse_node_get_string (parse_node_get_sub_ptr (node, 2))
  call model%set_parameter_unused (i, name)
end subroutine model_read_unused
```

⟨*Models: model: TBP*⟩+≡

```
procedure, private :: read_field => model_read_field
```

⟨*Models: procedures*⟩+≡

```
subroutine model_read_field (model, i, node)
  class(model_t), intent(inout), target :: model
  integer, intent(in) :: i
  type(parse_node_t), intent(in) :: node
  type(parse_node_t), pointer :: nd_src, nd_props, nd_prop
  type(string_t) :: longname
  integer :: pdg
  type(string_t) :: name_src
  type(string_t), dimension(:), allocatable :: name
  type(field_data_t), pointer :: field, field_src
```

```
longname = parse_node_get_string (parse_node_get_sub_ptr (node, 2))
pdg = parse_node_get_integer (parse_node_get_sub_ptr (node, 3))
field => model%get_field_ptr_by_index (i)
call field%init (longname, pdg)
nd_src => parse_node_get_sub_ptr (node, 4)
if (associated (nd_src)) then
   if (parse_node_get_rule_key (nd_src) == "prt_src") then
      name_src = parse_node_get_string (parse_node_get_sub_ptr (nd_src, 2))
      field_src => model%get_field_ptr (name_src, check=.true.)
      call field%copy_from (field_src)
      nd_props => parse_node_get_sub_ptr (nd_src, 3)
   else
      nd_props => nd_src
   end if
   nd_prop => parse_node_get_sub_ptr (nd_props)
   do while (associated (nd_prop))
      select case (char (parse_node_get_rule_key (nd_prop)))
      case ("invisible")
         call field%set (is_visible=.false.)
      case ("parton")
         call field%set (is_parton=.true.)
      case ("gauge")
         call field%set (is_gauge=.true.)
      case ("left")
         call field%set (is_left_handed=.true.)
      case ("right")
         call field%set (is_right_handed=.true.)
      case ("prt_name")
         call read_names (nd_prop, name)
         call field%set (name=name)
      case ("prt_anti")
         call read_names (nd_prop, name)
         call field%set (anti=name)
      case ("prt_tex_name")
         call field%set ( &
              tex_name = parse_node_get_string &
              (parse_node_get_sub_ptr (nd_prop, 2)))
      case ("prt_tex_anti")
         call field%set ( &
              tex_anti = parse_node_get_string &
              (parse_node_get_sub_ptr (nd_prop, 2)))
      case ("prt_spin")
         call field%set ( &
              spin_type = read_frac &
              (parse_node_get_sub_ptr (nd_prop, 2), 2))
      case ("prt_isospin")
         call field%set ( &
              isospin_type = read_frac &
              (parse_node_get_sub_ptr (nd_prop, 2), 2))
      case ("prt_charge")
         call field%set ( &
              charge_type = read_frac &
              (parse_node_get_sub_ptr (nd_prop, 2), 3))
      case ("prt_color")
```

```
            call field%set ( &
                 color_type = parse_node_get_integer &
                 (parse_node_get_sub_ptr (nd_prop, 2)))
         case ("prt_mass")
            call field%set ( &
                 mass_data = model%get_par_data_ptr &
                 (parse_node_get_string &
                 (parse_node_get_sub_ptr (nd_prop, 2))))
         case ("prt_width")
            call field%set ( &
                 width_data = model%get_par_data_ptr &
                 (parse_node_get_string &
                 (parse_node_get_sub_ptr (nd_prop, 2))))
         case default
            call msg_bug (" Unknown particle property '" &
                 // char (parse_node_get_rule_key (nd_prop)) // "'")
         end select
         if (allocated (name))  deallocate (name)
         nd_prop => parse_node_get_next_ptr (nd_prop)
      end do
   end if
   call field%freeze ()
end subroutine model_read_field
```

⟨*Models: model: TBP*⟩+≡

```
procedure, private :: read_vertex => model_read_vertex
```

⟨*Models: procedures*⟩+≡

```
subroutine model_read_vertex (model, i, node)
  class(model_t), intent(inout) :: model
  integer, intent(in) :: i
  type(parse_node_t), intent(in) :: node
  type(string_t), dimension(:), allocatable :: name
  call read_names (node, name)
  call model%set_vertex (i, name)
end subroutine model_read_vertex
```

⟨*Models: procedures*⟩+≡

```
subroutine read_names (node, name)
  type(parse_node_t), intent(in) :: node
  type(string_t), dimension(:), allocatable, intent(inout) :: name
  type(parse_node_t), pointer :: nd_name
  integer :: n_names, i
  n_names = parse_node_get_n_sub (node) - 1
  allocate (name (n_names))
  nd_name => parse_node_get_sub_ptr (node, 2)
  do i = 1, n_names
     name(i) = parse_node_get_string (nd_name)
     nd_name => parse_node_get_next_ptr (nd_name)
  end do
end subroutine read_names
```

⟨*Models: procedures*⟩+≡

```
function read_frac (nd_frac, base) result (qn_type)
```

```
    integer :: qn_type
    type(parse_node_t), intent(in) :: nd_frac
    integer, intent(in) :: base
    type(parse_node_t), pointer :: nd_num, nd_den
    integer :: num, den
    nd_num => parse_node_get_sub_ptr (nd_frac)
    nd_den => parse_node_get_next_ptr (nd_num)
    select case (char (parse_node_get_rule_key (nd_num)))
    case ("integer_literal")
       num = parse_node_get_integer (nd_num)
    case ("neg_int")
       num = - parse_node_get_integer (parse_node_get_sub_ptr (nd_num, 2))
    case ("pos_int")
       num = parse_node_get_integer (parse_node_get_sub_ptr (nd_num, 2))
    case default
       call parse_tree_bug (nd_num, "int|neg_int|pos_int")
    end select
    if (associated (nd_den)) then
       den = parse_node_get_integer (parse_node_get_sub_ptr (nd_den, 2))
    else
       den = 1
    end if
    if (den == 1) then
       qn_type = sign (1 + abs (num) * base, num)
    else if (den == base) then
       qn_type = sign (abs (num) + 1, num)
    else
       call parse_node_write_rec (nd_frac)
       call msg_fatal (" Fractional quantum number: wrong denominator")
    end if
  end function read_frac
```

Append field (PDG-array) variables to the variable list, based on the field content.

⟨*Models: model: TBP*⟩+≡

```
  procedure, private :: append_field_vars => model_append_field_vars
```

⟨*Models: procedures*⟩+≡

```
  subroutine model_append_field_vars (model)
    class(model_t), intent(inout) :: model
    type(pdg_array_t) :: aval
    type(field_data_t), dimension(:), pointer :: field_array
    type(field_data_t), pointer :: field
    type(string_t) :: name
    type(string_t), dimension(:), allocatable :: name_array
    integer, dimension(:), allocatable :: pdg
    logical, dimension(:), allocatable :: mask
    integer :: i, j
    field_array => model%get_field_array_ptr ()
    aval = UNDEFINED
    call var_list_append_pdg_array &
         (model%var_list, var_str ("particle"), &
          aval, locked = .true., intrinsic=.true.)
    do i = 1, size (field_array)
```

```
      aval = field_array(i)%get_pdg ()
      name = field_array(i)%get_longname ()
      call var_list_append_pdg_array &
           (model%var_list, name, aval, locked=.true., intrinsic=.true.)
      call field_array(i)%get_name_array (.false., name_array)
      do j = 1, size (name_array)
         call var_list_append_pdg_array &
              (model%var_list, name_array(j), &
              aval, locked=.true., intrinsic=.true.)
      end do
      model%max_field_name_length = &
           max (model%max_field_name_length, len (name_array(1)))
      aval = - field_array(i)%get_pdg ()
      call field_array(i)%get_name_array (.true., name_array)
      do j = 1, size (name_array)
         call var_list_append_pdg_array &
              (model%var_list, name_array(j), &
              aval, locked=.true., intrinsic=.true.)
      end do
      if (size (name_array) > 0) then
         model%max_field_name_length = &
              max (model%max_field_name_length, len (name_array(1)))
      end if
   end do
   call model%get_all_pdg (pdg)
   allocate (mask (size (pdg)))
   do i = 1, size (pdg)
      field => model%get_field_ptr (pdg(i))
      mask(i) = field%get_charge_type () /= 1
   end do
   aval = pack (pdg, mask)
   call var_list_append_pdg_array &
        (model%var_list, var_str ("charged"), &
         aval, locked = .true., intrinsic=.true.)
   do i = 1, size (pdg)
      field => model%get_field_ptr (pdg(i))
      mask(i) = field%get_charge_type () == 1
   end do
   aval = pack (pdg, mask)
   call var_list_append_pdg_array &
        (model%var_list, var_str ("neutral"), &
         aval, locked = .true., intrinsic=.true.)
   do i = 1, size (pdg)
      field => model%get_field_ptr (pdg(i))
      mask(i) = field%get_color_type () /= 1
   end do
   aval = pack (pdg, mask)
   call var_list_append_pdg_array &
        (model%var_list, var_str ("colored"), &
         aval, locked = .true., intrinsic=.true.)
end subroutine model_append_field_vars
```

28.4.7 Test models

⟨*Models: public*⟩+≡
```
public :: create_test_model
```

⟨*Models: procedures*⟩+≡
```
subroutine create_test_model (model_name, test_model)
  type(string_t), intent(in) :: model_name
  type(model_t), intent(out), pointer :: test_model
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  call syntax_model_file_init ()
  call os_data_init (os_data)
  call model_list%read_model &
     (model_name, model_name // var_str (".mdl"), os_data, test_model)
end subroutine create_test_model
```

28.4.8 Model list

List of currently active models

⟨*Models: types*⟩+≡
```
type, extends (model_t) :: model_entry_t
   type(model_entry_t), pointer :: next => null ()
end type model_entry_t
```

⟨*Models: public*⟩+≡
```
public :: model_list_t
```

⟨*Models: types*⟩+≡
```
type :: model_list_t
   type(model_entry_t), pointer :: first => null ()
   type(model_entry_t), pointer :: last => null ()
   type(model_list_t), pointer :: context => null ()
 contains
 ⟨Models: model list: TBP⟩
end type model_list_t
```

Write an account of the model list. We write linked lists first, starting from the global context.

⟨*Models: model list: TBP*⟩≡
```
procedure :: write => model_list_write
```

⟨*Models: procedures*⟩+≡
```
recursive subroutine model_list_write (object, unit, verbose, follow_link)
  class(model_list_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  logical, intent(in), optional :: follow_link
  type(model_entry_t), pointer :: current
  logical :: rec
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  rec = .true.;  if (present (follow_link))  rec = follow_link
```

```
    if (rec .and. associated (object%context)) then
       call object%context%write (unit, verbose, follow_link)
    end if
    current => object%first
    if (associated (current)) then
       do while (associated (current))
          call current%write (unit, verbose)
          current => current%next
          if (associated (current))  write (u, *)
       end do
    end if
  end subroutine model_list_write
```

Link this list to another one.

⟨*Models: model list: TBP*⟩+≡

```
  procedure :: link => model_list_link
```

⟨*Models: procedures*⟩+≡

```
  subroutine model_list_link (model_list, context)
    class(model_list_t), intent(inout) :: model_list
    type(model_list_t), intent(in), target :: context
    model_list%context => context
  end subroutine model_list_link
```

(Private, used below:) Append an existing model, for which we have allocated a pointer entry, to the model list. The original pointer becomes disassociated, and the model should now be considered as part of the list. We assume that this model is not yet part of the list.

If we provide a `model` argument, this returns a pointer to the new entry.

⟨*Models: model list: TBP*⟩+≡

```
  procedure, private :: import => model_list_import
```

⟨*Models: procedures*⟩+≡

```
  subroutine model_list_import (model_list, current, model)
    class(model_list_t), intent(inout) :: model_list
    type(model_entry_t), pointer, intent(inout) :: current
    type(model_t), optional, pointer, intent(out) :: model
    if (associated (current)) then
       if (associated (model_list%first)) then
          model_list%last%next => current
       else
          model_list%first => current
       end if
       model_list%last => current
       if (present (model))  model => current%model_t
       current => null ()
    end if
  end subroutine model_list_import
```

Currently test only:

Add a new model with given `name` to the list, if it does not yet exist. If successful, return a pointer to the new model.

⟨*Models: model list: TBP*⟩+≡

```
procedure :: add => model_list_add
```

⟨*Models: procedures*⟩+≡

```
subroutine model_list_add (model_list, &
     name, os_data, n_par, n_prt, n_vtx, model)
  class(model_list_t), intent(inout) :: model_list
  type(string_t), intent(in) :: name
  type(os_data_t), intent(in) :: os_data
  integer, intent(in) :: n_par, n_prt, n_vtx
  type(model_t), pointer :: model
  type(model_entry_t), pointer :: current
  if (model_list%model_exists (name, follow_link=.false.)) then
     model => null ()
  else
     allocate (current)
     call current%init (name, var_str (""), os_data, &
          n_par, n_prt, n_vtx)
     call model_list%import (current, model)
  end if
end subroutine model_list_add
```

Read a new model from file and add to the list, if it does not yet exist. Finalize the model by allocating the vertex table. Return a pointer to the new model. If unsuccessful, return the original pointer.

The model is always inserted in the last link of a chain of model lists. This way, we avoid loading models twice from different contexts. When a model is modified, we should first allocate a local copy.

⟨*Models: model list: TBP*⟩+≡

```
procedure :: read_model => model_list_read_model
```

⟨*Models: procedures*⟩+≡

```
subroutine model_list_read_model &
     (model_list, name, filename, os_data, model, scheme)
  class(model_list_t), intent(inout), target :: model_list
  type(string_t), intent(in) :: name, filename
  type(os_data_t), intent(in) :: os_data
  type(model_t), pointer, intent(inout) :: model
  type(string_t), intent(in), optional :: scheme
  class(model_list_t), pointer :: global_model_list
  type(model_entry_t), pointer :: current
  logical :: exist
  if (.not. model_list%model_exists (name, scheme, follow_link=.true.)) then
     allocate (current)
     call current%read (filename, os_data, exist, scheme=scheme)
     if (.not. exist)  return
     if (current%get_name () /= name) then
        call msg_fatal ("Model file '" // char (filename) // &
             "' contains model '" // char (current%get_name ()) // &
             "' instead of '" // char (name) // "'")
        call current%final ();  deallocate (current)
        return
     end if
     global_model_list => model_list
     do while (associated (global_model_list%context))
```

```
            global_model_list => global_model_list%context
         end do
         call global_model_list%import (current, model)
      else
         model => model_list%get_model_ptr (name, scheme)
      end if
   end subroutine model_list_read_model
```

Append a copy of an existing model to a model list. Optionally, return pointer to the new entry.

⟨*Models: model list: TBP*⟩+≡

```
   procedure :: append_copy => model_list_append_copy
```

⟨*Models: procedures*⟩+≡

```
   subroutine model_list_append_copy (model_list, orig, model)
      class(model_list_t), intent(inout) :: model_list
      type(model_t), intent(in), target :: orig
      type(model_t), intent(out), pointer, optional :: model
      type(model_entry_t), pointer :: copy
      allocate (copy)
      call copy%init_instance (orig)
      call model_list%import (copy, model)
   end subroutine model_list_append_copy
```

Check if a model exists by examining the list. Check recursively unless told otherwise.

⟨*Models: model list: TBP*⟩+≡

```
   procedure :: model_exists => model_list_model_exists
```

⟨*Models: procedures*⟩+≡

```
   recursive function model_list_model_exists &
         (model_list, name, scheme, follow_link) result (exists)
      class(model_list_t), intent(in) :: model_list
      logical :: exists
      type(string_t), intent(in) :: name
      type(string_t), intent(in), optional :: scheme
      logical, intent(in), optional :: follow_link
      type(model_entry_t), pointer :: current
      logical :: rec
      rec = .true.;  if (present (follow_link))  rec = follow_link
      current => model_list%first
      do while (associated (current))
         if (current%matches (name, scheme)) then
            exists = .true.
            return
         end if
         current => current%next
      end do
      if (rec .and. associated (model_list%context)) then
         exists = model_list%context%model_exists (name, scheme, follow_link)
      else
         exists = .false.
      end if
   end function model_list_model_exists
```

Return a pointer to a named model. Search recursively unless told otherwise.

⟨*Models: model list: TBP*⟩+≡

```
procedure :: get_model_ptr => model_list_get_model_ptr
```

⟨*Models: procedures*⟩+≡

```
recursive function model_list_get_model_ptr &
    (model_list, name, scheme, follow_link) result (model)
  class(model_list_t), intent(in) :: model_list
  type(model_t), pointer :: model
  type(string_t), intent(in) :: name
  type(string_t), intent(in), optional :: scheme
  logical, intent(in), optional :: follow_link
  type(model_entry_t), pointer :: current
  logical :: rec
  rec = .true.;  if (present (follow_link))  rec = follow_link
  current => model_list%first
  do while (associated (current))
     if (current%matches (name, scheme)) then
        model => current%model_t
        return
     end if
     current => current%next
  end do
  if (rec .and. associated (model_list%context)) then
     model => model_list%context%get_model_ptr (name, scheme, follow_link)
  else
     model => null ()
  end if
end function model_list_get_model_ptr
```

Delete the list of models. No recursion.

⟨*Models: model list: TBP*⟩+≡

```
procedure :: final => model_list_final
```

⟨*Models: procedures*⟩+≡

```
subroutine model_list_final (model_list)
  class(model_list_t), intent(inout) :: model_list
  type(model_entry_t), pointer :: current
  model_list%last => null ()
  do while (associated (model_list%first))
     current => model_list%first
     model_list%first => model_list%first%next
     call current%final ()
     deallocate (current)
  end do
end subroutine model_list_final
```

28.4.9 Model instances

A model instance is a copy of a model object. The parameters are true copies. The particle data and the variable list pointers should point to the copy, so

modifying the parameters has only a local effect. Hence, we build them up explicitly. The vertex array is also rebuilt, it contains particle pointers. Finally, the vertex hash table can be copied directly since it contains no pointers.

The `multiplicity` entry depends on the association of the `mass_data` entry and therefore has to be set at the end.

The instance must carry the `target` attribute.

Parameters: the `copy_parameter` method essentially copies the parameter decorations (parse node, expression etc.). The current parameter values are part of the `model_data_t` base type and are copied afterwards via its `copy_from` method.

⟨*Models: model: TBP*⟩+≡

```
procedure :: init_instance => model_copy
```

⟨*Models: procedures*⟩+≡

```
subroutine model_copy (model, orig)
  class(model_t), intent(out), target :: model
  type(model_t), intent(in) :: orig
  integer :: n_par, n_prt, n_vtx
  integer :: i
  n_par = size (orig%par)
  n_prt = orig%get_n_field ()
  n_vtx = orig%get_n_vtx ()
  call model%basic_init (orig%get_name (), n_par, n_prt, n_vtx)
  if (allocated (orig%schemes)) then
     allocate (model%schemes (size (orig%schemes)))
     model%schemes(:) = orig%schemes(:)
     if (allocated (orig%selected_scheme)) then
        allocate (model%selected_scheme)
        model%selected_scheme = orig%selected_scheme
        call model%set_scheme_num (orig%get_scheme_num ())
     end if
  end if
  model%md5sum = orig%md5sum
  do i = 1, n_par
     call model%copy_parameter (i, orig%par(i))
  end do
  model%init_external_parameters => orig%init_external_parameters
  call model%copy_from (orig)
  model%max_par_name_length = orig%max_par_name_length
  call model%append_field_vars ()
end subroutine model_copy
```

28.4.10 Unit tests

Test module, followed by the corresponding implementation module.

⟨`models_ut.f90`⟩≡

⟨*File header*⟩

```
module models_ut
  use unit_tests
  use models_uti
```

```
  ⟨Standard module head⟩

  ⟨Models: public test⟩

  contains

  ⟨Models: test driver⟩

  end module models_ut
```

⟨models_uti.f90⟩≡

```
  ⟨File header⟩

  module models_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use physics_defs, only: SCALAR, SPINOR
    use os_interface
    use model_data
    use variables

    use models

  ⟨Standard module head⟩

  ⟨Models: test declarations⟩

  contains

  ⟨Models: tests⟩

  end module models_uti
```

API: driver for the unit tests below.

⟨*Models: public test*⟩≡

```
    public :: models_test
```

⟨*Models: test driver*⟩≡

```
    subroutine models_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Models: execute tests⟩
    end subroutine models_test
```

Construct a Model

Here, we construct a toy model explicitly without referring to a file.

⟨*Models: execute tests*⟩≡

```
    call test (models_1, "models_1", &
         "construct model", &
         u, results)
```

⟨*Models: test declarations*⟩≡

```
    public :: models_1
```

⟨*Models: tests*⟩≡

```
subroutine models_1 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(string_t) :: model_name
  type(string_t) :: x_longname
  type(string_t), dimension(2) :: parname
  type(string_t), dimension(2) :: x_name
  type(string_t), dimension(1) :: x_anti
  type(string_t) :: x_tex_name, x_tex_anti
  type(string_t) :: y_longname
  type(string_t), dimension(2) :: y_name
  type(string_t) :: y_tex_name
  type(field_data_t), pointer :: field

  write (u, "(A)")  "* Test output: models_1"
  write (u, "(A)")  "*   Purpose: create a model"
  write (u, *)

  model_name = "Test model"
  call model_list%add (model_name, os_data, 2, 2, 3, model)
  parname(1) = "mx"
  parname(2) = "coup"
  call model%set_parameter_constant (1, parname(1), 10._default)
  call model%set_parameter_constant (2, parname(2), 1.3_default)
  x_longname = "X_LEPTON"
  x_name(1) = "X"
  x_name(2) = "x"
  x_anti(1) = "Xbar"
  x_tex_name = "X^+"
  x_tex_anti = "X^-"
  field => model%get_field_ptr_by_index (1)
  call field%init (x_longname, 99)
  call field%set ( &
       .true., .false., .false., .false., .false., &
       name=x_name, anti=x_anti, tex_name=x_tex_name, tex_anti=x_tex_anti, &
       spin_type=SPINOR, isospin_type=-3, charge_type=2, &
       mass_data=model%get_par_data_ptr (parname(1)))
  y_longname = "Y_COLORON"
  y_name(1) = "Y"
  y_name(2) = "yc"
  y_tex_name = "Y^0"
  field => model%get_field_ptr_by_index (2)
  call field%init (y_longname, 97)
  call field%set ( &
        .false., .false., .true., .false., .false., &
        name=y_name, tex_name=y_tex_name, &
        spin_type=SCALAR, isospin_type=2, charge_type=1, color_type=8)
  call model%set_vertex (1, [99, 99, 99])
  call model%set_vertex (2, [99, 99, 99, 99])
  call model%set_vertex (3, [99, 97, 99])
  call model_list%write (u)
```

```
    call model_list%final ()

    write (u, *)
    write (u, "(A)")  "* Test output end: models_1"

  end subroutine models_1
```

Read a Model

Read a predefined model from file.

⟨*Models: execute tests*⟩+≡

```
  call test (models_2, "models_2", &
       "read model", &
       u, results)
```

⟨*Models: test declarations*⟩+≡

```
  public :: models_2
```

⟨*Models: tests*⟩+≡

```
  subroutine models_2 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(model_list_t) :: model_list
    type(var_list_t), pointer :: var_list
    type(model_t), pointer :: model

    write (u, "(A)")  "* Test output: models_2"
    write (u, "(A)")  "*   Purpose: read a model from file"
    write (u, *)

    call syntax_model_file_init ()
    call os_data_init (os_data)

    call model_list%read_model (var_str ("Test"), var_str ("Test.mdl"), &
         os_data, model)
    call model_list%write (u)

    write (u, *)
    write (u, "(A)")  "* Variable list"
    write (u, *)

    var_list => model%get_var_list_ptr ()
    call var_list%write (u)

    write (u, *)
    write (u, "(A)")  "* Cleanup"

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, *)
    write (u, "(A)")  "* Test output end: models_2"
```

```
end subroutine models_2
```

Model Instance

Read a predefined model from file and create an instance.

⟨*Models: execute tests*⟩+≡

```
call test (models_3, "models_3", &
     "model instance", &
     u, results)
```

⟨*Models: test declarations*⟩+≡

```
public :: models_3
```

⟨*Models: tests*⟩+≡

```
subroutine models_3 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(var_list_t), pointer :: var_list
  type(model_t), pointer :: instance

  write (u, "(A)")  "* Test output: models_3"
  write (u, "(A)")  "*   Purpose: create a model instance"
  write (u, *)

  call syntax_model_file_init ()
  call os_data_init (os_data)

  call model_list%read_model (var_str ("Test"), var_str ("Test.mdl"), &
       os_data, model)
  allocate (instance)
  call instance%init_instance (model)

  call model%write (u)

  write (u, *)
  write (u, "(A)")  "* Variable list"
  write (u, *)

  var_list => instance%get_var_list_ptr ()
  call var_list%write (u)

  write (u, *)
  write (u, "(A)")  "* Cleanup"

  call instance%final ()
  deallocate (instance)

  call model_list%final ()
  call syntax_model_file_final ()

  write (u, *)
  write (u, "(A)")  "* Test output end: models_3"
```

```
end subroutine models_3
```

Unstable and Polarized Particles

Read a predefined model from file and define decays and polarization.

⟨*Models: execute tests*⟩+≡

```
call test (models_4, "models_4", &
     "handle decays and polarization", &
     u, results)
```

⟨*Models: test declarations*⟩+≡

```
public :: models_4
```

⟨*Models: tests*⟩+≡

```
subroutine models_4 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model, model_instance
  character(32) :: md5sum

  write (u, "(A)")  "* Test output: models_4"
  write (u, "(A)")  "*   Purpose: set and unset decays and polarization"
  write (u, *)

  call syntax_model_file_init ()
  call os_data_init (os_data)

  write (u, "(A)")  "* Read model from file"

  call model_list%read_model (var_str ("Test"), var_str ("Test.mdl"), &
       os_data, model)

  md5sum = model%get_parameters_md5sum ()
  write (u, *)
  write (u, "(1x,3A)")  "MD5 sum (parameters) = '", md5sum, "'"

  write (u, *)
  write (u, "(A)")  "* Set particle decays and polarization"
  write (u, *)

  call model%set_unstable (25, [var_str ("dec1"), var_str ("dec2")])
  call model%set_polarized (6)
  call model%set_unstable (-6, [var_str ("fdec")])

  call model%write (u)

  md5sum = model%get_parameters_md5sum ()
  write (u, *)
  write (u, "(1x,3A)")  "MD5 sum (parameters) = '", md5sum, "'"

  write (u, *)
  write (u, "(A)")  "* Create a model instance"
```

```
    allocate (model_instance)
    call model_instance%init_instance (model)

    write (u, *)
    write (u, "(A)")  "* Revert particle decays and polarization"
    write (u, *)

    call model%set_stable (25)
    call model%set_unpolarized (6)
    call model%set_stable (-6)

    call model%write (u)

    md5sum = model%get_parameters_md5sum ()
    write (u, *)
    write (u, "(1x,3A)")  "MD5 sum (parameters) = '", md5sum, "'"

    write (u, *)
    write (u, "(A)")  "* Show the model instance"
    write (u, *)

    call model_instance%write (u)

    md5sum = model_instance%get_parameters_md5sum ()
    write (u, *)
    write (u, "(1x,3A)")  "MD5 sum (parameters) = '", md5sum, "'"

    write (u, *)
    write (u, "(A)")  "* Cleanup"

    call model_instance%final ()
    deallocate (model_instance)
    call model_list%final ()
    call syntax_model_file_final ()

    write (u, *)
    write (u, "(A)")  "* Test output end: models_4"

  end subroutine models_4
```

Model Variables

Read a predefined model from file and modify some parameters.

Note that the MD5 sum is not modified by this.

⟨*Models: execute tests*⟩+≡

```
  call test (models_5, "models_5", &
       "handle parameters", &
       u, results)
```

⟨*Models: test declarations*⟩+≡

```
  public :: models_5
```

⟨*Models: tests*⟩+≡

```
subroutine models_5 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_list_t) :: model_list
  type(model_t), pointer :: model, model_instance
  character(32) :: md5sum

  write (u, "(A)")  "* Test output: models_5"
  write (u, "(A)")  "*   Purpose: access and modify model variables"
  write (u, *)

  call syntax_model_file_init ()
  call os_data_init (os_data)

  write (u, "(A)")  "* Read model from file"

  call model_list%read_model (var_str ("Test"), var_str ("Test.mdl"), &
       os_data, model)

  write (u, *)

  call model%write (u, &
       show_md5sum = .true., &
       show_variables = .true., &
       show_parameters = .true., &
       show_particles = .false., &
       show_vertices = .false.)

  write (u, *)
  write (u, "(A)")  "* Check parameter status"
  write (u, *)

  write (u, "(1x,A,L1)") "xy exists = ", model%var_exists (var_str ("xx"))
  write (u, "(1x,A,L1)") "ff exists = ", model%var_exists (var_str ("ff"))
  write (u, "(1x,A,L1)") "mf exists = ", model%var_exists (var_str ("mf"))
  write (u, "(1x,A,L1)") "ff locked = ", model%var_is_locked (var_str ("ff"))
  write (u, "(1x,A,L1)") "mf locked = ", model%var_is_locked (var_str ("mf"))

  write (u, *)
  write (u, "(1x,A,F6.2)") "ff = ", model%get_rval (var_str ("ff"))
  write (u, "(1x,A,F6.2)") "mf = ", model%get_rval (var_str ("mf"))

  write (u, *)
  write (u, "(A)")  "* Modify parameter"
  write (u, *)

  call model%set_real (var_str ("ff"), 1._default)

  call model%write (u, &
       show_md5sum = .true., &
       show_variables = .true., &
       show_parameters = .true., &
       show_particles = .false., &
```

```
        show_vertices = .false.)

    write (u, *)
    write (u, "(A)")  "* Cleanup"

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, *)
    write (u, "(A)")  "* Test output end: models_5"

  end subroutine models_5
```

Read model with disordered parameters

Read a model from file where the ordering of independent and derived parameters is non-canonical.

⟨*Models: execute tests*⟩+≡

```
  call test (models_6, "models_6", &
       "read model parameters", &
       u, results)
```

⟨*Models: test declarations*⟩+≡

```
  public :: models_6
```

⟨*Models: tests*⟩+≡

```
  subroutine models_6 (u)
    integer, intent(in) :: u
    integer :: um
    character(80) :: buffer
    type(os_data_t) :: os_data
    type(model_list_t) :: model_list
    type(var_list_t), pointer :: var_list
    type(model_t), pointer :: model

    write (u, "(A)")  "* Test output: models_6"
    write (u, "(A)")  "*   Purpose: read a model from file &
         &with non-canonical parameter ordering"
    write (u, *)

    open (newunit=um, file="Test6.mdl", status="replace", action="readwrite")
    write (um, "(A)")  'model "Test6"'
    write (um, "(A)")  '   parameter a =  1.000000000000E+00'
    write (um, "(A)")  '   derived   b =  2 * a'
    write (um, "(A)")  '   parameter c =  3.000000000000E+00'
    write (um, "(A)")  '   unused    d'

    rewind (um)
    do
       read (um, "(A)", end=1)  buffer
       write (u, "(A)")  trim (buffer)
    end do
1   continue
```

```
    close (um)

    call syntax_model_file_init ()
    call os_data_init (os_data)

    call model_list%read_model (var_str ("Test6"), var_str ("Test6.mdl"), &
         os_data, model)

    write (u, *)
    write (u, "(A)")  "* Variable list"
    write (u, *)

    var_list => model%get_var_list_ptr ()
    call var_list%write (u)

    write (u, *)
    write (u, "(A)")  "* Cleanup"

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, *)
    write (u, "(A)")  "* Test output end: models_6"

  end subroutine models_6
```

Read model with schemes

Read a model from file which supports scheme selection in the parameter list.

⟨*Models: execute tests*⟩+≡

```
  call test (models_7, "models_7", &
       "handle schemes", &
       u, results)
```

⟨*Models: test declarations*⟩+≡

```
  public :: models_7
```

⟨*Models: tests*⟩+≡

```
  subroutine models_7 (u)
    integer, intent(in) :: u
    integer :: um
    character(80) :: buffer
    type(os_data_t) :: os_data
    type(model_list_t) :: model_list
    type(var_list_t), pointer :: var_list
    type(model_t), pointer :: model

    write (u, "(A)")  "* Test output: models_7"
    write (u, "(A)")  "*   Purpose: read a model from file &
         &with scheme selection"
    write (u, *)

    open (newunit=um, file="Test7.mdl", status="replace", action="readwrite")
    write (um, "(A)")  'model "Test7"'
```

```
    write (um, "(A)")  '  schemes = "foo", "bar", "gee"'
    write (um, "(A)")  ''
    write (um, "(A)")  '  select scheme'
    write (um, "(A)")  '  scheme "foo"'
    write (um, "(A)")  '    parameter a = 1'
    write (um, "(A)")  '    derived   b = 2 * a'
    write (um, "(A)")  '  scheme other'
    write (um, "(A)")  '    parameter b = 4'
    write (um, "(A)")  '    derived   a = b / 2'
    write (um, "(A)")  '  end select'
    write (um, "(A)")  ''
    write (um, "(A)")  '  parameter c = 3'
    write (um, "(A)")  ''
    write (um, "(A)")  '  select scheme'
    write (um, "(A)")  '  scheme "foo", "gee"'
    write (um, "(A)")  '    derived   d = b + c'
    write (um, "(A)")  '  scheme other'
    write (um, "(A)")  '    unused    d'
    write (um, "(A)")  '  end select'

    rewind (um)
    do
       read (um, "(A)", end=1)  buffer
       write (u, "(A)")  trim (buffer)
    end do
1   continue
    close (um)

    call syntax_model_file_init ()
    call os_data_init (os_data)

    write (u, *)
    write (u, "(A)")  "* Model output, default scheme (= foo)"
    write (u, *)

    call model_list%read_model (var_str ("Test7"), var_str ("Test7.mdl"), &
         os_data, model)
    call model%write (u, show_md5sum=.false.)
    call show_var_list ()
    call show_par_array ()

    call model_list%final ()

    write (u, *)
    write (u, "(A)")  "* Model output, scheme foo"
    write (u, *)

    call model_list%read_model (var_str ("Test7"), var_str ("Test7.mdl"), &
         os_data, model, scheme = var_str ("foo"))
    call model%write (u, show_md5sum=.false.)
    call show_var_list ()
    call show_par_array ()

    call model_list%final ()
```

```
    write (u, *)
    write (u, "(A)")  "* Model output, scheme bar"
    write (u, *)

    call model_list%read_model (var_str ("Test7"), var_str ("Test7.mdl"), &
         os_data, model, scheme = var_str ("bar"))
    call model%write (u, show_md5sum=.false.)
    call show_var_list ()
    call show_par_array ()

    call model_list%final ()

    write (u, *)
    write (u, "(A)")  "* Model output, scheme gee"
    write (u, *)

    call model_list%read_model (var_str ("Test7"), var_str ("Test7.mdl"), &
         os_data, model, scheme = var_str ("gee"))
    call model%write (u, show_md5sum=.false.)
    call show_var_list ()
    call show_par_array ()

    write (u, *)
    write (u, "(A)")  "* Cleanup"

    call model_list%final ()
    call syntax_model_file_final ()

    write (u, *)
    write (u, "(A)")  "* Test output end: models_7"

  contains

    subroutine show_var_list ()
      write (u, *)
      write (u, "(A)")  "* Variable list"
      write (u, *)
      var_list => model%get_var_list_ptr ()
      call var_list%write (u)
    end subroutine show_var_list

    subroutine show_par_array ()
      real(default), dimension(:), allocatable :: par
      integer :: n
      write (u, *)
      write (u, "(A)")  "* Parameter array"
      write (u, *)
      n = model%get_n_real ()
      allocate (par (n))
      call model%real_parameters_to_array (par)
      write (u, 1)  par
1     format (1X,F6.3)
    end subroutine show_par_array
```

```
end subroutine models_7
```

28.5 The SUSY Les Houches Accord

The SUSY Les Houches Accord defines a standard interfaces for storing the physics data of SUSY models. Here, we provide the means for reading, storing, and writing such data.

⟨slha_interface.f90⟩≡

```
⟨File header⟩

module slha_interface

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use constants
  use string_utils, only: upper_case
  use system_defs, only: VERSION_STRING
  use system_defs, only: EOF
  use diagnostics
  use os_interface
  use ifiles
  use lexers
  use syntax_rules
  use parser
  use variables
  use models

⟨Standard module head⟩

⟨SLHA: public⟩

⟨SLHA: parameters⟩

⟨SLHA: variables⟩

  save

contains

⟨SLHA: procedures⟩

⟨SLHA: tests⟩

end module slha_interface
```

28.5.1 Preprocessor

SLHA is a mixed-format standard. It should be read in assuming free format (but line-oriented), but it has some fixed-format elements.

To overcome this difficulty, we implement a preprocessing step which transforms the SLHA into a format that can be swallowed by our generic free-format lexer and parser. Each line with a blank first character is assumed to be a data line. We prepend a 'DATA' keyword to these lines. Furthermore, to enforce line-orientation, each line is appended a '$' key which is recognized by the

parser. To do this properly, we first remove trailing comments, and skip lines consisting only of comments.

The preprocessor reads from a stream and puts out an `ifile`. Blocks that are not recognized are skipped. For some blocks, data items are quoted, so they can be read as strings if necessary.

⟨*SLHA: parameters*⟩≡

```
integer, parameter :: MODE_SKIP = 0, MODE_DATA = 1, MODE_INFO = 2
```

⟨*SLHA: procedures*⟩≡

```
subroutine slha_preprocess (stream, ifile)
  type(stream_t), intent(inout), target :: stream
  type(ifile_t), intent(out) :: ifile
  type(string_t) :: buffer, line, item
  integer :: iostat
  integer :: mode
  mode = MODE
  SCAN_FILE: do
     call stream_get_record (stream, buffer, iostat)
     select case (iostat)
     case (0)
        call split (buffer, line, "#")
        if (len_trim (line) == 0)  cycle SCAN_FILE
        select case (char (extract (line, 1, 1)))
        case ("B", "b")
           mode = check_block_handling (line)
           call ifile_append (ifile, line // "$")
        case ("D", "d")
           mode = MODE_DATA
           call ifile_append (ifile, line // "$")
        case (" ")
           select case (mode)
           case (MODE_DATA)
              call ifile_append (ifile, "DATA" // line // "$")
           case (MODE_INFO)
              line = adjustl (line)
              call split (line, item, " ")
              call ifile_append (ifile, "INFO" // " " // item // " " &
                   // '"' // trim (adjustl (line)) // '" $')
           end select
        case default
           call msg_message (char (line))
           call msg_fatal ("SLHA: Incomprehensible line")
        end select
     case (EOF)
        exit SCAN_FILE
     case default
        call msg_fatal ("SLHA: I/O error occured while reading SLHA input")
     end select
  end do SCAN_FILE
end subroutine slha_preprocess
```

Return the mode that we should treat this block with. We need to recognize only those blocks that we actually use.

⟨*SLHA: procedures*⟩+≡

```
function check_block_handling (line) result (mode)
  integer :: mode
  type(string_t), intent(in) :: line
  type(string_t) :: buffer, key, block_name
  buffer = trim (line)
  call split (buffer, key, " ")
  buffer = adjustl (buffer)
  call split (buffer, block_name, " ")
  block_name = trim (adjustl (upper_case (block_name)))
  select case (char (block_name))
  case ("MODSEL", "MINPAR", "SMINPUTS")
     mode = MODE_DATA
  case ("MASS")
     mode = MODE_DATA
  case ("NMIX", "UMIX", "VMIX", "STOPMIX", "SBOTMIX", "STAUMIX")
     mode = MODE_DATA
  case ("NMHMIX", "NMAMIX", "NMNMIX", "NMSSMRUN")
     mode = MODE_DATA
  case ("ALPHA", "HMIX")
     mode = MODE_DATA
  case ("AU", "AD", "AE")
     mode = MODE_DATA
  case ("SPINFO", "DCINFO")
     mode = MODE_INFO
  case default
     mode = MODE_SKIP
  end select
end function check_block_handling
```

28.5.2 Lexer and syntax

⟨*SLHA: variables*⟩≡

```
type(syntax_t), target :: syntax_slha
```

⟨*SLHA: public*⟩≡

```
public :: syntax_slha_init
```

⟨*SLHA: procedures*⟩+≡

```
subroutine syntax_slha_init ()
  type(ifile_t) :: ifile
  call define_slha_syntax (ifile)
  call syntax_init (syntax_slha, ifile)
  call ifile_final (ifile)
end subroutine syntax_slha_init
```

⟨*SLHA: public*⟩+≡

```
public :: syntax_slha_final
```

⟨*SLHA: procedures*⟩+≡

```
subroutine syntax_slha_final ()
  call syntax_final (syntax_slha)
end subroutine syntax_slha_final
```

⟨*SLHA: public*⟩+≡

```
public :: syntax_slha_write
```

⟨*SLHA: procedures*⟩+≡

```
subroutine syntax_slha_write (unit)
  integer, intent(in), optional :: unit
  call syntax_write (syntax_slha, unit)
end subroutine syntax_slha_write
```

⟨*SLHA: procedures*⟩+≡

```
subroutine define_slha_syntax (ifile)
  type(ifile_t), intent(inout) :: ifile
  call ifile_append (ifile, "SEQ slha = chunk*")
  call ifile_append (ifile, "ALT chunk = block_def | decay_def")
  call ifile_append (ifile, "SEQ block_def = " &
       // "BLOCK block_spec '$' block_line*")
  call ifile_append (ifile, "KEY BLOCK")
  call ifile_append (ifile, "SEQ block_spec = block_name qvalue?")
  call ifile_append (ifile, "IDE block_name")
  call ifile_append (ifile, "SEQ qvalue = qname '=' real")
  call ifile_append (ifile, "IDE qname")
  call ifile_append (ifile, "KEY '='")
  call ifile_append (ifile, "REA real")
  call ifile_append (ifile, "KEY '$'")
  call ifile_append (ifile, "ALT block_line = block_data | block_info")
  call ifile_append (ifile, "SEQ block_data = DATA data_line '$'")
  call ifile_append (ifile, "KEY DATA")
  call ifile_append (ifile, "SEQ data_line = data_item+")
  call ifile_append (ifile, "ALT data_item = signed_number | number")
  call ifile_append (ifile, "SEQ signed_number = sign number")
  call ifile_append (ifile, "ALT sign = '+' | '-'")
  call ifile_append (ifile, "ALT number = integer | real")
  call ifile_append (ifile, "INT integer")
  call ifile_append (ifile, "KEY '-'")
  call ifile_append (ifile, "KEY '+'")
  call ifile_append (ifile, "SEQ block_info = INFO info_line '$'")
  call ifile_append (ifile, "KEY INFO")
  call ifile_append (ifile, "SEQ info_line = integer string_literal")
  call ifile_append (ifile, "QUO string_literal = '""'...'""'")
  call ifile_append (ifile, "SEQ decay_def = " &
       // "DECAY decay_spec '$' decay_data*")
  call ifile_append (ifile, "KEY DECAY")
  call ifile_append (ifile, "SEQ decay_spec = pdg_code data_item")
  call ifile_append (ifile, "ALT pdg_code = signed_integer | integer")
  call ifile_append (ifile, "SEQ signed_integer = sign integer")
  call ifile_append (ifile, "SEQ decay_data = DATA decay_line '$'")
  call ifile_append (ifile, "SEQ decay_line = data_item integer pdg_code+")
end subroutine define_slha_syntax
```

The SLHA specification allows for string data items in certain places. Currently, we do not interpret them, but the strings, which are not quoted, must be parsed somehow. The hack for this problem is to allow essentially all characters as special characters, so the string can be read before it is discarded.

⟨*SLHA: public*⟩+≡

```
public :: lexer_init_slha
```

⟨*SLHA: procedures*⟩+≡

```
subroutine lexer_init_slha (lexer)
  type(lexer_t), intent(out) :: lexer
  call lexer_init (lexer, &
       comment_chars = "#", &
       quote_chars = '"', &
       quote_match = '"', &
       single_chars = "+-=$", &
       special_class = [ "" ], &
       keyword_list = syntax_get_keyword_list_ptr (syntax_slha), &
       upper_case_keywords = .true.)  ! $
end subroutine lexer_init_slha
```

28.5.3 Interpreter

Find blocks

From the parse tree, find the node that represents a particular block. If `required` is true, issue an error if not found. Since `block_name` is always invoked with capital letters, we have to capitalize `pn_block_name`.

⟨*SLHA: procedures*⟩+≡

```
function slha_get_block_ptr &
     (parse_tree, block_name, required) result (pn_block)
  type(parse_node_t), pointer :: pn_block
  type(parse_tree_t), intent(in) :: parse_tree
  type(string_t), intent(in) :: block_name
  logical, intent(in) :: required
  type(parse_node_t), pointer :: pn_root, pn_block_spec, pn_block_name
  pn_root => parse_tree%get_root_ptr ()
  pn_block => parse_node_get_sub_ptr (pn_root)
  do while (associated (pn_block))
     select case (char (parse_node_get_rule_key (pn_block)))
     case ("block_def")
        pn_block_spec => parse_node_get_sub_ptr (pn_block, 2)
        pn_block_name => parse_node_get_sub_ptr (pn_block_spec)
        if (trim (adjustl (upper_case (parse_node_get_string &
             (pn_block_name)))) == block_name) then
           return
        end if
     end select
     pn_block => parse_node_get_next_ptr (pn_block)
  end do
  if (required) then
     call msg_fatal ("SLHA: block '" // char (block_name) // "' not found")
  end if
end function slha_get_block_ptr
```

Scan the file for the first/next DECAY block.

⟨*SLHA: procedures*⟩+≡

```
function slha_get_first_decay_ptr (parse_tree) result (pn_decay)
```

```
  type(parse_node_t), pointer :: pn_decay
  type(parse_tree_t), intent(in) :: parse_tree
  type(parse_node_t), pointer :: pn_root
  pn_root => parse_tree%get_root_ptr ()
  pn_decay => parse_node_get_sub_ptr (pn_root)
  do while (associated (pn_decay))
     select case (char (parse_node_get_rule_key (pn_decay)))
     case ("decay_def")
        return
     end select
     pn_decay => parse_node_get_next_ptr (pn_decay)
  end do
end function slha_get_first_decay_ptr

function slha_get_next_decay_ptr (pn_block) result (pn_decay)
  type(parse_node_t), pointer :: pn_decay
  type(parse_node_t), intent(in), target :: pn_block
  pn_decay => parse_node_get_next_ptr (pn_block)
  do while (associated (pn_decay))
     select case (char (parse_node_get_rule_key (pn_decay)))
     case ("decay_def")
        return
     end select
     pn_decay => parse_node_get_next_ptr (pn_decay)
  end do
end function slha_get_next_decay_ptr
```

Extract and transfer data from blocks

Given the parse node of a block, find the parse node of a particular switch or data line. Return this node and the node of the data item following the integer code.

⟨*SLHA: procedures*⟩+≡

```
subroutine slha_find_index_ptr (pn_block, pn_data, pn_item, code)
  type(parse_node_t), intent(in), target :: pn_block
  type(parse_node_t), intent(out), pointer :: pn_data
  type(parse_node_t), intent(out), pointer :: pn_item
  integer, intent(in) :: code
  pn_data => parse_node_get_sub_ptr (pn_block, 4)
  call slha_next_index_ptr (pn_data, pn_item, code)
end subroutine slha_find_index_ptr

subroutine slha_find_index_pair_ptr (pn_block, pn_data, pn_item, code1, code2)
  type(parse_node_t), intent(in), target :: pn_block
  type(parse_node_t), intent(out), pointer :: pn_data
  type(parse_node_t), intent(out), pointer :: pn_item
  integer, intent(in) :: code1, code2
  pn_data => parse_node_get_sub_ptr (pn_block, 4)
  call slha_next_index_pair_ptr (pn_data, pn_item, code1, code2)
end subroutine slha_find_index_pair_ptr
```

Starting from the pointer to a data line, find a data line with the given integer code.

⟨*SLHA: procedures*⟩+≡

```
subroutine slha_next_index_ptr (pn_data, pn_item, code)
  type(parse_node_t), intent(inout), pointer :: pn_data
  integer, intent(in) :: code
  type(parse_node_t), intent(out), pointer :: pn_item
  type(parse_node_t), pointer :: pn_line, pn_code
  do while (associated (pn_data))
     pn_line => parse_node_get_sub_ptr (pn_data, 2)
     pn_code => parse_node_get_sub_ptr (pn_line)
     select case (char (parse_node_get_rule_key (pn_code)))
     case ("integer")
        if (parse_node_get_integer (pn_code) == code) then
           pn_item => parse_node_get_next_ptr (pn_code)
           return
        end if
     end select
     pn_data => parse_node_get_next_ptr (pn_data)
  end do
  pn_item => null ()
end subroutine slha_next_index_ptr
```

Starting from the pointer to a data line, find a data line with the given integer code pair.

⟨*SLHA: procedures*⟩+≡

```
subroutine slha_next_index_pair_ptr (pn_data, pn_item, code1, code2)
  type(parse_node_t), intent(inout), pointer :: pn_data
  integer, intent(in) :: code1, code2
  type(parse_node_t), intent(out), pointer :: pn_item
  type(parse_node_t), pointer :: pn_line, pn_code1, pn_code2
  do while (associated (pn_data))
     pn_line => parse_node_get_sub_ptr (pn_data, 2)
     pn_code1 => parse_node_get_sub_ptr (pn_line)
     select case (char (parse_node_get_rule_key (pn_code1)))
     case ("integer")
        if (parse_node_get_integer (pn_code1) == code1) then
           pn_code2 => parse_node_get_next_ptr (pn_code1)
           if (associated (pn_code2)) then
              select case (char (parse_node_get_rule_key (pn_code2)))
              case ("integer")
                 if (parse_node_get_integer (pn_code2) == code2) then
                    pn_item => parse_node_get_next_ptr (pn_code2)
                    return
                 end if
              end select
           end if
        end if
     end select
     pn_data => parse_node_get_next_ptr (pn_data)
  end do
  pn_item => null ()
end subroutine slha_next_index_pair_ptr
```

Handle info data

Return all strings with index `i`. The result is an allocated string array. Since we do not know the number of matching entries in advance, we build an intermediate list which is transferred to the final array and deleted before exiting.

⟨*SLHA: procedures*⟩+≡

```
subroutine retrieve_strings_in_block (pn_block, code, str_array)
  type(parse_node_t), intent(in), target :: pn_block
  integer, intent(in) :: code
  type(string_t), dimension(:), allocatable, intent(out) :: str_array
  type(parse_node_t), pointer :: pn_data, pn_item
  type :: str_entry_t
     type(string_t) :: str
     type(str_entry_t), pointer :: next => null ()
  end type str_entry_t
  type(str_entry_t), pointer :: first => null ()
  type(str_entry_t), pointer :: current => null ()
  integer :: n
  n = 0
  call slha_find_index_ptr (pn_block, pn_data, pn_item, code)
  if (associated (pn_item)) then
     n = n + 1
     allocate (first)
     first%str = parse_node_get_string (pn_item)
     current => first
     do while (associated (pn_data))
        pn_data => parse_node_get_next_ptr (pn_data)
        call slha_next_index_ptr (pn_data, pn_item, code)
        if (associated (pn_item)) then
           n = n + 1
           allocate (current%next)
           current => current%next
           current%str = parse_node_get_string (pn_item)
        end if
     end do
     allocate (str_array (n))
     n = 0
     do while (associated (first))
        n = n + 1
        current => first
        str_array(n) = current%str
        first => first%next
        deallocate (current)
     end do
  else
     allocate (str_array (0))
  end if
end subroutine retrieve_strings_in_block
```

Transfer data from SLHA to variables

Extract real parameter with index `i`. If it does not exist, retrieve it from the variable list, using the given name.

⟨*SLHA: procedures*⟩+≡

```
function get_parameter_in_block (pn_block, code, name, var_list) result (var)
  real(default) :: var
  type(parse_node_t), intent(in), target :: pn_block
  integer, intent(in) :: code
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in), target :: var_list
  type(parse_node_t), pointer :: pn_data, pn_item
  call slha_find_index_ptr (pn_block, pn_data, pn_item, code)
  if (associated (pn_item)) then
     var = get_real_parameter (pn_item)
  else
     var = var_list%get_rval (name)
  end if
end function get_parameter_in_block
```

Extract a real data item with index `i`. If it does exist, set it in the variable list, using the given name. If the variable is not present in the variable list, ignore it.

⟨*SLHA: procedures*⟩+≡

```
subroutine set_data_item (pn_block, code, name, var_list)
  type(parse_node_t), intent(in), target :: pn_block
  integer, intent(in) :: code
  type(string_t), intent(in) :: name
  type(var_list_t), intent(inout), target :: var_list
  type(parse_node_t), pointer :: pn_data, pn_item
  call slha_find_index_ptr (pn_block, pn_data, pn_item, code)
  if (associated (pn_item)) then
     call var_list%set_real (name,  get_real_parameter (pn_item), &
           is_known=.true., ignore=.true.)
  end if
end subroutine set_data_item
```

Extract a real matrix element with index `i,j`. If it does exists, set it in the variable list, using the given name. If the variable is not present in the variable list, ignore it.

⟨*SLHA: procedures*⟩+≡

```
subroutine set_matrix_element (pn_block, code1, code2, name, var_list)
  type(parse_node_t), intent(in), target :: pn_block
  integer, intent(in) :: code1, code2
  type(string_t), intent(in) :: name
  type(var_list_t), intent(inout), target :: var_list
  type(parse_node_t), pointer :: pn_data, pn_item
  call slha_find_index_pair_ptr (pn_block, pn_data, pn_item, code1, code2)
  if (associated (pn_item)) then
     call var_list%set_real (name, get_real_parameter (pn_item), &
           is_known=.true., ignore=.true.)
  end if
end subroutine set_matrix_element
```

Transfer data from variables to SLHA

Get a real/integer parameter with index `i` from the variable list and write it to the current output file. In the integer case, we account for the fact that the variable is type real. If it does not exist, do nothing.

⟨*SLHA: procedures*⟩+≡

```
subroutine write_integer_data_item (u, code, name, var_list, comment)
  integer, intent(in) :: u
  integer, intent(in) :: code
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in) :: var_list
  character(*), intent(in) :: comment
  integer :: item
  if (var_list%contains (name)) then
     item = nint (var_list%get_rval (name))
     call write_integer_parameter (u, code, item, comment)
  end if
end subroutine write_integer_data_item

subroutine write_real_data_item (u, code, name, var_list, comment)
  integer, intent(in) :: u
  integer, intent(in) :: code
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in) :: var_list
  character(*), intent(in) :: comment
  real(default) :: item
  if (var_list%contains (name)) then
     item = var_list%get_rval (name)
     call write_real_parameter (u, code, item, comment)
  end if
end subroutine write_real_data_item
```

Get a real data item with two integer indices from the variable list and write it to the current output file. If it does not exist, do nothing.

⟨*SLHA: procedures*⟩+≡

```
subroutine write_matrix_element (u, code1, code2, name, var_list, comment)
  integer, intent(in) :: u
  integer, intent(in) :: code1, code2
  type(string_t), intent(in) :: name
  type(var_list_t), intent(in) :: var_list
  character(*), intent(in) :: comment
  real(default) :: item
  if (var_list%contains (name)) then
     item = var_list%get_rval (name)
     call write_real_matrix_element (u, code1, code2, item, comment)
  end if
end subroutine write_matrix_element
```

28.5.4 Auxiliary function

Write a block header.

⟨*SLHA: procedures*⟩+≡

```
subroutine write_block_header (u, name, comment)
  integer, intent(in) :: u
  character(*), intent(in) :: name, comment
  write (u, "(A,1x,A,3x,'#',1x,A)")  "BLOCK", name, comment
end subroutine write_block_header
```

Extract a real parameter that may be defined real or integer, signed or unsigned.

⟨*SLHA: procedures*⟩+≡

```
function get_real_parameter (pn_item) result (var)
  real(default) :: var
  type(parse_node_t), intent(in), target :: pn_item
  type(parse_node_t), pointer :: pn_sign, pn_var
  integer :: sign
  select case (char (parse_node_get_rule_key (pn_item)))
  case ("signed_number")
     pn_sign => parse_node_get_sub_ptr (pn_item)
     pn_var  => parse_node_get_next_ptr (pn_sign)
     select case (char (parse_node_get_key (pn_sign)))
     case ("+");  sign = +1
     case ("-");  sign = -1
     end select
  case default
     sign = +1
     pn_var => pn_item
  end select
  select case (char (parse_node_get_rule_key (pn_var)))
  case ("integer");  var = sign * parse_node_get_integer (pn_var)
  case ("real");     var = sign * parse_node_get_real (pn_var)
  end select
end function get_real_parameter
```

Auxiliary: Extract an integer parameter that may be defined signed or unsigned. A real value is an error.

⟨*SLHA: procedures*⟩+≡

```
function get_integer_parameter (pn_item) result (var)
  integer :: var
  type(parse_node_t), intent(in), target :: pn_item
  type(parse_node_t), pointer :: pn_sign, pn_var
  integer :: sign
  select case (char (parse_node_get_rule_key (pn_item)))
  case ("signed_integer")
     pn_sign => parse_node_get_sub_ptr (pn_item)
     pn_var  => parse_node_get_next_ptr (pn_sign)
     select case (char (parse_node_get_key (pn_sign)))
     case ("+");  sign = +1
     case ("-");  sign = -1
     end select
  case ("integer")
     sign = +1
```

```
      pn_var => pn_item
    case default
      call parse_node_write (pn_var)
      call msg_error ("SLHA: Integer parameter expected")
      var = 0
      return
    end select
    var = sign * parse_node_get_integer (pn_var)
  end function get_integer_parameter
```

Write an integer parameter with a single index directly to file, using the required output format.

⟨*SLHA: procedures*⟩+≡

```
  subroutine write_integer_parameter (u, code, item, comment)
    integer, intent(in) :: u
    integer, intent(in) :: code
    integer, intent(in) :: item
    character(*), intent(in) :: comment
1   format (1x, I9, 3x, 3x, I9, 4x, 3x, '#', 1x, A)
    write (u, 1)  code, item, comment
  end subroutine write_integer_parameter
```

Write a real parameter with two indices directly to file, using the required output format.

⟨*SLHA: procedures*⟩+≡

```
  subroutine write_real_parameter (u, code, item, comment)
    integer, intent(in) :: u
    integer, intent(in) :: code
    real(default), intent(in) :: item
    character(*), intent(in) :: comment
1   format (1x, I9, 3x, 1P, E16.8, 0P, 3x, '#', 1x, A)
    write (u, 1)  code, item, comment
  end subroutine write_real_parameter
```

Write a real parameter with a single index directly to file, using the required output format.

⟨*SLHA: procedures*⟩+≡

```
  subroutine write_real_matrix_element (u, code1, code2, item, comment)
    integer, intent(in) :: u
    integer, intent(in) :: code1, code2
    real(default), intent(in) :: item
    character(*), intent(in) :: comment
1   format (1x, I2, 1x, I2, 3x, 1P, E16.8, 0P, 3x, '#', 1x, A)
    write (u, 1)  code1, code2, item, comment
  end subroutine write_real_matrix_element
```

The concrete SLHA interpreter

SLHA codes for particular physics models

⟨*SLHA: parameters*⟩+≡

```
integer, parameter :: MDL_MSSM = 0
integer, parameter :: MDL_NMSSM = 1
```

Take the parse tree and extract relevant data. Select the correct model and store all data that is present in the appropriate variable list. Finally, update the variable record.

Public for use in unit test.

⟨*SLHA: public*⟩+≡

```
public :: slha_interpret_parse_tree
```

⟨*SLHA: procedures*⟩+≡

```
subroutine slha_interpret_parse_tree &
     (parse_tree, model, input, spectrum, decays)
  type(parse_tree_t), intent(in) :: parse_tree
  type(model_t), intent(inout), target :: model
  logical, intent(in) :: input, spectrum, decays
  logical :: errors
  integer :: mssm_type
  call slha_handle_MODSEL (parse_tree, model, mssm_type)
  if (input) then
     call slha_handle_SMINPUTS (parse_tree, model)
     call slha_handle_MINPAR (parse_tree, model, mssm_type)
  end if
  if (spectrum) then
     call slha_handle_info_block (parse_tree, "SPINFO", errors)
     if (errors)  return
     call slha_handle_MASS (parse_tree, model)
     call slha_handle_matrix_block (parse_tree, "NMIX", "mn_", 4, 4, model)
     call slha_handle_matrix_block (parse_tree, "NMNMIX", "mixn_", 5, 5, model)
     call slha_handle_matrix_block (parse_tree, "UMIX", "mu_", 2, 2, model)
     call slha_handle_matrix_block (parse_tree, "VMIX", "mv_", 2, 2, model)
     call slha_handle_matrix_block (parse_tree, "STOPMIX", "mt_", 2, 2, model)
     call slha_handle_matrix_block (parse_tree, "SBOTMIX", "mb_", 2, 2, model)
     call slha_handle_matrix_block (parse_tree, "STAUMIX", "ml_", 2, 2, model)
     call slha_handle_matrix_block (parse_tree, "NMHMIX", "mixh0_", 3, 3, model)
     call slha_handle_matrix_block (parse_tree, "NMAMIX", "mixa0_", 2, 3, model)
     call slha_handle_ALPHA (parse_tree, model)
     call slha_handle_HMIX (parse_tree, model)
     call slha_handle_NMSSMRUN (parse_tree, model)
     call slha_handle_matrix_block (parse_tree, "AU", "Au_", 3, 3, model)
     call slha_handle_matrix_block (parse_tree, "AD", "Ad_", 3, 3, model)
     call slha_handle_matrix_block (parse_tree, "AE", "Ae_", 3, 3, model)
  end if
  if (decays) then
     call slha_handle_info_block (parse_tree, "DCINFO", errors)
     if (errors)  return
     call slha_handle_decays (parse_tree, model)
  end if
end subroutine slha_interpret_parse_tree
```

Info blocks

Handle the informational blocks SPINFO and DCINFO. The first two items are program name and version. Items with index 3 are warnings. Items with index

4 are errors. We reproduce these as WHIZARD warnings and errors.

⟨*SLHA: procedures*⟩+≡

```
subroutine slha_handle_info_block (parse_tree, block_name, errors)
  type(parse_tree_t), intent(in) :: parse_tree
  character(*), intent(in) :: block_name
  logical, intent(out) :: errors
  type(parse_node_t), pointer :: pn_block
  type(string_t), dimension(:), allocatable :: msg
  integer :: i
  pn_block => slha_get_block_ptr &
       (parse_tree, var_str (block_name), required=.true.)
  if (.not. associated (pn_block)) then
     call msg_error ("SLHA: Missing info block '" &
          // trim (block_name) // "'; ignored.")
     errors = .true.
     return
  end if
  select case (block_name)
  case ("SPINFO")
     call msg_message ("SLHA: SUSY spectrum program info:")
  case ("DCINFO")
     call msg_message ("SLHA: SUSY decay program info:")
  end select
  call retrieve_strings_in_block (pn_block, 1, msg)
  do i = 1, size (msg)
     call msg_message ("SLHA: " // char (msg(i)))
  end do
  call retrieve_strings_in_block (pn_block, 2, msg)
  do i = 1, size (msg)
     call msg_message ("SLHA: " // char (msg(i)))
  end do
  call retrieve_strings_in_block (pn_block, 3, msg)
  do i = 1, size (msg)
     call msg_warning ("SLHA: " // char (msg(i)))
  end do
  call retrieve_strings_in_block (pn_block, 4, msg)
  do i = 1, size (msg)
     call msg_error ("SLHA: " // char (msg(i)))
  end do
  errors = size (msg) > 0
end subroutine slha_handle_info_block
```

MODSEL

Handle the overall model definition. Only certain models are recognized. The soft-breaking model templates that determine the set of input parameters:

⟨*SLHA: parameters*⟩+≡

```
integer, parameter :: MSSM_GENERIC = 0
integer, parameter :: MSSM_SUGRA = 1
integer, parameter :: MSSM_GMSB = 2
integer, parameter :: MSSM_AMSB = 3
```

⟨*SLHA: procedures*⟩+≡

```
subroutine slha_handle_MODSEL (parse_tree, model, mssm_type)
  type(parse_tree_t), intent(in) :: parse_tree
  type(model_t), intent(in), target :: model
  integer, intent(out) :: mssm_type
  type(parse_node_t), pointer :: pn_block, pn_data, pn_item
  type(string_t) :: model_name
  pn_block => slha_get_block_ptr &
       (parse_tree, var_str ("MODSEL"), required=.true.)
  call slha_find_index_ptr (pn_block, pn_data, pn_item, 1)
  if (associated (pn_item)) then
     mssm_type = get_integer_parameter (pn_item)
  else
     mssm_type = MSSM_GENERIC
  end if
  call slha_find_index_ptr (pn_block, pn_data, pn_item, 3)
  if (associated (pn_item)) then
     select case (parse_node_get_integer (pn_item))
     case (MDL_MSSM);  model_name = "MSSM"
     case (MDL_NMSSM); model_name = "NMSSM"
     case default
        call msg_fatal ("SLHA: unknown model code in MODSEL")
        return
     end select
  else
     model_name = "MSSM"
  end if
  call slha_find_index_ptr (pn_block, pn_data, pn_item, 4)
  if (associated (pn_item)) then
    call msg_fatal (" R-parity violation is currently not supported by WHIZARD.")
  end if
  call slha_find_index_ptr (pn_block, pn_data, pn_item, 5)
  if (associated (pn_item)) then
    call msg_fatal (" CP violation is currently not supported by WHIZARD.")
  end if
  select case (char (model_name))
  case ("MSSM")
     select case (char (model%get_name ()))
     case ("MSSM","MSSM_CKM","MSSM_Grav","MSSM_Hgg")
        model_name = model%get_name ()
     case default
        call msg_fatal ("Selected model '" &
             // char (model%get_name ()) // "' does not match model '" &
             // char (model_name) // "' in SLHA input file.")
        return
     end select
  case ("NMSSM")
     select case (char (model%get_name ()))
     case ("NMSSM","NMSSM_CKM","NMSSM_Hgg")
        model_name = model%get_name ()
     case default
        call msg_fatal ("Selected model '" &
             // char (model%get_name ()) // "' does not match model '" &
             // char (model_name) // "' in SLHA input file.")
```

```
        return
      end select
    case default
      call msg_bug ("SLHA model name '" &
           // char (model_name) // "' not recognized.")
      return
    end select
    call msg_message ("SLHA: Initializing model '" // char (model_name) // "'")
  end subroutine slha_handle_MODSEL
```

Write a MODSEL block, based on the contents of the current model.

⟨*SLHA: procedures*⟩+≡

```
  subroutine slha_write_MODSEL (u, model, mssm_type)
    integer, intent(in) :: u
    type(model_t), intent(in), target :: model
    integer, intent(out) :: mssm_type
    type(var_list_t), pointer :: var_list
    integer :: model_id
    type(string_t) :: mtype_string
    var_list => model%get_var_list_ptr ()
    if (var_list%contains (var_str ("mtype"))) then
      mssm_type = nint (var_list%get_rval (var_str ("mtype")))
    else
      call msg_error ("SLHA: parameter 'mtype' (SUSY breaking scheme) " &
           // "is unknown in current model, no SLHA output possible")
      mssm_type = -1
      return
    end if
    call write_block_header (u, "MODSEL", "SUSY model selection")
    select case (mssm_type)
    case (0);  mtype_string = "Generic MSSM"
    case (1);  mtype_string = "SUGRA"
    case (2);  mtype_string = "GMSB"
    case (3);  mtype_string = "AMSB"
    case default
      mtype_string = "unknown"
    end select
    call write_integer_parameter (u, 1, mssm_type, &
         "SUSY-breaking scheme: " // char (mtype_string))
    select case (char (model%get_name ()))
    case ("MSSM");  model_id = MDL_MSSM
    case ("NMSSM"); model_id = MDL_NMSSM
    case default
      model_id = 0
    end select
    call write_integer_parameter (u, 3, model_id, &
         "SUSY model type: " // char (model%get_name ()))
  end subroutine slha_write_MODSEL
```

SMINPUTS

Read SM parameters and update the variable list accordingly. If a parameter is not defined in the block, we use the previous value from the model variable list. For the basic parameters we have to do a small recalculation, since SLHA uses the G_F-α-m_Z scheme, while WHIZARD derives them from G_F, m_W, and m_Z.

⟨*SLHA: procedures*⟩+≡

```
subroutine slha_handle_SMINPUTS (parse_tree, model)
  type(parse_tree_t), intent(in) :: parse_tree
  type(model_t), intent(inout), target :: model
  type(parse_node_t), pointer :: pn_block
  real(default) :: alpha_em_i, GF, alphas, mZ
  real(default) :: ee, vv, cw_sw, cw2, mW
  real(default) :: mb, mtop, mtau
  type(var_list_t), pointer :: var_list
  var_list => model%get_var_list_ptr ()
  pn_block => slha_get_block_ptr &
       (parse_tree, var_str ("SMINPUTS"), required=.true.)
  if (.not. (associated (pn_block)))  return
  alpha_em_i = &
       get_parameter_in_block (pn_block, 1, var_str ("alpha_em_i"), var_list)
  GF = get_parameter_in_block (pn_block, 2, var_str ("GF"), var_list)
  alphas = &
       get_parameter_in_block (pn_block, 3, var_str ("alphas"), var_list)
  mZ   = get_parameter_in_block (pn_block, 4, var_str ("mZ"), var_list)
  mb   = get_parameter_in_block (pn_block, 5, var_str ("mb"), var_list)
  mtop = get_parameter_in_block (pn_block, 6, var_str ("mtop"), var_list)
  mtau = get_parameter_in_block (pn_block, 7, var_str ("mtau"), var_list)
  ee = sqrt (4 * pi / alpha_em_i)
  vv = 1 / sqrt (sqrt (2._default) * GF)
  cw_sw = ee * vv / (2 * mZ)
  if (2*cw_sw <= 1) then
     cw2 = (1 + sqrt (1 - 4 * cw_sw**2)) / 2
     mW = mZ * sqrt (cw2)
     call var_list%set_real (var_str ("GF"), GF, .true.)
     call var_list%set_real (var_str ("mZ"), mZ, .true.)
     call var_list%set_real (var_str ("mW"), mW, .true.)
     call var_list%set_real (var_str ("mtau"), mtau, .true.)
     call var_list%set_real (var_str ("mb"), mb, .true.)
     call var_list%set_real (var_str ("mtop"), mtop, .true.)
     call var_list%set_real (var_str ("alphas"), alphas, .true.)
  else
     call msg_fatal ("SLHA: Unphysical SM parameter values")
     return
  end if
end subroutine slha_handle_SMINPUTS
```

Write a SMINPUTS block.

⟨*SLHA: procedures*⟩+≡

```
subroutine slha_write_SMINPUTS (u, model)
  integer, intent(in) :: u
  type(model_t), intent(in), target :: model
  type(var_list_t), pointer :: var_list
```

```
    var_list => model%get_var_list_ptr ()
    call write_block_header (u, "SMINPUTS", "SM input parameters")
    call write_real_data_item (u, 1, var_str ("alpha_em_i"), var_list, &
         "Inverse electromagnetic coupling alpha (Z pole)")
    call write_real_data_item (u, 2, var_str ("GF"), var_list, &
         "Fermi constant")
    call write_real_data_item (u, 3, var_str ("alphas"), var_list, &
         "Strong coupling alpha_s (Z pole)")
    call write_real_data_item (u, 4, var_str ("mZ"), var_list, &
         "Z mass")
    call write_real_data_item (u, 5, var_str ("mb"), var_list, &
         "b running mass (at mb)")
    call write_real_data_item (u, 6, var_str ("mtop"), var_list, &
         "top mass")
    call write_real_data_item (u, 7, var_str ("mtau"), var_list, &
         "tau mass")
  end subroutine slha_write_SMINPUTS
```

MINPAR

The block of SUSY input parameters. They are accessible to WHIZARD, but they only get used when an external spectrum generator is invoked. The precise set of parameters depends on the type of SUSY breaking, which by itself is one of the parameters.

⟨*SLHA: procedures*⟩+≡

```
  subroutine slha_handle_MINPAR (parse_tree, model, mssm_type)
    type(parse_tree_t), intent(in) :: parse_tree
    type(model_t), intent(inout), target :: model
    integer, intent(in) :: mssm_type
    type(var_list_t), pointer :: var_list
    type(parse_node_t), pointer :: pn_block
    var_list => model%get_var_list_ptr ()
    call var_list%set_real &
         (var_str ("mtype"),  real(mssm_type, default), is_known=.true.)
    pn_block => slha_get_block_ptr &
         (parse_tree, var_str ("MINPAR"), required=.true.)
    select case (mssm_type)
    case (MSSM_SUGRA)
       call set_data_item (pn_block, 1, var_str ("m_zero"), var_list)
       call set_data_item (pn_block, 2, var_str ("m_half"), var_list)
       call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
       call set_data_item (pn_block, 4, var_str ("sgn_mu"), var_list)
       call set_data_item (pn_block, 5, var_str ("A0"), var_list)
    case (MSSM_GMSB)
       call set_data_item (pn_block, 1, var_str ("Lambda"), var_list)
       call set_data_item (pn_block, 2, var_str ("M_mes"), var_list)
       call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
       call set_data_item (pn_block, 4, var_str ("sgn_mu"), var_list)
       call set_data_item (pn_block, 5, var_str ("N_5"), var_list)
       call set_data_item (pn_block, 6, var_str ("c_grav"), var_list)
    case (MSSM_AMSB)
       call set_data_item (pn_block, 1, var_str ("m_zero"), var_list)
```

```
      call set_data_item (pn_block, 2, var_str ("m_grav"), var_list)
      call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
      call set_data_item (pn_block, 4, var_str ("sgn_mu"), var_list)
    case default
      call set_data_item (pn_block, 3, var_str ("tanb"), var_list)
    end select
  end subroutine slha_handle_MINPAR
```

Write a MINPAR block as appropriate for the current model type.

⟨*SLHA: procedures*⟩+≡

```
  subroutine slha_write_MINPAR (u, model, mssm_type)
    integer, intent(in) :: u
    type(model_t), intent(in), target :: model
    integer, intent(in) :: mssm_type
    type(var_list_t), pointer :: var_list
    var_list => model%get_var_list_ptr ()
    call write_block_header (u, "MINPAR", "Basic SUSY input parameters")
    select case (mssm_type)
    case (MSSM_SUGRA)
      call write_real_data_item (u, 1, var_str ("m_zero"), var_list, &
           "Common scalar mass")
      call write_real_data_item (u, 2, var_str ("m_half"), var_list, &
           "Common gaugino mass")
      call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
           "tan(beta)")
      call write_integer_data_item (u, 4, &
           var_str ("sgn_mu"), var_list, &
           "Sign of mu")
      call write_real_data_item (u, 5, var_str ("A0"), var_list, &
           "Common trilinear coupling")
    case (MSSM_GMSB)
      call write_real_data_item (u, 1, var_str ("Lambda"), var_list, &
           "Soft-breaking scale")
      call write_real_data_item (u, 2, var_str ("M_mes"), var_list, &
           "Messenger scale")
      call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
           "tan(beta)")
      call write_integer_data_item (u, 4, &
           var_str ("sgn_mu"), var_list, &
           "Sign of mu")
      call write_integer_data_item (u, 5, var_str ("N_5"), var_list, &
           "Messenger index")
      call write_real_data_item (u, 6, var_str ("c_grav"), var_list, &
           "Gravitino mass factor")
    case (MSSM_AMSB)
      call write_real_data_item (u, 1, var_str ("m_zero"), var_list, &
           "Common scalar mass")
      call write_real_data_item (u, 2, var_str ("m_grav"), var_list, &
           "Gravitino mass")
      call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
           "tan(beta)")
      call write_integer_data_item (u, 4, &
           var_str ("sgn_mu"), var_list, &
           "Sign of mu")
```

```
    case default
       call write_real_data_item (u, 3, var_str ("tanb"), var_list, &
            "tan(beta)")
    end select
  end subroutine slha_write_MINPAR
```

Mass spectrum

Set masses. Since the particles are identified by PDG code, read the line and try to set the appropriate particle mass in the current model. At the end, update parameters, just in case the W or Z mass was included.

⟨*SLHA: procedures*⟩+≡

```
  subroutine slha_handle_MASS (parse_tree, model)
    type(parse_tree_t), intent(in) :: parse_tree
    type(model_t), intent(inout), target :: model
    type(parse_node_t), pointer :: pn_block, pn_data, pn_line, pn_code
    type(parse_node_t), pointer :: pn_mass
    integer :: pdg
    real(default) :: mass
    pn_block => slha_get_block_ptr &
         (parse_tree, var_str ("MASS"), required=.true.)
    if (.not. (associated (pn_block)))  return
    pn_data => parse_node_get_sub_ptr (pn_block, 4)
    do while (associated (pn_data))
       pn_line => parse_node_get_sub_ptr (pn_data, 2)
       pn_code => parse_node_get_sub_ptr (pn_line)
       if (associated (pn_code)) then
          pdg = get_integer_parameter (pn_code)
          pn_mass => parse_node_get_next_ptr (pn_code)
          if (associated (pn_mass)) then
             mass = get_real_parameter (pn_mass)
             call model%set_field_mass (pdg, mass)
          else
             call msg_error ("SLHA: Block MASS: Missing mass value")
          end if
       else
          call msg_error ("SLHA: Block MASS: Missing PDG code")
       end if
       pn_data => parse_node_get_next_ptr (pn_data)
    end do
  end subroutine slha_handle_MASS
```

Widths

Set widths. For each DECAY block, extract the header, read the PDG code and width, and try to set the appropriate particle width in the current model.

⟨*SLHA: procedures*⟩+≡

```
  subroutine slha_handle_decays (parse_tree, model)
    type(parse_tree_t), intent(in) :: parse_tree
    type(model_t), intent(inout), target :: model
    type(parse_node_t), pointer :: pn_decay, pn_decay_spec, pn_code, pn_width
```

```
    integer :: pdg
    real(default) :: width
    pn_decay => slha_get_first_decay_ptr (parse_tree)
    do while (associated (pn_decay))
       pn_decay_spec => parse_node_get_sub_ptr (pn_decay, 2)
       pn_code => parse_node_get_sub_ptr (pn_decay_spec)
       pdg = get_integer_parameter (pn_code)
       pn_width => parse_node_get_next_ptr (pn_code)
       width = get_real_parameter (pn_width)
       call model%set_field_width (pdg, width)
       pn_decay => slha_get_next_decay_ptr (pn_decay)
    end do
  end subroutine slha_handle_decays
```

Mixing matrices

Read mixing matrices. We can treat all matrices by a single procedure if we just know the block name, variable prefix, and matrix dimension. The matrix dimension must be less than 10. For the pseudoscalar Higgses in NMSSM-type models we need off-diagonal matrices, so we generalize the definition.

⟨*SLHA: procedures*⟩+≡

```
  subroutine slha_handle_matrix_block &
       (parse_tree, block_name, var_prefix, dim1, dim2, model)
    type(parse_tree_t), intent(in) :: parse_tree
    character(*), intent(in) :: block_name, var_prefix
    integer, intent(in) :: dim1, dim2
    type(model_t), intent(inout), target :: model
    type(parse_node_t), pointer :: pn_block
    type(var_list_t), pointer :: var_list
    integer :: i, j
    character(len=len(var_prefix)+2) :: var_name
    var_list => model%get_var_list_ptr ()
    pn_block => slha_get_block_ptr &
         (parse_tree, var_str (block_name), required=.false.)
    if (.not. (associated (pn_block)))  return
    do i = 1, dim1
       do j = 1, dim2
          write (var_name, "(A,I1,I1)")  var_prefix, i, j
          call set_matrix_element (pn_block, i, j, var_str (var_name), var_list)
       end do
    end do
  end subroutine slha_handle_matrix_block
```

Higgs data

Read the block ALPHA which holds just the Higgs mixing angle.

⟨*SLHA: procedures*⟩+≡

```
  subroutine slha_handle_ALPHA (parse_tree, model)
    type(parse_tree_t), intent(in) :: parse_tree
    type(model_t), intent(inout), target :: model
    type(parse_node_t), pointer :: pn_block, pn_line, pn_data, pn_item
```

```
    type(var_list_t), pointer :: var_list
    real(default) :: al_h
    var_list => model%get_var_list_ptr ()
    pn_block => slha_get_block_ptr &
         (parse_tree, var_str ("ALPHA"), required=.false.)
    if (.not. (associated (pn_block)))  return
    pn_data => parse_node_get_sub_ptr (pn_block, 4)
    pn_line => parse_node_get_sub_ptr (pn_data, 2)
    pn_item => parse_node_get_sub_ptr (pn_line)
    if (associated (pn_item)) then
       al_h = get_real_parameter (pn_item)
       call var_list%set_real (var_str ("al_h"), al_h, &
            is_known=.true., ignore=.true.)
    end if
  end subroutine slha_handle_ALPHA
```

Read the block HMIX for the Higgs mixing parameters

⟨*SLHA: procedures*⟩+≡

```
  subroutine slha_handle_HMIX (parse_tree, model)
    type(parse_tree_t), intent(in) :: parse_tree
    type(model_t), intent(inout), target :: model
    type(parse_node_t), pointer :: pn_block
    type(var_list_t), pointer :: var_list
    var_list => model%get_var_list_ptr ()
    pn_block => slha_get_block_ptr &
         (parse_tree, var_str ("HMIX"), required=.false.)
    if (.not. (associated (pn_block)))  return
    call set_data_item (pn_block, 1, var_str ("mu_h"), var_list)
    call set_data_item (pn_block, 2, var_str ("tanb_h"), var_list)
  end subroutine slha_handle_HMIX
```

Read the block NMSSMRUN for the specific NMSSM parameters

⟨*SLHA: procedures*⟩+≡

```
  subroutine slha_handle_NMSSMRUN (parse_tree, model)
    type(parse_tree_t), intent(in) :: parse_tree
    type(model_t), intent(inout), target :: model
    type(parse_node_t), pointer :: pn_block
    type(var_list_t), pointer :: var_list
    var_list => model%get_var_list_ptr ()
    pn_block => slha_get_block_ptr &
         (parse_tree, var_str ("NMSSMRUN"), required=.false.)
    if (.not. (associated (pn_block)))  return
    call set_data_item (pn_block, 1, var_str ("ls"), var_list)
    call set_data_item (pn_block, 2, var_str ("ks"), var_list)
    call set_data_item (pn_block, 3, var_str ("a_ls"), var_list)
    call set_data_item (pn_block, 4, var_str ("a_ks"), var_list)
    call set_data_item (pn_block, 5, var_str ("nmu"), var_list)
    end subroutine slha_handle_NMSSMRUN
```

28.5.5 Parser

Read a SLHA file from stream, including preprocessing, and make up a parse tree.

⟨*SLHA: procedures*⟩+≡

```
subroutine slha_parse_stream (stream, parse_tree)
  type(stream_t), intent(inout), target :: stream
  type(parse_tree_t), intent(out) :: parse_tree
  type(ifile_t) :: ifile
  type(lexer_t) :: lexer
  type(stream_t), target :: stream_tmp
  call slha_preprocess (stream, ifile)
  call stream_init (stream_tmp, ifile)
  call lexer_init_slha (lexer)
  call lexer_assign_stream (lexer, stream_tmp)
  call parse_tree_init (parse_tree, syntax_slha, lexer)
  call lexer_final (lexer)
  call stream_final (stream_tmp)
  call ifile_final (ifile)
end subroutine slha_parse_stream
```

Read a SLHA file chosen by name. Check first the current directory, then the directory where SUSY input files should be located.

Required for test:

⟨*SLHA: public*⟩+≡

```
public :: slha_parse_file
```

⟨*SLHA: procedures*⟩+≡

```
subroutine slha_parse_file (file, os_data, parse_tree)
  type(string_t), intent(in) :: file
  type(os_data_t), intent(in) :: os_data
  type(parse_tree_t), intent(out) :: parse_tree
  logical :: exist
  type(string_t) :: filename
  type(stream_t), target :: stream
  call msg_message ("Reading SLHA input file '" // char (file) // "'")
  filename = file
  inquire (file=char(filename), exist=exist)
  if (.not. exist) then
     filename = os_data%whizard_susypath // "/" // file
     inquire (file=char(filename), exist=exist)
     if (.not. exist) then
        call msg_fatal ("SLHA input file '" // char (file) // "' not found")
        return
     end if
  end if
  call stream_init (stream, char (filename))
  call slha_parse_stream (stream, parse_tree)
  call stream_final (stream)
end subroutine slha_parse_file
```

28.5.6 API

Read the SLHA file, parse it, and interpret the parse tree. The model parameters retrieved from the file will be inserted into the appropriate model, which is loaded and modified in the background. The pointer to this model is returned as the last argument.

⟨*SLHA: public*⟩+≡

```
public :: slha_read_file
```

⟨*SLHA: procedures*⟩+≡

```
subroutine slha_read_file &
     (file, os_data, model, input, spectrum, decays)
  type(string_t), intent(in) :: file
  type(os_data_t), intent(in) :: os_data
  type(model_t), intent(inout), target :: model
  logical, intent(in) :: input, spectrum, decays
  type(parse_tree_t) :: parse_tree
  call slha_parse_file (file, os_data, parse_tree)
  if (associated (parse_tree%get_root_ptr ())) then
     call slha_interpret_parse_tree &
          (parse_tree, model, input, spectrum, decays)
     call parse_tree_final (parse_tree)
     call model%update_parameters ()
  end if
end subroutine slha_read_file
```

Write the SLHA contents, as far as possible, to external file.

⟨*SLHA: public*⟩+≡

```
public :: slha_write_file
```

⟨*SLHA: procedures*⟩+≡

```
subroutine slha_write_file (file, model, input, spectrum, decays)
  type(string_t), intent(in) :: file
  type(model_t), target, intent(in) :: model
  logical, intent(in) :: input, spectrum, decays
  integer :: mssm_type
  integer :: u
  u = free_unit ()
  call msg_message ("Writing SLHA output file '" // char (file) // "'")
  open (unit=u, file=char(file), action="write", status="replace")
  write (u, "(A)")  "# SUSY Les Houches Accord"
  write (u, "(A)")  "# Output generated by " // trim (VERSION_STRING)
  call slha_write_MODSEL (u, model, mssm_type)
  if (input) then
     call slha_write_SMINPUTS (u, model)
     call slha_write_MINPAR (u, model, mssm_type)
  end if
  if (spectrum) then
     call msg_bug ("SLHA: spectrum output not supported yet")
  end if
  if (decays) then
     call msg_bug ("SLHA: decays output not supported yet")
  end if
  close (u)
```

```
    end subroutine slha_write_file
```

28.5.7 Dispatch

⟨*SLHA: public*⟩+≡
```
    public :: dispatch_slha
```

⟨*SLHA: procedures*⟩+≡
```
    subroutine dispatch_slha (var_list, input, spectrum, decays)
      type(var_list_t), intent(inout), target :: var_list
      logical, intent(out) :: input, spectrum, decays
      input = var_list%get_lval (var_str ("?slha_read_input"))
      spectrum = var_list%get_lval (var_str ("?slha_read_spectrum"))
      decays = var_list%get_lval (var_str ("?slha_read_decays"))
    end subroutine dispatch_slha
```

28.5.8 Unit tests

Test module, followed by the corresponding implementation module.

⟨slha_interface_ut.f90⟩≡
```
  ⟨File header⟩

  module slha_interface_ut
    use unit_tests
    use slha_interface_uti

  ⟨Standard module head⟩

  ⟨SLHA: public test⟩

  contains

  ⟨SLHA: test driver⟩

  end module slha_interface_ut
```

⟨slha_interface_uti.f90⟩≡
```
  ⟨File header⟩

  module slha_interface_uti

  ⟨Use strings⟩
    use io_units
    use os_interface
    use parser
    use model_data
    use variables
    use models

    use slha_interface

  ⟨Standard module head⟩
```

```
⟨SLHA: test declarations⟩

contains

⟨SLHA: tests⟩

end module slha_interface_uti
```

API: driver for the unit tests below.

⟨*SLHA: public test*⟩≡
```
public :: slha_test
```

⟨*SLHA: test driver*⟩≡
```
subroutine slha_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨SLHA: execute tests⟩
end subroutine slha_test
```

Checking the basics of the SLHA interface.

⟨*SLHA: execute tests*⟩≡
```
call test (slha_1, "slha_1", &
     "check SLHA interface", &
     u, results)
```

⟨*SLHA: test declarations*⟩≡
```
public :: slha_1
```

⟨*SLHA: tests*⟩≡
```
subroutine slha_1 (u)
  integer, intent(in) :: u
  type(os_data_t), pointer :: os_data => null ()
  type(parse_tree_t), pointer :: parse_tree => null ()
  integer :: u_file, iostat
  character(80) :: buffer
  character(*), parameter :: file_slha = "slha_test.dat"
  type(model_list_t) :: model_list
  type(model_t), pointer :: model => null ()

  write (u, "(A)")  "* Test output: SLHA Interface"
  write (u, "(A)")  "*   Purpose: test SLHA file reading and writing"
  write (u, "(A)")

  write (u, "(A)")  "* Initializing"
  write (u, "(A)")

  allocate (os_data)
  allocate (parse_tree)
  call os_data_init (os_data)
  call syntax_model_file_init ()
  call model_list%read_model &
       (var_str("MSSM"), var_str("MSSM.mdl"), os_data, model)
  call syntax_slha_init ()

  write (u, "(A)")  "* Reading SLHA file sps1ap_decays.slha"
```

```
  write (u, "(A)")

  call slha_parse_file (var_str ("sps1ap_decays.slha"), os_data, parse_tree)

  write (u, "(A)")  "* Writing the parse tree:"
  write (u, "(A)")

  call parse_tree_write (parse_tree, u)

  write (u, "(A)")  "* Interpreting the parse tree"
  write (u, "(A)")

  call slha_interpret_parse_tree (parse_tree, model, &
       input=.true., spectrum=.true., decays=.true.)
  call parse_tree_final (parse_tree)

  write (u, "(A)")  "* Writing out the list of variables (reals only):"
  write (u, "(A)")

  call var_list_write (model%get_var_list_ptr (), &
       only_type = V_REAL, unit = u)

  write (u, "(A)")
  write (u, "(A)")  "* Writing SLHA output to '" // file_slha // "'"
  write (u, "(A)")

  call slha_write_file (var_str (file_slha), model, input=.true., &
       spectrum=.false., decays=.false.)
  u_file = free_unit ()
  open (u_file, file = file_slha, action = "read", status = "old")
  do
     read (u_file, "(A)", iostat = iostat)  buffer
     if (buffer(1:37) == "# Output generated by WHIZARD version") then
        buffer = "[...]"
     end if
     if (iostat /= 0)  exit
     write (u, "(A)") trim (buffer)
  end do
  close (u_file)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"
  write (u, "(A)")

  call parse_tree_final (parse_tree)
  deallocate (parse_tree)
  deallocate (os_data)

  write (u, "(A)")  "* Test output end: slha_1"
  write (u, "(A)")

end subroutine slha_1
```

SLHA interface

This rather trivial sets all input values for the SLHA interface to `false`.

⟨*SLHA: execute tests*⟩+≡
```
    call test (slha_2, "slha_2", &
         "SLHA interface", &
         u, results)
```

⟨*SLHA: test declarations*⟩+≡
```
    public :: slha_2
```

⟨*SLHA: tests*⟩+≡
```
    subroutine slha_2 (u)
      integer, intent(in) :: u
      type(var_list_t) :: var_list
      logical :: input, spectrum, decays

      write (u, "(A)")  "* Test output: slha_2"
      write (u, "(A)")  "*   Purpose: SLHA interface settings"
      write (u, "(A)")

      write (u, "(A)")  "* Default settings"
      write (u, "(A)")

      call var_list%init_defaults (0)
      call dispatch_slha (var_list, &
           input = input, spectrum = spectrum, decays = decays)

      write (u, "(A,1x,L1)")  " slha_read_input     =", input
      write (u, "(A,1x,L1)")  " slha_read_spectrum  =", spectrum
      write (u, "(A,1x,L1)")  " slha_read_decays    =", decays

      call var_list%final ()
      call var_list%init_defaults (0)

      write (u, "(A)")
      write (u, "(A)")  "* Set all entries to [false]"
      write (u, "(A)")

      call var_list%set_log (var_str ("?slha_read_input"), &
           .false., is_known = .true.)
      call var_list%set_log (var_str ("?slha_read_spectrum"), &
           .false., is_known = .true.)
      call var_list%set_log (var_str ("?slha_read_decays"), &
           .false., is_known = .true.)

      call dispatch_slha (var_list, &
           input = input, spectrum = spectrum, decays = decays)

      write (u, "(A,1x,L1)")  " slha_read_input     =", input
      write (u, "(A,1x,L1)")  " slha_read_spectrum  =", spectrum
      write (u, "(A,1x,L1)")  " slha_read_decays    =", decays

      call var_list%final ()
```

```
  write (u, "(A)")
  write (u, "(A)")  "* Test output end: slha_2"

end subroutine slha_2
```

Chapter 29

Infrastructure for threshold processes

⟨interpolation.f90⟩≡
⟨*File header*⟩

```
module interpolation
  use kinds
  implicit none
  save
  private

  public :: interpolate_linear, strictly_monotonous

  interface interpolate_linear
    module  procedure interpolate_linear_1D_complex_array, &
      interpolate_linear_1D_complex_scalar, &
      interpolate_linear_1D_real_array, &
      interpolate_linear_1D_real_scalar, &
      interpolate_linear_2D_complex_array, &
      interpolate_linear_2D_complex_scalar, &
      interpolate_linear_2D_real_array, &
      interpolate_linear_2D_real_scalar, &
      interpolate_linear_3D_complex_array, &
      interpolate_linear_3D_complex_scalar, &
      interpolate_linear_3D_real_array, &
      interpolate_linear_3D_real_scalar
  end interface

  interface strictly_monotonous
    module procedure monotonous
  end interface strictly_monotonous

  interface find_nearest_left
    !!! recursive bisection is slower
    module procedure find_nearest_left_loop
  end interface find_nearest_left
```

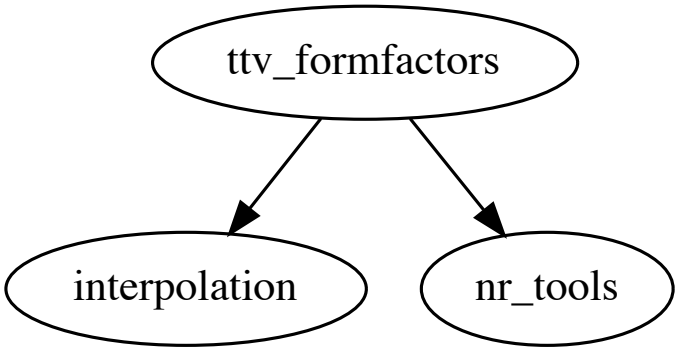

Figure 29.1: Module dependencies in `src/threshold`.

```
contains

  pure subroutine interpolate_linear_1D_complex_scalar (xa, ya, x, y)
    real(default), dimension(:), intent(in) :: xa
    complex(default), dimension(:), intent(in) :: ya
    real(default), intent(in) :: x
    complex(default), intent(out) :: y
    integer :: ixl
    real(default) :: t
    y = 0.0_default
    !!! don't check this at runtime:
    ! if ( .not.monotonous(xa) ) return
    if ( out_of_range(xa, x) ) return
    ixl = 0
    call find_nearest_left (xa, x, ixl)
    t = ( x - xa(ixl) ) / ( xa(ixl+1) - xa(ixl) )
    y = (1.-t)*ya(ixl) + t*ya(ixl+1)
  end subroutine interpolate_linear_1D_complex_scalar

  pure subroutine interpolate_linear_2D_complex_scalar (x1a, x2a, ya, x1, x2, y)
    real(default), dimension(:), intent(in) :: x1a
    real(default), dimension(:), intent(in) :: x2a
    complex(default), dimension(:,:), intent(in) :: ya
    real(default), intent(in) :: x1
    real(default), intent(in) :: x2
    complex(default), intent(out) :: y
    integer :: ix1l, ix2l
    real(default) :: t, u
    y = 0.0_default
    !!! don't check this at runtime:
```

```
  ! if ( (.not.monotonous(x1a)) .or. (.not.monotonous(x2a)) ) return
  if ( out_of_range(x1a, x1) .or. out_of_range(x2a, x2) ) return
  ix1l = 0
  call find_nearest_left (x1a, x1, ix1l)
  ix2l = 0
  call find_nearest_left (x2a, x2, ix2l)
  t = ( x1 - x1a(ix1l) ) / ( x1a(ix1l+1) - x1a(ix1l) )
  u = ( x2 - x2a(ix2l) ) / ( x2a(ix2l+1) - x2a(ix2l) )
  y =  (1.-t)*(1.-u)*ya(ix1l  ,ix2l  ) &
      +    t *(1.-u)*ya(ix1l+1,ix2l  ) &
      +    t *    u *ya(ix1l+1,ix2l+1) &
      +(1.-t)*    u *ya(ix1l  ,ix2l+1)
end subroutine interpolate_linear_2D_complex_scalar

pure subroutine interpolate_linear_3D_complex_scalar (x1a, x2a, x3a, ya, x1, x2, x3, y)
  real(default), dimension(:), intent(in) :: x1a
  real(default), dimension(:), intent(in) :: x2a
  real(default), dimension(:), intent(in) :: x3a
  complex(default), dimension(:,:,:), intent(in) :: ya
  real(default), intent(in) :: x1
  real(default), intent(in) :: x2
  real(default), intent(in) :: x3
  complex(default), intent(out) :: y
  integer :: ix1l, ix2l, ix3l
  real(default) :: t, u, v
  y = 0.0_default
  !!! don't check this at runtime:
  ! if ( (.not.monotonous(x1a)) .or. (.not.monotonous(x2a)) ) return
  if ( out_of_range(x1a, x1) .or. out_of_range(x2a, x2) .or. out_of_range(x3a, x3) ) return
  ix1l = 0
  call find_nearest_left (x1a, x1, ix1l)
  ix2l = 0
  call find_nearest_left (x2a, x2, ix2l)
  ix3l = 0
  call find_nearest_left (x3a, x3, ix3l)
  t = ( x1 - x1a(ix1l) ) / ( x1a(ix1l+1) - x1a(ix1l) )
  u = ( x2 - x2a(ix2l) ) / ( x2a(ix2l+1) - x2a(ix2l) )
  v = ( x3 - x3a(ix3l) ) / ( x3a(ix3l+1) - x3a(ix3l) )
  y =  (1.-t)*(1.-u)*(1.-v)*ya(ix1l  ,ix2l  ,ix3l  ) &
      +(1.-t)*(1.-u)*    v *ya(ix1l  ,ix2l  ,ix3l+1) &
      +(1.-t)*    u *(1.-v)*ya(ix1l  ,ix2l+1,ix3l  ) &
      +(1.-t)*    u *    v *ya(ix1l  ,ix2l+1,ix3l+1) &
      +    t *(1.-u)*(1.-v)*ya(ix1l+1,ix2l  ,ix3l  ) &
      +    t *(1.-u)*    v *ya(ix1l+1,ix2l  ,ix3l+1) &
      +    t *    u *(1.-v)*ya(ix1l+1,ix2l+1,ix3l  ) &
      +    t *    u *    v *ya(ix1l+1,ix2l+1,ix3l+1)
end subroutine interpolate_linear_3D_complex_scalar

pure subroutine find_nearest_left_loop (xa, x, ixl)
  real(default), dimension(:), intent(in) :: xa
  real(default), intent(in) :: x
  integer, intent(out) :: ixl
  integer :: ixm, ixr
  ixl = 1
```

```
  ixr = size(xa)
  do
    if ( ixr-ixl <= 1 ) return
    ixm = (ixr+ixl) / 2
    if ( x < xa(ixm) ) then
      ixr = ixm
    else
      ixl = ixm
    end if
  end do
end subroutine find_nearest_left_loop

pure recursive subroutine find_nearest_left_rec (xa, x, ixl)
  real(default), dimension(:), intent(in) :: xa
  real(default), intent(in) :: x
  integer, intent(inout) :: ixl
  integer :: nx, bs
  real(default), dimension(:), allocatable :: xa_new
  nx = size(xa)
  bs = nx/2 + 1
  if ( nx < 3 ) then
    ixl = ixl + bs - 1
    return
  else
    if ( x < xa(bs) ) then
      allocate( xa_new(1:bs) )
      xa_new = xa(1:bs)
    else
      ixl = ixl + bs - 1
      allocate( xa_new(bs:nx) )
      xa_new = xa(bs:nx)
    end if
    call find_nearest_left_rec (xa_new, x, ixl)
    deallocate( xa_new )
  end if
end subroutine find_nearest_left_rec

pure function monotonous (xa) result (flag)
  real(default), dimension(:), intent(in) :: xa
  integer :: ix
  logical :: flag
  flag = .false.
  do ix = 1, size(xa)-1
    flag = ( xa(ix) < xa(ix+1) )
    if ( .not. flag ) return
  end do
end function monotonous

pure function out_of_range (xa, x) result (flag)
  real(default), dimension(:), intent(in) :: xa
  real(default), intent(in) :: x
  logical :: flag
  flag = ( x < xa(1) .or. x > xa(size(xa)) )
end function out_of_range
```

```
pure subroutine interpolate_linear_1D_complex_array (xa, ya, x, y)
  real(default), dimension(:), intent(in) :: xa
  complex(default), dimension(:,:), intent(in) :: ya
  real(default), intent(in) :: x
  complex(default), dimension(size(ya(1,:))), intent(out) :: y
  integer :: iy
  do iy=1, size(y)
    call interpolate_linear_1D_complex_scalar (xa, ya(:,iy), x, y(iy))
  end do
end subroutine interpolate_linear_1D_complex_array

pure subroutine interpolate_linear_1D_real_array (xa, ya, x, y)
  real(default), dimension(:), intent(in) :: xa
  real(default), dimension(:,:), intent(in) :: ya
  real(default), intent(in) :: x
  real(default), dimension(:), intent(out) :: y
  complex(default), dimension(size(ya(1,:))) :: y_c
  call interpolate_linear_1D_complex_array (xa, cmplx(ya,kind=default), x, y_c)
  y = real(y_c,kind=default)
end subroutine interpolate_linear_1D_real_array

pure subroutine interpolate_linear_1D_real_scalar (xa, ya, x, y)
  real(default), dimension(:), intent(in) :: xa
  real(default), dimension(:), intent(in) :: ya
  real(default), intent(in) :: x
  real(default), intent(out) :: y
  complex(default), dimension(size(ya)) :: ya_c
  complex(default) :: y_c
  ya_c = cmplx(ya,kind=default)
  call interpolate_linear_1D_complex_scalar (xa, ya_c, x, y_c)
  y = real(y_c,kind=default)
end subroutine interpolate_linear_1D_real_scalar

pure subroutine interpolate_linear_2D_complex_array (x1a, x2a, ya, x1, x2, y)
  real(default), dimension(:), intent(in) :: x1a
  real(default), dimension(:), intent(in) :: x2a
  complex(default), dimension(:,:,:), intent(in) :: ya
  real(default), intent(in) :: x1
  real(default), intent(in) :: x2
  complex(default), dimension(size(ya(1,1,:))), intent(out) :: y
  integer :: iy
  do iy=1, size(y)
    call interpolate_linear_2D_complex_scalar (x1a, x2a, ya(:,:,iy), x1, x2, y(iy))
  end do
end subroutine interpolate_linear_2D_complex_array

pure subroutine interpolate_linear_2D_real_array (x1a, x2a, ya, x1, x2, y)
  real(default), dimension(:), intent(in) :: x1a
  real(default), dimension(:), intent(in) :: x2a
  real(default), dimension(:,:,:), intent(in) :: ya
  real(default), intent(in) :: x1
  real(default), intent(in) :: x2
  real(default), dimension(:), intent(out) :: y
```

```
  complex(default), dimension(size(ya(1,1,:))) :: y_c
  call interpolate_linear_2D_complex_array (x1a, x2a, cmplx(ya,kind=default), x1, x2, y_c)
  y = real(y_c,kind=default)
end subroutine interpolate_linear_2D_real_array

pure subroutine interpolate_linear_2D_real_scalar (x1a, x2a, ya, x1, x2, y)
  real(default), dimension(:), intent(in) :: x1a
  real(default), dimension(:), intent(in) :: x2a
  real(default), dimension(:,:), intent(in) :: ya
  real(default), intent(in) :: x1
  real(default), intent(in) :: x2
  real(default), intent(out) :: y
  complex(default), dimension(size(ya(:,1)),size(ya(1,:))) :: ya_c
  complex(default) :: y_c
  ya_c = reshape (ya_c, shape(ya))
  ya_c = cmplx(ya,kind=default)
  call interpolate_linear_2D_complex_scalar (x1a, x2a, ya_c, x1, x2, y_c)
  y = real(y_c,kind=default)
end subroutine interpolate_linear_2D_real_scalar

pure subroutine interpolate_linear_3D_complex_array (x1a, x2a, x3a, ya, x1, x2, x3, y)
  real(default), dimension(:), intent(in) :: x1a
  real(default), dimension(:), intent(in) :: x2a
  real(default), dimension(:), intent(in) :: x3a
  complex(default), dimension(:,:,:,:), intent(in) :: ya
  real(default), intent(in) :: x1
  real(default), intent(in) :: x2
  real(default), intent(in) :: x3
  complex(default), dimension(size(ya(1,1,1,:))), intent(out) :: y
  integer :: iy
  do iy=1, size(y)
    call interpolate_linear_3D_complex_scalar &
                        (x1a, x2a, x3a, ya(:,:,:,iy), x1, x2, x3, y(iy))
  end do
end subroutine interpolate_linear_3D_complex_array

pure subroutine interpolate_linear_3D_real_array (x1a, x2a, x3a, ya, x1, x2, x3, y)
  real(default), dimension(:), intent(in) :: x1a
  real(default), dimension(:), intent(in) :: x2a
  real(default), dimension(:), intent(in) :: x3a
  real(default), dimension(:,:,:,:), intent(in) :: ya
  real(default), intent(in) :: x1
  real(default), intent(in) :: x2
  real(default), intent(in) :: x3
  real(default), dimension(:), intent(out) :: y
  complex(default), dimension(size(ya(1,1,1,:))) :: y_c
  call interpolate_linear_3D_complex_array &
                    (x1a, x2a, x3a, cmplx(ya,kind=default), x1, x2, x3, y_c)
  y = real(y_c,kind=default)
end subroutine interpolate_linear_3D_real_array

pure subroutine interpolate_linear_3D_real_scalar (x1a, x2a, x3a, ya, x1, x2, x3, y)
  real(default), dimension(:), intent(in) :: x1a
  real(default), dimension(:), intent(in) :: x2a
```

```
      real(default), dimension(:), intent(in) :: x3a
      real(default), dimension(:,:,:), intent(in) :: ya
      real(default), intent(in) :: x1
      real(default), intent(in) :: x2
      real(default), intent(in) :: x3
      real(default), intent(out) :: y
      complex(default), dimension(size(ya(:,1,1)),size(ya(1,:,1)),size(ya(1,1,:))) :: ya_c
      complex(default) :: y_c
      ya_c = cmplx(ya,kind=default)
      call interpolate_linear_3D_complex_scalar (x1a, x2a, x3a, ya_c, x1, x2, x3, y_c)
      y = real(y_c,kind=default)
    end subroutine interpolate_linear_3D_real_scalar
  end module interpolation
```

⟨nr_tools.f90⟩≡

```
  !!!!
```

```
  REAL(DP), PARAMETER :: PIO2_D=1.5707963267948966192313216916397514420985 8_dp
  REAL(DP), PARAMETER :: TWOPI_D=6.2831853071795864769252867665590057683 94_dp
  TYPE sprs2_sp
    INTEGER(I4B) :: n,len
    REAL(SP), DIMENSION(:), POINTER :: val
    INTEGER(I4B), DIMENSION(:), POINTER :: irow
    INTEGER(I4B), DIMENSION(:), POINTER :: jcol
  END TYPE sprs2_sp
  TYPE sprs2_dp
    INTEGER(I4B) :: n,len
    REAL(DP), DIMENSION(:), POINTER :: val
    INTEGER(I4B), DIMENSION(:), POINTER :: irow
    INTEGER(I4B), DIMENSION(:), POINTER :: jcol
  END TYPE sprs2_dp
END MODULE nrtype

MODULE nrutil
  USE nrtype
  IMPLICIT NONE
  INTEGER(I4B), PARAMETER :: NPAR_ARTH=16,NPAR2_ARTH=8
  INTEGER(I4B), PARAMETER :: NPAR_GEOP=4,NPAR2_GEOP=2
  INTEGER(I4B), PARAMETER :: NPAR_CUMSUM=16
  INTEGER(I4B), PARAMETER :: NPAR_CUMPROD=8
  INTEGER(I4B), PARAMETER :: NPAR_POLY=8
  INTEGER(I4B), PARAMETER :: NPAR_POLYTERM=8
  INTERFACE array_copy
    MODULE PROCEDURE array_copy_r, array_copy_d, array_copy_i
  END INTERFACE
  INTERFACE swap
    MODULE PROCEDURE swap_i,swap_r,swap_rv,swap_c, &
      swap_cv,swap_cm,swap_z,swap_zv,swap_zm, &
      masked_swap_rs,masked_swap_rv,masked_swap_rm
  END INTERFACE
  INTERFACE reallocate
    MODULE PROCEDURE reallocate_rv,reallocate_rm,&
      reallocate_iv,reallocate_im,reallocate_hv
  END INTERFACE
  INTERFACE imaxloc
    MODULE PROCEDURE imaxloc_r,imaxloc_i
  END INTERFACE
  INTERFACE assert
    MODULE PROCEDURE assert1,assert2,assert3,assert4,assert_v
  END INTERFACE
  INTERFACE assert_eq
    MODULE PROCEDURE assert_eq2,assert_eq3,assert_eq4,assert_eqn
  END INTERFACE
  INTERFACE arth
    MODULE PROCEDURE arth_r, arth_d, arth_i
  END INTERFACE
  INTERFACE geop
    MODULE PROCEDURE geop_r, geop_d, geop_i, geop_c, geop_dv
  END INTERFACE
  INTERFACE cumsum
    MODULE PROCEDURE cumsum_r,cumsum_i
```

```
  END INTERFACE
  INTERFACE poly
    MODULE PROCEDURE poly_rr,poly_rrv,poly_dd,poly_ddv,&
      poly_rc,poly_cc,poly_msk_rrv,poly_msk_ddv
  END INTERFACE
  INTERFACE poly_term
    MODULE PROCEDURE poly_term_rr,poly_term_cc
  END INTERFACE
  INTERFACE outerprod
    MODULE PROCEDURE outerprod_r,outerprod_d
  END INTERFACE
  INTERFACE outerdiff
    MODULE PROCEDURE outerdiff_r,outerdiff_d,outerdiff_i
  END INTERFACE
  INTERFACE scatter_add
    MODULE PROCEDURE scatter_add_r,scatter_add_d
  END INTERFACE
  INTERFACE scatter_max
    MODULE PROCEDURE scatter_max_r,scatter_max_d
  END INTERFACE
  INTERFACE diagadd
    MODULE PROCEDURE diagadd_rv,diagadd_r
  END INTERFACE
  INTERFACE diagmult
    MODULE PROCEDURE diagmult_rv,diagmult_r
  END INTERFACE
  INTERFACE get_diag
    MODULE PROCEDURE get_diag_rv, get_diag_dv
  END INTERFACE
  INTERFACE put_diag
    MODULE PROCEDURE put_diag_rv, put_diag_r
  END INTERFACE
CONTAINS
!BL
  SUBROUTINE array_copy_r(src,dest,n_copied,n_not_copied)
  REAL(SP), DIMENSION(:), INTENT(IN) :: src
  REAL(SP), DIMENSION(:), INTENT(OUT) :: dest
  INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
  n_copied=min(size(src),size(dest))
  n_not_copied=size(src)-n_copied
  dest(1:n_copied)=src(1:n_copied)
  END SUBROUTINE array_copy_r
!BL
  SUBROUTINE array_copy_d(src,dest,n_copied,n_not_copied)
  REAL(DP), DIMENSION(:), INTENT(IN) :: src
  REAL(DP), DIMENSION(:), INTENT(OUT) :: dest
  INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
  n_copied=min(size(src),size(dest))
  n_not_copied=size(src)-n_copied
  dest(1:n_copied)=src(1:n_copied)
  END SUBROUTINE array_copy_d
!BL
  SUBROUTINE array_copy_i(src,dest,n_copied,n_not_copied)
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: src
```

```
  INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: dest
  INTEGER(I4B), INTENT(OUT) :: n_copied, n_not_copied
  n_copied=min(size(src),size(dest))
  n_not_copied=size(src)-n_copied
  dest(1:n_copied)=src(1:n_copied)
  END SUBROUTINE array_copy_i
!BL
!BL
  SUBROUTINE swap_i(a,b)
  INTEGER(I4B), INTENT(INOUT) :: a,b
  INTEGER(I4B) :: dum
  dum=a
  a=b
  b=dum
  END SUBROUTINE swap_i
!BL
  SUBROUTINE swap_r(a,b)
  REAL(SP), INTENT(INOUT) :: a,b
  REAL(SP) :: dum
  dum=a
  a=b
  b=dum
  END SUBROUTINE swap_r
!BL
  SUBROUTINE swap_rv(a,b)
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: a,b
  REAL(SP), DIMENSION(SIZE(a)) :: dum
  dum=a
  a=b
  b=dum
  END SUBROUTINE swap_rv
!BL
  SUBROUTINE swap_c(a,b)
  COMPLEX(SPC), INTENT(INOUT) :: a,b
  COMPLEX(SPC) :: dum
  dum=a
  a=b
  b=dum
  END SUBROUTINE swap_c
!BL
  SUBROUTINE swap_cv(a,b)
  COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: a,b
  COMPLEX(SPC), DIMENSION(SIZE(a)) :: dum
  dum=a
  a=b
  b=dum
  END SUBROUTINE swap_cv
!BL
  SUBROUTINE swap_cm(a,b)
  COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: a,b
  COMPLEX(SPC), DIMENSION(size(a,1),size(a,2)) :: dum
  dum=a
  a=b
  b=dum
```

```
  END SUBROUTINE swap_cm
!BL
  SUBROUTINE swap_z(a,b)
  COMPLEX(DPC), INTENT(INOUT) :: a,b
  COMPLEX(DPC) :: dum
  dum=a
  a=b
  b=dum
  END SUBROUTINE swap_z
!BL
  SUBROUTINE swap_zv(a,b)
  COMPLEX(DPC), DIMENSION(:), INTENT(INOUT) :: a,b
  COMPLEX(DPC), DIMENSION(SIZE(a)) :: dum
  dum=a
  a=b
  b=dum
  END SUBROUTINE swap_zv
!BL
  SUBROUTINE swap_zm(a,b)
  COMPLEX(DPC), DIMENSION(:,:), INTENT(INOUT) :: a,b
  COMPLEX(DPC), DIMENSION(size(a,1),size(a,2)) :: dum
  dum=a
  a=b
  b=dum
  END SUBROUTINE swap_zm
!BL
  SUBROUTINE masked_swap_rs(a,b,mask)
  REAL(SP), INTENT(INOUT) :: a,b
  LOGICAL(LGT), INTENT(IN) :: mask
  REAL(SP) :: swp
  if (mask) then
    swp=a
    a=b
    b=swp
  end if
  END SUBROUTINE masked_swap_rs
!BL
  SUBROUTINE masked_swap_rv(a,b,mask)
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: a,b
  LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: mask
  REAL(SP), DIMENSION(size(a)) :: swp
  where (mask)
    swp=a
    a=b
    b=swp
  end where
  END SUBROUTINE masked_swap_rv
!BL
  SUBROUTINE masked_swap_rm(a,b,mask)
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a,b
  LOGICAL(LGT), DIMENSION(:,:), INTENT(IN) :: mask
  REAL(SP), DIMENSION(size(a,1),size(a,2)) :: swp
  where (mask)
    swp=a
```

```
    a=b
    b=swp
  end where
  END SUBROUTINE masked_swap_rm
!BL
!BL
  FUNCTION reallocate_rv(p,n)
  REAL(SP), DIMENSION(:), POINTER :: p, reallocate_rv
  INTEGER(I4B), INTENT(IN) :: n
  INTEGER(I4B) :: nold,ierr
  allocate(reallocate_rv(n),stat=ierr)
  if (ierr /= 0) call &
    nrerror('reallocate_rv: problem in attempt to allocate memory')
  if (.not. associated(p)) RETURN
  nold=size(p)
  reallocate_rv(1:min(nold,n))=p(1:min(nold,n))
  deallocate(p)
  END FUNCTION reallocate_rv
!BL
  FUNCTION reallocate_iv(p,n)
  INTEGER(I4B), DIMENSION(:), POINTER :: p, reallocate_iv
  INTEGER(I4B), INTENT(IN) :: n
  INTEGER(I4B) :: nold,ierr
  allocate(reallocate_iv(n),stat=ierr)
  if (ierr /= 0) call &
    nrerror('reallocate_iv: problem in attempt to allocate memory')
  if (.not. associated(p)) RETURN
  nold=size(p)
  reallocate_iv(1:min(nold,n))=p(1:min(nold,n))
  deallocate(p)
  END FUNCTION reallocate_iv
!BL
  FUNCTION reallocate_hv(p,n)
  CHARACTER(1), DIMENSION(:), POINTER :: p, reallocate_hv
  INTEGER(I4B), INTENT(IN) :: n
  INTEGER(I4B) :: nold,ierr
  allocate(reallocate_hv(n),stat=ierr)
  if (ierr /= 0) call &
    nrerror('reallocate_hv: problem in attempt to allocate memory')
  if (.not. associated(p)) RETURN
  nold=size(p)
  reallocate_hv(1:min(nold,n))=p(1:min(nold,n))
  deallocate(p)
  END FUNCTION reallocate_hv
!BL
  FUNCTION reallocate_rm(p,n,m)
  REAL(SP), DIMENSION(:,:), POINTER :: p, reallocate_rm
  INTEGER(I4B), INTENT(IN) :: n,m
  INTEGER(I4B) :: nold,mold,ierr
  allocate(reallocate_rm(n,m),stat=ierr)
  if (ierr /= 0) call &
    nrerror('reallocate_rm: problem in attempt to allocate memory')
  if (.not. associated(p)) RETURN
  nold=size(p,1)
```

```
  mold=size(p,2)
  reallocate_rm(1:min(nold,n),1:min(mold,m))=&
    p(1:min(nold,n),1:min(mold,m))
  deallocate(p)
  END FUNCTION reallocate_rm
!BL
  FUNCTION reallocate_im(p,n,m)
  INTEGER(I4B), DIMENSION(:,:), POINTER :: p, reallocate_im
  INTEGER(I4B), INTENT(IN) :: n,m
  INTEGER(I4B) :: nold,mold,ierr
  allocate(reallocate_im(n,m),stat=ierr)
  if (ierr /= 0) call &
    nrerror('reallocate_im: problem in attempt to allocate memory')
  if (.not. associated(p)) RETURN
  nold=size(p,1)
  mold=size(p,2)
  reallocate_im(1:min(nold,n),1:min(mold,m))=&
    p(1:min(nold,n),1:min(mold,m))
  deallocate(p)
  END FUNCTION reallocate_im
!BL
  FUNCTION ifirstloc(mask)
  LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: mask
  INTEGER(I4B) :: ifirstloc
  INTEGER(I4B), DIMENSION(1) :: loc
  loc=maxloc(merge(1,0,mask))
  ifirstloc=loc(1)
  if (.not. mask(ifirstloc)) ifirstloc=size(mask)+1
  END FUNCTION ifirstloc
!BL
  FUNCTION imaxloc_r(arr)
  REAL(SP), DIMENSION(:), INTENT(IN) :: arr
  INTEGER(I4B) :: imaxloc_r
  INTEGER(I4B), DIMENSION(1) :: imax
  imax=maxloc(arr(:))
  imaxloc_r=imax(1)
  END FUNCTION imaxloc_r
!BL
  FUNCTION imaxloc_i(iarr)
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iarr
  INTEGER(I4B), DIMENSION(1) :: imax
  INTEGER(I4B) :: imaxloc_i
  imax=maxloc(iarr(:))
  imaxloc_i=imax(1)
  END FUNCTION imaxloc_i
!BL
  FUNCTION iminloc(arr)
  REAL(SP), DIMENSION(:), INTENT(IN) :: arr
  INTEGER(I4B), DIMENSION(1) :: imin
  INTEGER(I4B) :: iminloc
  imin=minloc(arr(:))
  iminloc=imin(1)
  END FUNCTION iminloc
!BL
```

```
  SUBROUTINE assert1(n1,string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  LOGICAL, INTENT(IN) :: n1
  if (.not. n1) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
      string
    STOP 'program terminated by assert1'
  end if
  END SUBROUTINE assert1
!BL
  SUBROUTINE assert2(n1,n2,string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  LOGICAL, INTENT(IN) :: n1,n2
  if (.not. (n1 .and. n2)) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
      string
    STOP 'program terminated by assert2'
  end if
  END SUBROUTINE assert2
!BL
  SUBROUTINE assert3(n1,n2,n3,string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  LOGICAL, INTENT(IN) :: n1,n2,n3
  if (.not. (n1 .and. n2 .and. n3)) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
      string
    STOP 'program terminated by assert3'
  end if
  END SUBROUTINE assert3
!BL
  SUBROUTINE assert4(n1,n2,n3,n4,string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  LOGICAL, INTENT(IN) :: n1,n2,n3,n4
  if (.not. (n1 .and. n2 .and. n3 .and. n4)) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
      string
    STOP 'program terminated by assert4'
  end if
  END SUBROUTINE assert4
!BL
  SUBROUTINE assert_v(n,string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  LOGICAL, DIMENSION(:), INTENT(IN) :: n
  if (.not. all(n)) then
    write (*,*) 'nrerror: an assertion failed with this tag:', &
      string
    STOP 'program terminated by assert_v'
  end if
  END SUBROUTINE assert_v
!BL
  FUNCTION assert_eq2(n1,n2,string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  INTEGER, INTENT(IN) :: n1,n2
  INTEGER :: assert_eq2
```

```
  if (n1 == n2) then
    assert_eq2=n1
  else
    write (*,*) 'nrerror: an assert_eq failed with this tag:', &
      string
    STOP 'program terminated by assert_eq2'
  end if
  END FUNCTION assert_eq2
!BL
  FUNCTION assert_eq3(n1,n2,n3,string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  INTEGER, INTENT(IN) :: n1,n2,n3
  INTEGER :: assert_eq3
  if (n1 == n2 .and. n2 == n3) then
    assert_eq3=n1
  else
    write (*,*) 'nrerror: an assert_eq failed with this tag:', &
      string
    STOP 'program terminated by assert_eq3'
  end if
  END FUNCTION assert_eq3
!BL
  FUNCTION assert_eq4(n1,n2,n3,n4,string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  INTEGER, INTENT(IN) :: n1,n2,n3,n4
  INTEGER :: assert_eq4
  if (n1 == n2 .and. n2 == n3 .and. n3 == n4) then
    assert_eq4=n1
  else
    write (*,*) 'nrerror: an assert_eq failed with this tag:', &
      string
    STOP 'program terminated by assert_eq4'
  end if
  END FUNCTION assert_eq4
!BL
  FUNCTION assert_eqn(nn,string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  INTEGER, DIMENSION(:), INTENT(IN) :: nn
  INTEGER :: assert_eqn
  if (all(nn(2:) == nn(1))) then
    assert_eqn=nn(1)
  else
    write (*,*) 'nrerror: an assert_eq failed with this tag:', &
      string
    STOP 'program terminated by assert_eqn'
  end if
  END FUNCTION assert_eqn
!BL
  SUBROUTINE nrerror(string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  write (*,*) 'nrerror: ',string
  STOP 'program terminated by nrerror'
  END SUBROUTINE nrerror
!BL
```

```
  FUNCTION arth_r(first,increment,n)
  REAL(SP), INTENT(IN) :: first,increment
  INTEGER(I4B), INTENT(IN) :: n
  REAL(SP), DIMENSION(n) :: arth_r
  INTEGER(I4B) :: k,k2
  REAL(SP) :: temp
  if (n > 0) arth_r(1)=first
  if (n <= NPAR_ARTH) then
    do k=2,n
      arth_r(k)=arth_r(k-1)+increment
    end do
  else
    do k=2,NPAR2_ARTH
      arth_r(k)=arth_r(k-1)+increment
    end do
    temp=increment*NPAR2_ARTH
    k=NPAR2_ARTH
    do
      if (k >= n) exit
      k2=k+k
      arth_r(k+1:min(k2,n))=temp+arth_r(1:min(k,n-k))
      temp=temp+temp
      k=k2
    end do
  end if
  END FUNCTION arth_r
!BL
  FUNCTION arth_d(first,increment,n)
  REAL(DP), INTENT(IN) :: first,increment
  INTEGER(I4B), INTENT(IN) :: n
  REAL(DP), DIMENSION(n) :: arth_d
  INTEGER(I4B) :: k,k2
  REAL(DP) :: temp
  if (n > 0) arth_d(1)=first
  if (n <= NPAR_ARTH) then
    do k=2,n
      arth_d(k)=arth_d(k-1)+increment
    end do
  else
    do k=2,NPAR2_ARTH
      arth_d(k)=arth_d(k-1)+increment
    end do
    temp=increment*NPAR2_ARTH
    k=NPAR2_ARTH
    do
      if (k >= n) exit
      k2=k+k
      arth_d(k+1:min(k2,n))=temp+arth_d(1:min(k,n-k))
      temp=temp+temp
      k=k2
    end do
  end if
  END FUNCTION arth_d
!BL
```

```
  FUNCTION arth_i(first,increment,n)
  INTEGER(I4B), INTENT(IN) :: first,increment,n
  INTEGER(I4B), DIMENSION(n) :: arth_i
  INTEGER(I4B) :: k,k2,temp
  if (n > 0) arth_i(1)=first
  if (n <= NPAR_ARTH) then
    do k=2,n
      arth_i(k)=arth_i(k-1)+increment
    end do
  else
    do k=2,NPAR2_ARTH
      arth_i(k)=arth_i(k-1)+increment
    end do
    temp=increment*NPAR2_ARTH
    k=NPAR2_ARTH
    do
      if (k >= n) exit
      k2=k+k
      arth_i(k+1:min(k2,n))=temp+arth_i(1:min(k,n-k))
      temp=temp+temp
      k=k2
    end do
  end if
  END FUNCTION arth_i
!BL
!BL
  FUNCTION geop_r(first,factor,n)
  REAL(SP), INTENT(IN) :: first,factor
  INTEGER(I4B), INTENT(IN) :: n
  REAL(SP), DIMENSION(n) :: geop_r
  INTEGER(I4B) :: k,k2
  REAL(SP) :: temp
  if (n > 0) geop_r(1)=first
  if (n <= NPAR_GEOP) then
    do k=2,n
      geop_r(k)=geop_r(k-1)*factor
    end do
  else
    do k=2,NPAR2_GEOP
      geop_r(k)=geop_r(k-1)*factor
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
      if (k >= n) exit
      k2=k+k
      geop_r(k+1:min(k2,n))=temp*geop_r(1:min(k,n-k))
      temp=temp*temp
      k=k2
    end do
  end if
  END FUNCTION geop_r
!BL
  FUNCTION geop_d(first,factor,n)
```

```
  REAL(DP), INTENT(IN) :: first,factor
  INTEGER(I4B), INTENT(IN) :: n
  REAL(DP), DIMENSION(n) :: geop_d
  INTEGER(I4B) :: k,k2
  REAL(DP) :: temp
  if (n > 0) geop_d(1)=first
  if (n <= NPAR_GEOP) then
    do k=2,n
      geop_d(k)=geop_d(k-1)*factor
    end do
  else
    do k=2,NPAR2_GEOP
      geop_d(k)=geop_d(k-1)*factor
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
      if (k >= n) exit
      k2=k+k
      geop_d(k+1:min(k2,n))=temp*geop_d(1:min(k,n-k))
      temp=temp*temp
      k=k2
    end do
  end if
  END FUNCTION geop_d
!BL
  FUNCTION geop_i(first,factor,n)
  INTEGER(I4B), INTENT(IN) :: first,factor,n
  INTEGER(I4B), DIMENSION(n) :: geop_i
  INTEGER(I4B) :: k,k2,temp
  if (n > 0) geop_i(1)=first
  if (n <= NPAR_GEOP) then
    do k=2,n
      geop_i(k)=geop_i(k-1)*factor
    end do
  else
    do k=2,NPAR2_GEOP
      geop_i(k)=geop_i(k-1)*factor
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
      if (k >= n) exit
      k2=k+k
      geop_i(k+1:min(k2,n))=temp*geop_i(1:min(k,n-k))
      temp=temp*temp
      k=k2
    end do
  end if
  END FUNCTION geop_i
!BL
  FUNCTION geop_c(first,factor,n)
  COMPLEX(SP), INTENT(IN) :: first,factor
  INTEGER(I4B), INTENT(IN) :: n
```

```
  COMPLEX(SP), DIMENSION(n) :: geop_c
  INTEGER(I4B) :: k,k2
  COMPLEX(SP) :: temp
  if (n > 0) geop_c(1)=first
  if (n <= NPAR_GEOP) then
    do k=2,n
      geop_c(k)=geop_c(k-1)*factor
    end do
  else
    do k=2,NPAR2_GEOP
      geop_c(k)=geop_c(k-1)*factor
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
      if (k >= n) exit
      k2=k+k
      geop_c(k+1:min(k2,n))=temp*geop_c(1:min(k,n-k))
      temp=temp*temp
      k=k2
    end do
  end if
  END FUNCTION geop_c
!BL
  FUNCTION geop_dv(first,factor,n)
  REAL(DP), DIMENSION(:), INTENT(IN) :: first,factor
  INTEGER(I4B), INTENT(IN) :: n
  REAL(DP), DIMENSION(size(first),n) :: geop_dv
  INTEGER(I4B) :: k,k2
  REAL(DP), DIMENSION(size(first)) :: temp
  if (n > 0) geop_dv(:,1)=first(:)
  if (n <= NPAR_GEOP) then
    do k=2,n
      geop_dv(:,k)=geop_dv(:,k-1)*factor(:)
    end do
  else
    do k=2,NPAR2_GEOP
      geop_dv(:,k)=geop_dv(:,k-1)*factor(:)
    end do
    temp=factor**NPAR2_GEOP
    k=NPAR2_GEOP
    do
      if (k >= n) exit
      k2=k+k
      geop_dv(:,k+1:min(k2,n))=geop_dv(:,1:min(k,n-k))*&
        spread(temp,2,size(geop_dv(:,1:min(k,n-k)),2))
      temp=temp*temp
      k=k2
    end do
  end if
  END FUNCTION geop_dv
!BL
!BL
  RECURSIVE FUNCTION cumsum_r(arr,seed) RESULT(ans)
```

```
  REAL(SP), DIMENSION(:), INTENT(IN) :: arr
  REAL(SP), OPTIONAL, INTENT(IN) :: seed
  REAL(SP), DIMENSION(size(arr)) :: ans
  INTEGER(I4B) :: n,j
  REAL(SP) :: sd
  n=size(arr)
  if (n == 0_i4b) RETURN
  sd=0.0_sp
  if (present(seed)) sd=seed
  ans(1)=arr(1)+sd
  if (n < NPAR_CUMSUM) then
    do j=2,n
      ans(j)=ans(j-1)+arr(j)
    end do
  else
    ans(2:n:2)=cumsum_r(arr(2:n:2)+arr(1:n-1:2),sd)
    ans(3:n:2)=ans(2:n-1:2)+arr(3:n:2)
  end if
  END FUNCTION cumsum_r
!BL
  RECURSIVE FUNCTION cumsum_i(arr,seed) RESULT(ans)
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: arr
  INTEGER(I4B), OPTIONAL, INTENT(IN) :: seed
  INTEGER(I4B), DIMENSION(size(arr)) :: ans
  INTEGER(I4B) :: n,j,sd
  n=size(arr)
  if (n == 0_i4b) RETURN
  sd=0_i4b
  if (present(seed)) sd=seed
  ans(1)=arr(1)+sd
  if (n < NPAR_CUMSUM) then
    do j=2,n
      ans(j)=ans(j-1)+arr(j)
    end do
  else
    ans(2:n:2)=cumsum_i(arr(2:n:2)+arr(1:n-1:2),sd)
    ans(3:n:2)=ans(2:n-1:2)+arr(3:n:2)
  end if
  END FUNCTION cumsum_i
!BL
!BL
  RECURSIVE FUNCTION cumprod(arr,seed) RESULT(ans)
  REAL(SP), DIMENSION(:), INTENT(IN) :: arr
  REAL(SP), OPTIONAL, INTENT(IN) :: seed
  REAL(SP), DIMENSION(size(arr)) :: ans
  INTEGER(I4B) :: n,j
  REAL(SP) :: sd
  n=size(arr)
  if (n == 0_i4b) RETURN
  sd=1.0_sp
  if (present(seed)) sd=seed
  ans(1)=arr(1)*sd
  if (n < NPAR_CUMPROD) then
    do j=2,n
```

```
      ans(j)=ans(j-1)*arr(j)
    end do
  else
    ans(2:n:2)=cumprod(arr(2:n:2)*arr(1:n-1:2),sd)
    ans(3:n:2)=ans(2:n-1:2)*arr(3:n:2)
  end if
  END FUNCTION cumprod
!BL
!BL
  FUNCTION poly_rr(x,coeffs)
  REAL(SP), INTENT(IN) :: x
  REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs
  REAL(SP) :: poly_rr
  REAL(SP) :: pow
  REAL(SP), DIMENSION(:), ALLOCATABLE :: vec
  INTEGER(I4B) :: i,n,nn
  n=size(coeffs)
  if (n <= 0) then
    poly_rr=0.0_sp
  else if (n < NPAR_POLY) then
    poly_rr=coeffs(n)
    do i=n-1,1,-1
      poly_rr=x*poly_rr+coeffs(i)
    end do
  else
    allocate(vec(n+1))
    pow=x
    vec(1:n)=coeffs
    do
      vec(n+1)=0.0_sp
      nn=ishft(n+1,-1)
      vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
      if (nn == 1) exit
      pow=pow*pow
      n=nn
    end do
    poly_rr=vec(1)
    deallocate(vec)
  end if
  END FUNCTION poly_rr
!BL
  FUNCTION poly_dd(x,coeffs)
  REAL(DP), INTENT(IN) :: x
  REAL(DP), DIMENSION(:), INTENT(IN) :: coeffs
  REAL(DP) :: poly_dd
  REAL(DP) :: pow
  REAL(DP), DIMENSION(:), ALLOCATABLE :: vec
  INTEGER(I4B) :: i,n,nn
  n=size(coeffs)
  if (n <= 0) then
    poly_dd=0.0_dp
  else if (n < NPAR_POLY) then
    poly_dd=coeffs(n)
    do i=n-1,1,-1
```

```
      poly_dd=x*poly_dd+coeffs(i)
    end do
  else
    allocate(vec(n+1))
    pow=x
    vec(1:n)=coeffs
    do
      vec(n+1)=0.0_dp
      nn=ishft(n+1,-1)
      vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
      if (nn == 1) exit
      pow=pow*pow
      n=nn
    end do
    poly_dd=vec(1)
    deallocate(vec)
  end if
  END FUNCTION poly_dd
!BL
  FUNCTION poly_rc(x,coeffs)
  COMPLEX(SPC), INTENT(IN) :: x
  REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs
  COMPLEX(SPC) :: poly_rc
  COMPLEX(SPC) :: pow
  COMPLEX(SPC), DIMENSION(:), ALLOCATABLE :: vec
  INTEGER(I4B) :: i,n,nn
  n=size(coeffs)
  if (n <= 0) then
    poly_rc=0.0_sp
  else if (n < NPAR_POLY) then
    poly_rc=coeffs(n)
    do i=n-1,1,-1
      poly_rc=x*poly_rc+coeffs(i)
    end do
  else
    allocate(vec(n+1))
    pow=x
    vec(1:n)=coeffs
    do
      vec(n+1)=0.0_sp
      nn=ishft(n+1,-1)
      vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
      if (nn == 1) exit
      pow=pow*pow
      n=nn
    end do
    poly_rc=vec(1)
    deallocate(vec)
  end if
  END FUNCTION poly_rc
!BL
  FUNCTION poly_cc(x,coeffs)
  COMPLEX(SPC), INTENT(IN) :: x
  COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: coeffs
```

```
  COMPLEX(SPC) :: poly_cc
  COMPLEX(SPC) :: pow
  COMPLEX(SPC), DIMENSION(:), ALLOCATABLE :: vec
  INTEGER(I4B) :: i,n,nn
  n=size(coeffs)
  if (n <= 0) then
    poly_cc=0.0_sp
  else if (n < NPAR_POLY) then
    poly_cc=coeffs(n)
    do i=n-1,1,-1
      poly_cc=x*poly_cc+coeffs(i)
    end do
  else
    allocate(vec(n+1))
    pow=x
    vec(1:n)=coeffs
    do
      vec(n+1)=0.0_sp
      nn=ishft(n+1,-1)
      vec(1:nn)=vec(1:n:2)+pow*vec(2:n+1:2)
      if (nn == 1) exit
      pow=pow*pow
      n=nn
    end do
    poly_cc=vec(1)
    deallocate(vec)
  end if
  END FUNCTION poly_cc
!BL
  FUNCTION poly_rrv(x,coeffs)
  REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs,x
  REAL(SP), DIMENSION(size(x)) :: poly_rrv
  INTEGER(I4B) :: i,n,m
  m=size(coeffs)
  n=size(x)
  if (m <= 0) then
    poly_rrv=0.0_sp
  else if (m < n .or. m < NPAR_POLY) then
    poly_rrv=coeffs(m)
    do i=m-1,1,-1
      poly_rrv=x*poly_rrv+coeffs(i)
    end do
  else
    do i=1,n
      poly_rrv(i)=poly_rr(x(i),coeffs)
    end do
  end if
  END FUNCTION poly_rrv
!BL
  FUNCTION poly_ddv(x,coeffs)
  REAL(DP), DIMENSION(:), INTENT(IN) :: coeffs,x
  REAL(DP), DIMENSION(size(x)) :: poly_ddv
  INTEGER(I4B) :: i,n,m
  m=size(coeffs)
```

```
  n=size(x)
  if (m <= 0) then
    poly_ddv=0.0_dp
  else if (m < n .or. m < NPAR_POLY) then
    poly_ddv=coeffs(m)
    do i=m-1,1,-1
      poly_ddv=x*poly_ddv+coeffs(i)
    end do
  else
    do i=1,n
      poly_ddv(i)=poly_dd(x(i),coeffs)
    end do
  end if
  END FUNCTION poly_ddv
!BL
  FUNCTION poly_msk_rrv(x,coeffs,mask)
  REAL(SP), DIMENSION(:), INTENT(IN) :: coeffs,x
  LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: mask
  REAL(SP), DIMENSION(size(x)) :: poly_msk_rrv
  poly_msk_rrv=unpack(poly_rrv(pack(x,mask),coeffs),mask,0.0_sp)
  END FUNCTION poly_msk_rrv
!BL
  FUNCTION poly_msk_ddv(x,coeffs,mask)
  REAL(DP), DIMENSION(:), INTENT(IN) :: coeffs,x
  LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: mask
  REAL(DP), DIMENSION(size(x)) :: poly_msk_ddv
  poly_msk_ddv=unpack(poly_ddv(pack(x,mask),coeffs),mask,0.0_dp)
  END FUNCTION poly_msk_ddv
!BL
!BL
  RECURSIVE FUNCTION poly_term_rr(a,b) RESULT(u)
  REAL(SP), DIMENSION(:), INTENT(IN) :: a
  REAL(SP), INTENT(IN) :: b
  REAL(SP), DIMENSION(size(a)) :: u
  INTEGER(I4B) :: n,j
  n=size(a)
  if (n <= 0) RETURN
  u(1)=a(1)
  if (n < NPAR_POLYTERM) then
    do j=2,n
      u(j)=a(j)+b*u(j-1)
    end do
  else
    u(2:n:2)=poly_term_rr(a(2:n:2)+a(1:n-1:2)*b,b*b)
    u(3:n:2)=a(3:n:2)+b*u(2:n-1:2)
  end if
  END FUNCTION poly_term_rr
!BL
  RECURSIVE FUNCTION poly_term_cc(a,b) RESULT(u)
  COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
  COMPLEX(SPC), INTENT(IN) :: b
  COMPLEX(SPC), DIMENSION(size(a)) :: u
  INTEGER(I4B) :: n,j
  n=size(a)
```

```
  if (n <= 0) RETURN
  u(1)=a(1)
  if (n < NPAR_POLYTERM) then
    do j=2,n
      u(j)=a(j)+b*u(j-1)
    end do
  else
    u(2:n:2)=poly_term_cc(a(2:n:2)+a(1:n-1:2)*b,b*b)
    u(3:n:2)=a(3:n:2)+b*u(2:n-1:2)
  end if
  END FUNCTION poly_term_cc
!BL
!BL
  FUNCTION zroots_unity(n,nn)
  INTEGER(I4B), INTENT(IN) :: n,nn
  COMPLEX(SPC), DIMENSION(nn) :: zroots_unity
  INTEGER(I4B) :: k
  REAL(SP) :: theta
  zroots_unity(1)=1.0
  theta=TWOPI/n
  k=1
  do
    if (k >= nn) exit
    zroots_unity(k+1)=cmplx(cos(k*theta),sin(k*theta),SPC)
    zroots_unity(k+2:min(2*k,nn))=zroots_unity(k+1)*&
      zroots_unity(2:min(k,nn-k))
    k=2*k
  end do
  END FUNCTION zroots_unity
!BL
  FUNCTION outerprod_r(a,b)
  REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
  REAL(SP), DIMENSION(size(a),size(b)) :: outerprod_r
  outerprod_r = spread(a,dim=2,ncopies=size(b)) * &
    spread(b,dim=1,ncopies=size(a))
  END FUNCTION outerprod_r
!BL
  FUNCTION outerprod_d(a,b)
  REAL(DP), DIMENSION(:), INTENT(IN) :: a,b
  REAL(DP), DIMENSION(size(a),size(b)) :: outerprod_d
  outerprod_d = spread(a,dim=2,ncopies=size(b)) * &
    spread(b,dim=1,ncopies=size(a))
  END FUNCTION outerprod_d
!BL
  FUNCTION outerdiv(a,b)
  REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
  REAL(SP), DIMENSION(size(a),size(b)) :: outerdiv
  outerdiv = spread(a,dim=2,ncopies=size(b)) / &
    spread(b,dim=1,ncopies=size(a))
  END FUNCTION outerdiv
!BL
  FUNCTION outersum(a,b)
  REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
  REAL(SP), DIMENSION(size(a),size(b)) :: outersum
```

```
  outersum = spread(a,dim=2,ncopies=size(b)) + &
    spread(b,dim=1,ncopies=size(a))
  END FUNCTION outersum
!BL
  FUNCTION outerdiff_r(a,b)
  REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
  REAL(SP), DIMENSION(size(a),size(b)) :: outerdiff_r
  outerdiff_r = spread(a,dim=2,ncopies=size(b)) - &
    spread(b,dim=1,ncopies=size(a))
  END FUNCTION outerdiff_r
!BL
  FUNCTION outerdiff_d(a,b)
  REAL(DP), DIMENSION(:), INTENT(IN) :: a,b
  REAL(DP), DIMENSION(size(a),size(b)) :: outerdiff_d
  outerdiff_d = spread(a,dim=2,ncopies=size(b)) - &
    spread(b,dim=1,ncopies=size(a))
  END FUNCTION outerdiff_d
!BL
  FUNCTION outerdiff_i(a,b)
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: a,b
  INTEGER(I4B), DIMENSION(size(a),size(b)) :: outerdiff_i
  outerdiff_i = spread(a,dim=2,ncopies=size(b)) - &
    spread(b,dim=1,ncopies=size(a))
  END FUNCTION outerdiff_i
!BL
  FUNCTION outerand(a,b)
  LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: a,b
  LOGICAL(LGT), DIMENSION(size(a),size(b)) :: outerand
  outerand = spread(a,dim=2,ncopies=size(b)) .and. &
    spread(b,dim=1,ncopies=size(a))
  END FUNCTION outerand
!BL
  SUBROUTINE scatter_add_r(dest,source,dest_index)
  REAL(SP), DIMENSION(:), INTENT(OUT) :: dest
  REAL(SP), DIMENSION(:), INTENT(IN) :: source
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
  INTEGER(I4B) :: m,n,j,i
  n=assert_eq2(size(source),size(dest_index),'scatter_add_r')
  m=size(dest)
  do j=1,n
    i=dest_index(j)
    if (i > 0 .and. i <= m) dest(i)=dest(i)+source(j)
  end do
  END SUBROUTINE scatter_add_r
  SUBROUTINE scatter_add_d(dest,source,dest_index)
  REAL(DP), DIMENSION(:), INTENT(OUT) :: dest
  REAL(DP), DIMENSION(:), INTENT(IN) :: source
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
  INTEGER(I4B) :: m,n,j,i
  n=assert_eq2(size(source),size(dest_index),'scatter_add_d')
  m=size(dest)
  do j=1,n
    i=dest_index(j)
    if (i > 0 .and. i <= m) dest(i)=dest(i)+source(j)
```

```
  end do
  END SUBROUTINE scatter_add_d
  SUBROUTINE scatter_max_r(dest,source,dest_index)
  REAL(SP), DIMENSION(:), INTENT(OUT) :: dest
  REAL(SP), DIMENSION(:), INTENT(IN) :: source
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
  INTEGER(I4B) :: m,n,j,i
  n=assert_eq2(size(source),size(dest_index),'scatter_max_r')
  m=size(dest)
  do j=1,n
    i=dest_index(j)
    if (i > 0 .and. i <= m) dest(i)=max(dest(i),source(j))
  end do
  END SUBROUTINE scatter_max_r
  SUBROUTINE scatter_max_d(dest,source,dest_index)
  REAL(DP), DIMENSION(:), INTENT(OUT) :: dest
  REAL(DP), DIMENSION(:), INTENT(IN) :: source
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: dest_index
  INTEGER(I4B) :: m,n,j,i
  n=assert_eq2(size(source),size(dest_index),'scatter_max_d')
  m=size(dest)
  do j=1,n
    i=dest_index(j)
    if (i > 0 .and. i <= m) dest(i)=max(dest(i),source(j))
  end do
  END SUBROUTINE scatter_max_d
!BL
  SUBROUTINE diagadd_rv(mat,diag)
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
  REAL(SP), DIMENSION(:), INTENT(IN) :: diag
  INTEGER(I4B) :: j,n
  n = assert_eq2(size(diag),min(size(mat,1),size(mat,2)),'diagadd_rv')
  do j=1,n
    mat(j,j)=mat(j,j)+diag(j)
  end do
  END SUBROUTINE diagadd_rv
!BL
  SUBROUTINE diagadd_r(mat,diag)
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
  REAL(SP), INTENT(IN) :: diag
  INTEGER(I4B) :: j,n
  n = min(size(mat,1),size(mat,2))
  do j=1,n
    mat(j,j)=mat(j,j)+diag
  end do
  END SUBROUTINE diagadd_r
!BL
  SUBROUTINE diagmult_rv(mat,diag)
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
  REAL(SP), DIMENSION(:), INTENT(IN) :: diag
  INTEGER(I4B) :: j,n
  n = assert_eq2(size(diag),min(size(mat,1),size(mat,2)),'diagmult_rv')
  do j=1,n
    mat(j,j)=mat(j,j)*diag(j)
```

```
  end do
  END SUBROUTINE diagmult_rv
!BL
  SUBROUTINE diagmult_r(mat,diag)
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
  REAL(SP), INTENT(IN) :: diag
  INTEGER(I4B) :: j,n
  n = min(size(mat,1),size(mat,2))
  do j=1,n
    mat(j,j)=mat(j,j)*diag
  end do
  END SUBROUTINE diagmult_r
!BL
  FUNCTION get_diag_rv(mat)
  REAL(SP), DIMENSION(:,:), INTENT(IN) :: mat
  REAL(SP), DIMENSION(size(mat,1)) :: get_diag_rv
  INTEGER(I4B) :: j
  j=assert_eq2(size(mat,1),size(mat,2),'get_diag_rv')
  do j=1,size(mat,1)
    get_diag_rv(j)=mat(j,j)
  end do
  END FUNCTION get_diag_rv
!BL
  FUNCTION get_diag_dv(mat)
  REAL(DP), DIMENSION(:,:), INTENT(IN) :: mat
  REAL(DP), DIMENSION(size(mat,1)) :: get_diag_dv
  INTEGER(I4B) :: j
  j=assert_eq2(size(mat,1),size(mat,2),'get_diag_dv')
  do j=1,size(mat,1)
    get_diag_dv(j)=mat(j,j)
  end do
  END FUNCTION get_diag_dv
!BL
  SUBROUTINE put_diag_rv(diagv,mat)
  REAL(SP), DIMENSION(:), INTENT(IN) :: diagv
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
  INTEGER(I4B) :: j,n
  n=assert_eq2(size(diagv),min(size(mat,1),size(mat,2)),'put_diag_rv')
  do j=1,n
    mat(j,j)=diagv(j)
  end do
  END SUBROUTINE put_diag_rv
!BL
  SUBROUTINE put_diag_r(scal,mat)
  REAL(SP), INTENT(IN) :: scal
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: mat
  INTEGER(I4B) :: j,n
  n = min(size(mat,1),size(mat,2))
  do j=1,n
    mat(j,j)=scal
  end do
  END SUBROUTINE put_diag_r
!BL
  SUBROUTINE unit_matrix(mat)
```

```
  REAL(SP), DIMENSION(:,:), INTENT(OUT) :: mat
  INTEGER(I4B) :: i,n
  n=min(size(mat,1),size(mat,2))
  mat(:,:)=0.0_sp
  do i=1,n
    mat(i,i)=1.0_sp
  end do
  END SUBROUTINE unit_matrix
!BL
  FUNCTION upper_triangle(j,k,extra)
  INTEGER(I4B), INTENT(IN) :: j,k
  INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
  LOGICAL(LGT), DIMENSION(j,k) :: upper_triangle
  INTEGER(I4B) :: n
  n=0
  if (present(extra)) n=extra
  upper_triangle=(outerdiff(arth_i(1,1,j),arth_i(1,1,k)) < n)
  END FUNCTION upper_triangle
!BL
  FUNCTION lower_triangle(j,k,extra)
  INTEGER(I4B), INTENT(IN) :: j,k
  INTEGER(I4B), OPTIONAL, INTENT(IN) :: extra
  LOGICAL(LGT), DIMENSION(j,k) :: lower_triangle
  INTEGER(I4B) :: n
  n=0
  if (present(extra)) n=extra
  lower_triangle=(outerdiff(arth_i(1,1,j),arth_i(1,1,k)) > -n)
  END FUNCTION lower_triangle
!BL
  FUNCTION vabs(v)
  REAL(SP), DIMENSION(:), INTENT(IN) :: v
  REAL(SP) :: vabs
  vabs=sqrt(dot_product(v,v))
  END FUNCTION vabs
!BL
END MODULE nrutil

MODULE ode_path
  USE nrtype
  INTEGER(I4B) :: nok,nbad,kount
  LOGICAL(LGT), SAVE :: save_steps=.false.
  REAL(SP) :: dxsav
  REAL(SP), DIMENSION(:), POINTER :: xp
  REAL(SP), DIMENSION(:,:), POINTER :: yp
END MODULE ode_path

MODULE hypgeo_info
  USE nrtype
  COMPLEX(SPC) :: hypgeo_aa,hypgeo_bb,hypgeo_cc,hypgeo_dz,hypgeo_z0
END MODULE hypgeo_info

MODULE nr
  INTERFACE
    SUBROUTINE airy(x,ai,bi,aip,bip)
```

```
  USE nrtype
  REAL(SP), INTENT(IN) :: x
  REAL(SP), INTENT(OUT) :: ai,bi,aip,bip
  END SUBROUTINE airy
END INTERFACE
INTERFACE
  SUBROUTINE amebsa(p,y,pb,yb,ftol,func,iter,temptr)
  USE nrtype
  INTEGER(I4B), INTENT(INOUT) :: iter
  REAL(SP), INTENT(INOUT) :: yb
  REAL(SP), INTENT(IN) :: ftol,temptr
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: y,pb
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: p
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
  END INTERFACE
  END SUBROUTINE amebsa
END INTERFACE
INTERFACE
  SUBROUTINE amoeba(p,y,ftol,func,iter)
  USE nrtype
  INTEGER(I4B), INTENT(OUT) :: iter
  REAL(SP), INTENT(IN) :: ftol
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: p
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
  END INTERFACE
  END SUBROUTINE amoeba
END INTERFACE
INTERFACE
  SUBROUTINE anneal(x,y,iorder)
  USE nrtype
  INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: iorder
  REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
  END SUBROUTINE anneal
END INTERFACE
INTERFACE
  SUBROUTINE asolve(b,x,itrnsp)
  USE nrtype
  REAL(DP), DIMENSION(:), INTENT(IN) :: b
  REAL(DP), DIMENSION(:), INTENT(OUT) :: x
  INTEGER(I4B), INTENT(IN) :: itrnsp
  END SUBROUTINE asolve
END INTERFACE
INTERFACE
```

```
  SUBROUTINE atimes(x,r,itrnsp)
  USE nrtype
  REAL(DP), DIMENSION(:), INTENT(IN) :: x
  REAL(DP), DIMENSION(:), INTENT(OUT) :: r
  INTEGER(I4B), INTENT(IN) :: itrnsp
  END SUBROUTINE atimes
END INTERFACE
INTERFACE
  SUBROUTINE avevar(data,ave,var)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: data
  REAL(SP), INTENT(OUT) :: ave,var
  END SUBROUTINE avevar
END INTERFACE
INTERFACE
  SUBROUTINE balanc(a)
  USE nrtype
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
  END SUBROUTINE balanc
END INTERFACE
INTERFACE
  SUBROUTINE banbks(a,m1,m2,al,indx,b)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: m1,m2
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
  REAL(SP), DIMENSION(:,:), INTENT(IN) :: a,al
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
  END SUBROUTINE banbks
END INTERFACE
INTERFACE
  SUBROUTINE bandec(a,m1,m2,al,indx,d)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: m1,m2
  INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: indx
  REAL(SP), INTENT(OUT) :: d
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
  REAL(SP), DIMENSION(:,:), INTENT(OUT) :: al
  END SUBROUTINE bandec
END INTERFACE
INTERFACE
  SUBROUTINE banmul(a,m1,m2,x,b)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: m1,m2
  REAL(SP), DIMENSION(:), INTENT(IN) :: x
  REAL(SP), DIMENSION(:), INTENT(OUT) :: b
  REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
  END SUBROUTINE banmul
END INTERFACE
INTERFACE
  SUBROUTINE bcucof(y,y1,y2,y12,d1,d2,c)
  USE nrtype
  REAL(SP), INTENT(IN) :: d1,d2
  REAL(SP), DIMENSION(4), INTENT(IN) :: y,y1,y2,y12
  REAL(SP), DIMENSION(4,4), INTENT(OUT) :: c
```

```
    END SUBROUTINE bcucof
  END INTERFACE
  INTERFACE
    SUBROUTINE bcuint(y,y1,y2,y12,x1l,x1u,x2l,x2u,x1,x2,ansy,&
      ansy1,ansy2)
    USE nrtype
    REAL(SP), DIMENSION(4), INTENT(IN) :: y,y1,y2,y12
    REAL(SP), INTENT(IN) :: x1l,x1u,x2l,x2u,x1,x2
    REAL(SP), INTENT(OUT) :: ansy,ansy1,ansy2
    END SUBROUTINE bcuint
  END INTERFACE
  INTERFACE beschb
    SUBROUTINE beschb_s(x,gam1,gam2,gampl,gammi)
    USE nrtype
    REAL(DP), INTENT(IN) :: x
    REAL(DP), INTENT(OUT) :: gam1,gam2,gampl,gammi
    END SUBROUTINE beschb_s
!BL
    SUBROUTINE beschb_v(x,gam1,gam2,gampl,gammi)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(IN) :: x
    REAL(DP), DIMENSION(:), INTENT(OUT) :: gam1,gam2,gampl,gammi
    END SUBROUTINE beschb_v
  END INTERFACE
  INTERFACE bessi
    FUNCTION bessi_s(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessi_s
    END FUNCTION bessi_s
!BL
    FUNCTION bessi_v(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessi_v
    END FUNCTION bessi_v
  END INTERFACE
  INTERFACE bessi0
    FUNCTION bessi0_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessi0_s
    END FUNCTION bessi0_s
!BL
    FUNCTION bessi0_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessi0_v
    END FUNCTION bessi0_v
  END INTERFACE
  INTERFACE bessi1
    FUNCTION bessi1_s(x)
```

```
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessi1_s
    END FUNCTION bessi1_s
!BL
    FUNCTION bessi1_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessi1_v
    END FUNCTION bessi1_v
  END INTERFACE
  INTERFACE
    SUBROUTINE bessik(x,xnu,ri,rk,rip,rkp)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,xnu
    REAL(SP), INTENT(OUT) :: ri,rk,rip,rkp
    END SUBROUTINE bessik
  END INTERFACE
  INTERFACE bessj
    FUNCTION bessj_s(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessj_s
    END FUNCTION bessj_s
!BL
    FUNCTION bessj_v(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessj_v
    END FUNCTION bessj_v
  END INTERFACE
  INTERFACE bessj0
    FUNCTION bessj0_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessj0_s
    END FUNCTION bessj0_s
!BL
    FUNCTION bessj0_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessj0_v
    END FUNCTION bessj0_v
  END INTERFACE
  INTERFACE bessj1
    FUNCTION bessj1_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessj1_s
    END FUNCTION bessj1_s
!BL
    FUNCTION bessj1_v(x)
```

```
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessj1_v
    END FUNCTION bessj1_v
  END INTERFACE
  INTERFACE bessjy
    SUBROUTINE bessjy_s(x,xnu,rj,ry,rjp,ryp)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,xnu
    REAL(SP), INTENT(OUT) :: rj,ry,rjp,ryp
    END SUBROUTINE bessjy_s
!BL
    SUBROUTINE bessjy_v(x,xnu,rj,ry,rjp,ryp)
    USE nrtype
    REAL(SP), INTENT(IN) :: xnu
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(OUT) :: rj,rjp,ry,ryp
    END SUBROUTINE bessjy_v
  END INTERFACE
  INTERFACE bessk
    FUNCTION bessk_s(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessk_s
    END FUNCTION bessk_s
!BL
    FUNCTION bessk_v(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessk_v
    END FUNCTION bessk_v
  END INTERFACE
  INTERFACE bessk0
    FUNCTION bessk0_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessk0_s
    END FUNCTION bessk0_s
!BL
    FUNCTION bessk0_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessk0_v
    END FUNCTION bessk0_v
  END INTERFACE
  INTERFACE bessk1
    FUNCTION bessk1_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessk1_s
    END FUNCTION bessk1_s
!BL
```

```
    FUNCTION bessk1_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessk1_v
    END FUNCTION bessk1_v
  END INTERFACE
  INTERFACE bessy
    FUNCTION bessy_s(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessy_s
    END FUNCTION bessy_s
!BL
    FUNCTION bessy_v(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessy_v
    END FUNCTION bessy_v
  END INTERFACE
  INTERFACE bessy0
    FUNCTION bessy0_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessy0_s
    END FUNCTION bessy0_s
!BL
    FUNCTION bessy0_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessy0_v
    END FUNCTION bessy0_v
  END INTERFACE
  INTERFACE bessy1
    FUNCTION bessy1_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: bessy1_s
    END FUNCTION bessy1_s
!BL
    FUNCTION bessy1_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: bessy1_v
    END FUNCTION bessy1_v
  END INTERFACE
  INTERFACE beta
    FUNCTION beta_s(z,w)
    USE nrtype
    REAL(SP), INTENT(IN) :: z,w
    REAL(SP) :: beta_s
    END FUNCTION beta_s
!BL
```

```
    FUNCTION beta_v(z,w)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: z,w
    REAL(SP), DIMENSION(size(z)) :: beta_v
    END FUNCTION beta_v
  END INTERFACE
  INTERFACE betacf
    FUNCTION betacf_s(a,b,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b,x
    REAL(SP) :: betacf_s
    END FUNCTION betacf_s
!BL
    FUNCTION betacf_v(a,b,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,x
    REAL(SP), DIMENSION(size(x)) :: betacf_v
    END FUNCTION betacf_v
  END INTERFACE
  INTERFACE betai
    FUNCTION betai_s(a,b,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b,x
    REAL(SP) :: betai_s
    END FUNCTION betai_s
!BL
    FUNCTION betai_v(a,b,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,x
    REAL(SP), DIMENSION(size(a)) :: betai_v
    END FUNCTION betai_v
  END INTERFACE
  INTERFACE bico
    FUNCTION bico_s(n,k)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n,k
    REAL(SP) :: bico_s
    END FUNCTION bico_s
!BL
    FUNCTION bico_v(n,k)
    USE nrtype
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n,k
    REAL(SP), DIMENSION(size(n)) :: bico_v
    END FUNCTION bico_v
  END INTERFACE
  INTERFACE
    FUNCTION bnldev(pp,n)
    USE nrtype
    REAL(SP), INTENT(IN) :: pp
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP) :: bnldev
    END FUNCTION bnldev
  END INTERFACE
  INTERFACE
```

```
  FUNCTION brent(ax,bx,cx,func,tol,xmin)
  USE nrtype
  REAL(SP), INTENT(IN) :: ax,bx,cx,tol
  REAL(SP), INTENT(OUT) :: xmin
  REAL(SP) :: brent
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
  END INTERFACE
  END FUNCTION brent
END INTERFACE
INTERFACE
  SUBROUTINE broydn(x,check)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
  LOGICAL(LGT), INTENT(OUT) :: check
  END SUBROUTINE broydn
END INTERFACE
INTERFACE
  SUBROUTINE bsstep(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
  REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
  REAL(SP), INTENT(INOUT) :: x
  REAL(SP), INTENT(IN) :: htry,eps
  REAL(SP), INTENT(OUT) :: hdid,hnext
  INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
  END INTERFACE
  END SUBROUTINE bsstep
END INTERFACE
INTERFACE
  SUBROUTINE caldat(julian,mm,id,iyyy)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: julian
  INTEGER(I4B), INTENT(OUT) :: mm,id,iyyy
  END SUBROUTINE caldat
END INTERFACE
INTERFACE
  FUNCTION chder(a,b,c)
  USE nrtype
  REAL(SP), INTENT(IN) :: a,b
  REAL(SP), DIMENSION(:), INTENT(IN) :: c
  REAL(SP), DIMENSION(size(c)) :: chder
  END FUNCTION chder
END INTERFACE
```

```
  INTERFACE chebev
    FUNCTION chebev_s(a,b,c,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b,x
    REAL(SP), DIMENSION(:), INTENT(IN) :: c
    REAL(SP) :: chebev_s
    END FUNCTION chebev_s
!BL
    FUNCTION chebev_v(a,b,c,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(:), INTENT(IN) :: c,x
    REAL(SP), DIMENSION(size(x)) :: chebev_v
    END FUNCTION chebev_v
  END INTERFACE
  INTERFACE
    FUNCTION chebft(a,b,n,func)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(n) :: chebft
    INTERFACE
      FUNCTION func(x)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(IN) :: x
      REAL(SP), DIMENSION(size(x)) :: func
      END FUNCTION func
    END INTERFACE
    END FUNCTION chebft
  END INTERFACE
  INTERFACE
    FUNCTION chebpc(c)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: c
    REAL(SP), DIMENSION(size(c)) :: chebpc
    END FUNCTION chebpc
  END INTERFACE
  INTERFACE
    FUNCTION chint(a,b,c)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(:), INTENT(IN) :: c
    REAL(SP), DIMENSION(size(c)) :: chint
    END FUNCTION chint
  END INTERFACE
  INTERFACE
    SUBROUTINE choldc(a,p)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    REAL(SP), DIMENSION(:), INTENT(OUT) :: p
    END SUBROUTINE choldc
  END INTERFACE
  INTERFACE
    SUBROUTINE cholsl(a,p,b,x)
```

```
  USE nrtype
  REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
  REAL(SP), DIMENSION(:), INTENT(IN) :: p,b
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
  END SUBROUTINE cholsl
END INTERFACE
INTERFACE
  SUBROUTINE chsone(bins,ebins,knstrn,df,chsq,prob)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: knstrn
  REAL(SP), INTENT(OUT) :: df,chsq,prob
  REAL(SP), DIMENSION(:), INTENT(IN) :: bins,ebins
  END SUBROUTINE chsone
END INTERFACE
INTERFACE
  SUBROUTINE chstwo(bins1,bins2,knstrn,df,chsq,prob)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: knstrn
  REAL(SP), INTENT(OUT) :: df,chsq,prob
  REAL(SP), DIMENSION(:), INTENT(IN) :: bins1,bins2
  END SUBROUTINE chstwo
END INTERFACE
INTERFACE
  SUBROUTINE cisi(x,ci,si)
  USE nrtype
  REAL(SP), INTENT(IN) :: x
  REAL(SP), INTENT(OUT) :: ci,si
  END SUBROUTINE cisi
END INTERFACE
INTERFACE
  SUBROUTINE cntab1(nn,chisq,df,prob,cramrv,ccc)
  USE nrtype
  INTEGER(I4B), DIMENSION(:,:), INTENT(IN) :: nn
  REAL(SP), INTENT(OUT) :: chisq,df,prob,cramrv,ccc
  END SUBROUTINE cntab1
END INTERFACE
INTERFACE
  SUBROUTINE cntab2(nn,h,hx,hy,hygx,hxgy,uygx,uxgy,uxy)
  USE nrtype
  INTEGER(I4B), DIMENSION(:,:), INTENT(IN) :: nn
  REAL(SP), INTENT(OUT) :: h,hx,hy,hygx,hxgy,uygx,uxgy,uxy
  END SUBROUTINE cntab2
END INTERFACE
INTERFACE
  FUNCTION convlv(data,respns,isign)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: data
  REAL(SP), DIMENSION(:), INTENT(IN) :: respns
  INTEGER(I4B), INTENT(IN) :: isign
  REAL(SP), DIMENSION(size(data)) :: convlv
  END FUNCTION convlv
END INTERFACE
INTERFACE
  FUNCTION correl(data1,data2)
```

```
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
    REAL(SP), DIMENSION(size(data1)) :: correl
    END FUNCTION correl
  END INTERFACE
  INTERFACE
    SUBROUTINE cosft1(y)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
    END SUBROUTINE cosft1
  END INTERFACE
  INTERFACE
    SUBROUTINE cosft2(y,isign)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE cosft2
  END INTERFACE
  INTERFACE
    SUBROUTINE covsrt(covar,maska)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: covar
    LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
    END SUBROUTINE covsrt
  END INTERFACE
  INTERFACE
    SUBROUTINE cyclic(a,b,c,alpha,beta,r,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN):: a,b,c,r
    REAL(SP), INTENT(IN) :: alpha,beta
    REAL(SP), DIMENSION(:), INTENT(OUT):: x
    END SUBROUTINE cyclic
  END INTERFACE
  INTERFACE
    SUBROUTINE daub4(a,isign)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE daub4
  END INTERFACE
  INTERFACE dawson
    FUNCTION dawson_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: dawson_s
    END FUNCTION dawson_s
!BL
    FUNCTION dawson_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: dawson_v
    END FUNCTION dawson_v
  END INTERFACE
  INTERFACE
```

```
    FUNCTION dbrent(ax,bx,cx,func,dbrent_dfunc,tol,xmin)
    USE nrtype
    REAL(SP), INTENT(IN) :: ax,bx,cx,tol
    REAL(SP), INTENT(OUT) :: xmin
    REAL(SP) :: dbrent
    INTERFACE
      FUNCTION func(x)
      USE nrtype
      REAL(SP), INTENT(IN) :: x
      REAL(SP) :: func
      END FUNCTION func
!BL
      FUNCTION dbrent_dfunc(x)
      USE nrtype
      REAL(SP), INTENT(IN) :: x
      REAL(SP) :: dbrent_dfunc
      END FUNCTION dbrent_dfunc
    END INTERFACE
    END FUNCTION dbrent
  END INTERFACE
  INTERFACE
    SUBROUTINE ddpoly(c,x,pd)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: c
    REAL(SP), DIMENSION(:), INTENT(OUT) :: pd
    END SUBROUTINE ddpoly
  END INTERFACE
  INTERFACE
    FUNCTION decchk(string,ch)
    USE nrtype
    CHARACTER(1), DIMENSION(:), INTENT(IN) :: string
    CHARACTER(1), INTENT(OUT) :: ch
    LOGICAL(LGT) :: decchk
    END FUNCTION decchk
  END INTERFACE
  INTERFACE
    SUBROUTINE dfpmin(p,gtol,iter,fret,func,dfunc)
    USE nrtype
    INTEGER(I4B), INTENT(OUT) :: iter
    REAL(SP), INTENT(IN) :: gtol
    REAL(SP), INTENT(OUT) :: fret
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
    INTERFACE
      FUNCTION func(p)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(IN) :: p
      REAL(SP) :: func
      END FUNCTION func
!BL
      FUNCTION dfunc(p)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(IN) :: p
      REAL(SP), DIMENSION(size(p)) :: dfunc
```

```
    END FUNCTION dfunc
  END INTERFACE
  END SUBROUTINE dfpmin
END INTERFACE
INTERFACE
  FUNCTION dfridr(func,x,h,err)
  USE nrtype
  REAL(SP), INTENT(IN) :: x,h
  REAL(SP), INTENT(OUT) :: err
  REAL(SP) :: dfridr
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
  END INTERFACE
  END FUNCTION dfridr
END INTERFACE
INTERFACE
  SUBROUTINE dftcor(w,delta,a,b,endpts,corre,corim,corfac)
  USE nrtype
  REAL(SP), INTENT(IN) :: w,delta,a,b
  REAL(SP), INTENT(OUT) :: corre,corim,corfac
  REAL(SP), DIMENSION(:), INTENT(IN) :: endpts
  END SUBROUTINE dftcor
END INTERFACE
INTERFACE
  SUBROUTINE dftint(func,a,b,w,cosint,sinint)
  USE nrtype
  REAL(SP), INTENT(IN) :: a,b,w
  REAL(SP), INTENT(OUT) :: cosint,sinint
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
  END INTERFACE
  END SUBROUTINE dftint
END INTERFACE
INTERFACE
  SUBROUTINE difeq(k,k1,k2,jsf,is1,isf,indexv,s,y)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: is1,isf,jsf,k,k1,k2
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indexv
  REAL(SP), DIMENSION(:,:), INTENT(OUT) :: s
  REAL(SP), DIMENSION(:,:), INTENT(IN) :: y
  END SUBROUTINE difeq
END INTERFACE
INTERFACE
  FUNCTION eclass(lista,listb,n)
  USE nrtype
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: lista,listb
```

```
    INTEGER(I4B), INTENT(IN) :: n
    INTEGER(I4B), DIMENSION(n) :: eclass
    END FUNCTION eclass
  END INTERFACE
  INTERFACE
    FUNCTION eclazz(equiv,n)
    USE nrtype
    INTERFACE
      FUNCTION equiv(i,j)
      USE nrtype
      LOGICAL(LGT) :: equiv
      INTEGER(I4B), INTENT(IN) :: i,j
      END FUNCTION equiv
    END INTERFACE
    INTEGER(I4B), INTENT(IN) :: n
    INTEGER(I4B), DIMENSION(n) :: eclazz
    END FUNCTION eclazz
  END INTERFACE
  INTERFACE
    FUNCTION ei(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: ei
    END FUNCTION ei
  END INTERFACE
  INTERFACE
    SUBROUTINE eigsrt(d,v)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: v
    END SUBROUTINE eigsrt
  END INTERFACE
  INTERFACE elle
    FUNCTION elle_s(phi,ak)
    USE nrtype
    REAL(SP), INTENT(IN) :: phi,ak
    REAL(SP) :: elle_s
    END FUNCTION elle_s
!BL
    FUNCTION elle_v(phi,ak)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: phi,ak
    REAL(SP), DIMENSION(size(phi)) :: elle_v
    END FUNCTION elle_v
  END INTERFACE
  INTERFACE ellf
    FUNCTION ellf_s(phi,ak)
    USE nrtype
    REAL(SP), INTENT(IN) :: phi,ak
    REAL(SP) :: ellf_s
    END FUNCTION ellf_s
!BL
    FUNCTION ellf_v(phi,ak)
    USE nrtype
```

```
    REAL(SP), DIMENSION(:), INTENT(IN) :: phi,ak
    REAL(SP), DIMENSION(size(phi)) :: ellf_v
    END FUNCTION ellf_v
  END INTERFACE
  INTERFACE ellpi
    FUNCTION ellpi_s(phi,en,ak)
    USE nrtype
    REAL(SP), INTENT(IN) :: phi,en,ak
    REAL(SP) :: ellpi_s
    END FUNCTION ellpi_s
!BL
    FUNCTION ellpi_v(phi,en,ak)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: phi,en,ak
    REAL(SP), DIMENSION(size(phi)) :: ellpi_v
    END FUNCTION ellpi_v
  END INTERFACE
  INTERFACE
    SUBROUTINE elmhes(a)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    END SUBROUTINE elmhes
  END INTERFACE
  INTERFACE erf
    FUNCTION erf_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: erf_s
    END FUNCTION erf_s
!BL
    FUNCTION erf_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: erf_v
    END FUNCTION erf_v
  END INTERFACE
  INTERFACE erfc
    FUNCTION erfc_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: erfc_s
    END FUNCTION erfc_s
!BL
    FUNCTION erfc_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: erfc_v
    END FUNCTION erfc_v
  END INTERFACE
  INTERFACE erfcc
    FUNCTION erfcc_s(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: erfcc_s
```

```
    END FUNCTION erfcc_s
!BL
    FUNCTION erfcc_v(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: erfcc_v
    END FUNCTION erfcc_v
  END INTERFACE
  INTERFACE
    SUBROUTINE eulsum(sum,term,jterm)
    USE nrtype
    REAL(SP), INTENT(INOUT) :: sum
    REAL(SP), INTENT(IN) :: term
    INTEGER(I4B), INTENT(IN) :: jterm
    END SUBROUTINE eulsum
  END INTERFACE
  INTERFACE
    FUNCTION evlmem(fdt,d,xms)
    USE nrtype
    REAL(SP), INTENT(IN) :: fdt,xms
    REAL(SP), DIMENSION(:), INTENT(IN) :: d
    REAL(SP) :: evlmem
    END FUNCTION evlmem
  END INTERFACE
  INTERFACE expdev
    SUBROUTINE expdev_s(harvest)
    USE nrtype
    REAL(SP), INTENT(OUT) :: harvest
    END SUBROUTINE expdev_s
!BL
    SUBROUTINE expdev_v(harvest)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
    END SUBROUTINE expdev_v
  END INTERFACE
  INTERFACE
    FUNCTION expint(n,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: expint
    END FUNCTION expint
  END INTERFACE
  INTERFACE factln
    FUNCTION factln_s(n)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP) :: factln_s
    END FUNCTION factln_s
!BL
    FUNCTION factln_v(n)
    USE nrtype
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
    REAL(SP), DIMENSION(size(n)) :: factln_v
```

```
    END FUNCTION factln_v
  END INTERFACE
  INTERFACE factrl
    FUNCTION factrl_s(n)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP) :: factrl_s
    END FUNCTION factrl_s
!BL
    FUNCTION factrl_v(n)
    USE nrtype
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: n
    REAL(SP), DIMENSION(size(n)) :: factrl_v
    END FUNCTION factrl_v
  END INTERFACE
  INTERFACE
    SUBROUTINE fasper(x,y,ofac,hifac,px,py,jmax,prob)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    REAL(SP), INTENT(IN) :: ofac,hifac
    INTEGER(I4B), INTENT(OUT) :: jmax
    REAL(SP), INTENT(OUT) :: prob
    REAL(SP), DIMENSION(:), POINTER :: px,py
    END SUBROUTINE fasper
  END INTERFACE
  INTERFACE
    SUBROUTINE fdjac(x,fvec,df)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: fvec
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: df
    END SUBROUTINE fdjac
  END INTERFACE
  INTERFACE
    SUBROUTINE fgauss(x,a,y,dyda)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,a
    REAL(SP), DIMENSION(:), INTENT(OUT) :: y
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dyda
    END SUBROUTINE fgauss
  END INTERFACE
  INTERFACE
    SUBROUTINE fit(x,y,a,b,siga,sigb,chi2,q,sig)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    REAL(SP), INTENT(OUT) :: a,b,siga,sigb,chi2,q
    REAL(SP), DIMENSION(:), OPTIONAL, INTENT(IN) :: sig
    END SUBROUTINE fit
  END INTERFACE
  INTERFACE
    SUBROUTINE fitexy(x,y,sigx,sigy,a,b,siga,sigb,chi2,q)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sigx,sigy
    REAL(SP), INTENT(OUT) :: a,b,siga,sigb,chi2,q
```

```
    END SUBROUTINE fitexy
  END INTERFACE
  INTERFACE
    SUBROUTINE fixrts(d)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
    END SUBROUTINE fixrts
  END INTERFACE
  INTERFACE
    FUNCTION fleg(x,n)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(n) :: fleg
    END FUNCTION fleg
  END INTERFACE
  INTERFACE
    SUBROUTINE flmoon(n,nph,jd,frac)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n,nph
    INTEGER(I4B), INTENT(OUT) :: jd
    REAL(SP), INTENT(OUT) :: frac
    END SUBROUTINE flmoon
  END INTERFACE
  INTERFACE four1
    SUBROUTINE four1_dp(data,isign)
    USE nrtype
    COMPLEX(DPC), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE four1_dp
!BL
    SUBROUTINE four1_sp(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE four1_sp
  END INTERFACE
  INTERFACE
    SUBROUTINE four1_alt(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE four1_alt
  END INTERFACE
  INTERFACE
    SUBROUTINE four1_gather(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE four1_gather
  END INTERFACE
  INTERFACE
    SUBROUTINE four2(data,isign)
    USE nrtype
```

```
    COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
    INTEGER(I4B),INTENT(IN) :: isign
    END SUBROUTINE four2
  END INTERFACE
  INTERFACE
    SUBROUTINE four2_alt(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE four2_alt
  END INTERFACE
  INTERFACE
    SUBROUTINE four3(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
    INTEGER(I4B),INTENT(IN) :: isign
    END SUBROUTINE four3
  END INTERFACE
  INTERFACE
    SUBROUTINE four3_alt(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE four3_alt
  END INTERFACE
  INTERFACE
    SUBROUTINE fourcol(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE fourcol
  END INTERFACE
  INTERFACE
    SUBROUTINE fourcol_3d(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE fourcol_3d
  END INTERFACE
  INTERFACE
    SUBROUTINE fourn_gather(data,nn,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nn
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE fourn_gather
  END INTERFACE
  INTERFACE fourrow
    SUBROUTINE fourrow_dp(data,isign)
    USE nrtype
    COMPLEX(DPC), DIMENSION(:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE fourrow_dp
!BL
```

```
    SUBROUTINE fourrow_sp(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE fourrow_sp
  END INTERFACE
  INTERFACE
    SUBROUTINE fourrow_3d(data,isign)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:,:,:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE fourrow_3d
  END INTERFACE
  INTERFACE
    FUNCTION fpoly(x,n)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(n) :: fpoly
    END FUNCTION fpoly
  END INTERFACE
  INTERFACE
    SUBROUTINE fred2(a,b,t,f,w,g,ak)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(:), INTENT(OUT) :: t,f,w
    INTERFACE
      FUNCTION g(t)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(IN) :: t
      REAL(SP), DIMENSION(size(t)) :: g
      END FUNCTION g
!BL
      FUNCTION ak(t,s)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(IN) :: t,s
      REAL(SP), DIMENSION(size(t),size(s)) :: ak
      END FUNCTION ak
    END INTERFACE
    END SUBROUTINE fred2
  END INTERFACE
  INTERFACE
    FUNCTION fredin(x,a,b,t,f,w,g,ak)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,t,f,w
    REAL(SP), DIMENSION(size(x)) :: fredin
    INTERFACE
      FUNCTION g(t)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(IN) :: t
      REAL(SP), DIMENSION(size(t)) :: g
      END FUNCTION g
!BL
```

```
      FUNCTION ak(t,s)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(IN) :: t,s
      REAL(SP), DIMENSION(size(t),size(s)) :: ak
      END FUNCTION ak
    END INTERFACE
    END FUNCTION fredin
  END INTERFACE
  INTERFACE
    SUBROUTINE frenel(x,s,c)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: s,c
    END SUBROUTINE frenel
  END INTERFACE
  INTERFACE
    SUBROUTINE frprmn(p,ftol,iter,fret)
    USE nrtype
    INTEGER(I4B), INTENT(OUT) :: iter
    REAL(SP), INTENT(IN) :: ftol
    REAL(SP), INTENT(OUT) :: fret
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
    END SUBROUTINE frprmn
  END INTERFACE
  INTERFACE
    SUBROUTINE ftest(data1,data2,f,prob)
    USE nrtype
    REAL(SP), INTENT(OUT) :: f,prob
    REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
    END SUBROUTINE ftest
  END INTERFACE
  INTERFACE
    FUNCTION gamdev(ia)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: ia
    REAL(SP) :: gamdev
    END FUNCTION gamdev
  END INTERFACE
  INTERFACE gammln
    FUNCTION gammln_s(xx)
    USE nrtype
    REAL(SP), INTENT(IN) :: xx
    REAL(SP) :: gammln_s
    END FUNCTION gammln_s
!BL
    FUNCTION gammln_v(xx)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: xx
    REAL(SP), DIMENSION(size(xx)) :: gammln_v
    END FUNCTION gammln_v
  END INTERFACE
  INTERFACE gammp
    FUNCTION gammp_s(a,x)
    USE nrtype
```

```
    REAL(SP), INTENT(IN) :: a,x
    REAL(SP) :: gammp_s
    END FUNCTION gammp_s
!BL
    FUNCTION gammp_v(a,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
    REAL(SP), DIMENSION(size(a)) :: gammp_v
    END FUNCTION gammp_v
  END INTERFACE
  INTERFACE gammq
    FUNCTION gammq_s(a,x)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,x
    REAL(SP) :: gammq_s
    END FUNCTION gammq_s
!BL
    FUNCTION gammq_v(a,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
    REAL(SP), DIMENSION(size(a)) :: gammq_v
    END FUNCTION gammq_v
  END INTERFACE
  INTERFACE gasdev
    SUBROUTINE gasdev_s(harvest)
    USE nrtype
    REAL(SP), INTENT(OUT) :: harvest
    END SUBROUTINE gasdev_s
!BL
    SUBROUTINE gasdev_v(harvest)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
    END SUBROUTINE gasdev_v
  END INTERFACE
  INTERFACE
    SUBROUTINE gaucof(a,b,amu0,x,w)
    USE nrtype
    REAL(SP), INTENT(IN) :: amu0
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a,b
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gaucof
  END INTERFACE
  INTERFACE
    SUBROUTINE gauher(x,w)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gauher
  END INTERFACE
  INTERFACE
    SUBROUTINE gaujac(x,w,alf,bet)
    USE nrtype
    REAL(SP), INTENT(IN) :: alf,bet
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gaujac
```

```
  END INTERFACE
  INTERFACE
    SUBROUTINE gaulag(x,w,alf)
    USE nrtype
    REAL(SP), INTENT(IN) :: alf
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gaulag
  END INTERFACE
  INTERFACE
    SUBROUTINE gauleg(x1,x2,x,w)
    USE nrtype
    REAL(SP), INTENT(IN) :: x1,x2
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x,w
    END SUBROUTINE gauleg
  END INTERFACE
  INTERFACE
    SUBROUTINE gaussj(a,b)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a,b
    END SUBROUTINE gaussj
  END INTERFACE
  INTERFACE gcf
    FUNCTION gcf_s(a,x,gln)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,x
    REAL(SP), OPTIONAL, INTENT(OUT) :: gln
    REAL(SP) :: gcf_s
    END FUNCTION gcf_s
!BL
    FUNCTION gcf_v(a,x,gln)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
    REAL(SP), DIMENSION(:), OPTIONAL, INTENT(OUT) :: gln
    REAL(SP), DIMENSION(size(a)) :: gcf_v
    END FUNCTION gcf_v
  END INTERFACE
  INTERFACE
    FUNCTION golden(ax,bx,cx,func,tol,xmin)
    USE nrtype
    REAL(SP), INTENT(IN) :: ax,bx,cx,tol
    REAL(SP), INTENT(OUT) :: xmin
    REAL(SP) :: golden
    INTERFACE
      FUNCTION func(x)
      USE nrtype
      REAL(SP), INTENT(IN) :: x
      REAL(SP) :: func
      END FUNCTION func
    END INTERFACE
    END FUNCTION golden
  END INTERFACE
  INTERFACE gser
    FUNCTION gser_s(a,x,gln)
    USE nrtype
```

```
    REAL(SP), INTENT(IN) :: a,x
    REAL(SP), OPTIONAL, INTENT(OUT) :: gln
    REAL(SP) :: gser_s
    END FUNCTION gser_s
!BL
    FUNCTION gser_v(a,x,gln)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,x
    REAL(SP), DIMENSION(:), OPTIONAL, INTENT(OUT) :: gln
    REAL(SP), DIMENSION(size(a)) :: gser_v
    END FUNCTION gser_v
  END INTERFACE
  INTERFACE
    SUBROUTINE hqr(a,wr,wi)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: wr,wi
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    END SUBROUTINE hqr
  END INTERFACE
  INTERFACE
    SUBROUTINE hunt(xx,x,jlo)
    USE nrtype
    INTEGER(I4B), INTENT(INOUT) :: jlo
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: xx
    END SUBROUTINE hunt
  END INTERFACE
  INTERFACE
    SUBROUTINE hypdrv(s,ry,rdyds)
    USE nrtype
    REAL(SP), INTENT(IN) :: s
    REAL(SP), DIMENSION(:), INTENT(IN) :: ry
    REAL(SP), DIMENSION(:), INTENT(OUT) :: rdyds
    END SUBROUTINE hypdrv
  END INTERFACE
  INTERFACE
    FUNCTION hypgeo(a,b,c,z)
    USE nrtype
    COMPLEX(SPC), INTENT(IN) :: a,b,c,z
    COMPLEX(SPC) :: hypgeo
    END FUNCTION hypgeo
  END INTERFACE
  INTERFACE
    SUBROUTINE hypser(a,b,c,z,series,deriv)
    USE nrtype
    COMPLEX(SPC), INTENT(IN) :: a,b,c,z
    COMPLEX(SPC), INTENT(OUT) :: series,deriv
    END SUBROUTINE hypser
  END INTERFACE
  INTERFACE
    FUNCTION icrc(crc,buf,jinit,jrev)
    USE nrtype
    CHARACTER(1), DIMENSION(:), INTENT(IN) :: buf
    INTEGER(I2B), INTENT(IN) :: crc,jinit
```

```
  INTEGER(I4B), INTENT(IN) :: jrev
  INTEGER(I2B) :: icrc
  END FUNCTION icrc
END INTERFACE
INTERFACE
  FUNCTION igray(n,is)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: n,is
  INTEGER(I4B) :: igray
  END FUNCTION igray
END INTERFACE
INTERFACE
  RECURSIVE SUBROUTINE index_bypack(arr,index,partial)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: arr
  INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: index
  INTEGER, OPTIONAL, INTENT(IN) :: partial
  END SUBROUTINE index_bypack
END INTERFACE
INTERFACE indexx
  SUBROUTINE indexx_sp(arr,index)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: arr
  INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: index
  END SUBROUTINE indexx_sp
  SUBROUTINE indexx_i4b(iarr,index)
  USE nrtype
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: iarr
  INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: index
  END SUBROUTINE indexx_i4b
END INTERFACE
INTERFACE
  FUNCTION interp(uc)
  USE nrtype
  REAL(DP), DIMENSION(:,:), INTENT(IN) :: uc
  REAL(DP), DIMENSION(2*size(uc,1)-1,2*size(uc,1)-1) :: interp
  END FUNCTION interp
END INTERFACE
INTERFACE
  FUNCTION rank(indx)
  USE nrtype
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
  INTEGER(I4B), DIMENSION(size(indx)) :: rank
  END FUNCTION rank
END INTERFACE
INTERFACE
  FUNCTION irbit1(iseed)
  USE nrtype
  INTEGER(I4B), INTENT(INOUT) :: iseed
  INTEGER(I4B) :: irbit1
  END FUNCTION irbit1
END INTERFACE
INTERFACE
  FUNCTION irbit2(iseed)
```

```
  USE nrtype
  INTEGER(I4B), INTENT(INOUT) :: iseed
  INTEGER(I4B) :: irbit2
  END FUNCTION irbit2
END INTERFACE
INTERFACE
  SUBROUTINE jacobi(a,d,v,nrot)
  USE nrtype
  INTEGER(I4B), INTENT(OUT) :: nrot
  REAL(SP), DIMENSION(:), INTENT(OUT) :: d
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
  REAL(SP), DIMENSION(:,:), INTENT(OUT) :: v
  END SUBROUTINE jacobi
END INTERFACE
INTERFACE
  SUBROUTINE jacobn(x,y,dfdx,dfdy)
  USE nrtype
  REAL(SP), INTENT(IN) :: x
  REAL(SP), DIMENSION(:), INTENT(IN) :: y
  REAL(SP), DIMENSION(:), INTENT(OUT) :: dfdx
  REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dfdy
  END SUBROUTINE jacobn
END INTERFACE
INTERFACE
  FUNCTION julday(mm,id,iyyy)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: mm,id,iyyy
  INTEGER(I4B) :: julday
  END FUNCTION julday
END INTERFACE
INTERFACE
  SUBROUTINE kendl1(data1,data2,tau,z,prob)
  USE nrtype
  REAL(SP), INTENT(OUT) :: tau,z,prob
  REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
  END SUBROUTINE kendl1
END INTERFACE
INTERFACE
  SUBROUTINE kendl2(tab,tau,z,prob)
  USE nrtype
  REAL(SP), DIMENSION(:,:), INTENT(IN) :: tab
  REAL(SP), INTENT(OUT) :: tau,z,prob
  END SUBROUTINE kendl2
END INTERFACE
INTERFACE
  FUNCTION kermom(y,m)
  USE nrtype
  REAL(DP), INTENT(IN) :: y
  INTEGER(I4B), INTENT(IN) :: m
  REAL(DP), DIMENSION(m) :: kermom
  END FUNCTION kermom
END INTERFACE
INTERFACE
  SUBROUTINE ks2d1s(x1,y1,quadvl,d1,prob)
```

```
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: x1,y1
  REAL(SP), INTENT(OUT) :: d1,prob
  INTERFACE
    SUBROUTINE quadvl(x,y,fa,fb,fc,fd)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
    END SUBROUTINE quadvl
  END INTERFACE
  END SUBROUTINE ks2d1s
END INTERFACE
INTERFACE
  SUBROUTINE ks2d2s(x1,y1,x2,y2,d,prob)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: x1,y1,x2,y2
  REAL(SP), INTENT(OUT) :: d,prob
  END SUBROUTINE ks2d2s
END INTERFACE
INTERFACE
  SUBROUTINE ksone(data,func,d,prob)
  USE nrtype
  REAL(SP), INTENT(OUT) :: d,prob
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
  END INTERFACE
  END SUBROUTINE ksone
END INTERFACE
INTERFACE
  SUBROUTINE kstwo(data1,data2,d,prob)
  USE nrtype
  REAL(SP), INTENT(OUT) :: d,prob
  REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
  END SUBROUTINE kstwo
END INTERFACE
INTERFACE
  SUBROUTINE laguer(a,x,its)
  USE nrtype
  INTEGER(I4B), INTENT(OUT) :: its
  COMPLEX(SPC), INTENT(INOUT) :: x
  COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
  END SUBROUTINE laguer
END INTERFACE
INTERFACE
  SUBROUTINE lfit(x,y,sig,a,maska,covar,chisq,funcs)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sig
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
  LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
```

```
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: covar
  REAL(SP), INTENT(OUT) :: chisq
  INTERFACE
    SUBROUTINE funcs(x,arr)
    USE nrtype
    REAL(SP),INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(OUT) :: arr
    END SUBROUTINE funcs
  END INTERFACE
  END SUBROUTINE lfit
END INTERFACE
INTERFACE
  SUBROUTINE linbcg(b,x,itol,tol,itmax,iter,err)
  USE nrtype
  REAL(DP), DIMENSION(:), INTENT(IN) :: b
  REAL(DP), DIMENSION(:), INTENT(INOUT) :: x
  INTEGER(I4B), INTENT(IN) :: itol,itmax
  REAL(DP), INTENT(IN) :: tol
  INTEGER(I4B), INTENT(OUT) :: iter
  REAL(DP), INTENT(OUT) :: err
  END SUBROUTINE linbcg
END INTERFACE
INTERFACE
  SUBROUTINE linmin(p,xi,fret)
  USE nrtype
  REAL(SP), INTENT(OUT) :: fret
  REAL(SP), DIMENSION(:), TARGET, INTENT(INOUT) :: p,xi
  END SUBROUTINE linmin
END INTERFACE
INTERFACE
  SUBROUTINE lnsrch(xold,fold,g,p,x,f,stpmax,check,func)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: xold,g
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
  REAL(SP), INTENT(IN) :: fold,stpmax
  REAL(SP), DIMENSION(:), INTENT(OUT) :: x
  REAL(SP), INTENT(OUT) :: f
  LOGICAL(LGT), INTENT(OUT) :: check
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP) :: func
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    END FUNCTION func
  END INTERFACE
  END SUBROUTINE lnsrch
END INTERFACE
INTERFACE
  FUNCTION locatenr(xx,x)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: xx
  REAL(SP), INTENT(IN) :: x
  INTEGER(I4B) :: locatenr
  END FUNCTION locatenr
```

```
END INTERFACE
INTERFACE
  FUNCTION lop(u)
  USE nrtype
  REAL(DP), DIMENSION(:,:), INTENT(IN) :: u
  REAL(DP), DIMENSION(size(u,1),size(u,1)) :: lop
  END FUNCTION lop
END INTERFACE
INTERFACE
  SUBROUTINE lubksb(a,indx,b)
  USE nrtype
  REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
  END SUBROUTINE lubksb
END INTERFACE
INTERFACE
  SUBROUTINE ludcmp(a,indx,d)
  USE nrtype
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
  INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: indx
  REAL(SP), INTENT(OUT) :: d
  END SUBROUTINE ludcmp
END INTERFACE
INTERFACE
  SUBROUTINE machar(ibeta,it,irnd,ngrd,machep,negep,iexp,minexp,&
    maxexp,eps,epsneg,xmin,xmax)
  USE nrtype
  INTEGER(I4B), INTENT(OUT) :: ibeta,iexp,irnd,it,machep,maxexp,&
    minexp,negep,ngrd
  REAL(SP), INTENT(OUT) :: eps,epsneg,xmax,xmin
  END SUBROUTINE machar
END INTERFACE
INTERFACE
  SUBROUTINE medfit(x,y,a,b,abdev)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
  REAL(SP), INTENT(OUT) :: a,b,abdev
  END SUBROUTINE medfit
END INTERFACE
INTERFACE
  SUBROUTINE memcof(data,xms,d)
  USE nrtype
  REAL(SP), INTENT(OUT) :: xms
  REAL(SP), DIMENSION(:), INTENT(IN) :: data
  REAL(SP), DIMENSION(:), INTENT(OUT) :: d
  END SUBROUTINE memcof
END INTERFACE
INTERFACE
  SUBROUTINE mgfas(u,maxcyc)
  USE nrtype
  REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
  INTEGER(I4B), INTENT(IN) :: maxcyc
  END SUBROUTINE mgfas
```

```
END INTERFACE
INTERFACE
  SUBROUTINE mglin(u,ncycle)
  USE nrtype
  REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
  INTEGER(I4B), INTENT(IN) :: ncycle
  END SUBROUTINE mglin
END INTERFACE
INTERFACE
  SUBROUTINE midexp(funk,aa,bb,s,n)
  USE nrtype
  REAL(SP), INTENT(IN) :: aa,bb
  REAL(SP), INTENT(INOUT) :: s
  INTEGER(I4B), INTENT(IN) :: n
  INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
  END INTERFACE
  END SUBROUTINE midexp
END INTERFACE
INTERFACE
  SUBROUTINE midinf(funk,aa,bb,s,n)
  USE nrtype
  REAL(SP), INTENT(IN) :: aa,bb
  REAL(SP), INTENT(INOUT) :: s
  INTEGER(I4B), INTENT(IN) :: n
  INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
  END INTERFACE
  END SUBROUTINE midinf
END INTERFACE
INTERFACE
  SUBROUTINE midpnt(func,a,b,s,n)
  USE nrtype
  REAL(SP), INTENT(IN) :: a,b
  REAL(SP), INTENT(INOUT) :: s
  INTEGER(I4B), INTENT(IN) :: n
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
  END INTERFACE
  END SUBROUTINE midpnt
END INTERFACE
INTERFACE
```

```
  SUBROUTINE midsql(funk,aa,bb,s,n)
  USE nrtype
  REAL(SP), INTENT(IN) :: aa,bb
  REAL(SP), INTENT(INOUT) :: s
  INTEGER(I4B), INTENT(IN) :: n
  INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
  END INTERFACE
  END SUBROUTINE midsql
END INTERFACE
INTERFACE
  SUBROUTINE midsqu(funk,aa,bb,s,n)
  USE nrtype
  REAL(SP), INTENT(IN) :: aa,bb
  REAL(SP), INTENT(INOUT) :: s
  INTEGER(I4B), INTENT(IN) :: n
  INTERFACE
    FUNCTION funk(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: funk
    END FUNCTION funk
  END INTERFACE
  END SUBROUTINE midsqu
END INTERFACE
INTERFACE
  RECURSIVE SUBROUTINE miser(func,regn,ndim,npts,dith,ave,var)
  USE nrtype
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP) :: func
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    END FUNCTION func
  END INTERFACE
  REAL(SP), DIMENSION(:), INTENT(IN) :: regn
  INTEGER(I4B), INTENT(IN) :: ndim,npts
  REAL(SP), INTENT(IN) :: dith
  REAL(SP), INTENT(OUT) :: ave,var
  END SUBROUTINE miser
END INTERFACE
INTERFACE
  SUBROUTINE mmid(y,dydx,xs,htot,nstep,yout,derivs)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: nstep
  REAL(SP), INTENT(IN) :: xs,htot
  REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
  REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
  INTERFACE
    SUBROUTINE derivs(x,y,dydx)
```

```
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
  END INTERFACE
  END SUBROUTINE mmid
END INTERFACE
INTERFACE
  SUBROUTINE mnbrak(ax,bx,cx,fa,fb,fc,func)
  USE nrtype
  REAL(SP), INTENT(INOUT) :: ax,bx
  REAL(SP), INTENT(OUT) :: cx,fa,fb,fc
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
  END INTERFACE
  END SUBROUTINE mnbrak
END INTERFACE
INTERFACE
  SUBROUTINE mnewt(ntrial,x,tolx,tolf,usrfun)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: ntrial
  REAL(SP), INTENT(IN) :: tolx,tolf
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
  INTERFACE
    SUBROUTINE usrfun(x,fvec,fjac)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(OUT) :: fvec
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: fjac
    END SUBROUTINE usrfun
  END INTERFACE
  END SUBROUTINE mnewt
END INTERFACE
INTERFACE
  SUBROUTINE moment(data,ave,adev,sdev,var,skew,curt)
  USE nrtype
  REAL(SP), INTENT(OUT) :: ave,adev,sdev,var,skew,curt
  REAL(SP), DIMENSION(:), INTENT(IN) :: data
  END SUBROUTINE moment
END INTERFACE
INTERFACE
  SUBROUTINE mp2dfr(a,s,n,m)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: n
  INTEGER(I4B), INTENT(OUT) :: m
  CHARACTER(1), DIMENSION(:), INTENT(INOUT) :: a
  CHARACTER(1), DIMENSION(:), INTENT(OUT) :: s
  END SUBROUTINE mp2dfr
END INTERFACE
```

```
INTERFACE
  SUBROUTINE mpdiv(q,r,u,v,n,m)
  USE nrtype
  CHARACTER(1), DIMENSION(:), INTENT(OUT) :: q,r
  CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
  INTEGER(I4B), INTENT(IN) :: n,m
  END SUBROUTINE mpdiv
END INTERFACE
INTERFACE
  SUBROUTINE mpinv(u,v,n,m)
  USE nrtype
  CHARACTER(1), DIMENSION(:), INTENT(OUT) :: u
  CHARACTER(1), DIMENSION(:), INTENT(IN) :: v
  INTEGER(I4B), INTENT(IN) :: n,m
  END SUBROUTINE mpinv
END INTERFACE
INTERFACE
  SUBROUTINE mpmul(w,u,v,n,m)
  USE nrtype
  CHARACTER(1), DIMENSION(:), INTENT(IN) :: u,v
  CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w
  INTEGER(I4B), INTENT(IN) :: n,m
  END SUBROUTINE mpmul
END INTERFACE
INTERFACE
  SUBROUTINE mppi(n)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: n
  END SUBROUTINE mppi
END INTERFACE
INTERFACE
  SUBROUTINE mprove(a,alud,indx,b,x)
  USE nrtype
  REAL(SP), DIMENSION(:,:), INTENT(IN) :: a,alud
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indx
  REAL(SP), DIMENSION(:), INTENT(IN) :: b
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
  END SUBROUTINE mprove
END INTERFACE
INTERFACE
  SUBROUTINE mpsqrt(w,u,v,n,m)
  USE nrtype
  CHARACTER(1), DIMENSION(:), INTENT(OUT) :: w,u
  CHARACTER(1), DIMENSION(:), INTENT(IN) :: v
  INTEGER(I4B), INTENT(IN) :: n,m
  END SUBROUTINE mpsqrt
END INTERFACE
INTERFACE
  SUBROUTINE mrqcof(x,y,sig,a,maska,alpha,beta,chisq,funcs)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,a,sig
  REAL(SP), DIMENSION(:), INTENT(OUT) :: beta
  REAL(SP), DIMENSION(:,:), INTENT(OUT) :: alpha
  REAL(SP), INTENT(OUT) :: chisq
```

```
    LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
    INTERFACE
      SUBROUTINE funcs(x,a,yfit,dyda)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(IN) :: x,a
      REAL(SP), DIMENSION(:), INTENT(OUT) :: yfit
      REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dyda
      END SUBROUTINE funcs
    END INTERFACE
    END SUBROUTINE mrqcof
  END INTERFACE
  INTERFACE
    SUBROUTINE mrqmin(x,y,sig,a,maska,covar,alpha,chisq,funcs,alamda)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sig
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: covar,alpha
    REAL(SP), INTENT(OUT) :: chisq
    REAL(SP), INTENT(INOUT) :: alamda
    LOGICAL(LGT), DIMENSION(:), INTENT(IN) :: maska
    INTERFACE
      SUBROUTINE funcs(x,a,yfit,dyda)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(IN) :: x,a
      REAL(SP), DIMENSION(:), INTENT(OUT) :: yfit
      REAL(SP), DIMENSION(:,:), INTENT(OUT) :: dyda
      END SUBROUTINE funcs
    END INTERFACE
    END SUBROUTINE mrqmin
  END INTERFACE
  INTERFACE
    SUBROUTINE newt(x,check)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: x
    LOGICAL(LGT), INTENT(OUT) :: check
    END SUBROUTINE newt
  END INTERFACE
  INTERFACE
    SUBROUTINE odeint(ystart,x1,x2,eps,h1,hmin,derivs,rkqs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: ystart
    REAL(SP), INTENT(IN) :: x1,x2,eps,h1,hmin
    INTERFACE
      SUBROUTINE derivs(x,y,dydx)
      USE nrtype
      REAL(SP), INTENT(IN) :: x
      REAL(SP), DIMENSION(:), INTENT(IN) :: y
      REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
      END SUBROUTINE derivs
!BL
      SUBROUTINE rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
      REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
```

```
    REAL(SP), INTENT(INOUT) :: x
    REAL(SP), INTENT(IN) :: htry,eps
    REAL(SP), INTENT(OUT) :: hdid,hnext
      INTERFACE
      SUBROUTINE derivs(x,y,dydx)
        USE nrtype
        REAL(SP), INTENT(IN) :: x
        REAL(SP), DIMENSION(:), INTENT(IN) :: y
        REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
        END SUBROUTINE derivs
      END INTERFACE
    END SUBROUTINE rkqs
  END INTERFACE
  END SUBROUTINE odeint
END INTERFACE
INTERFACE
  SUBROUTINE orthog(anu,alpha,beta,a,b)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: anu,alpha,beta
  REAL(SP), DIMENSION(:), INTENT(OUT) :: a,b
  END SUBROUTINE orthog
END INTERFACE
INTERFACE
  SUBROUTINE pade(cof,resid)
  USE nrtype
  REAL(DP), DIMENSION(:), INTENT(INOUT) :: cof
  REAL(SP), INTENT(OUT) :: resid
  END SUBROUTINE pade
END INTERFACE
INTERFACE
  FUNCTION pccheb(d)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: d
  REAL(SP), DIMENSION(size(d)) :: pccheb
  END FUNCTION pccheb
END INTERFACE
INTERFACE
  SUBROUTINE pcshft(a,b,d)
  USE nrtype
  REAL(SP), INTENT(IN) :: a,b
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: d
  END SUBROUTINE pcshft
END INTERFACE
INTERFACE
  SUBROUTINE pearsn(x,y,r,prob,z)
  USE nrtype
  REAL(SP), INTENT(OUT) :: r,prob,z
  REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
  END SUBROUTINE pearsn
END INTERFACE
INTERFACE
  SUBROUTINE period(x,y,ofac,hifac,px,py,jmax,prob)
  USE nrtype
  INTEGER(I4B), INTENT(OUT) :: jmax
```

```
    REAL(SP), INTENT(IN) :: ofac,hifac
    REAL(SP), INTENT(OUT) :: prob
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    REAL(SP), DIMENSION(:), POINTER :: px,py
    END SUBROUTINE period
  END INTERFACE
  INTERFACE plgndr
    FUNCTION plgndr_s(l,m,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: l,m
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: plgndr_s
    END FUNCTION plgndr_s
!BL
    FUNCTION plgndr_v(l,m,x)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: l,m
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: plgndr_v
    END FUNCTION plgndr_v
  END INTERFACE
  INTERFACE
    FUNCTION poidev(xm)
    USE nrtype
    REAL(SP), INTENT(IN) :: xm
    REAL(SP) :: poidev
    END FUNCTION poidev
  END INTERFACE
  INTERFACE
    FUNCTION polcoe(x,y)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    REAL(SP), DIMENSION(size(x)) :: polcoe
    END FUNCTION polcoe
  END INTERFACE
  INTERFACE
    FUNCTION polcof(xa,ya)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
    REAL(SP), DIMENSION(size(xa)) :: polcof
    END FUNCTION polcof
  END INTERFACE
  INTERFACE
    SUBROUTINE poldiv(u,v,q,r)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: u,v
    REAL(SP), DIMENSION(:), INTENT(OUT) :: q,r
    END SUBROUTINE poldiv
  END INTERFACE
  INTERFACE
    SUBROUTINE polin2(x1a,x2a,ya,x1,x2,y,dy)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: ya
```

```
    REAL(SP), INTENT(IN) :: x1,x2
    REAL(SP), INTENT(OUT) :: y,dy
    END SUBROUTINE polin2
  END INTERFACE
  INTERFACE
    SUBROUTINE polint(xa,ya,x,y,dy)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: y,dy
    END SUBROUTINE polint
  END INTERFACE
  INTERFACE
    SUBROUTINE powell(p,xi,ftol,iter,fret)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: p
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: xi
    INTEGER(I4B), INTENT(OUT) :: iter
    REAL(SP), INTENT(IN) :: ftol
    REAL(SP), INTENT(OUT) :: fret
    END SUBROUTINE powell
  END INTERFACE
  INTERFACE
    FUNCTION predic(data,d,nfut)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: data,d
    INTEGER(I4B), INTENT(IN) :: nfut
    REAL(SP), DIMENSION(nfut) :: predic
    END FUNCTION predic
  END INTERFACE
  INTERFACE
    FUNCTION probks(alam)
    USE nrtype
    REAL(SP), INTENT(IN) :: alam
    REAL(SP) :: probks
    END FUNCTION probks
  END INTERFACE
  INTERFACE psdes
    SUBROUTINE psdes_s(lword,rword)
    USE nrtype
    INTEGER(I4B), INTENT(INOUT) :: lword,rword
    END SUBROUTINE psdes_s
!BL
    SUBROUTINE psdes_v(lword,rword)
    USE nrtype
    INTEGER(I4B), DIMENSION(:), INTENT(INOUT) :: lword,rword
    END SUBROUTINE psdes_v
  END INTERFACE
  INTERFACE
    SUBROUTINE pwt(a,isign)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    INTEGER(I4B), INTENT(IN) :: isign
    END SUBROUTINE pwt
```

```
  END INTERFACE
  INTERFACE
    SUBROUTINE pwtset(n)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    END SUBROUTINE pwtset
  END INTERFACE
  INTERFACE pythag
    FUNCTION pythag_dp(a,b)
    USE nrtype
    REAL(DP), INTENT(IN) :: a,b
    REAL(DP) :: pythag_dp
    END FUNCTION pythag_dp
!BL
    FUNCTION pythag_sp(a,b)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP) :: pythag_sp
    END FUNCTION pythag_sp
  END INTERFACE
  INTERFACE
    SUBROUTINE pzextr(iest,xest,yest,yz,dy)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: iest
    REAL(SP), INTENT(IN) :: xest
    REAL(SP), DIMENSION(:), INTENT(IN) :: yest
    REAL(SP), DIMENSION(:), INTENT(OUT) :: yz,dy
    END SUBROUTINE pzextr
  END INTERFACE
!!! FB:
!   INTERFACE
!     FUNCTION qgaus(func,a,b)
!     USE nrtype
!     REAL(SP), INTENT(IN) :: a,b
!     REAL(SP) :: qgaus
!     INTERFACE
!       FUNCTION func(x)
!       USE nrtype
!       REAL(SP), DIMENSION(:), INTENT(IN) :: x
!       REAL(SP), DIMENSION(size(x)) :: func
!       END FUNCTION func
!     END INTERFACE
!     END FUNCTION qgaus
!   END INTERFACE
!!! /FB
  INTERFACE
    SUBROUTINE qrdcmp(a,c,d,sing)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    REAL(SP), DIMENSION(:), INTENT(OUT) :: c,d
    LOGICAL(LGT), INTENT(OUT) :: sing
    END SUBROUTINE qrdcmp
  END INTERFACE
  INTERFACE
```

```
  FUNCTION qromb(func,a,b)
  USE nrtype
  REAL(SP), INTENT(IN) :: a,b
  REAL(SP) :: qromb
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
  END INTERFACE
  END FUNCTION qromb
END INTERFACE
INTERFACE
  FUNCTION qromo(func,a,b,choose)
  USE nrtype
  REAL(SP), INTENT(IN) :: a,b
  REAL(SP) :: qromo
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
  END INTERFACE
  INTERFACE
    SUBROUTINE choose(funk,aa,bb,s,n)
    USE nrtype
    REAL(SP), INTENT(IN) :: aa,bb
    REAL(SP), INTENT(INOUT) :: s
    INTEGER(I4B), INTENT(IN) :: n
    INTERFACE
      FUNCTION funk(x)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(IN) :: x
      REAL(SP), DIMENSION(size(x)) :: funk
      END FUNCTION funk
    END INTERFACE
    END SUBROUTINE choose
  END INTERFACE
  END FUNCTION qromo
END INTERFACE
INTERFACE
  SUBROUTINE qroot(p,b,c,eps)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: p
  REAL(SP), INTENT(INOUT) :: b,c
  REAL(SP), INTENT(IN) :: eps
  END SUBROUTINE qroot
END INTERFACE
INTERFACE
  SUBROUTINE qrsolv(a,c,d,b)
  USE nrtype
  REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
```

```
  REAL(SP), DIMENSION(:), INTENT(IN) :: c,d
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
  END SUBROUTINE qrsolv
END INTERFACE
INTERFACE
  SUBROUTINE qrupdt(r,qt,u,v)
  USE nrtype
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: r,qt
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: u
  REAL(SP), DIMENSION(:), INTENT(IN) :: v
  END SUBROUTINE qrupdt
END INTERFACE
INTERFACE
  FUNCTION qsimp(func,a,b)
  USE nrtype
  REAL(SP), INTENT(IN) :: a,b
  REAL(SP) :: qsimp
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
  END INTERFACE
  END FUNCTION qsimp
END INTERFACE
INTERFACE
  FUNCTION qtrap(func,a,b)
  USE nrtype
  REAL(SP), INTENT(IN) :: a,b
  REAL(SP) :: qtrap
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
  END INTERFACE
  END FUNCTION qtrap
END INTERFACE
INTERFACE
  SUBROUTINE quadct(x,y,xx,yy,fa,fb,fc,fd)
  USE nrtype
  REAL(SP), INTENT(IN) :: x,y
  REAL(SP), DIMENSION(:), INTENT(IN) :: xx,yy
  REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
  END SUBROUTINE quadct
END INTERFACE
INTERFACE
  SUBROUTINE quadmx(a)
  USE nrtype
  REAL(SP), DIMENSION(:,:), INTENT(OUT) :: a
  END SUBROUTINE quadmx
END INTERFACE
```

```
  INTERFACE
    SUBROUTINE quadvl(x,y,fa,fb,fc,fd)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
    END SUBROUTINE quadvl
  END INTERFACE
  INTERFACE
    FUNCTION ran(idum)
    INTEGER(selected_int_kind(9)), INTENT(INOUT) :: idum
    REAL :: ran
    END FUNCTION ran
  END INTERFACE
  INTERFACE ran0
    SUBROUTINE ran0_s(harvest)
    USE nrtype
    REAL(SP), INTENT(OUT) :: harvest
    END SUBROUTINE ran0_s
!BL
    SUBROUTINE ran0_v(harvest)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
    END SUBROUTINE ran0_v
  END INTERFACE
  INTERFACE ran1
    SUBROUTINE ran1_s(harvest)
    USE nrtype
    REAL(SP), INTENT(OUT) :: harvest
    END SUBROUTINE ran1_s
!BL
    SUBROUTINE ran1_v(harvest)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
    END SUBROUTINE ran1_v
  END INTERFACE
  INTERFACE ran2
    SUBROUTINE ran2_s(harvest)
    USE nrtype
    REAL(SP), INTENT(OUT) :: harvest
    END SUBROUTINE ran2_s
!BL
    SUBROUTINE ran2_v(harvest)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
    END SUBROUTINE ran2_v
  END INTERFACE
  INTERFACE ran3
    SUBROUTINE ran3_s(harvest)
    USE nrtype
    REAL(SP), INTENT(OUT) :: harvest
    END SUBROUTINE ran3_s
!BL
    SUBROUTINE ran3_v(harvest)
    USE nrtype
```

```
    REAL(SP), DIMENSION(:), INTENT(OUT) :: harvest
    END SUBROUTINE ran3_v
  END INTERFACE
  INTERFACE
    SUBROUTINE ratint(xa,ya,x,y,dy)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: y,dy
    END SUBROUTINE ratint
  END INTERFACE
  INTERFACE
    SUBROUTINE ratlsq(func,a,b,mm,kk,cof,dev)
    USE nrtype
    REAL(DP), INTENT(IN) :: a,b
    INTEGER(I4B), INTENT(IN) :: mm,kk
    REAL(DP), DIMENSION(:), INTENT(OUT) :: cof
    REAL(DP), INTENT(OUT) :: dev
    INTERFACE
      FUNCTION func(x)
      USE nrtype
      REAL(DP), DIMENSION(:), INTENT(IN) :: x
      REAL(DP), DIMENSION(size(x)) :: func
      END FUNCTION func
    END INTERFACE
    END SUBROUTINE ratlsq
  END INTERFACE
  INTERFACE ratval
    FUNCTION ratval_s(x,cof,mm,kk)
    USE nrtype
    REAL(DP), INTENT(IN) :: x
    INTEGER(I4B), INTENT(IN) :: mm,kk
    REAL(DP), DIMENSION(mm+kk+1), INTENT(IN) :: cof
    REAL(DP) :: ratval_s
    END FUNCTION ratval_s
!BL
    FUNCTION ratval_v(x,cof,mm,kk)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(IN) :: x
    INTEGER(I4B), INTENT(IN) :: mm,kk
    REAL(DP), DIMENSION(mm+kk+1), INTENT(IN) :: cof
    REAL(DP), DIMENSION(size(x)) :: ratval_v
    END FUNCTION ratval_v
  END INTERFACE
  INTERFACE rc
    FUNCTION rc_s(x,y)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP) :: rc_s
    END FUNCTION rc_s
!BL
    FUNCTION rc_v(x,y)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
```

```
    REAL(SP), DIMENSION(size(x)) :: rc_v
    END FUNCTION rc_v
  END INTERFACE
  INTERFACE rd
    FUNCTION rd_s(x,y,z)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y,z
    REAL(SP) :: rd_s
    END FUNCTION rd_s
!BL
    FUNCTION rd_v(x,y,z)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,z
    REAL(SP), DIMENSION(size(x)) :: rd_v
    END FUNCTION rd_v
  END INTERFACE
  INTERFACE realft
    SUBROUTINE realft_dp(data,isign,zdata)
    USE nrtype
    REAL(DP), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    COMPLEX(DPC), DIMENSION(:), OPTIONAL, TARGET :: zdata
    END SUBROUTINE realft_dp
!BL
    SUBROUTINE realft_sp(data,isign,zdata)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
    INTEGER(I4B), INTENT(IN) :: isign
    COMPLEX(SPC), DIMENSION(:), OPTIONAL, TARGET :: zdata
    END SUBROUTINE realft_sp
  END INTERFACE
  INTERFACE
    RECURSIVE FUNCTION recur1(a,b) RESULT(u)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,b
    REAL(SP), DIMENSION(size(a)) :: u
    END FUNCTION recur1
  END INTERFACE
  INTERFACE
    FUNCTION recur2(a,b,c)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c
    REAL(SP), DIMENSION(size(a)) :: recur2
    END FUNCTION recur2
  END INTERFACE
  INTERFACE
    SUBROUTINE relax(u,rhs)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: rhs
    END SUBROUTINE relax
  END INTERFACE
  INTERFACE
    SUBROUTINE relax2(u,rhs)
```

```
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: rhs
    END SUBROUTINE relax2
  END INTERFACE
  INTERFACE
  FUNCTION resid(u,rhs)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: u,rhs
    REAL(DP), DIMENSION(size(u,1),size(u,1)) :: resid
    END FUNCTION resid
  END INTERFACE
  INTERFACE rf
    FUNCTION rf_s(x,y,z)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y,z
    REAL(SP) :: rf_s
    END FUNCTION rf_s
!BL
    FUNCTION rf_v(x,y,z)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,z
    REAL(SP), DIMENSION(size(x)) :: rf_v
    END FUNCTION rf_v
  END INTERFACE
  INTERFACE rj
    FUNCTION rj_s(x,y,z,p)
    USE nrtype
    REAL(SP), INTENT(IN) :: x,y,z,p
    REAL(SP) :: rj_s
    END FUNCTION rj_s
!BL
    FUNCTION rj_v(x,y,z,p)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,z,p
    REAL(SP), DIMENSION(size(x)) :: rj_v
    END FUNCTION rj_v
  END INTERFACE
  INTERFACE
    SUBROUTINE rk4(y,dydx,x,h,yout,derivs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
    REAL(SP), INTENT(IN) :: x,h
    REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
    INTERFACE
      SUBROUTINE derivs(x,y,dydx)
      USE nrtype
      REAL(SP), INTENT(IN) :: x
      REAL(SP), DIMENSION(:), INTENT(IN) :: y
      REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
      END SUBROUTINE derivs
    END INTERFACE
    END SUBROUTINE rk4
  END INTERFACE
```

```
INTERFACE
  SUBROUTINE rkck(y,dydx,x,h,yout,yerr,derivs)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
  REAL(SP), INTENT(IN) :: x,h
  REAL(SP), DIMENSION(:), INTENT(OUT) :: yout,yerr
  INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
  END INTERFACE
  END SUBROUTINE rkck
END INTERFACE
INTERFACE
  SUBROUTINE rkdumb(vstart,x1,x2,nstep,derivs)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: vstart
  REAL(SP), INTENT(IN) :: x1,x2
  INTEGER(I4B), INTENT(IN) :: nstep
  INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
  END INTERFACE
  END SUBROUTINE rkdumb
END INTERFACE
INTERFACE
  SUBROUTINE rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
  REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
  REAL(SP), INTENT(INOUT) :: x
  REAL(SP), INTENT(IN) :: htry,eps
  REAL(SP), INTENT(OUT) :: hdid,hnext
  INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
  END INTERFACE
  END SUBROUTINE rkqs
END INTERFACE
INTERFACE
  SUBROUTINE rlft2(data,spec,speq,isign)
  USE nrtype
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: data
```

```
  COMPLEX(SPC), DIMENSION(:,:), INTENT(OUT) :: spec
  COMPLEX(SPC), DIMENSION(:), INTENT(OUT) :: speq
  INTEGER(I4B), INTENT(IN) :: isign
  END SUBROUTINE rlft2
END INTERFACE
INTERFACE
  SUBROUTINE rlft3(data,spec,speq,isign)
  USE nrtype
  REAL(SP), DIMENSION(:,:,:), INTENT(INOUT) :: data
  COMPLEX(SPC), DIMENSION(:,:,:), INTENT(OUT) :: spec
  COMPLEX(SPC), DIMENSION(:,:), INTENT(OUT) :: speq
  INTEGER(I4B), INTENT(IN) :: isign
  END SUBROUTINE rlft3
END INTERFACE
INTERFACE
  SUBROUTINE rotate(r,qt,i,a,b)
  USE nrtype
  REAL(SP), DIMENSION(:,:), TARGET, INTENT(INOUT) :: r,qt
  INTEGER(I4B), INTENT(IN) :: i
  REAL(SP), INTENT(IN) :: a,b
  END SUBROUTINE rotate
END INTERFACE
INTERFACE
  SUBROUTINE rsolv(a,d,b)
  USE nrtype
  REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
  REAL(SP), DIMENSION(:), INTENT(IN) :: d
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: b
  END SUBROUTINE rsolv
END INTERFACE
INTERFACE
  FUNCTION rstrct(uf)
  USE nrtype
  REAL(DP), DIMENSION(:,:), INTENT(IN) :: uf
  REAL(DP), DIMENSION((size(uf,1)+1)/2,(size(uf,1)+1)/2) :: rstrct
  END FUNCTION rstrct
END INTERFACE
INTERFACE
  FUNCTION rtbis(func,x1,x2,xacc)
  USE nrtype
  REAL(SP), INTENT(IN) :: x1,x2,xacc
  REAL(SP) :: rtbis
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
  END INTERFACE
  END FUNCTION rtbis
END INTERFACE
INTERFACE
  FUNCTION rtflsp(func,x1,x2,xacc)
  USE nrtype
```

```
  REAL(SP), INTENT(IN) :: x1,x2,xacc
  REAL(SP) :: rtflsp
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
  END INTERFACE
  END FUNCTION rtflsp
END INTERFACE
INTERFACE
  FUNCTION rtnewt(funcd,x1,x2,xacc)
  USE nrtype
  REAL(SP), INTENT(IN) :: x1,x2,xacc
  REAL(SP) :: rtnewt
  INTERFACE
    SUBROUTINE funcd(x,fval,fderiv)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: fval,fderiv
    END SUBROUTINE funcd
  END INTERFACE
  END FUNCTION rtnewt
END INTERFACE
INTERFACE
  FUNCTION rtsafe(funcd,x1,x2,xacc)
  USE nrtype
  REAL(SP), INTENT(IN) :: x1,x2,xacc
  REAL(SP) :: rtsafe
  INTERFACE
    SUBROUTINE funcd(x,fval,fderiv)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: fval,fderiv
    END SUBROUTINE funcd
  END INTERFACE
  END FUNCTION rtsafe
END INTERFACE
INTERFACE
  FUNCTION rtsec(func,x1,x2,xacc)
  USE nrtype
  REAL(SP), INTENT(IN) :: x1,x2,xacc
  REAL(SP) :: rtsec
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
  END INTERFACE
  END FUNCTION rtsec
END INTERFACE
INTERFACE
```

```
  SUBROUTINE rzextr(iest,xest,yest,yz,dy)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: iest
  REAL(SP), INTENT(IN) :: xest
  REAL(SP), DIMENSION(:), INTENT(IN) :: yest
  REAL(SP), DIMENSION(:), INTENT(OUT) :: yz,dy
  END SUBROUTINE rzextr
END INTERFACE
INTERFACE
  FUNCTION savgol(nl,nrr,ld,m)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
  REAL(SP), DIMENSION(nl+nrr+1) :: savgol
  END FUNCTION savgol
END INTERFACE
INTERFACE
  SUBROUTINE scrsho(func)
  USE nrtype
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
  END INTERFACE
  END SUBROUTINE scrsho
END INTERFACE
INTERFACE
  FUNCTION select(k,arr)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: k
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
  REAL(SP) :: select
  END FUNCTION select
END INTERFACE
INTERFACE
  FUNCTION select_bypack(k,arr)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: k
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
  REAL(SP) :: select_bypack
  END FUNCTION select_bypack
END INTERFACE
INTERFACE
  SUBROUTINE select_heap(arr,heap)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: arr
  REAL(SP), DIMENSION(:), INTENT(OUT) :: heap
  END SUBROUTINE select_heap
END INTERFACE
INTERFACE
  FUNCTION select_inplace(k,arr)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: k
```

```
  REAL(SP), DIMENSION(:), INTENT(IN) :: arr
  REAL(SP) :: select_inplace
  END FUNCTION select_inplace
END INTERFACE
INTERFACE
  SUBROUTINE simplx(a,m1,m2,m3,icase,izrov,iposv)
  USE nrtype
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
  INTEGER(I4B), INTENT(IN) :: m1,m2,m3
  INTEGER(I4B), INTENT(OUT) :: icase
  INTEGER(I4B), DIMENSION(:), INTENT(OUT) :: izrov,iposv
  END SUBROUTINE simplx
END INTERFACE
INTERFACE
  SUBROUTINE simpr(y,dydx,dfdx,dfdy,xs,htot,nstep,yout,derivs)
  USE nrtype
  REAL(SP), INTENT(IN) :: xs,htot
  REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx,dfdx
  REAL(SP), DIMENSION(:,:), INTENT(IN) :: dfdy
  INTEGER(I4B), INTENT(IN) :: nstep
  REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
  INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
  END INTERFACE
  END SUBROUTINE simpr
END INTERFACE
INTERFACE
  SUBROUTINE sinft(y)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
  END SUBROUTINE sinft
END INTERFACE
INTERFACE
  SUBROUTINE slvsm2(u,rhs)
  USE nrtype
  REAL(DP), DIMENSION(3,3), INTENT(OUT) :: u
  REAL(DP), DIMENSION(3,3), INTENT(IN) :: rhs
  END SUBROUTINE slvsm2
END INTERFACE
INTERFACE
  SUBROUTINE slvsml(u,rhs)
  USE nrtype
  REAL(DP), DIMENSION(3,3), INTENT(OUT) :: u
  REAL(DP), DIMENSION(3,3), INTENT(IN) :: rhs
  END SUBROUTINE slvsml
END INTERFACE
INTERFACE
  SUBROUTINE sncndn(uu,emmc,sn,cn,dn)
  USE nrtype
```

```
  REAL(SP), INTENT(IN) :: uu,emmc
  REAL(SP), INTENT(OUT) :: sn,cn,dn
  END SUBROUTINE sncndn
END INTERFACE
INTERFACE
  FUNCTION snrm(sx,itol)
  USE nrtype
  REAL(DP), DIMENSION(:), INTENT(IN) :: sx
  INTEGER(I4B), INTENT(IN) :: itol
  REAL(DP) :: snrm
  END FUNCTION snrm
END INTERFACE
INTERFACE
  SUBROUTINE sobseq(x,init)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(OUT) :: x
  INTEGER(I4B), OPTIONAL, INTENT(IN) :: init
  END SUBROUTINE sobseq
END INTERFACE
INTERFACE
  SUBROUTINE solvde(itmax,conv,slowc,scalv,indexv,nb,y)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: itmax,nb
  REAL(SP), INTENT(IN) :: conv,slowc
  REAL(SP), DIMENSION(:), INTENT(IN) :: scalv
  INTEGER(I4B), DIMENSION(:), INTENT(IN) :: indexv
  REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: y
  END SUBROUTINE solvde
END INTERFACE
INTERFACE
  SUBROUTINE sor(a,b,c,d,e,f,u,rjac)
  USE nrtype
  REAL(DP), DIMENSION(:,:), INTENT(IN) :: a,b,c,d,e,f
  REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: u
  REAL(DP), INTENT(IN) :: rjac
  END SUBROUTINE sor
END INTERFACE
INTERFACE
  SUBROUTINE sort(arr)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
  END SUBROUTINE sort
END INTERFACE
INTERFACE
  SUBROUTINE sort2(arr,slave)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr,slave
  END SUBROUTINE sort2
END INTERFACE
INTERFACE
  SUBROUTINE sort3(arr,slave1,slave2)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr,slave1,slave2
  END SUBROUTINE sort3
```

```
END INTERFACE
INTERFACE
  SUBROUTINE sort_bypack(arr)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
  END SUBROUTINE sort_bypack
END INTERFACE
INTERFACE
  SUBROUTINE sort_byreshape(arr)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
  END SUBROUTINE sort_byreshape
END INTERFACE
INTERFACE
  SUBROUTINE sort_heap(arr)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
  END SUBROUTINE sort_heap
END INTERFACE
INTERFACE
  SUBROUTINE sort_pick(arr)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
  END SUBROUTINE sort_pick
END INTERFACE
INTERFACE
  SUBROUTINE sort_radix(arr)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
  END SUBROUTINE sort_radix
END INTERFACE
INTERFACE
  SUBROUTINE sort_shell(arr)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
  END SUBROUTINE sort_shell
END INTERFACE
INTERFACE
  SUBROUTINE spctrm(p,k,ovrlap,unit,n_window)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(OUT) :: p
  INTEGER(I4B), INTENT(IN) :: k
  LOGICAL(LGT), INTENT(IN) :: ovrlap
  INTEGER(I4B), OPTIONAL, INTENT(IN) :: n_window,unit
  END SUBROUTINE spctrm
END INTERFACE
INTERFACE
  SUBROUTINE spear(data1,data2,d,zd,probd,rs,probrs)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
  REAL(SP), INTENT(OUT) :: d,zd,probd,rs,probrs
  END SUBROUTINE spear
END INTERFACE
INTERFACE sphbes
```

```
    SUBROUTINE sphbes_s(n,x,sj,sy,sjp,syp)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), INTENT(IN) :: x
    REAL(SP), INTENT(OUT) :: sj,sy,sjp,syp
    END SUBROUTINE sphbes_s
!BL
    SUBROUTINE sphbes_v(n,x,sj,sy,sjp,syp)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(OUT) :: sj,sy,sjp,syp
    END SUBROUTINE sphbes_v
  END INTERFACE
  INTERFACE
    SUBROUTINE splie2(x1a,x2a,ya,y2a)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: ya
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: y2a
    END SUBROUTINE splie2
  END INTERFACE
  INTERFACE
    FUNCTION splin2(x1a,x2a,ya,y2a,x1,x2)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x1a,x2a
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: ya,y2a
    REAL(SP), INTENT(IN) :: x1,x2
    REAL(SP) :: splin2
    END FUNCTION splin2
  END INTERFACE
  INTERFACE
    SUBROUTINE spline(x,y,yp1,ypn,y2)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
    REAL(SP), INTENT(IN) :: yp1,ypn
    REAL(SP), DIMENSION(:), INTENT(OUT) :: y2
    END SUBROUTINE spline
  END INTERFACE
  INTERFACE
    FUNCTION splint(xa,ya,y2a,x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya,y2a
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: splint
    END FUNCTION splint
  END INTERFACE
  INTERFACE sprsax
    SUBROUTINE sprsax_dp(sa,x,b)
    USE nrtype
    TYPE(sprs2_dp), INTENT(IN) :: sa
    REAL(DP), DIMENSION (:), INTENT(IN) :: x
    REAL(DP), DIMENSION (:), INTENT(OUT) :: b
    END SUBROUTINE sprsax_dp
```

```
!BL
    SUBROUTINE sprsax_sp(sa,x,b)
    USE nrtype
    TYPE(sprs2_sp), INTENT(IN) :: sa
    REAL(SP), DIMENSION (:), INTENT(IN) :: x
    REAL(SP), DIMENSION (:), INTENT(OUT) :: b
    END SUBROUTINE sprsax_sp
  END INTERFACE
  INTERFACE sprsdiag
    SUBROUTINE sprsdiag_dp(sa,b)
    USE nrtype
    TYPE(sprs2_dp), INTENT(IN) :: sa
    REAL(DP), DIMENSION(:), INTENT(OUT) :: b
    END SUBROUTINE sprsdiag_dp
!BL
    SUBROUTINE sprsdiag_sp(sa,b)
    USE nrtype
    TYPE(sprs2_sp), INTENT(IN) :: sa
    REAL(SP), DIMENSION(:), INTENT(OUT) :: b
    END SUBROUTINE sprsdiag_sp
  END INTERFACE
  INTERFACE sprsin
    SUBROUTINE sprsin_sp(a,thresh,sa)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: a
    REAL(SP), INTENT(IN) :: thresh
    TYPE(sprs2_sp), INTENT(OUT) :: sa
    END SUBROUTINE sprsin_sp
!BL
    SUBROUTINE sprsin_dp(a,thresh,sa)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: a
    REAL(DP), INTENT(IN) :: thresh
    TYPE(sprs2_dp), INTENT(OUT) :: sa
    END SUBROUTINE sprsin_dp
  END INTERFACE
  INTERFACE
    SUBROUTINE sprstp(sa)
    USE nrtype
    TYPE(sprs2_sp), INTENT(INOUT) :: sa
    END SUBROUTINE sprstp
  END INTERFACE
  INTERFACE sprstx
    SUBROUTINE sprstx_dp(sa,x,b)
    USE nrtype
    TYPE(sprs2_dp), INTENT(IN) :: sa
    REAL(DP), DIMENSION (:), INTENT(IN) :: x
    REAL(DP), DIMENSION (:), INTENT(OUT) :: b
    END SUBROUTINE sprstx_dp
!BL
    SUBROUTINE sprstx_sp(sa,x,b)
    USE nrtype
    TYPE(sprs2_sp), INTENT(IN) :: sa
    REAL(SP), DIMENSION (:), INTENT(IN) :: x
```

```
  REAL(SP), DIMENSION (:), INTENT(OUT) :: b
  END SUBROUTINE sprstx_sp
END INTERFACE
INTERFACE
  SUBROUTINE stifbs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
  REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
  REAL(SP), INTENT(IN) :: htry,eps
  REAL(SP), INTENT(INOUT) :: x
  REAL(SP), INTENT(OUT) :: hdid,hnext
  INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
  END INTERFACE
  END SUBROUTINE stifbs
END INTERFACE
INTERFACE
  SUBROUTINE stiff(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
  REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
  REAL(SP), INTENT(INOUT) :: x
  REAL(SP), INTENT(IN) :: htry,eps
  REAL(SP), INTENT(OUT) :: hdid,hnext
  INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
  END INTERFACE
  END SUBROUTINE stiff
END INTERFACE
INTERFACE
  SUBROUTINE stoerm(y,d2y,xs,htot,nstep,yout,derivs)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: y,d2y
  REAL(SP), INTENT(IN) :: xs,htot
  INTEGER(I4B), INTENT(IN) :: nstep
  REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
  INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
  END INTERFACE
```

```
    END SUBROUTINE stoerm
  END INTERFACE
  INTERFACE svbksb
    SUBROUTINE svbksb_dp(u,w,v,b,x)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(IN) :: u,v
    REAL(DP), DIMENSION(:), INTENT(IN) :: w,b
    REAL(DP), DIMENSION(:), INTENT(OUT) :: x
    END SUBROUTINE svbksb_dp
!BL
    SUBROUTINE svbksb_sp(u,w,v,b,x)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: u,v
    REAL(SP), DIMENSION(:), INTENT(IN) :: w,b
    REAL(SP), DIMENSION(:), INTENT(OUT) :: x
    END SUBROUTINE svbksb_sp
  END INTERFACE
  INTERFACE svdcmp
    SUBROUTINE svdcmp_dp(a,w,v)
    USE nrtype
    REAL(DP), DIMENSION(:,:), INTENT(INOUT) :: a
    REAL(DP), DIMENSION(:), INTENT(OUT) :: w
    REAL(DP), DIMENSION(:,:), INTENT(OUT) :: v
    END SUBROUTINE svdcmp_dp
!BL
    SUBROUTINE svdcmp_sp(a,w,v)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    REAL(SP), DIMENSION(:), INTENT(OUT) :: w
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: v
    END SUBROUTINE svdcmp_sp
  END INTERFACE
  INTERFACE
    SUBROUTINE svdfit(x,y,sig,a,v,w,chisq,funcs)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x,y,sig
    REAL(SP), DIMENSION(:), INTENT(OUT) :: a,w
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: v
    REAL(SP), INTENT(OUT) :: chisq
    INTERFACE
      FUNCTION funcs(x,n)
      USE nrtype
      REAL(SP), INTENT(IN) :: x
      INTEGER(I4B), INTENT(IN) :: n
      REAL(SP), DIMENSION(n) :: funcs
      END FUNCTION funcs
    END INTERFACE
    END SUBROUTINE svdfit
  END INTERFACE
  INTERFACE
    SUBROUTINE svdvar(v,w,cvm)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(IN) :: v
    REAL(SP), DIMENSION(:), INTENT(IN) :: w
```

```
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: cvm
    END SUBROUTINE svdvar
  END INTERFACE
  INTERFACE
    FUNCTION toeplz(r,y)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: r,y
    REAL(SP), DIMENSION(size(y)) :: toeplz
    END FUNCTION toeplz
  END INTERFACE
  INTERFACE
    SUBROUTINE tptest(data1,data2,t,prob)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
    REAL(SP), INTENT(OUT) :: t,prob
    END SUBROUTINE tptest
  END INTERFACE
  INTERFACE
    SUBROUTINE tqli(d,e,z)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: d,e
    REAL(SP), DIMENSION(:,:), OPTIONAL, INTENT(INOUT) :: z
    END SUBROUTINE tqli
  END INTERFACE
  INTERFACE
    SUBROUTINE trapzd(func,a,b,s,n)
    USE nrtype
    REAL(SP), INTENT(IN) :: a,b
    REAL(SP), INTENT(INOUT) :: s
    INTEGER(I4B), INTENT(IN) :: n
    INTERFACE
      FUNCTION func(x)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(IN) :: x
      REAL(SP), DIMENSION(size(x)) :: func
      END FUNCTION func
    END INTERFACE
    END SUBROUTINE trapzd
  END INTERFACE
  INTERFACE
    SUBROUTINE tred2(a,d,e,novectors)
    USE nrtype
    REAL(SP), DIMENSION(:,:), INTENT(INOUT) :: a
    REAL(SP), DIMENSION(:), INTENT(OUT) :: d,e
    LOGICAL(LGT), OPTIONAL, INTENT(IN) :: novectors
    END SUBROUTINE tred2
  END INTERFACE
!  On a purely serial machine, for greater efficiency, remove
!  the generic name tridag from the following interface,
!  and put it on the next one after that.
  INTERFACE tridag
    RECURSIVE SUBROUTINE tridag_par(a,b,c,r,u)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
```

```
  REAL(SP), DIMENSION(:), INTENT(OUT) :: u
  END SUBROUTINE tridag_par
END INTERFACE
INTERFACE
  SUBROUTINE tridag_ser(a,b,c,r,u)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
  REAL(SP), DIMENSION(:), INTENT(OUT) :: u
  END SUBROUTINE tridag_ser
END INTERFACE
INTERFACE
  SUBROUTINE ttest(data1,data2,t,prob)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
  REAL(SP), INTENT(OUT) :: t,prob
  END SUBROUTINE ttest
END INTERFACE
INTERFACE
  SUBROUTINE tutest(data1,data2,t,prob)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
  REAL(SP), INTENT(OUT) :: t,prob
  END SUBROUTINE tutest
END INTERFACE
INTERFACE
  SUBROUTINE twofft(data1,data2,fft1,fft2)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
  COMPLEX(SPC), DIMENSION(:), INTENT(OUT) :: fft1,fft2
  END SUBROUTINE twofft
END INTERFACE
INTERFACE
  FUNCTION vander(x,q)
  USE nrtype
  REAL(DP), DIMENSION(:), INTENT(IN) :: x,q
  REAL(DP), DIMENSION(size(x)) :: vander
  END FUNCTION vander
END INTERFACE
INTERFACE
  SUBROUTINE vegas(region,func,init,ncall,itmx,nprn,tgral,sd,chi2a)
  USE nrtype
  REAL(SP), DIMENSION(:), INTENT(IN) :: region
  INTEGER(I4B), INTENT(IN) :: init,ncall,itmx,nprn
  REAL(SP), INTENT(OUT) :: tgral,sd,chi2a
  INTERFACE
    FUNCTION func(pt,wgt)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: pt
    REAL(SP), INTENT(IN) :: wgt
    REAL(SP) :: func
    END FUNCTION func
  END INTERFACE
  END SUBROUTINE vegas
END INTERFACE
```

```
  INTERFACE
    SUBROUTINE voltra(t0,h,t,f,g,ak)
    USE nrtype
    REAL(SP), INTENT(IN) :: t0,h
    REAL(SP), DIMENSION(:), INTENT(OUT) :: t
    REAL(SP), DIMENSION(:,:), INTENT(OUT) :: f
    INTERFACE
      FUNCTION g(t)
      USE nrtype
      REAL(SP), INTENT(IN) :: t
      REAL(SP), DIMENSION(:), POINTER :: g
      END FUNCTION g
!BL
      FUNCTION ak(t,s)
      USE nrtype
      REAL(SP), INTENT(IN) :: t,s
      REAL(SP), DIMENSION(:,:), POINTER :: ak
      END FUNCTION ak
    END INTERFACE
    END SUBROUTINE voltra
  END INTERFACE
  INTERFACE
    SUBROUTINE wt1(a,isign,wtstep)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    INTEGER(I4B), INTENT(IN) :: isign
    INTERFACE
      SUBROUTINE wtstep(a,isign)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
      INTEGER(I4B), INTENT(IN) :: isign
      END SUBROUTINE wtstep
    END INTERFACE
    END SUBROUTINE wt1
  END INTERFACE
  INTERFACE
    SUBROUTINE wtn(a,nn,isign,wtstep)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
    INTEGER(I4B), DIMENSION(:), INTENT(IN) :: nn
    INTEGER(I4B), INTENT(IN) :: isign
    INTERFACE
      SUBROUTINE wtstep(a,isign)
      USE nrtype
      REAL(SP), DIMENSION(:), INTENT(INOUT) :: a
      INTEGER(I4B), INTENT(IN) :: isign
      END SUBROUTINE wtstep
    END INTERFACE
    END SUBROUTINE wtn
  END INTERFACE
  INTERFACE
    FUNCTION wwghts(n,h,kermom)
    USE nrtype
    INTEGER(I4B), INTENT(IN) :: n
```

```
  REAL(SP), INTENT(IN) :: h
  REAL(SP), DIMENSION(n) :: wwghts
  INTERFACE
    FUNCTION kermom(y,m)
    USE nrtype
    REAL(DP), INTENT(IN) :: y
    INTEGER(I4B), INTENT(IN) :: m
    REAL(DP), DIMENSION(m) :: kermom
    END FUNCTION kermom
  END INTERFACE
  END FUNCTION wwghts
END INTERFACE
INTERFACE
  SUBROUTINE zbrac(func,x1,x2,succes)
  USE nrtype
  REAL(SP), INTENT(INOUT) :: x1,x2
  LOGICAL(LGT), INTENT(OUT) :: succes
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
  END INTERFACE
  END SUBROUTINE zbrac
END INTERFACE
INTERFACE
  SUBROUTINE zbrak(func,x1,x2,n,xb1,xb2,nb)
  USE nrtype
  INTEGER(I4B), INTENT(IN) :: n
  INTEGER(I4B), INTENT(OUT) :: nb
  REAL(SP), INTENT(IN) :: x1,x2
  REAL(SP), DIMENSION(:), POINTER :: xb1,xb2
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
  END INTERFACE
  END SUBROUTINE zbrak
END INTERFACE
INTERFACE
  FUNCTION zbrent(func,x1,x2,tol)
  USE nrtype
  REAL(SP), INTENT(IN) :: x1,x2,tol
  REAL(SP) :: zbrent
  INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), INTENT(IN) :: x
    REAL(SP) :: func
    END FUNCTION func
  END INTERFACE
```

```
    END FUNCTION zbrent
  END INTERFACE
  INTERFACE
    SUBROUTINE zrhqr(a,rtr,rti)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: a
    REAL(SP), DIMENSION(:), INTENT(OUT) :: rtr,rti
    END SUBROUTINE zrhqr
  END INTERFACE
  INTERFACE
    FUNCTION zriddr(func,x1,x2,xacc)
    USE nrtype
    REAL(SP), INTENT(IN) :: x1,x2,xacc
    REAL(SP) :: zriddr
    INTERFACE
      FUNCTION func(x)
      USE nrtype
      REAL(SP), INTENT(IN) :: x
      REAL(SP) :: func
      END FUNCTION func
    END INTERFACE
    END FUNCTION zriddr
  END INTERFACE
  INTERFACE
    SUBROUTINE zroots(a,roots,polish)
    USE nrtype
    COMPLEX(SPC), DIMENSION(:), INTENT(IN) :: a
    COMPLEX(SPC), DIMENSION(:), INTENT(OUT) :: roots
    LOGICAL(LGT), INTENT(IN) :: polish
    END SUBROUTINE zroots
  END INTERFACE
END MODULE nr



  SUBROUTINE rkck(y,dydx,x,h,yout,yerr,derivs)
  USE nrtype; USE nrutil, ONLY : assert_eq
  IMPLICIT NONE
  REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
  REAL(SP), INTENT(IN) :: x,h
  REAL(SP), DIMENSION(:), INTENT(OUT) :: yout,yerr
  INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
  END INTERFACE
  INTEGER(I4B) :: ndum
  REAL(SP), DIMENSION(size(y)) :: ak2,ak3,ak4,ak5,ak6,ytemp
  REAL(SP), PARAMETER :: A2=0.2_sp,A3=0.3_sp,A4=0.6_sp,A5=1.0_sp,&
    A6=0.875_sp,B21=0.2_sp,B31=3.0_sp/40.0_sp,B32=9.0_sp/40.0_sp,&
```

```
  B41=0.3_sp,B42=-0.9_sp,B43=1.2_sp,B51=-11.0_sp/54.0_sp,&
  B52=2.5_sp,B53=-70.0_sp/27.0_sp,B54=35.0_sp/27.0_sp,&
  B61=1631.0_sp/55296.0_sp,B62=175.0_sp/512.0_sp,&
  B63=575.0_sp/13824.0_sp,B64=44275.0_sp/110592.0_sp,&
  B65=253.0_sp/4096.0_sp,C1=37.0_sp/378.0_sp,&
  C3=250.0_sp/621.0_sp,C4=125.0_sp/594.0_sp,&
  C6=512.0_sp/1771.0_sp,DC1=C1-2825.0_sp/27648.0_sp,&
  DC3=C3-18575.0_sp/48384.0_sp,DC4=C4-13525.0_sp/55296.0_sp,&
  DC5=-277.0_sp/14336.0_sp,DC6=C6-0.25_sp
ndum=assert_eq(size(y),size(dydx),size(yout),size(yerr),'rkck')
ytemp=y+B21*h*dydx
call derivs(x+A2*h,ytemp,ak2)
ytemp=y+h*(B31*dydx+B32*ak2)
call derivs(x+A3*h,ytemp,ak3)
ytemp=y+h*(B41*dydx+B42*ak2+B43*ak3)
call derivs(x+A4*h,ytemp,ak4)
ytemp=y+h*(B51*dydx+B52*ak2+B53*ak3+B54*ak4)
call derivs(x+A5*h,ytemp,ak5)
ytemp=y+h*(B61*dydx+B62*ak2+B63*ak3+B64*ak4+B65*ak5)
call derivs(x+A6*h,ytemp,ak6)
yout=y+h*(C1*dydx+C3*ak3+C4*ak4+C6*ak6)
yerr=h*(DC1*dydx+DC3*ak3+DC4*ak4+DC5*ak5+DC6*ak6)
END SUBROUTINE rkck

SUBROUTINE rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : rkck
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(OUT) :: hdid,hnext
INTERFACE
  SUBROUTINE derivs(x,y,dydx)
  USE nrtype
  IMPLICIT NONE
  REAL(SP), INTENT(IN) :: x
  REAL(SP), DIMENSION(:), INTENT(IN) :: y
  REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
  END SUBROUTINE derivs
END INTERFACE
INTEGER(I4B) :: ndum
REAL(SP) :: errmax,h,htemp,xnew
REAL(SP), DIMENSION(size(y)) :: yerr,ytemp
REAL(SP), PARAMETER :: SAFETY=0.9_sp,PGROW=-0.2_sp,PSHRNK=-0.25_sp,&
  ERRCON=1.89e-4
ndum=assert_eq(size(y),size(dydx),size(yscal),'rkqs')
h=htry
do
  call rkck(y,dydx,x,h,ytemp,yerr,derivs)
  errmax=maxval(abs(yerr(:)/yscal(:)))/eps
  if (errmax <= 1.0) exit
  htemp=SAFETY*h*(errmax**PSHRNK)
```

```
  h=sign(max(abs(htemp),0.1_sp*abs(h)),h)
  xnew=x+h
  if (xnew == x) call nrerror('stepsize underflow in rkqs')
end do
if (errmax > ERRCON) then
  hnext=SAFETY*h*(errmax**PGROW)
else
  hnext=5.0_sp*h
end if
hdid=h
x=x+h
y(:)=ytemp(:)
END SUBROUTINE rkqs

SUBROUTINE mmid(y,dydx,xs,htot,nstep,yout,derivs)
USE nrtype; USE nrutil, ONLY : assert_eq,swap
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: nstep
REAL(SP), INTENT(IN) :: xs,htot
REAL(SP), DIMENSION(:), INTENT(IN) :: y,dydx
REAL(SP), DIMENSION(:), INTENT(OUT) :: yout
INTERFACE
  SUBROUTINE derivs(x,y,dydx)
  USE nrtype
  IMPLICIT NONE
  REAL(SP), INTENT(IN) :: x
  REAL(SP), DIMENSION(:), INTENT(IN) :: y
  REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
  END SUBROUTINE derivs
END INTERFACE
INTEGER(I4B) :: n,ndum
REAL(SP) :: h,h2,x
REAL(SP), DIMENSION(size(y)) :: ym,yn
ndum=assert_eq(size(y),size(dydx),size(yout),'mmid')
h=htot/nstep
ym=y
yn=y+h*dydx
x=xs+h
call derivs(x,yn,yout)
h2=2.0_sp*h
do n=2,nstep
  call swap(ym,yn)
  yn=yn+h2*yout
  x=x+h
  call derivs(x,yn,yout)
end do
yout=0.5_sp*(ym+yn+h*yout)
END SUBROUTINE mmid

SUBROUTINE pzextr(iest,xest,yest,yz,dy)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: iest
REAL(SP), INTENT(IN) :: xest
```

```
REAL(SP), DIMENSION(:), INTENT(IN) :: yest
REAL(SP), DIMENSION(:), INTENT(OUT) :: yz,dy
INTEGER(I4B), PARAMETER :: IEST_MAX=16
INTEGER(I4B) :: j,nv
INTEGER(I4B), SAVE :: nvold=-1
REAL(SP) :: delta,f1,f2
REAL(SP), DIMENSION(size(yz)) :: d,tmp,q
REAL(SP), DIMENSION(IEST_MAX), SAVE :: x
REAL(SP), DIMENSION(:,:), ALLOCATABLE, SAVE :: qcol
nv=assert_eq(size(yz),size(yest),size(dy),'pzextr')
if (iest > IEST_MAX) call &
  nrerror('pzextr: probable misuse, too much extrapolation')
if (nv /= nvold) then
  if (allocated(qcol)) deallocate(qcol)
  allocate(qcol(nv,IEST_MAX))
  nvold=nv
end if
x(iest)=xest
dy(:)=yest(:)
yz(:)=yest(:)
if (iest == 1) then
  qcol(:,1)=yest(:)
else
  d(:)=yest(:)
  do j=1,iest-1
    delta=1.0_sp/(x(iest-j)-xest)
    f1=xest*delta
    f2=x(iest-j)*delta
    q(:)=qcol(:,j)
    qcol(:,j)=dy(:)
    tmp(:)=d(:)-q(:)
    dy(:)=f1*tmp(:)
    d(:)=f2*tmp(:)
    yz(:)=yz(:)+dy(:)
  end do
  qcol(:,iest)=dy(:)
end if
END SUBROUTINE pzextr

SUBROUTINE bsstep(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,cumsum,iminloc,nrerror,&
  outerdiff,outerprod,upper_triangle
USE nr, ONLY : mmid,pzextr
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
REAL(SP), INTENT(INOUT) :: x
REAL(SP), INTENT(IN) :: htry,eps
REAL(SP), INTENT(OUT) :: hdid,hnext
INTERFACE
  SUBROUTINE derivs(x,y,dydx)
  USE nrtype
  IMPLICIT NONE
  REAL(SP), INTENT(IN) :: x
```

```
  REAL(SP), DIMENSION(:), INTENT(IN) :: y
  REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
  END SUBROUTINE derivs
END INTERFACE
INTEGER(I4B), PARAMETER :: IMAX=9, KMAXX=IMAX-1
REAL(SP), PARAMETER :: SAFE1=0.25_sp,SAFE2=0.7_sp,REDMAX=1.0e-5_sp,&
  REDMIN=0.7_sp,TINY=1.0e-30_sp,SCALMX=0.1_sp
INTEGER(I4B) :: k,km,ndum
INTEGER(I4B), DIMENSION(IMAX) :: nseq = (/ 2,4,6,8,10,12,14,16,18 /)
INTEGER(I4B), SAVE :: kopt,kmax
REAL(SP), DIMENSION(KMAXX,KMAXX), SAVE :: alf
REAL(SP), DIMENSION(KMAXX) :: err
REAL(SP), DIMENSION(IMAX), SAVE :: a
REAL(SP), SAVE :: epsold = -1.0_sp,xnew
REAL(SP) :: eps1,errmax,fact,h,red,scale,wrkmin,xest
REAL(SP), DIMENSION(size(y)) :: yerr,ysav,yseq
LOGICAL(LGT) :: reduct
LOGICAL(LGT), SAVE :: first=.true.
ndum=assert_eq(size(y),size(dydx),size(yscal),'bsstep')
if (eps /= epsold) then
  hnext=-1.0e29_sp
  xnew=-1.0e29_sp
  eps1=SAFE1*eps
  a(:)=cumsum(nseq,1)
  where (upper_triangle(KMAXX,KMAXX)) alf=eps1** &
    (outerdiff(a(2:),a(2:))/outerprod(arth( &
    3.0_sp,2.0_sp,KMAXX),(a(2:)-a(1)+1.0_sp)))
  epsold=eps
  do kopt=2,KMAXX-1
    if (a(kopt+1) > a(kopt)*alf(kopt-1,kopt)) exit
  end do
  kmax=kopt
end if
h=htry
ysav(:)=y(:)
if (h /= hnext .or. x /= xnew) then
  first=.true.
  kopt=kmax
end if
reduct=.false.
main_loop: do
  do k=1,kmax
    xnew=x+h
    if (xnew == x) call nrerror('step size underflow in bsstep')
    call mmid(ysav,dydx,x,h,nseq(k),yseq,derivs)
    xest=(h/nseq(k))**2
    call pzextr(k,xest,yseq,y,yerr)
    if (k /= 1) then
      errmax=maxval(abs(yerr(:)/yscal(:)))
      errmax=max(TINY,errmax)/eps
      km=k-1
      err(km)=(errmax/SAFE1)**(1.0_sp/(2*km+1))
    end if
    if (k /= 1 .and. (k >= kopt-1 .or. first)) then
```

```
      if (errmax < 1.0) exit main_loop
      if (k == kmax .or. k == kopt+1) then
        red=SAFE2/err(km)
        exit
      else if (k == kopt) then
        if (alf(kopt-1,kopt) < err(km)) then
          red=1.0_sp/err(km)
          exit
        end if
      else if (kopt == kmax) then
        if (alf(km,kmax-1) < err(km)) then
          red=alf(km,kmax-1)*SAFE2/err(km)
          exit
        end if
      else if (alf(km,kopt) < err(km)) then
        red=alf(km,kopt-1)/err(km)
        exit
      end if
    end if
  end do
  red=max(min(red,REDMIN),REDMAX)
  h=h*red
  reduct=.true.
end do main_loop
x=xnew
hdid=h
first=.false.
kopt=1+iminloc(a(2:km+1)*max(err(1:km),SCALMX))
scale=max(err(kopt-1),SCALMX)
wrkmin=scale*a(kopt)
hnext=h/scale
if (kopt >= k .and. kopt /= kmax .and. .not. reduct) then
  fact=max(scale/alf(kopt-1,kopt),SCALMX)
  if (a(kopt+1)*fact <= wrkmin) then
    hnext=h/fact
    kopt=kopt+1
  end if
end if
END SUBROUTINE bsstep

FUNCTION hypgeo(a,b,c,z)
USE nrtype
USE hypgeo_info
USE nr, ONLY : bsstep,hypdrv,hypser,odeint
IMPLICIT NONE
COMPLEX(SPC), INTENT(IN) :: a,b,c,z
COMPLEX(SPC) :: hypgeo
REAL(SP), PARAMETER :: EPS=1.0e-6_sp
COMPLEX(SPC), DIMENSION(2) :: y
REAL(SP), DIMENSION(4) :: ry
if (real(z)**2+aimag(z)**2 <= 0.25) then
  call hypser(a,b,c,z,hypgeo,y(2))
  RETURN
else if (real(z) < 0.0) then
```

```
    hypgeo_z0=cmplx(-0.5_sp,0.0_sp,kind=spc)
  else if (real(z) <= 1.0) then
    hypgeo_z0=cmplx(0.5_sp,0.0_sp,kind=spc)
  else
    hypgeo_z0=cmplx(0.0_sp,sign(0.5_sp,aimag(z)),kind=spc)
  end if
  hypgeo_aa=a
  hypgeo_bb=b
  hypgeo_cc=c
  hypgeo_dz=z-hypgeo_z0
  call hypser(hypgeo_aa,hypgeo_bb,hypgeo_cc,hypgeo_z0,y(1),y(2))
  ry(1:4:2)=real(y)
  ry(2:4:2)=aimag(y)
!  call odeint(ry,0.0_sp,1.0_sp,EPS,0.1_sp,0.0001_sp,hypdrv,bsstep)
  call odeint(ry,0.0_sp,1.0_sp,EPS,0.1_sp,0.000001_sp,hypdrv,bsstep) !!! FB
  y=cmplx(ry(1:4:2),ry(2:4:2),kind=spc)
  hypgeo=y(1)
  END FUNCTION hypgeo

  SUBROUTINE hypdrv(s,ry,rdyds)
  USE nrtype
  USE hypgeo_info
  IMPLICIT NONE
  REAL(SP), INTENT(IN) :: s
  REAL(SP), DIMENSION(:), INTENT(IN) :: ry
  REAL(SP), DIMENSION(:), INTENT(OUT) :: rdyds
  COMPLEX(SPC), DIMENSION(2) :: y,dyds
  COMPLEX(SPC) :: z
  y=cmplx(ry(1:4:2),ry(2:4:2),kind=spc)
  z=hypgeo_z0+s*hypgeo_dz
  dyds(1)=y(2)*hypgeo_dz
  dyds(2)=((hypgeo_aa*hypgeo_bb)*y(1)-(hypgeo_cc-&
    ((hypgeo_aa+hypgeo_bb)+1.0_sp)*z)*y(2))*hypgeo_dz/(z*(1.0_sp-z))
  rdyds(1:4:2)=real(dyds)
  rdyds(2:4:2)=aimag(dyds)
  END SUBROUTINE hypdrv

  SUBROUTINE hypser(a,b,c,z,series,deriv)
  USE nrtype; USE nrutil, ONLY : nrerror
  IMPLICIT NONE
  COMPLEX(SPC), INTENT(IN) :: a,b,c,z
  COMPLEX(SPC), INTENT(OUT) :: series,deriv
  INTEGER(I4B) :: n
  INTEGER(I4B), PARAMETER :: MAXIT=1000
  COMPLEX(SPC) :: aa,bb,cc,fac,temp
  deriv=cmplx(0.0_sp,0.0_sp,kind=spc)
  fac=cmplx(1.0_sp,0.0_sp,kind=spc)
  temp=fac
  aa=a
  bb=b
  cc=c
  do n=1,MAXIT
    fac=((aa*bb)/cc)*fac
    deriv=deriv+fac
```

```
    fac=fac*z/n
    series=temp+fac
    if (series == temp) RETURN
    temp=series
    aa=aa+1.0
    bb=bb+1.0
    cc=cc+1.0
  end do
  call nrerror('hypser: convergence failure')
  END SUBROUTINE hypser

  SUBROUTINE odeint(ystart,x1,x2,eps,h1,hmin,derivs,rkqs)
  USE nrtype; USE nrutil, ONLY : nrerror,reallocate
  USE ode_path
  IMPLICIT NONE
  REAL(SP), DIMENSION(:), INTENT(INOUT) :: ystart
  REAL(SP), INTENT(IN) :: x1,x2,eps,h1,hmin
  INTERFACE
    SUBROUTINE derivs(x,y,dydx)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x
    REAL(SP), DIMENSION(:), INTENT(IN) :: y
    REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
    END SUBROUTINE derivs
!BL
    SUBROUTINE rkqs(y,dydx,x,htry,eps,yscal,hdid,hnext,derivs)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(INOUT) :: y
    REAL(SP), DIMENSION(:), INTENT(IN) :: dydx,yscal
    REAL(SP), INTENT(INOUT) :: x
    REAL(SP), INTENT(IN) :: htry,eps
    REAL(SP), INTENT(OUT) :: hdid,hnext
    INTERFACE
      SUBROUTINE derivs(x,y,dydx)
      USE nrtype
      IMPLICIT NONE
      REAL(SP), INTENT(IN) :: x
      REAL(SP), DIMENSION(:), INTENT(IN) :: y
      REAL(SP), DIMENSION(:), INTENT(OUT) :: dydx
      END SUBROUTINE derivs
    END INTERFACE
    END SUBROUTINE rkqs
  END INTERFACE
  REAL(SP), PARAMETER :: TINY=1.0e-30_sp
  INTEGER(I4B), PARAMETER :: MAXSTP=10000
  INTEGER(I4B) :: nstp
  REAL(SP) :: h,hdid,hnext,x,xsav
  REAL(SP), DIMENSION(size(ystart)) :: dydx,y,yscal
  x=x1
  h=sign(h1,x2-x1)
  nok=0
  nbad=0
```

```
  kount=0
  y(:)=ystart(:)
  nullify(xp,yp)
  if (save_steps) then
    xsav=x-2.0_sp*dxsav
    allocate(xp(256))
    allocate(yp(size(ystart),size(xp)))
  end if
  do nstp=1,MAXSTP
    call derivs(x,y,dydx)
    yscal(:)=abs(y(:))+abs(h*dydx(:))+TINY
    if (save_steps .and. (abs(x-xsav) > abs(dxsav))) &
      call save_a_step
    if ((x+h-x2)*(x+h-x1) > 0.0) h=x2-x
    call rkqs(y,dydx,x,h,eps,yscal,hdid,hnext,derivs)
    if (hdid == h) then
      nok=nok+1
    else
      nbad=nbad+1
    end if
    if ((x-x2)*(x2-x1) >= 0.0) then
      ystart(:)=y(:)
      if (save_steps) call save_a_step
      RETURN
    end if
    if (abs(hnext) < hmin) then
print *, "abs(hnext) = ", abs(hnext)
print *, "hmin = ", hmin
      call nrerror('stepsize smaller than minimum in odeint')
    end if
    h=hnext
  end do
  call nrerror('too many steps in odeint')
  CONTAINS
!BL
  SUBROUTINE save_a_step
  kount=kount+1
  if (kount > size(xp)) then
    xp=>reallocate(xp,2*size(xp))
    yp=>reallocate(yp,size(yp,1),size(xp))
  end if
  xp(kount)=x
  yp(:,kount)=y(:)
  xsav=x
  END SUBROUTINE save_a_step
  END SUBROUTINE odeint

  FUNCTION gammln_s(xx)
  USE nrtype; USE nrutil, ONLY : arth,assert
  IMPLICIT NONE
  REAL(SP), INTENT(IN) :: xx
  REAL(SP) :: gammln_s
  REAL(DP) :: tmp,x
  REAL(DP) :: stp = 2.5066282746310005_dp
```

```
  REAL(DP), DIMENSION(6) :: coef = (/76.18009172947146_dp,&
    -86.50532032941677_dp,24.01409824083091_dp,&
    -1.231739572450155_dp,0.1208650973866179e-2_dp,&
    -0.5395239384953e-5_dp/)
  call assert(xx > 0.0, 'gammln_s arg')
  x=xx
  tmp=x+5.5_dp
  tmp=(x+0.5_dp)*log(tmp)-tmp
  gammln_s=tmp+log(stp*(1.000000000190015_dp+&
  sum(coef(:)/arth(x+1.0_dp,1.0_dp,size(coef))))/x)
  END FUNCTION gammln_s

  FUNCTION gammln_v(xx)
  USE nrtype; USE nrutil, ONLY: assert
  IMPLICIT NONE
  INTEGER(I4B) :: i
  REAL(SP), DIMENSION(:), INTENT(IN) :: xx
  REAL(SP), DIMENSION(size(xx)) :: gammln_v
  REAL(DP), DIMENSION(size(xx)) :: ser,tmp,x,y
  REAL(DP) :: stp = 2.5066282746310005_dp
  REAL(DP), DIMENSION(6) :: coef = (/76.18009172947146_dp,&
    -86.50532032941677_dp,24.01409824083091_dp,&
    -1.231739572450155_dp,0.1208650973866179e-2_dp,&
    -0.5395239384953e-5_dp/)
  if (size(xx) == 0) RETURN
  call assert(all(xx > 0.0), 'gammln_v arg')
  x=xx
  tmp=x+5.5_dp
  tmp=(x+0.5_dp)*log(tmp)-tmp
  ser=1.000000000190015_dp
  y=x
  do i=1,size(coef)
    y=y+1.0_dp
    ser=ser+coef(i)/y
  end do
  gammln_v=tmp+log(stp*ser/x)
  END FUNCTION gammln_v

!   FUNCTION qgaus(func,a,b)
!   USE nrtype
!   REAL(SP), INTENT(IN) :: a,b
!   REAL(SP) :: qgaus
!   INTERFACE
!     FUNCTION func(x)
!     USE nrtype
!     REAL(SP), DIMENSION(:), INTENT(IN) :: x
!     REAL(SP), DIMENSION(size(x)) :: func
!     END FUNCTION func
!   END INTERFACE
!   REAL(SP) :: xm,xr
!   REAL(SP), DIMENSION(5) :: dx, w = (/ 0.2955242247_sp,0.2692667193_sp,&
!       0.2190863625_sp,0.1494513491_sp,0.0666713443_sp /),&
!     x = (/ 0.1488743389_sp,0.4333953941_sp,0.6794095682_sp,&
!       0.8650633666_sp,0.9739065285_sp /)
```

```
!   xm=0.5_sp*(b+a)
!   xr=0.5_sp*(b-a)
!   dx(:)=xr*x(:)
!   qgaus=xr*sum(w(:)*(func(xm+dx)+func(xm-dx)))
!   END FUNCTION qgaus

  FUNCTION locatenr(xx,x)
  USE nrtype
  IMPLICIT NONE
  REAL(SP), DIMENSION(:), INTENT(IN) :: xx
  REAL(SP), INTENT(IN) :: x
  INTEGER(I4B) :: locatenr
  INTEGER(I4B) :: n,jl,jm,ju
  LOGICAL :: ascnd
  n=size(xx)
  ascnd = (xx(n) >= xx(1))
  jl=0
  ju=n+1
  do
    if (ju-jl <= 1) exit
    jm=(ju+jl)/2
    if (ascnd .eqv. (x >= xx(jm))) then
      jl=jm
    else
      ju=jm
    end if
  end do
  if (x == xx(1)) then
    locatenr=1
  else if (x == xx(n)) then
    locatenr=n-1
  else
    locatenr=jl
  end if
  END FUNCTION locatenr

  SUBROUTINE tridag_ser(a,b,c,r,u)
  USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
  IMPLICIT NONE
  REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
  REAL(SP), DIMENSION(:), INTENT(OUT) :: u
  REAL(SP), DIMENSION(size(b)) :: gam
  INTEGER(I4B) :: n,j
  REAL(SP) :: bet
  n=assert_eq((/size(a)+1,size(b),size(c)+1,size(r),size(u)/),'tridag_ser')
  bet=b(1)
  if (bet == 0.0) call nrerror('tridag_ser: Error at code stage 1')
  u(1)=r(1)/bet
  do j=2,n
    gam(j)=c(j-1)/bet
    bet=b(j)-a(j-1)*gam(j)
    if (bet == 0.0) &
      call nrerror('tridag_ser: Error at code stage 2')
    u(j)=(r(j)-a(j-1)*u(j-1))/bet
```

```
end do
do j=n-1,1,-1
  u(j)=u(j)-gam(j+1)*u(j+1)
end do
END SUBROUTINE tridag_ser

RECURSIVE SUBROUTINE tridag_par(a,b,c,r,u)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : tridag_ser
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: a,b,c,r
REAL(SP), DIMENSION(:), INTENT(OUT) :: u
INTEGER(I4B), PARAMETER :: NPAR_TRIDAG=4
INTEGER(I4B) :: n,n2,nm,nx
REAL(SP), DIMENSION(size(b)/2) :: y,q,piva
REAL(SP), DIMENSION(size(b)/2-1) :: x,z
REAL(SP), DIMENSION(size(a)/2) :: pivc
n=assert_eq((/size(a)+1,size(b),size(c)+1,size(r),size(u)/),'tridag_par')
if (n < NPAR_TRIDAG) then
  call tridag_ser(a,b,c,r,u)
else
  if (maxval(abs(b(1:n))) == 0.0) &
    call nrerror('tridag_par: possible singular matrix')
  n2=size(y)
  nm=size(pivc)
  nx=size(x)
  piva = a(1:n-1:2)/b(1:n-1:2)
  pivc = c(2:n-1:2)/b(3:n:2)
  y(1:nm) = b(2:n-1:2)-piva(1:nm)*c(1:n-2:2)-pivc*a(2:n-1:2)
  q(1:nm) = r(2:n-1:2)-piva(1:nm)*r(1:n-2:2)-pivc*r(3:n:2)
  if (nm < n2) then
    y(n2) = b(n)-piva(n2)*c(n-1)
    q(n2) = r(n)-piva(n2)*r(n-1)
  end if
  x = -piva(2:n2)*a(2:n-2:2)
  z = -pivc(1:nx)*c(3:n-1:2)
  call tridag_par(x,y,z,q,u(2:n:2))
  u(1) = (r(1)-c(1)*u(2))/b(1)
  u(3:n-1:2) = (r(3:n-1:2)-a(2:n-2:2)*u(2:n-2:2) &
               -c(3:n-1:2)*u(4:n:2))/b(3:n-1:2)
  if (nm == n2) u(n)=(r(n)-a(n-1)*u(n-1))/b(n)
end if
END SUBROUTINE tridag_par

SUBROUTINE spline(x,y,yp1,ypn,y2)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : tridag
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), INTENT(IN) :: yp1,ypn
REAL(SP), DIMENSION(:), INTENT(OUT) :: y2
INTEGER(I4B) :: n
REAL(SP), DIMENSION(size(x)) :: a,b,c,r
n=assert_eq(size(x),size(y),size(y2),'spline')
```

```
c(1:n-1)=x(2:n)-x(1:n-1)
r(1:n-1)=6.0_sp*((y(2:n)-y(1:n-1))/c(1:n-1))
r(2:n-1)=r(2:n-1)-r(1:n-2)
a(2:n-1)=c(1:n-2)
b(2:n-1)=2.0_sp*(c(2:n-1)+a(2:n-1))
b(1)=1.0
b(n)=1.0
if (yp1 > 0.99e30_sp) then
  r(1)=0.0
  c(1)=0.0
else
  r(1)=(3.0_sp/(x(2)-x(1)))*((y(2)-y(1))/(x(2)-x(1))-yp1)
  c(1)=0.5
end if
if (ypn > 0.99e30_sp) then
  r(n)=0.0
  a(n)=0.0
else
  r(n)=(-3.0_sp/(x(n)-x(n-1)))*((y(n)-y(n-1))/(x(n)-x(n-1))-ypn)
  a(n)=0.5
end if
call tridag(a(2:n),b(1:n),c(1:n-1),r(1:n),y2(1:n))
END SUBROUTINE spline

FUNCTION splint(xa,ya,y2a,x)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY: locatenr
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: xa,ya,y2a
REAL(SP), INTENT(IN) :: x
REAL(SP) :: splint
INTEGER(I4B) :: khi,klo,n
REAL(SP) :: a,b,h
n=assert_eq(size(xa),size(ya),size(y2a),'splint')
klo=max(min(locatenr(xa,x),n-1),1)
khi=klo+1
h=xa(khi)-xa(klo)
if (h == 0.0) call nrerror('bad xa input in splint')
a=(xa(khi)-x)/h
b=(x-xa(klo))/h
splint=a*ya(klo)+b*ya(khi)+((a**3-a)*y2a(klo)+(b**3-b)*y2a(khi))*(h**2)/6.0_sp
END FUNCTION splint

SUBROUTINE sort(arr)
USE nrtype; USE nrutil, ONLY : swap,nrerror
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(INOUT) :: arr
INTEGER(I4B), PARAMETER :: NN=15, NSTACK=50
REAL(SP) :: a
INTEGER(I4B) :: n,k,i,j,jstack,l,r
INTEGER(I4B), DIMENSION(NSTACK) :: istack
n=size(arr)
jstack=0
l=1
```

```
r=n
do
  if (r-l < NN) then
    do j=l+1,r
      a=arr(j)
      do i=j-1,l,-1
        if (arr(i) <= a) exit
        arr(i+1)=arr(i)
      end do
      arr(i+1)=a
    end do
    if (jstack == 0) RETURN
    r=istack(jstack)
    l=istack(jstack-1)
    jstack=jstack-2
  else
    k=(l+r)/2
    call swap(arr(k),arr(l+1))
    call swap(arr(l),arr(r),arr(l)>arr(r))
    call swap(arr(l+1),arr(r),arr(l+1)>arr(r))
    call swap(arr(l),arr(l+1),arr(l)>arr(l+1))
    i=l+1
    j=r
    a=arr(l+1)
    do
      do
        i=i+1
        if (arr(i) >= a) exit
      end do
      do
        j=j-1
        if (arr(j) <= a) exit
      end do
      if (j < i) exit
      call swap(arr(i),arr(j))
    end do
    arr(l+1)=arr(j)
    arr(j)=a
    jstack=jstack+2
    if (jstack > NSTACK) call nrerror('sort: NSTACK too small')
    if (r-i+1 >= j-l) then
      istack(jstack)=r
      istack(jstack-1)=i
      r=j-1
    else
      istack(jstack)=j-1
      istack(jstack-1)=l
      l=i
    end if
  end if
end do
END SUBROUTINE sort
```

```
!!! Whizard wrapper for NR tools
module nr_tools
  use kinds, only: default !NODEP!
  use nrtype, only: i4b, sp, spc !NODEP!
  use nr, only: gammln, hypgeo, locatenr, sort, spline, splint !NODEP!
  implicit none
  save
  private

  public :: nr_hypgeo, nr_gamma, nr_locate, nr_sort, nr_spline_t

  type :: nr_spline_t
    real(sp), dimension(:), allocatable :: xa, ya_re, ya_im, y2a_re, y2a_im
  contains
    procedure :: init => nr_spline_init
    procedure :: interpolate => nr_spline_interpolate
    procedure :: dealloc => nr_spline_dealloc
  end type nr_spline_t

contains

  function nr_hypgeo (a, b, c, d) result (h)
    complex(default), intent(in) :: a, b, c, d
    complex(default) :: h
    complex(spc) :: a_sp, b_sp, c_sp, d_sp
    a_sp = cmplx(a,kind=sp)
    b_sp = cmplx(b,kind=sp)
    c_sp = cmplx(c,kind=sp)
    d_sp = cmplx(d,kind=sp)
    h = cmplx( hypgeo (a_sp, b_sp, c_sp, d_sp) , kind=default )
  end function nr_hypgeo

  function nr_gamma (x) result (y)
    real(default), intent(in) :: x
    real(default) :: y
    y = real( exp(gammln(real(x,kind=sp))) , kind=default )
  end function nr_gamma

  function nr_locate (xa, x) result (pos)
    real(default), dimension(:), intent(in) :: xa
    real(default), intent(in) :: x
    integer :: pos
    pos = locatenr (real(xa,kind=sp), real(x,kind=sp))
  end function

!   function nr_qgaus (fun, pts) result (res)
!     real(default), dimension(:), intent(in) :: pts
!     complex(default) :: res
!     integer :: i_pts
!     real(sp) :: lo, hi, re, im
!     interface
!       function fun (x)
!         use kinds, only: default !NODEP!
```

```
!         real(default), intent(in) :: x
!         complex(default) :: fun
!       end function fun
!     end interface
!     res = 0.0_default
!     if ( size(pts) < 2 ) return
!     do i_pts=1, size(pts)-1
!       lo = real(pts(i_pts  ),kind=sp)
!       hi = real(pts(i_pts+1),kind=sp)
!       re = qgaus (fun_re, lo, hi)
!       im = qgaus (fun_im, lo, hi)
!       res = res + cmplx(re,im,kind=default)
!     end do
!     contains
!       function fun_re (xa_sp)
!         use kinds, only: default !NODEP!
!         use nrtype, only: sp !NODEP!
!         real(sp), dimension(:), intent(in) :: xa_sp
!         real(sp), dimension(size(xa_sp)) :: fun_re
!         real(default), dimension(size(xa_sp)) :: xa
!         integer :: ix
!         xa = real(xa_sp,kind=default)
!         fun_re = (/ (real(fun(xa(ix)),kind=sp), ix=1, size(xa)) /)
!       end function fun_re
!       function fun_im (xa_sp)
!         use kinds, only: default !NODEP!
!         use nrtype, only: sp !NODEP!
!         real(sp), dimension(:), intent(in) :: xa_sp
!         real(sp), dimension(size(xa_sp)) :: fun_im
!         real(default), dimension(size(xa_sp)) :: xa
!         integer :: ix
!         xa = real(xa_sp,kind=default)
!         fun_im = (/ (real(aimag(fun(xa(ix))),kind=sp), ix=1, size(xa)) /)
!       end function fun_im
!   end function nr_qgaus

  subroutine nr_sort (array)
    real(default), dimension(:), intent(inout) :: array
    real(sp), dimension(size(array)) :: array_sp
    array_sp = real(array,kind=sp)
    call sort (array_sp)
    array = real(array_sp,kind=default)
  end subroutine nr_sort

  subroutine nr_spline_init (spl, xa_in, ya_in)
    class(nr_spline_t), intent(inout) :: spl
    real(default), dimension(:), intent(in) :: xa_in
    complex(default), dimension(:), intent(in) :: ya_in
    integer :: n
    if ( allocated(spl%xa) ) then
      print *, "ERROR: nr_spline: init: already initialized!"
      stop
    end if
    n = size(xa_in)
```

```
    allocate( spl%xa(n) )
    allocate( spl%ya_re(n) )
    allocate( spl%ya_im(n) )
    allocate( spl%y2a_re(n) )
    allocate( spl%y2a_im(n) )
    spl%xa = real(xa_in,kind=sp)
    spl%ya_re = real(ya_in,kind=sp)
    spl%ya_im = real(aimag(ya_in),kind=sp)
    call spline (spl%xa, spl%ya_re, 1.e30, 1.e30, spl%y2a_re)
    call spline (spl%xa, spl%ya_im, 1.e30, 1.e30, spl%y2a_im)
  end subroutine nr_spline_init

  function nr_spline_interpolate (spl, x) result (y)
    complex(default) :: y
    class(nr_spline_t), intent(in) :: spl
    real(default), intent(in) :: x
    real(sp) :: y_re, y_im
    if ( .not.allocated(spl%xa) ) then
      print *, "ERROR: nr_spline: interpolate: not initialized!"
      stop
    end if
    y_re = splint (spl%xa, spl%ya_re, spl%y2a_re, real(x,kind=sp))
    y_im = splint (spl%xa, spl%ya_im, spl%y2a_im, real(x,kind=sp))
    y = cmplx(y_re,y_im,kind=default)
  end function nr_spline_interpolate

  subroutine nr_spline_dealloc (spl)
    class(nr_spline_t), intent(inout) :: spl
    if ( .not.allocated(spl%xa) ) then
      print *, "ERROR: nr_spline: dealloc: not initialized!"
      stop
    end if
    deallocate( spl%xa )
    deallocate( spl%ya_re )
    deallocate( spl%ya_im )
    deallocate( spl%y2a_re )
    deallocate( spl%y2a_im )
  end subroutine nr_spline_dealloc
end module nr_tools
```

⟨toppik.f⟩≡

```
! WHIZARD ⟨Version⟩ ⟨Date⟩

! TOPPIK code by M. Jezabek, T. Teubner (v1.1, 1992), T. Teubner (1998)
!
! FB: -commented out numerical recipes code for hypergeometric 2F1
!      included in hypgeo.f90;
!     -commented out unused function 'ZAPVQ1';
!     -replaced function 'cdabs' by 'abs';
!     -replaced function 'dimag' by 'aimag';
!     -replaced function 'dcmplx(,)' by 'cmplx(,,kind=kind(0d0))';
!     -replaced function 'dreal' by 'real';
!     -replaced function 'cdlog' by 'log';
!     -replaced PAUSE by PRINT statement to avoid compiler warning;
!     -initialized 'idum' explicitly as real to avoid compiler warning.
```

```
!     -modified 'adglg1', 'adglg2' and 'tttoppik' to catch unstable runs.
!
!!!
```

```
c    xcdeltc  :  constant of the delta(r)-
c                 [= constant in momentum space-] potential
c    xcdeltl  :  constant for the additional log(q^2/mu^2)-part of the
c                 delta-potential:
c                  xcdeltc*1 + xcdeltl*log(q^2/mu^2)
c    xcfullc  :  constant of the (p^2+q^2)/(p-q)^2-potential
c    xcfulll  :  constant for the additional log(q^2/mu^2)-part of the
c                 (p^2+q^2)/(p-q)^2-potential
c    xcrm2    :  constant of the 1/r^2-[1/|p-q|]-potential
c    xkincm   :  } kinetic corrections in the 0th order Green-function:
c    xkinca   :  }  G_0(p):=1/[E+iGamma_t-p^2/m_t]*(1+xkincm)+xkinca
c     !!! WATCH THE SIGN IN G_0 !!!
c    jknflg   :  flag for these kinetic corrections:
c                 0 : no kinetic corrections applied
c                 1 : kinetic corrections applied with cutoff xcutv
c                      for  xkinca  only
c                 2 : kinetic corrections applied with cutoff xcutv
c                      for  xkinca  AND  xkincm
c    jgcflg   :  flag for G_0(p) in the LS equation:
c                 0 (standard choice) : G_0(p) as given above
c                 1 (for TIPT)        : G_0(p) = G_c^{0}(p) the 0th
c                                          order Coulomb-Green-function
c                                          in analytical form; not for
c                                          momenta  p > 1000*topmass
c    xkincv   :  additional kinematic vertexcorrection in G_0, see below:
c    jvflg    :  flag for the additional vertexcorrection  xkincv  in the
c                 ``zeroth order'' G_0(p) in the LS-equation:
c                 0 : no correction, means  G = G_0 + G_0 int V G
c                      with G_0=1/[E+iGamma_t-p^2/m_t]*(1+xkincm)+xkinca
c                 1 : apply the correction in the LS equation as
c                      G = G_0 + xkincv*p^2/m_t^2/[E+iGamma_t-p^2/m_t] +
c                          G_0 int V G
c                     and correct the integral over Im[G(p)] to get sigma_tot
c                     from the optical theorem by the same factor.
c                     The cutoff  xcutv  is applied for these corrections.
c
c -- Output:
c
c    xim      :  R_{ttbar} from the imaginary part of the green
c                 function
c    xdi      :  R_{ttbar} form the integral over the momentum
c                 distribution (no cutoff but the numerical one here!!)
c    np       :  number of points used for the grid; fixed in tttoppik
c    xpp      :  1-dim array (max. 400 elements) giving the momenta of
c                 the Gauss-Legendre grid (pp(i) in the code)
c    xww      :  1-dim array (max. 400 elements) giving the corresponding
c                 Gauss-Legendre weights for the grid
c    xdsdp    :  1-dim array (max. 400 elements) giving the
c                 momentum distribution of top: d\sigma/dp,
c                  normalized to R,
c                  at the momenta of the Gauss-Legendre grid xpp(i)
c    zvfct    :  1-dim array (max. 400 elements) of COMPLEX*16 numbers
c                 giving the vertex function K(p), G(p)=K(p)*G_0(p)
c                 at the momenta of the grid
```

```
c
c ******************************************************************
c
c
          implicit none
          real*8
     u       pi,energy,vzero,eps,
     u       pp,
     u       tmass,tgamma,zmass,alphas,alamb5,
     u       wmass,wgamma,bmass,GFERMI,
     u       xx,critp,consde,
     u       w1,w2,sig1,sig2,const,
     u       gtpcor,etot,
     u       xenergy,xtm,xtg,xalphas,xscale,xc0,xc1,xc2,xim,xdi,
     u       xdsdp,xpp,xww,
     u       cplas,scale,c0,c1,c2,cdeltc,cdeltl,cfullc,cfulll,crm2,
     u       xcutn,dcut,xcutv,
     u       xp,xpmax,
     u       kincom,kincoa,kincov,xkincm,xkinca,xkincv,
     u       xcdeltc,xcdeltl,xcfullc,xcfulll,xcrm2
          complex*16 bb,gg,a1,a,g0,g0c,zvfct
          integer i,n,nmax,npot,np,gcflg,kinflg,jknflg,jgcflg,
     u            jvflg,vflag
          parameter (nmax=400)
          dimension pp(nmax), bb(nmax), xx(nmax), gg(nmax),
     u              w1(nmax), w2(nmax), a1(nmax),
     u              xdsdp(nmax),xpp(nmax),xww(nmax),zvfct(nmax)
c
          external a,gtpcor,g0,g0c
c
          common/ovalco/ pi, energy, vzero, eps, npot
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
          common/cplcns/cplas,scale,c0,c1,c2,
     u                  cdeltc,cdeltl,cfullc,cfulll,crm2
          common/mom/ xp,xpmax,dcut
          common/g0inf/kincom,kincoa,kincov,kinflg,gcflg,vflag
c
          pi=3.141592653589793238d0
c
c Number of points to evaluate on the integral equation
c  (<=400 and n mod 3 = 0 !!):
c         n=66
          n=360
          np=n
c
c For second order potential with free parameters:
c
          npot=5
c Internal accuracy for TOPPIK, the reachable limit may be smaller,
c  depending on the parameters. But increase in real accuracy only
c  in combination with large number of points.
          eps=1.d-3
c Some physical parameters:
```

```
          wgamma=2.07d0
          zmass=91.187d0
          wmass=80.33d0
          bmass=4.7d0
c
c Input:
          energy=xenergy
          tmass=xtm
          tgamma=xtg
          cplas=xalphas
          scale=xscale
          c0=xc0
          c1=xc1
          c2=xc2
          cdeltc=xcdeltc
          cdeltl=xcdeltl
          cfullc=xcfullc
          cfulll=xcfulll
          crm2=xcrm2
          kincom=xkincm
          kincoa=xkinca
          kincov=xkincv
          kinflg=jknflg
          gcflg=jgcflg
          vflag=jvflg
c
          alphas=xalphas
c
c Cut for divergent potential-terms for large momenta in the function vhat
c  and in the integrals a(p):
          dcut=xcutv
c
c Numerical Cutoff of all momenta (maximal momenta of the grid):
          xpmax=xcutn
          if (dcut.gt.xpmax) then
             write(*,*) ' dcut > xpmax  makes no sense! Stop.'
             stop
          endif
c
c Not needed for the fixed order potentials:
          alamb5=0.2d0
c
c      WRITE(*,*) 'INPUT TGAMMA=',TGAMMA
c Needed in subroutine GAMMAT:
          GFERMI=1.16637d-5
c           CALL GAMMAT
c           WRITE(*,*) 'CALCULATED TGAMMA=',TGAMMA
c
          etot=2.d0*tmass+energy
c
          if ((npot.eq.1).or.(npot.eq.3).or.(npot.eq.4).or.
     u         (npot.eq.5)) then
c For pure coulomb and fixed order potentials there is no delta-part:
             consde = 0.d0
```

```
          else if (npot.eq.2) then
c Initialize QCD-potential common-blocks and calculate constant multiplying
c  the delta-part of the 'qcutted' potential in momentum-space:
             call iniphc(1)
             call vqdelt(consde)
          else
             write (*,*) ' Potential not implemented! Stop.'
             stop
          endif
c Delta-part of potential is absorbed by subtracting vzero from the
c  original energy (shift from the potential to the free Hamiltonian):
          vzero = consde / (2.d0*pi)**3
c         write (*,*) 'vzero=', vzero
c
c Find x-values pp(i) and weigths w1(i) for the gaussian quadrature;
c  care about large number of points in the important intervals:
c       if (energy-vzero.le.0.d0) then
cc        call gauleg(0.d0, 1.d0, pp, w1, n/3)
cc        call gauleg(1.d0, 5.d0, pp(n/3+1), w1(n/3+1), n/3)
cc        call gauleg(0.d0, 0.2d0, pp(2*n/3+1), w1(2*n/3+1), n/3)
c         call gauleg(0.d0, 5.d0, pp, w1, n/3)
c         call gauleg(5.d0, 20.d0, pp(n/3+1), w1(n/3+1), n/3)
c         call gauleg(0.d0, 0.05d0, pp(2*n/3+1), w1(2*n/3+1), n/3)
c       else
cc Avoid numerical singular points in the inner of the intervals:
c         critp = dsqrt((energy-vzero)*tmass)
c         if (critp.le.1.d0) then
cc Gauss-Legendre is symmetric => automatically principal-value prescription:
c            call gauleg(0.d0, 2.d0*critp, pp, w1, n/3)
c            call gauleg(2.d0*critp, 20.d0, pp(n/3+1),
c     u                  w1(n/3+1), n/3)
c            call gauleg(0.d0, 0.05d0, pp(2*n/3+1), w1(2*n/3+1), n/3)
c         else
cc Better behaviour at the border of the intervals:
c            call gauleg(0.d0, critp, pp, w1, n/3)
c            call gauleg(critp, 2.d0*critp, pp(n/3+1),
c     u                  w1(n/3+1), n/3)
c            call gauleg(0.d0, 1.d0/(2.d0*critp), pp(2*n/3+1),
c     u                  w1(2*n/3+1), n/3)
c         endif
c       endif
c
c Or different (simpler) method, good for V_JKT:
          if (energy.le.0.d0) then
             critp=tmass/3.d0
          else
             critp=max(tmass/3.d0,2.d0*dsqrt(energy*tmass))
          endif
          call gauleg(0.d0, critp, pp, w1, 2*n/3)
          call gauleg(1.d0/xpmax, 1.d0/critp, pp(2*n/3+1),
     u                w1(2*n/3+1), n/3)
c
c Do substitution p => 1/p for the last interval explicitly:
          do 10 i=2*n/3+1,n
```

```
               pp(i) = 1.d0/pp(i)
10          continue
c
c Reorder the arrays for the third interval:
            do 20 i=1,n/3
               xx(i) = pp(2*n/3+i)
               w2(i) = w1(2*n/3+i)
20          continue
            do 30 i=1,n/3
               pp(n-i+1) = xx(i)
               w1(n-i+1) = w2(i)
30          continue
c
c Calculate the integrals a(p) for the given momenta pp(i)
c  and store weights and momenta for the output arrays:
            do 40 i=1,n
               a1(i) = a(pp(i)) !!! FB: can get stuck in original Toppik!
               !!! FB: abuse 'np' as a flag to communicate unstable runs
               if ( abs(a1(i)) .gt. 1d10 ) then
                 np = -1
                 return
               endif
               xpp(i)=pp(i)
               xww(i)=w1(i)
40          continue
            do 41 i=n+1,nmax
               xpp(i)=0.d0
               xww(i)=0.d0
41          continue
c
c Solve the integral-equation by solving a system of algebraic equations:
            call sae(pp, w1, bb, a1, n)
c
c (The substitution for the integration to infinity  pp => 1/pp
c  is done already.)
            do 50 i=1,n
               zvfct(i)=bb(i)
               gg(i) = bb(i)*g0c(pp(i))
cc             gg(i) = (1.d0 + bb(i))*g0c(pp(i))
cc Urspruenglich anderes (Minus) VZ hier, dafuer kein Minus mehr bei der
cc  Definition des WQs ueber Im G, 2.6.1998, tt.
cc             gg(i) = - (1.d0 + bb(i))*g0c(pp(i))
50          continue
c
c Normalisation on R:
            const = 8.d0*pi/tmass**2
c
c Proove of the optical theorem for the output values of sae:
c  Simply check if sig1 = sig2.
            sig1 = 0.d0
            sig2 = 0.d0
            do 60 i=1,n*2/3
c              write(*,*) 'check! p(',i,') = ',pp(i)
cvv
```

```
              if (pp(i).lt.dcut.and.vflag.eq.1) then
                 sig1 = sig1 + w1(i)*pp(i)**2*aimag(gg(i)
cc     u                 *(1.d0+kincov*(pp(i)/tmass)**2)
     u   *(1.d0+kincov*g0(pp(i))*(pp(i)/tmass)**2/g0c(pp(i)))
     u                 )
              else
                 sig1 = sig1 + w1(i)*pp(i)**2*aimag(gg(i))
              endif
              if (pp(i).lt.dcut.and.kinflg.ne.0) then
                 sig2 = sig2 + w1(i)*pp(i)**2*abs(gg(i))**2 *
     u                 tgamma*gtpcor(pp(i),etot)
     u                 *(1.d0-pp(i)**2/2.d0/tmass**2)
cc     u                   *tmass/dsqrt(tmass**2+pp(i)**2)
                 xdsdp(i)=pp(i)**2*abs(gg(i))**2 *
     u                 tgamma*gtpcor(pp(i),etot)
     u                 *(1.d0-pp(i)**2/2.d0/tmass**2)
     u                 /(2.d0*pi**2)*const
              else
                 sig2 = sig2 + w1(i)*pp(i)**2*abs(gg(i))**2 *
     u                 tgamma*gtpcor(pp(i),etot)
                 xdsdp(i)=pp(i)**2*abs(gg(i))**2 *
     u                 tgamma*gtpcor(pp(i),etot)
     u                 /(2.d0*pi**2)*const
              endif
c             write(*,*) 'xdsdp = ',xdsdp(i)
c             write(*,*) 'zvfct = ',zvfct(i)
60         continue
c '*p**2' because of substitution p => 1/p in the integration of p**2*G(p)
c  to infinity
           do 70 i=n*2/3+1,n
c             write(*,*) 'check! p(',i,') = ',pp(i)
cvv
              if (pp(i).lt.dcut.and.vflag.eq.1) then
                 sig1 = sig1 + w1(i)*pp(i)**4*aimag(gg(i)
cc     u                 *(1.d0+kincov*(pp(i)/tmass)**2)
     u   *(1.d0+kincov*g0(pp(i))*(pp(i)/tmass)**2/g0c(pp(i)))
     u                 )
              else
                 sig1 = sig1 + w1(i)*pp(i)**4*aimag(gg(i))
              endif
              if (pp(i).lt.dcut.and.kinflg.ne.0) then
                 sig2 = sig2 + w1(i)*pp(i)**4*abs(gg(i))**2 *
     u                 tgamma*gtpcor(pp(i),etot)
     u                 *(1.d0-pp(i)**2/2.d0/tmass**2)
cc     u                   *tmass/dsqrt(tmass**2+pp(i)**2)
                 xdsdp(i)=pp(i)**2*abs(gg(i))**2 *
     u                 tgamma*gtpcor(pp(i),etot)
     u                 *(1.d0-pp(i)**2/2.d0/tmass**2)
     u                 /(2.d0*pi**2)*const
              else
                 sig2 = sig2 + w1(i)*pp(i)**4*abs(gg(i))**2 *
     u                 tgamma*gtpcor(pp(i),etot)
                 xdsdp(i)=pp(i)**2*abs(gg(i))**2 *
     u                 tgamma*gtpcor(pp(i),etot)
```

```
     u                  /(2.d0*pi**2)*const
              endif
c
c             write(*,*) 'xdsdp = ',xdsdp(i)
c             write(*,*) 'zvfct = ',zvfct(i)
70         continue
           do 71 i=n+1,nmax
             xdsdp(i)=0.d0
             zvfct(i)=(0.d0,0.d0)
71         continue
c
c Normalisation on R:
           sig1  = sig1 / (2.d0*pi**2) * const
           sig2  = sig2 / (2.d0*pi**2) * const
c
c The results from the momentum space approach finally are:
cc Jetzt Minus hier, 2.6.98, tt.
           xim=-sig1
           xdi=sig2
c
        end
c
c
        complex*16 function g0(p)
c
           implicit none
           real*8
     u        tmass,tgamma,zmass,alphas,alamb5,
     u        wmass,wgamma,bmass,GFERMI,
     u        pi,energy,vzero,eps,
     u        p,gtpcor
           integer npot
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
           common/ovalco/ pi, energy, vzero, eps, npot
           external gtpcor
           save
           g0=1.d0/cmplx(energy-vzero-p**2/tmass,
     u                   tgamma*gtpcor(p,2.d0*tmass+energy),
     u                   kind=kind(0d0))
        end
c
        complex*16 function g0c(p)
c
           implicit none
           complex*16 hypgeo,green,zk,zi,amd2k,aa,bb,cc,zzp,zzm,
     u                hypp,hypm,g0
           real*8
     u        tmass,tgamma,zmass,alphas,alamb5,
     u        wmass,wgamma,bmass,GFERMI,
     u        pi,energy,vzero,eps,
     u        p,gtpcor,
     u        kincom,kincoa,kincov,xp,xpmax,dcut
           integer npot,kinflg,gcflg,vflag
```

```
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
            common/ovalco/ pi, energy, vzero, eps, npot
            common/g0inf/kincom,kincoa,kincov,kinflg,gcflg,vflag
            common/mom/ xp,xpmax,dcut
            external hypgeo,gtpcor,g0
            save
c
            if (gcflg.eq.0) then
               if (kinflg.eq.0) then
                  g0c=g0(p)
               else if (kinflg.eq.1.and.p.lt.dcut) then
                  g0c=g0(p)*(1.d0+kincom)+kincoa
               else if (kinflg.eq.1.and.p.ge.dcut) then
                  g0c=g0(p)*(1.d0+kincom)
               else if (kinflg.eq.2.and.p.lt.dcut) then
                  g0c=g0(p)*(1.d0+kincom)+kincoa
               else if (kinflg.eq.2.and.p.ge.dcut) then
                  g0c=g0(p)
               else
                  write(*,*) ' kinflg wrong! Stop.'
                  stop
               endif
            else if (gcflg.eq.1) then
               zi=(0.d0,1.d0)
               zk=-tmass*cmplx(energy,tgamma
     u                         *gtpcor(p,2.d0*tmass+energy),
     u                         kind=kind(0d0))
               zk=sqrt(zk)
               amd2k=4.d0/3.d0*alphas*tmass/2.d0/zk
               aa=(2.d0,0.d0)
               bb=(1.d0,0.d0)
               cc=2.d0-amd2k
               zzp=(1.d0+zi*p/zk)/2.d0
               zzm=(1.d0-zi*p/zk)/2.d0
               if (abs(zzp).gt.20.d0) then
                  hypp=(1.d0-zzp)**(-aa)*
     u                 hypgeo(aa,cc-bb,cc,zzp/(zzp-1.d0))
               else
                  hypp=hypgeo(aa,bb,cc,zzp)
               endif
               if (abs(zzm).gt.20.d0) then
                  hypm=(1.d0-zzm)**(-aa)*
     u                 hypgeo(aa,cc-bb,cc,zzm/(zzm-1.d0))
               else
                  hypm=hypgeo(aa,bb,cc,zzm)
               endif
               green=-zi*tmass/(4.d0*p*zk)/(1.d0-amd2k)*(hypp-hypm)
c VZ anders herum als in Andres Konvention, da bei ihm G_0=1/[-E-i G+p^2/m]:
               g0c=-green
               if (p.gt.1.d3*tmass) then
                  write(*,*) ' g0cana = ',g0c,' not reliable. Stop.'
                  stop
               endif
```

```
            else
               write(*,*) ' gcflg wrong! Stop.'
               stop
            endif
c
         end
c
c
         complex*16 function a(p)
c
            implicit none
            real*8
     u         tmass,tgamma,zmass,alphas,alamb5,
     u         wmass,wgamma,bmass,GFERMI,
     u         pi, energy,ETOT,vzero, eps,
     $         QCUT,QMAT1,ALR,PCUT,
     u         p,
     u         xp,xpmax, xb1,xb2,dcut,ddcut,
     u         a1, a2, a3, a4,a5,a6,
     u         adglg1, fretil1, fretil2, fimtil1, fimtil2,
     u         ALEFVQ, gtpcor, ad8gle, buf,adglg2,
c     u         xerg,
     u         kincom,kincoa,kincov
!           complex*16 zapvq1,ZAPVGP
            complex*16 ZAPVGP !!! FB
c     u                ,acomp
            integer npot,ILFLAG,kinflg,gcflg,vflag
c
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      COMMON/PARFLG/ QCUT,QMAT1,ALR,ILFLAG
            common/ovalco/ pi, energy, vzero, eps, npot
            common/mom/ xp,xpmax,dcut
            common/g0inf/kincom,kincoa,kincov,kinflg,gcflg,vflag
c
            external adglg1, fretil1, fretil2, fimtil1, fimtil2,
!     u              zapvq1, ALEFVQ, gtpcor,ZAPVGP,ad8gle,adglg2
     u              ALEFVQ, gtpcor,ZAPVGP,ad8gle,adglg2 !!! FB
c
            if ((npot.eq.1).or.(npot.eq.3).or.(npot.eq.4).or.
     u         (npot.eq.5)) then
c
               xp=p
               buf=0.d0
c
               a1=0.d0
               a2=0.d0
               a3=0.d0
               a4=0.d0
               a5=0.d0
               a6=0.d0
               if (gcflg.eq.0) then
                  ddcut=xpmax
               else if (gcflg.eq.1) then
```

```
                 ddcut=dcut
              else
                 write(*,*) ' gcflg wrong! Stop.'
                 stop
              endif
c
              if (2.d0*xp.lt.ddcut) then
                 xb1=xp
                 xb2=2.d0*xp
c
c More stable for logarithmically divergent fixed order potentials:
c
                 a1=adglg1(fretil1, buf, xb1, eps) !!! FB: can get stuck!
                 a2=adglg1(fimtil1, buf, xb1, eps)
c Slightly unstable:
                 a3=adglg2(fretil1,xb1,xb2,eps) !!! FB: can get stuck!
c No good:
c                a3=adglg1(fretil1,xb1,xb2,eps)
c Not better:
c                call adqua(xb1,xb2,fretil1,xerg,eps)
c                a3=xerg
c Also not better:
c                a1=adglg1(fretil1, buf, xb2, eps)
c
                 a4=adglg2(fimtil1,xb1,xb2,eps)
c                a5 = adglg2(fretil1, xb2, ddcut, eps)
c                a6 = adglg2(fimtil1, xb2, ddcut, eps)
                 a5 = adglg2(fretil2, 1.d0/ddcut, 1.d0/xb2, eps)
                 a6 = adglg2(fimtil2, 1.d0/ddcut, 1.d0/xb2, eps)
              else if (xp.lt.ddcut) then
                 xb1=xp
                 xb2=ddcut
                 a1=adglg1(fretil1, buf, xb1, eps)
                 a2=adglg1(fimtil1, buf, xb1, eps)
                 a3=adglg2(fretil1,xb1,xb2,eps)
                 a4=adglg2(fimtil1,xb1,xb2,eps)
              else if (ddcut.le.xp) then
              else
                 write(*,*) ' Constellation not possible! Stop.'
                 stop
              endif
c
              a  = 1.d0/(4.d0*pi**2)*cmplx(a1+a3+a5,a2+a4+a6,
     u                    kind=kind(0d0))
c
           else if (npot.eq.2) then
      PCUT=QCUT
      ETOT=ENERGY+2*TMASS
              a  = ZAPVGP(P,ETOT,VZERO-ENERGY,PCUT,EPS)
c             acomp = zapvq1(ALEFVQ, p, vzero-energy, gtpcor, eps)
c             a = zapvq1(ALEFVQ, p, vzero-energy, gtpcor, eps)
c             acomp = acomp/a
c             if (abs(acomp-1.d0).gt.1.d-3) then
c                write (*,*) 'p=', p
```

```
c                write (*,*) 'acomp/a=', acomp
c             endif
           else
              write (*,*) ' Potential not implemented! Stop.'
              stop
           endif
c
        end
c
        real*8 function fretil1(xk)
           implicit none
           real*8 xk, freal
           external freal
           fretil1 = freal(xk)
        end
c
        real*8 function fretil2(xk)
           implicit none
           real*8 xk, freal
           external freal
           fretil2 = freal(1.d0/xk) * xk**(-2)
        end
c
        real*8 function fimtil1(xk)
           implicit none
           real*8 xk, fim
           external fim
           fimtil1 = fim(xk)
        end
c
        real*8 function fimtil2(xk)
           implicit none
           real*8 xk, fim
           external fim
           fimtil2 = fim(1.d0/xk) * xk**(-2)
        end
c
        real*8 function freal(xk)
           implicit none
           complex*16 vhat
           real*8
     u        tmass,tgamma,zmass,alphas,alamb5,
     u        wmass,wgamma,bmass,GFERMI,
     u        pi, energy, vzero, eps,
     u        p,pmax, xk, gtpcor,dcut
           complex*16 g0,g0c
           integer npot
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
           common/ovalco/ pi, energy, vzero, eps, npot
           common/mom/ p,pmax,dcut
           external vhat, g0, g0c, gtpcor
c
           freal = real(g0c(xk)*vhat(p, xk)) !!! FB: NaN?
```

```
      end
c
      real*8 function fim(xk)
         implicit none
         complex*16 vhat
         real*8
     u      tmass,tgamma,zmass,alphas,alamb5,
     u      wmass,wgamma,bmass,GFERMI,
     u      pi, energy, vzero, eps,
     u      p,pmax, xk, gtpcor,dcut
         complex*16 g0,g0c
         integer npot
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
         common/ovalco/ pi, energy, vzero, eps, npot
         common/mom/ p,pmax,dcut
         external vhat, g0, g0c, gtpcor
         fim = aimag(g0c(xk)*vhat(p, xk))
      end
c
c
      complex*16 function vhat(p, xk)
c
         implicit none
         complex*16 zi
         real*8
     u      tmass,tgamma,zmass,alphas,alamb5,
     u      wmass,wgamma,bmass,GFERMI,
     u      pi, energy, vzero, eps,
     u      p, xk,
     u      cnspot, phiint, phfqcd, AD8GLE,
     u      pm, xkm, ALPHEF,
     u      zeta3,cf,ca,tf,xnf,a1,a2,b0,b1,
     u      cplas,scale,c0,c1,c2,
     u      cdeltc,cdeltl,cfullc,cfulll,crm2,
     u      xkpln1st,xkpln2nd,xkpln3rd,
     u      pp,pmax,dcut
         integer npot
         parameter(zi=(0.d0,1.d0))
         parameter(zeta3=1.20205690316d0,
     u            cf=4.d0/3.d0,ca=3.d0,tf=1.d0/2.d0,
     u            xnf=5.d0)
c
         external AD8GLE, phfqcd, ALPHEF
c
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
         common/ovalco/ pi, energy, vzero, eps, npot
         common/pmaxkm/ pm, xkm
         common/mom/ pp,pmax,dcut
         common/cplcns/cplas,scale,c0,c1,c2,
     u                 cdeltc,cdeltl,cfullc,cfulll,crm2
c
         b0=11.d0-2.d0/3.d0*xnf
```

```
      b1=102.d0-38.d0/3.d0*xnf
c
      a1=31.d0/9.d0*ca-20.d0/9.d0*tf*xnf
      a2=(4343.d0/162.d0+4.d0*pi**2-pi**4/4.d0+
     u     22.d0/3.d0*zeta3)*ca**2-
     u    (1798.d0/81.d0+56.d0/3.d0*zeta3)*ca*tf*xnf-
     u    (55.d0/3.d0-16.d0*zeta3)*cf*tf*xnf+
     u    (20.d0/9.d0*tf*xnf)**2
c
      pm=p
      xkm=xk
      cnspot=-4.d0/3.d0*4.d0*pi
c
      if (p/xk.le.1.d-5.and.p.le.1.d-5) then
         xkpln1st=2.d0
         xkpln2nd=-4.d0*dlog(scale/xk)
         xkpln3rd=-6.d0*dlog(scale/xk)**2
      else if (xk/p.le.1.d-5.and.xk.le.1.d-5) then
         xkpln1st=2.d0*(xk/p)**2
         xkpln2nd=-4.d0*(xk/p)**2*dlog(scale/p)
         xkpln3rd=-6.d0*(xk/p)**2*dlog(scale/p)**2
      else
c        xkpln1st=xk/p*dlog(dabs((p+xk)/(p-xk)))
         xkpln1st=xk/p*(dlog(p+xk)-dlog(dabs(p-xk)))
         xkpln2nd=xk/p*(-1.d0)*(dlog(scale/(p+xk))**2-
     u                          dlog(scale/dabs(p-xk))**2)
         xkpln3rd=xk/p*(-4.d0/3.d0)*(dlog(scale/(p+xk))**3-
     u                                dlog(scale/dabs(p-xk))**3)
      endif
c
      if (npot.eq.2) then
         if (p/xk.le.1.d-5.and.p.le.1.d-5) then
            vhat = 2.d0 * cnspot * ALPHEF(xk)
         else if (xk/p.le.1.d-5.and.xk.le.1.d-5) then
            vhat = 2.d0 * cnspot * xk**2 / p**2 * ALPHEF(p)
         else
            phiint = cnspot * (AD8GLE(phfqcd, 0.d0, 0.3d0, 1.d-5)
     u                        +AD8GLE(phfqcd, 0.3d0, 1.d0, 1.d-5))
            vhat   = xk / p * dlog(dabs((p+xk)/(p-xk))) * phiint
         endif
      else
         if (npot.eq.1) then
            c0=1.d0
            c1=0.d0
            c2=0.d0
         else if (npot.eq.3) then
            c0=1.d0+alphas/(4.d0*pi)*a1
            c1=alphas/(4.d0*pi)*b0
            c2=0
         else if (npot.eq.4) then
            c0=1.d0+alphas/(4.d0*pi)*a1+(alphas/(4.d0*pi))**2*a2
            c1=alphas/(4.d0*pi)*b0+
     u        (alphas/(4.d0*pi))**2*(b1+2.d0*b0*a1)
            c2=(alphas/(4.d0*pi))**2*b0**2
```

```
              else if (npot.eq.5) then
              else
                 write (*,*) ' Potential not implemented! Stop.'
                 stop
              endif
              phiint=cnspot*alphas
c
c             if ((xk+p).le.dcut) then
c                vhat=phiint*(c0*xkpln1st+c1*xkpln2nd+c2*xkpln3rd)
c     u               -1.d0/2.d0*(1.d0+2.d0*ca/cf)
c     u                *(pi*cf*alphas)**2/tmass
c     u                *xk/p*(p+xk-dabs(xk-p))
c             else if (dabs(xk-p).lt.dcut) then
c                vhat=phiint*(c0*xkpln1st+c1*xkpln2nd+c2*xkpln3rd)
c     u               -1.d0/2.d0*(1.d0+2.d0*ca/cf)
c     u                *(pi*cf*alphas)**2/tmass
c     u                *xk/p*(dcut-dabs(xk-p))
c             else if (dcut.le.dabs(xk-p)) then
c                vhat=phiint*(c0*xkpln1st+c1*xkpln2nd+c2*xkpln3rd)
c             else
c                write(*,*) ' Not possible! Stop.'
c                stop
c             endif
c
              if (max(xk,p).lt.dcut) then
c Coulomb + first + second order corrections:
                 vhat=phiint*(c0*xkpln1st+c1*xkpln2nd+c2*xkpln3rd)
c All other potentials:
     u               +cdeltc*2.d0*xk**2
     u               +cdeltl*xk/p/2.d0*(
     u                (p+xk)**2*(dlog(((p+xk)/scale)**2)-1.d0)-
     u                (p-xk)**2*(dlog(((p-xk)/scale)**2)-1.d0))
     u               +cfullc*(p**2+xk**2)*xkpln1st
     u               +cfulll*(p**2+xk**2)*xk/p/4.d0*
     u                 (dlog(((p+xk)/scale)**2)**2-
     u                  dlog(((p-xk)/scale)**2)**2)
     u               +crm2*xk/p*(p+xk-dabs(xk-p))
              else
                 vhat=phiint*(c0*xkpln1st+c1*xkpln2nd+c2*xkpln3rd)
              endif
           endif
c
        end
c
c
c
c --- Routines needed for use of phenomenological potentials ---
c
      SUBROUTINE INIPHC(INIFLG)
      implicit real*8(a-h,o-z)
      save
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      common/ovalco/ pi, energy, vzero, eps, npot
```

```
      COMMON/PARFLG/ QCUT,QMAT1,ALR,ILFLAG
      CHARACTER QCTCHR,QMTCHR,ALFCHR
      DATA QCUT0/.100d0/,QMT1S/5.0d0/
c
      zmass= 91.187d0
      if(INIFLG.eq.0) then
c     standard set of parameters
        ilflag= 1
        alphas=.12d0
        qcut= qcut0
        qmat1= qmt1s
      else
c     Parameters of QCD potential specified by USER
  5     write(*,*) 'QCD coupling at M_z:   ALPHAS  or  LAMBDA  ?'
        write(*,*) 'A/L  :'
        read(*,895) ALFCHR
          if(ALFCHR.eq.'A'.or.ALFCHR.eq.'a') then
              ilflag= 1
              write(*,*) 'alpha_s(M_z)= ?'
              read(*,*) alphas
          elseif(ALFCHR.eq.'L'.or.ALFCHR.eq.'l') then
              write(*,*) 'Lambda(nf=5) =?'
              read(*,*) alamb5
              ilflag= 0
          else
              write(*,*) '!!!  PLEASE TYPE: A OR L  !!!'
              goto 5
          endif
   10   write(*,896) qcut0
        read(*,895) QCTCHR
          if(QCTCHR.eq.'Y'.or.QCTCHR.eq.'y') then
              qcut=qcut0
          elseif(QCTCHR.eq.'N'.or.QCTCHR.eq.'n') then
              write(*,*) 'QCUT (GeV) = ?'
              read(*,*) qcut
          else
              write(*,*) '!!!   PLEASE TYPE: Y OR N   !!!'
              goto 10
          endif
   15   write(*,902) qmt1s
        read(*,895) QMTCHR
          if(QMTCHR.eq.'Y'.or.QMTCHR.eq.'y') then
              qmat1=qmt1s
          elseif(QMTCHR.eq.'N'.or.QMTCHR.eq.'n') then
              write(*,*) 'QMAT1 (GeV) = ?'
              read(*,*) qmat1
          else
              write(*,*) '!!!   PLEASE TYPE: Y OR N   !!!'
              goto 15
          endif
      endif
  895 format(1A)
  896 format(1x,'Long distance cut off for QCD potential'/
     $ 1x,'QCUT = ',f5.4,' GeV.  OK ? Y/N')
```

```
  902 format(1x,
     $ 'Matching QCD for NF=5 and Richardson for NF=3 at QMAT1 =',
     $  f5.2,' GeV.'/1x,'  OK ? Y/N')
      end
c
c
        real*8 function phfqcd(x)
c     integrand over k   ?
           real*8 pm, xkm, x, ALPHEF
           external ALPHEF
           common/pmaxkm/ pm, xkm
           phfqcd = ALPHEF((pm+xkm)*(dabs(pm-xkm)/(pm+xkm))**x)
        end
c
c
      FUNCTION ALEFVQ(x)
      implicit real*8(a-h,o-z)
      external ALPHEF
      common/xtr101/ p0
      data pi/3.1415926535897930d0/
      q= p0*x
      ALEFVQ= - 4d0/3* 4*pi*ALPHEF(q)
      return
      end
C
C
!!! FB:
!       COMPLEX*16 FUNCTION ZAPVQ1(F,P,VME,gtpcor,ACC)
! C for testing only! Original ZAPVQ1 for const. tgamma !!
! C     A(p,E)= ZAPVQ1(F,p,V0-E,Gtpcor,acc)
! C             where  F(q)=(q/p)**2*VQ(q/p)
! C
!       implicit real*8(a-c,d-h,o-y),complex*16(z)
!       real*8 zmass
!       EXTERNAL F, gtpcor
!       common/xtr101/ p0
!       COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
!      $ WMASS,WGAMMA,BMASS,GFERMI
! c           common/phcons/ TM,tgamma,zmass,alphas,alamb5,
! c     u                    wmass,wgamma,bmass
!          common/ovalco/ pi, energy, vzero, eps, npot
!       DIMENSION W(12),X(12)
!       data pi/3.1415926535897930d0/
!       DATA W / 0.10122 85362 90376 25915 25313 543D0,
!      1         0.22238 10344 53374 47054 43559 944D0,
!      2         0.31370 66458 77887 28733 79622 020D0,
!      3         0.36268 37833 78361 98296 51504 493D0,
!      4         0.27152 45941 17540 94851 78057 246D-1,
!      5         0.62253 52393 86478 92862 84383 699D-1,
!      6         0.95158 51168 24927 84809 92510 760D-1,
!      7         0.12462 89712 55533 87205 24762 822D0,
!      8         0.14959 59888 16576 73208 15017 305D0,
!      9         0.16915 65193 95002 53818 93120 790D0,
!      A         0.18260 34150 44923 58886 67636 680D0,
```

```
!      B         0.18945 06104 55068 49628 53967 232D0/
! C
!       DATA X / 0.96028 98564 97536 23168 35608 686D0,
!      1         0.79666 64774 13626 73959 15539 365D0,
!      2         0.52553 24099 16328 98581 77390 492D0,
!      3         0.18343 46424 95649 80493 94761 424D0,
!      4         0.98940 09349 91649 93259 61541 735D0,
!      5         0.94457 50230 73232 57607 79884 155D0,
!      6         0.86563 12023 87831 74388 04678 977D0,
!      7         0.75540 44083 55003 03389 51011 948D0,
!      8         0.61787 62444 02643 74844 66717 640D0,
!      9         0.45801 67776 57227 38634 24194 430D0,
!      A         0.28160 35507 79258 91323 04605 015D0,
!      B         0.95012 50983 76374 40185 31933 543D-1/
! C
!       ZA(xk)= log( cmplx((xk+1)**2+ vme*tm/p**2,-tm*tgamma*
!      $              gtpcor(xk*p,2.d0*tm+energy)/p**2)/
!      $             cmplx((xk-1)**2+ vme*tm/p**2,-tm*tgamma*
!      $              gtpcor(xk*p,2.d0*tm+energy)/p**2) )
! C
!       TM=TMASS
!       p0= p
! c     integral from 0 to 1  for IEX=1 and from 1 to 1/0 for IEX=-1
!       sumr=0d0
!       sumi=0d0
!       IEX=1
!    7  a=0d0
!       b=1d0
!       BB=A
! C
!    11 AA=BB
!       BB=B
!    12    C1=0.5D0*(BB+AA)
!          C2=0.5D0*(BB-AA)
!          S8r=0.0D0
!          S8i=0.0D0
!          DO 13 I=1,4
!             U=C2*X(I)
!             z1=ZA((c1+u)**IEX)
!             xz1r=DBLE(z1)/(c1+u)
!             xz1i=AIMAG(z1)/(c1+u)
!             z2=ZA((c1-u)**IEX)
!             xz2r=DBLE(z2)/(c1-u)
!             xz2i=AIMAG(z2)/(c1-u)
!             S8r=S8r+W(I)*(F((C1+U)**IEX)*xz1r+F((C1-U)**IEX)*xz2r)
!             S8i=S8i+W(I)*(F((C1+U)**IEX)*xz1i+F((C1-U)**IEX)*xz2i)
!    13    CONTINUE
!          S8r=C2*S8r
!          S8i=C2*S8i
!          S16r=0.0D0
!          S16i=0.0D0
!          DO 14 I=5,12
!             U=C2*X(I)
!             z1=ZA((c1+u)**IEX)
```

```
!             xz1r=DBLE(z1)/(c1+u)
!             xz1i=AIMAG(z1)/(c1+u)
!             z2=ZA((c1-u)**IEX)
!             xz2r=DBLE(z2)/(c1-u)
!             xz2i=AIMAG(z2)/(c1-u)
!             S16r=S16r+W(I)*(F((C1+U)**IEX)*xz1r+F((C1-U)**IEX)*xz2r)
!             S16i=S16i+W(I)*(F((C1+U)**IEX)*xz1i+F((C1-U)**IEX)*xz2i)
!    14    CONTINUE
!          S16r=C2*S16r
!          S16i=C2*S16i
!          IF(ABS(S16r-S8r) .LE. acc*(abs(s8r)+ABS(S16r))/2  .AND.
!      $      ABS(S16i-S8i) .LE. acc*(abs(s8i)+ABS(S16i))/2 ) GO TO 15
!          BB=C1
!          IF( 1.D0+ABS(C2) .NE. 1.D0) GO TO 12
!          ZAPVQ1 =(0d0,0d0)
!          WRITE(*,*) 'Too high accuracy required'
!          STOP 10101
!    15 sumr=sumr+s16r
!       sumi=sumi+s16i
!       IF(BB.NE.B) GO TO 11
!       IF(IEX.EQ.1) THEN
!          IEX= -1
!          GOTO 7
!       ELSE
!          ZAPVQ1= -TM/(8*pi**2*p)*CMPLX(SUMR,SUMI)
!       ENDIF
!       RETURN
!       END
!!! /FB
C
C
      COMPLEX*16 FUNCTION ZAPVGP(P,ETOT,VME,PCUT,ACC)
C
C     A(p,E)= ZAPVGP(P,ETOT,VME,PCUT,ACC)
C     for QCD potential VQQBAR(q) and GAMTPE(P,E)  - momentum
C     dependent width of top quark in t-tbar system.
C     2-dimensional integration
C     P - intrinsic momentum of t quark, ETOT - total energy of t-tbar,
C     VME=V0-E, where V0-potential at spatial infinity, E=ETOT-2*TMASS,
C     PCUT - cut off in momentum space; e.g. for QCD potential
C     given by ALPHEF  PCUT=QCUT in COMMON/parflg/,
C     ACC - accuracy
C     external functions: VQQBAR,GAMTPE,ADQUA,AD8GLE,ADGLG1,ADGLG2
C
      IMPLICIT REAL*8(A-Z)
      EXTERNAL FIN01P,FIN02P,FIN03P,FIN04P,AD8GLE,ADGLG1,ADGLG2
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      COMMON/XTR102/ P0,E0,VMEM,TM,ACC0
      DATA PI/3.14159265/,BUF/1D-10/,SMALL/1D-2/
C For Testing only
      small = 1.d-1
C
      CONST= -TMASS/(8*PI**2*P)
```

```
      TM= TMASS
      ACC0=ACC*SMALL
      P0=P
      E0=ETOT
      VMEM=VME*TMASS
      IF(PCUT.LE.P) THEN
         XXRE=AD8GLE(FIN01P,BUF,PCUT,ACC)+ADGLG1(FIN01P,PCUT,P,ACC)+
     $        ADGLG1(FIN02P,BUF,1/P,ACC)
         XXIM=AD8GLE(FIN03P,BUF,PCUT,ACC)+ADGLG1(FIN03P,PCUT,P,ACC)+
     $        ADGLG1(FIN04P,BUF,1/P,ACC)
      ELSE
         XXRE=ADGLG1(FIN01P,BUF,P,ACC)+ADGLG2(FIN01P,P,PCUT,ACC)+
     $        AD8GLE(FIN02P,BUF,1/PCUT,ACC)
         XXIM=ADGLG1(FIN03P,BUF,P,ACC)+ADGLG2(FIN03P,P,PCUT,ACC)+
     $        AD8GLE(FIN04P,BUF,1/PCUT,ACC)
      ENDIF
      ZAPVGP=CONST*CMPLX(XXRE,XXIM,KIND=KIND(0d0))
      END
C
      REAL*8 FUNCTION FIN01P(Q)
C     this segment contains FIN01P,FIN02P,FIN03P,FIN04P
      IMPLICIT REAL*8(A-C,D-H,O-Z)
      EXTERNAL VQQBAR,FIN11P, FIN12P
      COMMON/XTR102/ P0,E0,VMEM,TM,ACC0
      DATA PI/3.14159265/,BUF/1d-10/
      Q0=Q
      XL=(P0-Q0)**2
      XU=(P0+Q0)**2
      CALL ADQUA(XL,XU,FIN11P,Y,ACC0)
      FIN01P= VQQBAR(Q0)*Q0*Y
      RETURN
      ENTRY FIN02P(Q)
      Q0=1/Q
      XL=(P0-Q0)**2
      XU=(P0+Q0)**2
      CALL ADQUA(XL,XU,FIN11P,Y,ACC0)
      FIN02P= VQQBAR(Q0)*Q0**3*Y
      RETURN
      ENTRY FIN03P(Q)
      Q0=Q
      XL=(P0-Q0)**2
      XU=(P0+Q0)**2
      CALL ADQUA(XL,XU,FIN12P,Y,ACC0)
      FIN03P= VQQBAR(Q0)*Q0*Y
      RETURN
      ENTRY FIN04P(Q)
      Q0=1/Q
      XL=(P0-Q0)**2
      XU=(P0+Q0)**2
      CALL ADQUA(XL,XU,FIN12P,Y,ACC0)
      FIN04P= VQQBAR(Q0)*Q0**3*Y
      END
      REAL*8 FUNCTION FIN11P(T)
C     this segment contains FIN11P,FIN12P
```

```
      IMPLICIT REAL*8(A-C,D-H,O-Z)
      EXTERNAL GAMTPE
      COMMON/XTR102/ P0,E0,VMEM,TM,ACC0
      T1= T+VMEM
      TSQRT= SQRT(T)
      GAMMA= TM*GAMTPE(TSQRT,E0)
      FIN11P= T1/(T1**2+GAMMA**2)
      RETURN
      ENTRY FIN12P(T)
      T1= T+VMEM
      TSQRT= SQRT(T)
      GAMMA= TM*GAMTPE(TSQRT,E0)
      FIN12P= GAMMA/(T1**2+GAMMA**2)
      END
C
c
      SUBROUTINE VQDELT(VQ)
c
c     evaluates constants multiplying Dirac delta in potentials VQCUT
c     calls: ADQUA
c
      implicit real*8(a-h,o-z)
      external alphef,fncqct
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      COMMON/PARFLG/ QCUT,QMAT1,ALR,ILFLAG
      data pi/3.141592653589793238D0/
c
      call adqua(1d-8,1d4,fncqct,y,1d-4)
      v=-4d0/3*2/pi*y
      VQ=(-.25-v)*(2*pi)**3
      end
c
      function fncqct(q)
      implicit real*8(a-h,o-z)
      fncqct=sin(q)/q*alphef(q)
      end
c
C
      REAL*8 FUNCTION VQQBAR(P)
C
C     interquark potential for q- qbar singlet state
C
      IMPLICIT REAL*8(A-C,D-H,O-Z)
      EXTERNAL ALPHEF
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      DATA PI/3.14159265/
      VQQBAR = -4D0/3*4*PI*ALPHEF(P)/P**2
      END
C
      FUNCTION ALPHEF(q)
c
c     V(q) = -4/3 * 4*pi*ALPHEF(q)/q**2
```

```
c     input: alphas or alamb5 in COMMON/PHCONS/.  If:
c     ILFLAG.EQ.0   alamb5= \Lambda_\{\bar MS}^{(5)} at M_z
c     ILFLAG.EQ.1   alphas = alpha_{strong} at M_z (91.161)
c
c     effective coupling ALPHEF is defined as follows:
c     for q > qmat1=m_b:
c       alphas*( 1 +(31/3-10*nf/9)*alphas/(4*pi) )
c       where alphas=\alpha_\bar{MS} for nf=5, i.e.
c        alpha=4*pi/( b0(nf=5)*x + b1(5)/b0(5)*ln(x) )
c        and x = ln(q**2/alamb5**2)
c     for qmat1 > q > qcut:
c       4*pi/b0(nefr=3)*(alfmt+1/log(1+q**2/alr**2))
c       where alr=.4 GeV, nefr=3, and continuity --> alfmt
c     below qcut:  alphrc*2*q**2/(q**2+qcut**2)  (cont.-->alphrc)
c
      implicit real*8(a-h,o-z)
      SAVE
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      COMMON/PARFLG/ QCUT,QMAT1,ALR,ILFLAG
      common/parpot/ a5,b5,c5,alfmt,d,alphrc
      data pi/3.141592653589793238D0/,
     $ zold/-1d0/,qctold/-1d0/,alfold/-1d0/,
     $olmbd/-1d0/
c
      if(zmass.le.0d0 .or. qcut.le.0d0) STOP 10001
      if(zold.ne.zmass .or. qcut.ne.qctold) num=0
      if(ilflag.eq.0 .and. olmbd.ne.alamb5) num=0
      if(ilflag.eq.1 .and. alfold.ne.alphas) num=0
      if(num.eq.0)then
          num=num+1
          zold=zmass
          qctold=qcut
          call potpar
          alfold= alphas
          olmbd= alamb5
      endif
      if(q.le.qcut) then
         alphef=alphrc*(2*q**2)/(qcut**2+q**2)
      elseif(q.le.qmat1) then
         alphef=alfmt+d/log(1+q**2/alr**2)
      else
         x=2*log(q/alamb5)
         alfas5=1/(a5*x+b5*log(x))
         alphef=alfas5*(1+c5*alfas5)
      endif
      end
c
c Only called by ALPHEF:
      SUBROUTINE POTPAR
      implicit real*8(a-h,o-z)
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      COMMON/PARFLG/ QCUT,QMAT1,ALR,ILFLAG
```

```
      common/parpot/ a5,b5,c5,alfmt,d,alphrc
      data pi/3.14159265358979323 8D0/,nefr/3/
      b0(nf)=11-2./3*nf
      b1(nf)=102-38./3*nf
      cn(nf)=31./3-10./9*nf
      alr=400d-3
      a5=b0(5)/(4*pi)
      b5=b1(5)/b0(5)/(4*pi)
      c5=cn(5)/(4*pi)
      d=4*pi/b0(nefr)
      if(ilflag.eq.0) then
         if(alamb5.le.0d0) STOP 10002
         xa=2*log(zmass/alamb5)
         alphas= 1/(a5*xa + b5*log(xa))
      else
         if(alphas.le.0d0) STOP 10003
         t0=0
         t1=max(1d0,alphas*a5)
 10      tm=(t0+t1)/2
         fm=tm/alphas+b5*tm*log(tm)-a5
         if(fm.lt.-1d-10) then
           t0=tm
           goto 10
         elseif(fm.gt.1d-10) then
           t1=tm
           goto 10
         endif
         alamb5=zmass*exp(-5d-1/tm)
      endif
      x=2*log(qmat1/alamb5)
      alfas=1/(a5*x+b5*log(x))
      alfmt=alfas*(1+c5*alfas)-d/log(1+qmat1**2/alr**2)
      alphrc=alfmt+ d/log(1+qcut**2/alr**2)
      return
      end
c
c --- End of routines for phenomenological potentials ---
c
c
c --- Routines for Gamma_top ---
C
      SUBROUTINE GAMMAT
C
C     on shell width of top quark including QCD corrections, c.f.
C     M.Jezabek and J.H. Kuhn, Nucl. Phys. B314(1989)1
C
      IMPLICIT REAL*8(A-C,D-H,O-Z)
      EXTERNAL DILOG
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      DATA PI/3.14159265/
      F(X)= PI**2+2*DILOG(X)-2*DILOG(1-X)+( 4*X*(1-X-2*X**2)*LOG(X)+
     $2*(1-X)**2*(5+4*X)*LOG(1-X) - (1-X)*(5+9*X-6*X**2) ) /
     $(2*(1-X)**2*(1+2*X))
```

```
      Y= (WMASS/TMASS)**2
cc alpha_s(M_t) corresponding to alpha_s(M_Z)=0.118:
cc      alphas=0.107443d0
cc      write(*,*) 'alphas=',alphas
c Usage of alpha_s as given as input for the potential.. better use
c alpha_s at a scale close to m_t..
      TGAMMA= GFERMI*TMASS**3/(8*SQRT(2D0)*PI)*(1-Y)**2*(1+2*Y)*
     $(1- 2D0/3*ALPHAS/PI*F(Y))
      END
C
C
      REAL*8 FUNCTION GAMTPE(P,ETOT)
C
C     momentum dependent width of top quark in t-tbar system
C     GAMTPE = TGAMMA*GTPCOR(P,E), where TGAMMA includes
C     QCD corrections, see JKT, eq.(8), and
C     GTPCOR - correction factor for bound t quark
C
      IMPLICIT REAL*8(A-C,D-H,O-Z)
      EXTERNAL GTPCOR
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      GAMTPE= TGAMMA*GTPCOR(P,ETOT)
      END
C
C
C    GTPCOR and GTPCOR1 should be merged  (M.J.) !!!!
c
        real*8 function gtpcor(topp,etot)
        real*8 topp,etot,
     u         tmass,tgamma,zmass,alphas,alamb5,
     u         wmass,wgamma,bmass,GFERMI
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
c       if (topp.ge.tmass/2.d0) then
c          gtpcor1=0.001d0
c       else
           gtpcor=1.d0
c       endif
        end
c
c
c Correction function for non-constant (energy and momentum dependent) width:
      FUNCTION GTPCOR1(TOPP,ETOT)
c
c     TOPP - momentum of t quark = - momentum of tbar
c     ETOT - total energy of t-tbar system
c     calls: GENWDS, RAN2
c
c     Evaluates a correction factor to the width of t-tbar system.
c     in future has to be replaced by a function evaluating
c     width including radiative corrections and GTPCOR.
c     I include two factors reducing the width:
c     a - time dilatation: for decay in flight lifetime
```

```
c         increased accordingly to relativistic kinematics
c     b - overall energy-momentum conservation: I assume that
c         t and tbar decay in flight and in this decays energies
c         of Ws follow from 2-body kinematics. Then I calculate
c         effective mass squared of b-bar system (it may be
c         negative!) from en-momentum conservation.
c         If effective mass is < 2*Mb + 2 GeV configuration
c         is rejected. The weight is acceptance.
c
      IMPLICIT REAL*8(A-H,O-Z)
      real ran2
      external ran2
      PARAMETER(NG=20,NC=4)
      dimension gamma(0:NG),pw1(0:3),pw2(0:3),AIJ(NC,NC),BJ(NC),
     $AI(NC),SIG2IN(0:NG),XIK(0:NG,NC),INDX(NC)
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      SAVE NUM,EOLD,TOLD,AI
      data nevent/10000/, num/0/, eold/-1d5/, told/-1d0/
c
C     for test runs!!
C      nevent=1000
C
      if(etot.ne.eold) num=0
      if(tmass.ne.told) num=0
   5  if(num.eq.0) then
c            xdumm= ran2(-2)
            do 10 itp=0,NG
            tp=itp*tmass/NG*2
            gamma(itp)=0
            do 10 ix=1,nevent
            call GENWDS(tp,etot,pw1,pw2,efmsq)
            if(efmsq.gt.0d0) then
               efms=sqrt(efmsq)
               if(efms.ge. 2*bmass+2) gamma(itp)=gamma(itp)+1
            endif
  10        continue
            do 15 ix=0,NG
  15        SIG2IN(IX)= MAX(1D0,GAMMA(IX))
            DO 17 JX=1,NC
              IF(JX.EQ.1)THEN
                XIK(0,JX)= .5D0
              ELSE
                XIK(0,JX)= 0D0
              ENDIF
            DO 17 IX=1,NG
            tp= 2D0*ix/NG
  17        XIK(IX,JX)= tp**(JX-1)/(1+EXP(tp*3))
            DO 20 I=1,NC
            BJ(I)=0
            DO 20 J=1,NC
  20        AIJ(I,J)=0
            DO 30 I=1,NC
            DO 25 IX=0,NG
```

```
 25        BJ(I)= BJ(I)+GAMMA(IX)*XIK(IX,I)*SIG2IN(IX)
           DO 30 J=1,I
           DO 30 IX=0,NG
 30        AIJ(I,J)= AIJ(I,J)+XIK(IX,I)*XIK(IX,J)*SIG2IN(IX)
           DO 35 I=1,NC
           DO 35 J=I,NC
 35        AIJ(I,J)= AIJ(J,I)
           CALL LUDCMP(AIJ,NC,NC,INDX,D)
           CALL LUBKSB(AIJ,NC,NC,INDX,BJ)
           DO 40 I=1,NC
 40        AI(I)= BJ(I)/NEVENT
     do 42 i=1,nc
 42  write(*,*)'a(',i,')=',ai(i)
           do 100 ix=0,NG
 100       gamma(ix)= gamma(ix)/nevent
           eold=etot
           told=tmass
           num= 1
     endif
     SUM=AI(1)
     DO 110 I=2,NC
 110 SUM= SUM+AI(I)*(TOPP/TMASS)**(I-1)
C     CORRF2= SUM/(1+ EXP(TOPP/TMASS*3))
     CORRF2= SUM/(1+ EXP(MIN(1d1,TOPP/TMASS*3)))
C     if(topp.gt. 2d0*tmass) then
C         corrf1= 0.001d0
C     else
C          ip= NG*topp/tmass/2
C         corrf1= gamma(ip)
C     endif
C     write(*,*)'ratio=',corrf1/corrf2
C     GTPCOR1 = CORRF2
     GTPCOR1 = CORRF2*SQRT(1-TOPP**2/(TOPP**2+TMASS**2))
     END
c
c Generator: only called by GTPCOR1
     SUBROUTINE GENWDS(tp,etot,pw1,pw2,efm2)
c
c    generates 4-momenta of W's  and effective mass of b-bbar
c    from t and tbar quarks decays at flight (tp = momentum of t
c    = - momentum of tbar (in GeV) ) in Oz direction
c
     implicit real*8(a-h,o-z)
c     real ran2
     real ranf
c     external ran2
     external ranf
     dimension pw1(0:3),pw2(0:3)
     save
     COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
    $ WMASS,WGAMMA,BMASS,GFERMI
     data PI/3.141592653589793238D0/
     real idum
c  3   s1= wmass**2+wmass*wgamma*TAN((2*ran2(idum)-1)*pi/2)
```

```
  3   s1= wmass**2+wmass*wgamma*TAN((2*ranf(idum)-1)*pi/2)
      if(s1.le.0d0) goto 3
      wmass1= sqrt(s1)
      if(abs(wmass1-wmass).ge.3*wgamma) goto 3
c  4   s2= wmass**2+wmass*wgamma*TAN((2*ran2(idum)-1)*pi/2)
  4   s2= wmass**2+wmass*wgamma*TAN((2*ranf(idum)-1)*pi/2)
      if(s2.le.0d0) goto 4
      wmass2= sqrt(s2)
      if(abs(wmass2-wmass).ge.3*wgamma) goto 4
      ew1= (tmass**2+wmass1**2-bmass**2)/(2*tmass)
      pwt1= sqrt(ew1**2-wmass1**2)
      ew2= (tmass**2+wmass2**2-bmass**2)/(2*tmass)
      pwt2= sqrt(ew2**2-wmass2**2)
  5   p=tp
c      u1= 2*ran2(idum)-1
      u1= 2*ranf(idum)-1
      pw1z= pwt1*u1
c      u2= 2*ran2(idum)-1
      u2= 2*ranf(idum)-1
      pw2z= pwt2*u2
      et= sqrt(tmass**2+p**2)
      bet= p/et
      gam= et/tmass
      pw1(0)= gam*(ew1+bet*pw1z)
      pw1(3)= gam*(pw1z+bet*ew1)
      pw2(0)= gam*(ew2-bet*pw2z)
      pw2(3)= gam*(pw2z-bet*ew2)
      pw1tr= sqrt(pw1(0)**2-pw1(3)**2-wmass1**2)
      pw2tr= sqrt(pw2(0)**2-pw2(3)**2-wmass2**2)
c      phi1= 2*pi*ran2(idum)
      phi1= 2*pi*ranf(idum)
c      phi2= 2*pi*ran2(idum)
      phi2= 2*pi*ranf(idum)
      pw1(1)= pw1tr*cos(phi1)
      pw1(2)= pw1tr*sin(phi1)
      pw2(1)= pw2tr*cos(phi2)
      pw2(2)= pw2tr*sin(phi2)
      prec2= (pw1(1)+pw2(1))**2+(pw1(2)+pw2(2))**2+(pw1(3)+pw2(3))**2
      erest=etot-pw1(0)-pw2(0)
c
      efm2= erest*abs(erest)-prec2
      END
c
c --- End of routines for Gamma_top ---
c
c --- Routines for solving linear equations and matrix inversion (complex) ---
c
        subroutine sae(pp, w1, bb, a1, n)
c
           implicit none
           complex*16 vhat
           real*8
     u        tmass,tgamma,zmass,alphas,alamb5,
     u        wmass,wgamma,bmass,GFERMI,
```

```
     u        pi, energy, vzero, eps,
     u        d, pp, w1, gtpcor,
     u        xp,xpmax,dcut,kincom,kincoa,kincov
           complex*16 a, a1, bb, ff, cw, svw, g0, g0c
           integer i, j, npot, n, nmax, indx,kinflg,gcflg,vflag
           parameter (nmax=400)
           dimension bb(nmax), ff(nmax,nmax), pp(nmax), w1(nmax),
     u               indx(nmax), cw(nmax), a1(nmax)
c
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
           common/ovalco/ pi, energy, vzero, eps, npot
           common/mom/ xp,xpmax,dcut
           common/g0inf/kincom,kincoa,kincov,kinflg,gcflg,vflag
c
           external a, vhat, gtpcor, g0, g0c
c
           do 10 i=1,n*2/3
              cw(i) = w1(i) / (4.d0*pi**2) * g0c(pp(i))
c             cw(i) = w1(i) / (4.d0*pi**2 *
c     u                (cmplx(energy-vzero, tgamma*
c     u                 gtpcor(pp(i),2.d0*tmass+energy),
c     u                    kind=kind(0d0))-pp(i)**2/tmass))
10         continue
           do 20 i=n*2/3+1,n
              cw(i) = w1(i) / (4.d0*pi**2) * g0c(pp(i)) * pp(i)**2
c             cw(i) = w1(i) / (4.d0*pi**2 *
c     u          (cmplx(energy-vzero, tgamma*
c     u           gtpcor(pp(i),2.d0*tmass+energy),
c     u                    kind=kind(0d0)) /
c     u           pp(i)**2 - 1.d0/tmass))
20         continue
c
           do 30 i=1,n
cc            bb(i) = a1(i)
cvv
              if (pp(i).lt.dcut.and.vflag.eq.1) then
c                bb(i) = cmplx(1.d0+kincov*(pp(i)/tmass)**2,0.d0,
c     u                    kind=kind(0d0))
                 bb(i)=1.d0+kincov*
     u                      g0(pp(i))*(pp(i)/tmass)**2/g0c(pp(i))
              else
                 bb(i) = (1.d0,0.d0)
              endif
              svw = (0.d0,0.d0)
              do 40 j=1,n
                 if (i.ne.j) then
                    ff(i,j) = - vhat(pp(i),pp(j)) * cw(j)
                    svw = svw + ff(i,j)
                 endif
40            continue
              ff(i,i) = 1.d0 - a1(i) - svw
30         continue
c
```

```
          call zldcmp(ff, n, nmax, indx, d)
          call zlbksb(ff, n, nmax, indx, bb)
c
        end
c
c
      SUBROUTINE ZLBKSB(A,N,NP,INDX,B)
C complex version of lubksb
      IMPLICIT NONE
      INTEGER I, II, INDX, J, LL, N, NP
      COMPLEX*16 A, B, SUM
      DIMENSION A(NP,NP),INDX(N),B(N)
      II=0
      DO 12 I=1,N
        LL=INDX(I)
        SUM=B(LL)
        B(LL)=B(I)
        IF (II.NE.0)THEN
          DO 11 J=II,I-1
            SUM=SUM-A(I,J)*B(J)
11        CONTINUE
        ELSE IF (SUM.NE.(0.D0,0.D0)) THEN
          II=I
        ENDIF
        B(I)=SUM
12    CONTINUE
      DO 14 I=N,1,-1
        SUM=B(I)
        IF(I.LT.N)THEN
          DO 13 J=I+1,N
            SUM=SUM-A(I,J)*B(J)
13        CONTINUE
        ENDIF
        B(I)=SUM/A(I,I)
14    CONTINUE
      RETURN
      END
c
      SUBROUTINE ZLDCMP(A,N,NP,INDX,D)
C complex version of ludcmp
      IMPLICIT NONE
      INTEGER I, IMAX, INDX, J, K, N, NP, NMAX
      REAL*8 AAMAX, D, TINY, VV
      COMPLEX*16 A, DUM, SUM
      PARAMETER (NMAX=400)
      DIMENSION A(NP,NP), INDX(N), VV(NMAX)
c
        tiny=1.d-5
c
      D=1.D0
      DO 12 I=1,N
        AAMAX=0.D0
        DO 11 J=1,N
          IF (ABS(A(I,J)).GT.AAMAX) AAMAX=ABS(A(I,J))
```

```
11      CONTINUE
c        IF (AAMAX.EQ.0.D0) PAUSE 'Singular matrix.'
        IF (AAMAX.EQ.0.D0) print *, "Singular matrix."
        VV(I)=1.D0/AAMAX
12    CONTINUE
      DO 19 J=1,N
        IF (J.GT.1) THEN
          DO 14 I=1,J-1
            SUM=A(I,J)
            IF (I.GT.1)THEN
              DO 13 K=1,I-1
                SUM=SUM-A(I,K)*A(K,J)
13            CONTINUE
              A(I,J)=SUM
            ENDIF
14        CONTINUE
        ENDIF
        AAMAX=0.D0
        DO 16 I=J,N
          SUM=A(I,J)
          IF (J.GT.1)THEN
            DO 15 K=1,J-1
              SUM=SUM-A(I,K)*A(K,J)
15          CONTINUE
            A(I,J)=SUM
          ENDIF
          DUM=VV(I)*ABS(SUM)
          IF (ABS(DUM).GE.AAMAX) THEN
            IMAX=I
            AAMAX=DUM
          ENDIF
16      CONTINUE
        IF (J.NE.IMAX) THEN
          DO 17 K=1,N
            DUM=A(IMAX,K)
            A(IMAX,K)=A(J,K)
            A(J,K)=DUM
17        CONTINUE
          D=-D
          VV(IMAX)=VV(J)
        ENDIF
        INDX(J)=IMAX
        IF (J.NE.N) THEN
          IF (A(J,J).EQ.(0.D0,0.D0)) A(J,J)=cmplx(TINY, 0.d0,
     u                    kind=kind(0d0))
          DUM=1.D0/A(J,J)
          DO 18 I=J+1,N
            A(I,J)=A(I,J)*DUM
18        CONTINUE
        ENDIF
19    CONTINUE
      IF(A(N,N).EQ.(0.D0,0.D0)) A(N,N)=cmplx(TINY, 0.d0,
     u                    kind=kind(0d0))
      RETURN
```

```
      END
C
C
C *** TOOLS ***
C
C
C     ******* ROUTINES FOR GAUSSIAN INTEGRATIONS
C
C
      SUBROUTINE GAULEG(X1,X2,X,W,N)
C
C     Given the lower and upper limits of integration X1 and X2
C     and given N, this routine returns arrays X(N) and W(N)
C     containing the abscissas and weights of the Gauss-Legendre
C     N-point quadrature formula
C
      IMPLICIT REAL*8 (A-H,O-Z)
      REAL*8 X1,X2,X(N),W(N)
      PARAMETER (EPS=3.D-14)
      save
      M=(N+1)/2
      XM=0.5D0*(X2+X1)
      XL=0.5D0*(X2-X1)
      DO 12 I=1,M
        Z=DCOS(3.14159265358979323 8D0*(I-.25D0)/(N+.5D0))
1       CONTINUE
          P1=1.D0
          P2=0.D0
          DO 11 J=1,N
            P3=P2
            P2=P1
            P1=((2.D0*J-1.D0)*Z*P2-(J-1.D0)*P3)/J
11        CONTINUE
          PP=N*(Z*P1-P2)/(Z*Z-1.D0)
          Z1=Z
          Z=Z1-P1/PP
        IF(DABS(Z-Z1).GT.EPS)GO TO 1
        X(I)=XM-XL*Z
        X(N+1-I)=XM+XL*Z
        W(I)=2.D0*XL/((1.D0-Z*Z)*PP*PP)
        W(N+1-I)=W(I)
12    CONTINUE
      RETURN
      END
C
C
      DOUBLE PRECISION FUNCTION AD8GLE(F,A,B,EPS)
      implicit double precision (a-h,o-z)
      EXTERNAL F
      DIMENSION W(12),X(12)
c      SAVE W, X
      SAVE
C
C     ****************************************************************
```

```
C
C     ADAPTIVE GAUSSIAN QUADRATURE.
C
C     AD8GLE IS SET EQUAL TO THE APPROXIMATE VALUE OF THE INTEGRAL OF
C     THE FUNCTION F OVER THE INTERVAL (A,B), WITH ACCURACY PARAMETER
C     EPS.
C
C     ****************************************************************
C
      DATA W / 0.10122 85362 90376 25915 25313 543D0,
     $         0.22238 10344 53374 47054 43559 944D0,
     $         0.31370 66458 77887 28733 79622 020D0,
     $         0.36268 37833 78361 98296 51504 493D0,
     $         0.27152 45941 17540 94851 78057 246D-1,
     $         0.62253 52393 86478 92862 84383 699D-1,
     $         0.95158 51168 24927 84809 92510 760D-1,
     $         0.12462 89712 55533 87205 24762 822D0,
     $         0.14959 59888 16576 73208 15017 305D0,
     $         0.16915 65193 95002 53818 93120 790D0,
     $         0.18260 34150 44923 58886 67636 680D0,
     $         0.18945 06104 55068 49628 53967 232D0/
C
      DATA X / 0.96028 98564 97536 23168 35608 686D0,
     $         0.79666 64774 13626 73959 15539 365D0,
     $         0.52553 24099 16328 98581 77390 492D0,
     $         0.18343 46424 95649 80493 94761 424D0,
     $         0.98940 09349 91649 93259 61541 735D0,
     $         0.94457 50230 73232 57607 79884 155D0,
     $         0.86563 12023 87831 74388 04678 977D0,
     $         0.75540 44083 55003 03389 51011 948D0,
     $         0.61787 62444 02643 74844 66717 640D0,
     $         0.45801 67776 57227 38634 24194 430D0,
     $         0.28160 35507 79258 91323 04605 015D0,
     $         0.95012 50983 76374 40185 31933 543D-1/
C
C     ****************************************************************
C
      GAUSS=0.0D0
      AD8GLE=GAUSS
      IF(B.EQ.A) RETURN
      CONST=EPS/(B-A)
      BB=A
C
C  COMPUTATIONAL LOOP.
    1 AA=BB
      BB=B
    2    C1=0.5D0*(BB+AA)
         C2=0.5D0*(BB-AA)
         S8=0.0D0
         DO 3 I=1,4
            U=C2*X(I)
            S8=S8+W(I)*(F(C1+U)+F(C1-U))
    3    CONTINUE
         S8=C2*S8
```

```
         S16=0.0D0
         DO 4 I=5,12
            U=C2*X(I)
            S16=S16+W(I)*(F(C1+U)+F(C1-U))
    4    CONTINUE
         S16=C2*S16
         IF( ABS(S16-S8) .LE. EPS*(abs(s8)+ABS(S16))*0.5D0 ) GO TO 5
         BB=C1
         IF( 1.D0+ABS(CONST*C2) .NE. 1.D0) GO TO 2
      AD8GLE=0.0D0
      write(*,*)'too high accuracy required in function ad8gle!'
      RETURN
    5 GAUSS=GAUSS+S16
      IF(BB.NE.B) GO TO 1
      AD8GLE=GAUSS
      RETURN
      END
C
C
      DOUBLE PRECISION FUNCTION ADGLG1(F,A,B,EPS)
      IMPLICIT REAL*8 (A-H,O-Z)
      EXTERNAL F,AD8GLE,adqua
      DIMENSION W(6),X(6),xx(6)
c      SAVE W, XX, NUM
      SAVE
C
C     ******************************************************************
C
C     ADAPTIVE GAUSSIAN QUADRATURE.
C     For x->b   f(x) = O (ln^k (b-x) )
C     A - lower limit, B - upper limit (integrable singularity)
C     AD8GLE IS SET EQUAL TO THE APPROXIMATE VALUE OF THE INTEGRAL OF
C     THE FUNCTION F OVER THE INTERVAL (A,B), WITH ACCURACY PARAMETER
C     EPS.
C
C     ******************************************************************
      DATA W / 4.58964 673950d-1,
     $         4.17000 830772d-1,
     $         1.13373 382074d-1,
     $         1.03991 974531d-2,
     $         2.61017 202815d-4,
     $         8.98547 906430d-7/
C
      DATA X / 0.22284 66041 79d0,
     $         1.18893 21016 73d0,
     $         2.99273 63260 59d0,
     $         5.77514 35691 05d0,
     $         9.83746 74183 83d0,
     $        15.98287 39806 02d0/
      DATA NUM/0/
      IF(NUM.eq.0d0) then
      do 1 ix=1,6
  1   xx(ix)= EXP(-x(ix))
      ENDIF
```

```
      num=num+1
      sum=0d0
      c=b-a
      sum6=0d0
      do 10 in=1,6
 10   sum6= sum6+ w(in)*f(b-c*xx(in))
      sum6=sum6*c
      a1=a
 15   a2= (a1+b)/2
      c=b-a2
      sumn=0d0
      do 20 in=1,6
      !!! FB: catch NaN
      if ( c/b .lt. 1d-9 ) then
        adglg1 = 1d15
        return
      endif
 20   sumn= sumn+ w(in)*f(b-c*xx(in)) !!! FB: f(b) = NaN !
      sumn=sumn*c
ctt
c      call adqua(a1,a2,f,sum1,eps)
c      sum1=sum1+sum
      sum1=AD8GLE(F,A1,A2,eps)+sum
      IF(ABS( (sum+sum6)/(sum1+sumn)-1d0 ).lt.EPS) THEN
ctt
c      call adqua(a,a2,f,sum2,eps)
         sum2=AD8GLE(F,A,A2,eps)
         IF(ABS( (sum2+sumn)/(sum1+sumn)-1d0 ).gt.EPS) THEN
            sum=sum2
            a1=a2
            sum6=sumn
            goto 15
         ENDIF
         ADGLG1= SUM1+SUMN
         RETURN
      ELSE
         sum=sum1
         a1=a2
         sum6=sumn
         goto 15
      ENDIF
      END
C
      DOUBLE PRECISION FUNCTION ADGLG2(F,A,B,EPS)
      IMPLICIT REAL*8 (A-H,O-Z)
      EXTERNAL F,AD8GLE
      DIMENSION W(6),X(6),xx(6)
c      SAVE W,XX,NUM
      SAVE
C
C     ******************************************************************
C
C     ADAPTIVE GAUSSIAN QUADRATURE.
C     For x->A   f(x) = O (ln^k (x-a) )
```

```
C     A - lower limit  (integrable singularity), B - upper limit
C     AD8GLE IS SET EQUAL TO THE APPROXIMATE VALUE OF THE INTEGRAL OF
C     THE FUNCTION F OVER THE INTERVAL (A,B), WITH ACCURACY PARAMETER
C     EPS.
C
C     ****************************************************************
      DATA W / 4.58964 673950d-1,
     $         4.17000 830772d-1,
     $         1.13373 382074d-1,
     $         1.03991 974531d-2,
     $         2.61017 202815d-4,
     $         8.98547 906430d-7/
C
      DATA X / 0.22284 66041 79d0,
     $         1.18893 21016 73d0,
     $         2.99273 63260 59d0,
     $         5.77514 35691 05d0,
     $         9.83746 74183 83d0,
     $        15.98287 39806 02d0/
      DATA NUM/0/
      IF(NUM.eq.0d0) then
      do 1 ix=1,6
  1   xx(ix)= EXP(-x(ix))
      ENDIF
      num=num+1
      sum=0d0
      c=b-a
      sum6=0d0
      do 10 in=1,6
 10   sum6= sum6+ w(in)*f(A+c*xx(in))
      sum6=sum6*c
      b1=b
 15   b2= (a+b1)/2
      c=b2-a
      sumn=0d0
      do 20 in=1,6
      !!! FB: catch NaN
      if ( c/a .lt. 1d-9 ) then
        adglg2 = 1d15
        return
      endif
 20   sumn= sumn+ w(in)*f(a+c*xx(in)) !!! FB: f(a) = NaN !
      sumn=sumn*c
      sum1=AD8GLE(F,b2,b1,eps)+sum
      IF(ABS( (sum+sum6)/(sum1+sumn)-1d0 ).lt.EPS) THEN
         sum2=AD8GLE(F,b2,b,eps)
         IF(ABS( (sum2+sumn)/(sum1+sumn)-1d0 ).gt.EPS) THEN
            sum=sum2
            b1=b2
            sum6=sumn
            goto 15
         ENDIF
         ADGLG2= SUM1+SUMN
         RETURN
```

```
      ELSE
         sum=sum1
         b1=b2
         sum6=sumn
         goto 15
      ENDIF
      END
C
C
C-----------------------------------------------------------------
C INTEGRATION ROUTINE ADQUA written by M. Jezabek            ------
C-----------------------------------------------------------------
C
      SUBROUTINE ADQUA(XL,XU,F,Y,ACC)
C
C     ADAPTIVE GAUSS-LEGENDRE + SIMPSON'S RULE QUADRATURE
C     XL - LOWER LIMIT, XU - UPPER LIMIT, F - FUNCTION TO INTEGRATE
C     Y - INTEGRAL
C     ACC - ACCURACY (IF .LE. 0.  ACC=1.D-6)
c     ****** new constants,  1 error removed, Oct '92
C
C     CALLS: SIMPSA
C
C     PARAMETERS: NSUB > NO OF SUBDIVISION LEVELS IN GAUSS INTEGRATION
C          100*2**IMAX > NO OF POINTS IN SIMPSON INTEGRATION
C
      IMPLICIT REAL*8 (A-H,O-Z)
      EXTERNAL F
      DIMENSION VAL(25,2), BOUND(25,2,2), LEV(25),SING(25,3)
      DIMENSION W8(4),X8(4)
      DATA W8
     $/0.101228536290376D0, 0.222381034453374D0, 0.313706645877887D0,
     $ 0.362683783378362D0/
      DATA X8
     $/0.960289856497536D0, 0.796666477413627D0, 0.525532409916329D0,
     $ 0.183434642495650D0/
      save
C
      IF(ACC.LE.0.D0) ACC=1.D-6
      NSUB=24
      NSG=25
      NSC=0
      A=XL
      B=XU
      C1=0.5d0*(A+B)
      C2=C1-A
      S8=0d0
      DO 1 I=1,4
      U=X8(I)*C2
    1 S8=S8+W8(I)*(F(C1+U)+F(C1-U))
      S8=S8*C2
      XM=(XL+XU)/2.d0
      BOUND(1,1,1)=XL
      BOUND(1,1,2)=XM
```

```
      BOUND(1,2,1)=XM
      BOUND(1,2,2)=XU
      NC=1
      DO 3 IX=1,2
      A=BOUND(NC,IX,1)
      B=BOUND(NC,IX,2)
      C1=0.5d0*(A+B)
      C2=C1-A
      VAL(NC,IX)=0.d0
      DO 2 I=1,4
      U=X8(I)*C2
    2 VAL(NC,IX)=VAL(NC,IX)+W8(I)*(F(C1+U)+F(C1-U))
    3 VAL(NC,IX)=VAL(NC,IX)*C2
      S16=VAL(NC,1)+VAL(NC,2)
      IF(DABS(S8-S16).GT.ACC*DABS(S16)) GOTO 4
      Y=S16
      RETURN
    4 DO 5 I=1,NSUB
    5 LEV(I)=0
      NC1= NC+1
   11 XM=(BOUND(NC,1,1)+BOUND(NC,1,2))/2.d0
      BOUND(NC1,1,1)=BOUND(NC,1,1)
      BOUND(NC1,1,2)=XM
      BOUND(NC1,2,1)=XM
      BOUND(NC1,2,2)=BOUND(NC,1,2)
      DO 13 IX=1,2
      A=BOUND(NC1,IX,1)
      B=BOUND(NC1,IX,2)
      C1=0.5d0*(A+B)
      C2=C1-A
      VAL(NC1,IX)=0.d0
      DO 12 I=1,4
      U=X8(I)*C2
   12 VAL(NC1,IX)=VAL(NC1,IX)+W8(I)*(F(C1+U)+F(C1-U))
   13 VAL(NC1,IX)=VAL(NC1,IX)*C2
      S16=VAL(NC1,1)+VAL(NC1,2)
      S8=VAL(NC,1)
      IF(DABS(S8-S16).LE.ACC*DABS(S16)) GOTO 20
      NC=NC1
      NC1= NC+1
      IF(NC1.LE.NSUB) GOTO 11
C     NC=NSUB   USE SIMPSON'S RULE
      NSC=NSC+1
      IF(NSC.LE.NSG) GOTO 15
      WRITE(*,911)
  911 FORMAT(1X,'ADQUA: TOO MANY SINGULARITIES')
      STOP
   15 SING(NSC,1)=BOUND(NC,1,1)
      SING(NSC,2)=BOUND(NC,2,2)
      SING(NSC,3)=S16
      S16=0.d0
      NC=NC-1
   20 VAL(NC,1)= S16
  121 LEV(NC)=1
```

```
   21 XM=(BOUND(NC,2,1)+BOUND(NC,2,2))/2.d0
      BOUND(NC1,1,1)=BOUND(NC,2,1)
      BOUND(NC1,1,2)=XM
      BOUND(NC1,2,1)=XM
      BOUND(NC1,2,2)=BOUND(NC,2,2)
      DO 23 IX=1,2
      A=BOUND(NC1,IX,1)
      B=BOUND(NC1,IX,2)
      C1=0.5d0*(A+B)
      C2=C1-A
      VAL(NC1,IX)=0.d0
      DO 22 I=1,4
      U=X8(I)*C2
   22 VAL(NC1,IX)=VAL(NC1,IX)+W8(I)*(F(C1+U)+F(C1-U))
   23 VAL(NC1,IX)=VAL(NC1,IX)*C2
      S16=VAL(NC1,1)+VAL(NC1,2)
      S8=VAL(NC,2)
      IF(DABS(S8-S16).LE.ACC*DABS(S16)) GOTO 40
      NC=NC+1
      NC1=NC+1
      IF(NC1.LE.NSUB) GOTO 11
C     NC=NSUB   USE SIMPSON'S RULE
      NSC=NSC+1
      IF(NSC.LE.NSG) GOTO 35
      WRITE(*,911)
      STOP
   35 SING(NSC,1)=BOUND(NC,1,1)
      SING(NSC,2)=BOUND(NC,2,2)
      SING(NSC,3)=S16
      S16=0.d0
      NC=NC-1
   40 VAL(NC,2)= S16
   45 IF(NC.GT.1) GOTO 50
      Y1=VAL(1,1)+VAL(1,2)
      GOTO 100
   50 NC0=NC-1
      IF(LEV(NC0).EQ.0) IX=1
      IF(LEV(NC0).EQ.1) IX=2
      LEV(NC)=0
      NC1=NC
      VAL(NC0,IX)=VAL(NC,1)+VAL(NC,2)
      NC=NC0
      IF(IX.EQ.1) GOTO 121
      GOTO 45
  100 CONTINUE
      IF(NSC.GT.0) GOTO 101
      Y=Y1
      RETURN
  101 FSUM=0.d0
      DO 102 IK=1,NSC
  102 FSUM=FSUM+DABS(SING(IK,3))
      ACCR=ACC*DMAX1(FSUM,DABS(Y1))/FSUM/10.d0
      DO 104 IK=1,NSC
  104 CALL SIMPSA(SING(IK,1),SING(IK,2),F,SING(IK,3),ACCR)
```

```
      DO 106 IK=1,NSC
  106 Y1=Y1+SING(IK,3)
      Y=Y1
      RETURN
      END
C
      SUBROUTINE SIMPSA(A,B,F,F0,ACC)
C     SIMPSON'S ADAPTIVE QUADRATURE
      IMPLICIT REAL*8 (A-H,O-Z)
      save
      EXTERNAL F
      IMAX=5
      N0=100
      H=(B-A)/N0
      N02=N0/2
      S2=0.d0
      IC=1
      S0=F(A)+F(B)
      DO 5 K=1,N02
    5 S2=S2+F(A+2.d0*K*H)
    7 S1=0.d0
      DO 10 K=1,N02
   10 S1=S1+F(A+(2.d0*K-1.d0)*H)
      Y=H/3.d0*(S0+4.d0*S1+2.d0*S2)
      IF(DABS(F0/Y-1.d0).GT.ACC) GOTO 20
      RETURN
   20 N02=N0
      N0=2*N0
      S2=S1+S2
      H=H/2.d0
      IF(IC.GT.IMAX) GOTO 30
      F0=Y
      IC=IC+1
      GOTO 7
   30 ACC0=DABS(Y/F0-1.d0)
      WRITE(*,900) A,B,ACC0
      STOP
  900 FORMAT(1H ,'SIMPSA: TOO HIGH ACCURACY REQUIRED'/
     /1X,   29HSINGULARITY IN THE INTERVAL  ,D20.12,1X,D20.12/
     /1X,   29HACCURACY ACHIEVED            ,D20.12)
      END
C
C
C  ******* matrix-inversion-routines
C
      SUBROUTINE LUDCMP(A,N,NP,INDX,D)
      IMPLICIT REAL*8(A-H,O-Z)
      PARAMETER (NMAX=100,TINY=1.0E-20)
      DIMENSION A(NP,NP),INDX(N),VV(NMAX)
      D=1.
      DO 12 I=1,N
        AAMAX=0.
        DO 11 J=1,N
          IF (ABS(A(I,J)).GT.AAMAX) AAMAX=ABS(A(I,J))
```

```
11      CONTINUE
!        IF (AAMAX.EQ.0.) PAUSE 'Singular matrix.'
        IF (AAMAX.EQ.0.) print *, 'Singular matrix.'
        VV(I)=1./AAMAX
12    CONTINUE
      DO 19 J=1,N
        IF (J.GT.1) THEN
          DO 14 I=1,J-1
            SUM=A(I,J)
            IF (I.GT.1)THEN
              DO 13 K=1,I-1
                SUM=SUM-A(I,K)*A(K,J)
13            CONTINUE
              A(I,J)=SUM
            ENDIF
14        CONTINUE
        ENDIF
        AAMAX=0.
        DO 16 I=J,N
          SUM=A(I,J)
          IF (J.GT.1)THEN
            DO 15 K=1,J-1
              SUM=SUM-A(I,K)*A(K,J)
15          CONTINUE
            A(I,J)=SUM
          ENDIF
          DUM=VV(I)*ABS(SUM)
          IF (DUM.GE.AAMAX) THEN
            IMAX=I
            AAMAX=DUM
          ENDIF
16      CONTINUE
        IF (J.NE.IMAX)THEN
          DO 17 K=1,N
            DUM=A(IMAX,K)
            A(IMAX,K)=A(J,K)
            A(J,K)=DUM
17        CONTINUE
          D=-D
          VV(IMAX)=VV(J)
        ENDIF
        INDX(J)=IMAX
        IF(J.NE.N)THEN
          IF(A(J,J).EQ.0.)A(J,J)=TINY
          DUM=1./A(J,J)
          DO 18 I=J+1,N
            A(I,J)=A(I,J)*DUM
18        CONTINUE
        ENDIF
19    CONTINUE
      IF(A(N,N).EQ.0.)A(N,N)=TINY
      RETURN
      END
c
```

```
      SUBROUTINE LUBKSB(A,N,NP,INDX,B)
      IMPLICIT REAL*8(A-H,O-Z)
      DIMENSION A(NP,NP),INDX(N),B(N)
      II=0
      DO 12 I=1,N
        LL=INDX(I)
        SUM=B(LL)
        B(LL)=B(I)
        IF (II.NE.0)THEN
          DO 11 J=II,I-1
            SUM=SUM-A(I,J)*B(J)
11        CONTINUE
        ELSE IF (SUM.NE.0.) THEN
          II=I
        ENDIF
        B(I)=SUM
12    CONTINUE
      DO 14 I=N,1,-1
        SUM=B(I)
        IF(I.LT.N)THEN
          DO 13 J=I+1,N
            SUM=SUM-A(I,J)*B(J)
13        CONTINUE
        ENDIF
        B(I)=SUM/A(I,I)
14    CONTINUE
      RETURN
      END
C
C
C     *******  RANDOM NUMBER GENERATORS
C
C
      FUNCTION RANF(DUMMY)
C
C   RANDOM NUMBER FUNCTION TAKEN FROM KNUTH
C   (SEMINUMERICAL ALGORITHMS).
C   METHOD IS X(N)=MOD(X(N-55)-X(N-24),1/FMODUL)
C   NO PROVISION YET FOR CONTROL OVER THE SEED NUMBER.
C
C   RANF GIVES ONE RANDOM NUMBER BETWEEN 0 AND 1.
C   IRN55 GENERATES 55 RANDOM NUMBERS BETWEEN 0 AND 1/FMODUL.
C   IN55  INITIALIZES THE 55 NUMBERS AND WARMS UP THE SEQUENCE.
C
      PARAMETER (FMODUL=1.E-09)
      SAVE /CIRN55/
      COMMON /CIRN55/NCALL,MCALL,IA(55)
      INTEGER IA
      CALL RANDAT
      IF( NCALL.EQ.0 ) THEN
          CALL IN55 ( IA,234612947 )
          MCALL = 55
          NCALL = 1
      ENDIF
```

```
      IF ( MCALL.EQ.0 ) THEN
          CALL IRN55(IA)
          MCALL=55
      ENDIF
      RANF=IA(MCALL)*FMODUL
      MCALL=MCALL-1
      RETURN
      END
C
      SUBROUTINE RANDAT
C
C  INITIALISES THE NUMBER NCALL TO 0 TO FLAG THE FIRST CALL
C  OF THE RANDOM NUMBER GENERATOR
C
C      SAVE /CIRN55/
C      SAVE FIRST
      SAVE
      COMMON /CIRN55/NCALL,MCALL,IA(55)
      INTEGER IA
      LOGICAL FIRST
      DATA FIRST /.TRUE./
      IF(FIRST)THEN
         FIRST=.FALSE.
         NCALL=0
      ENDIF
      RETURN
      END
C
      SUBROUTINE IN55(IA,IX)
      PARAMETER (MODULO=1000000000)
      INTEGER IA(55)
C
      IA(55)=IX
      J=IX
      K=1
      DO 10 I=1,54
         II=MOD(21*I,55)
         IA(II)=K
         K=J-K
         IF(K.LT.0)K=K+MODULO
         J=IA(II)
10    CONTINUE
      DO 20 I=1,10
         CALL IRN55(IA)
20    CONTINUE
      RETURN
      END
C
      SUBROUTINE IRN55(IA)
      PARAMETER (MODULO=1000000000)
      INTEGER IA(55)
      DO 10 I=1,24
         J=IA(I)-IA(I+31)
         IF(J.LT.0)J=J+MODULO
```

```
         IA(I)=J
10    CONTINUE
      DO 20 I=25,55
         J=IA(I)-IA(I-24)
         IF(J.LT.0)J=J+MODULO
         IA(I)=J
20    CONTINUE
      RETURN
      END
C
C
      FUNCTION RAN2(IDUM)
C     *******************
      REAL RDM(31)
      DATA  IWARM/0/
C
      IF (IDUM.LT.0.OR.IWARM.EQ.0) THEN
C INITIALIZATION OR REINITIALISATION
      IWARM=1
      IA1=         1279
      IC1=       351762
      M1=       1664557
      IA2=         2011
      IC2=       221592
      M2=       1048583
      IA3=        15091
      IC3=         6171
      M3=         29201
      IX1=MOD(-IDUM,M1)
      IX1=MOD(IA1*IX1+IC1,M1)
      IX2=MOD(IX1,M2)
      IX1=MOD(IA1*IX1+IC1,M1)
      IX3=MOD(IX1,M3)
      RM1=1./FLOAT(M1)
      RM2=1./FLOAT(M2)
      DO 10 J=1,31
      IX1=MOD(IA1*IX1+IC1,M1)
      IX2=MOD(IA2*IX2+IC2,M2)
10    RDM(J)=(FLOAT(IX1)+FLOAT(IX2)*RM2)*RM1
      ENDIF
C
C GENERATE NEXT NUMBER IN SEQUENCE
      IF(IWARM.EQ.0) GOTO 901
      IX1=MOD(IA1*IX1+IC1,M1)
      IX2=MOD(IA2*IX2+IC2,M2)
      IX3=MOD(IA3*IX3+IC3,M3)
      J=1+(31*IX3)/M3
      RAN2=RDM(J)
      RDM(J)=(FLOAT(IX1)+FLOAT(IX2)*RM2)*RM1
      RETURN
901   PRINT 9010
9010  FORMAT('   RAN2: LACK OF ITINIALISATION')
      STOP
      END
```

```
C
C
C     *******    SPECIAL FUNCTIONS
C
C
      DOUBLE PRECISION FUNCTION DILOG(X)
C
C     SPENCE'S DILOGARITHM IN DOUBLE PRECISION
C
      IMPLICIT REAL*8 (A-H,O-Z)
      Z=-1.644934066848226
      IF(X .LT.-1.0) GO TO 1
      IF(X .LE. 0.5) GO TO 2
      IF(X .EQ. 1.0) GO TO 3
      IF(X .LE. 2.0) GO TO 4
      Z=3.289868133696453
    1 T=1.0/X
      S=-0.5
      Z=Z-0.5*DLOG(DABS(X))**2
      GO TO 5
    2 T=X
      S=0.5
      Z=0.
      GO TO 5
    3 DILOG=1.644934066848226
      RETURN
    4 T=1.0-X
      S=-0.5
      Z=1.644934066848226-DLOG(X)*DLOG(DABS(T))
    5 Y=2.666666666666667*T+0.666666666666667
      B=      0.00000 00000 00001
      A=Y*B  +0.00000 00000 00004
      B=Y*A-B+0.00000 00000 00011
      A=Y*B-A+0.00000 00000 00037
      B=Y*A-B+0.00000 00000 00121
      A=Y*B-A+0.00000 00000 00398
      B=Y*A-B+0.00000 00000 01312
      A=Y*B-A+0.00000 00000 04342
      B=Y*A-B+0.00000 00000 14437
      A=Y*B-A+0.00000 00000 48274
      B=Y*A-B+0.00000 00001 62421
      A=Y*B-A+0.00000 00005 50291
      B=Y*A-B+0.00000 00018 79117
      A=Y*B-A+0.00000 00064 74338
      B=Y*A-B+0.00000 00225 36705
      A=Y*B-A+0.00000 00793 87055
      B=Y*A-B+0.00000 02835 75385
      A=Y*B-A+0.00000 10299 04264
      B=Y*A-B+0.00000 38163 29463
      A=Y*B-A+0.00001 44963 00557
      B=Y*A-B+0.00005 68178 22718
      A=Y*B-A+0.00023 20021 96094
      B=Y*A-B+0.00100 16274 96164
      A=Y*B-A+0.00468 63619 59447
```

```
      B=Y*A-B+0.02487 93229 24228
      A=Y*B-A+0.16607 30329 27855
      A=Y*A-B+1.93506 43008 69969
      DILOG=S*T*(A-B)+Z
      RETURN
      END
C
c Everything for hypergeometric function F_{2,1},
c  taken from Numerical Recipes.
c
!!! FB:
!       FUNCTION hypgeo(a,b,c,z)
!       implicit none
!       COMPLEX*16 hypgeo,a,b,c,z,hp
!       REAL*8 EPS
! cttt
! c      PARAMETER (EPS=1.d-6)
!       PARAMETER (EPS=1.d-8)
!       INTEGER kmax,nbad,nok
!       COMPLEX*16 z0,dz,aa,bb,cc,y(2)
! cu    needs bsstep,hypser,odeint,mmid,hypdrv
!       COMMON /hypg/ aa,bb,cc,z0,dz
!       COMMON /path1/ kmax
!       kmax=0
!       if (real(z)**2+aimag(z)**2.le.0.25d0) then
!         call hypser(a,b,c,z,hp,y(2))
!         hypgeo=hp
!         return
!       else if (real(z).lt.0.d0) then
!         z0=cmplx(-0.5d0,0.d0,kind=kind(0d0))
!       else if (real(z).le.1.d0) then
!         z0=cmplx(0.5d0,0.d0,kind=kind(0d0))
!       else
!         z0=cmplx(0.d0,sign(0.5d0,aimag(z)),kind=kind(0d0))
!       endif
!       aa=a
!       bb=b
!       cc=c
!       dz=z-z0
!       call hypser(aa,bb,cc,z0,y(1),y(2))
!       call odeint(y,4,0.d0,1.d0,EPS,.1d0,.00001d0,nok,nbad)
!       hypgeo=y(1)
!       return
!       END
! c
!       SUBROUTINE odeint(ystart,nvar,x1,x2,eps,h1,hmin,nok,nbad)
!       implicit none
!       INTEGER nbad,nok,nvar,KMAXX,MAXSTP,NMAX
!       REAL*8 eps,h1,hmin,x1,x2,ystart(nvar),TINY
!       PARAMETER (MAXSTP=10000,NMAX=50,KMAXX=200,TINY=1.d-30)
!       INTEGER i,kmax,kount,nstp
!       REAL*8 dxsav,h,hdid,hnext,x,xsav,dydx(NMAX),xp(KMAXX),y(NMAX),
!      *yp(NMAX,KMAXX),yscal(NMAX)
!       COMMON /path/ dxsav,xp,yp,kount
```

```
!       COMMON /path1/ kmax
!       x=x1
!       h=sign(h1,x2-x1)
!       nok=0
!       nbad=0
!       kount=0
!       do 11 i=1,nvar
!         y(i)=ystart(i)
! 11    continue
!       if (kmax.gt.0) xsav=x-2.d0*dxsav
!       do 16 nstp=1,MAXSTP
!         call hypdrv(x,y,dydx)
!         do 12 i=1,nvar
!           yscal(i)=abs(y(i))+abs(h*dydx(i))+TINY
! 12      continue
!         if(kmax.gt.0)then
!           if(abs(x-xsav).gt.abs(dxsav)) then
!             if(kount.lt.kmax-1)then
!               kount=kount+1
!               xp(kount)=x
!               do 13 i=1,nvar
!                 yp(i,kount)=y(i)
! 13            continue
!               xsav=x
!             endif
!           endif
!         endif
!         if((x+h-x2)*(x+h-x1).gt.0.d0) h=x2-x
!         call bsstep(y,dydx,nvar,x,h,eps,yscal,hdid,hnext)
!         if(hdid.eq.h)then
!           nok=nok+1
!         else
!           nbad=nbad+1
!         endif
!         if((x-x2)*(x2-x1).ge.0.d0)then
!           do 14 i=1,nvar
!             ystart(i)=y(i)
! 14        continue
!           if(kmax.ne.0)then
!             kount=kount+1
!             xp(kount)=x
!             do 15 i=1,nvar
!               yp(i,kount)=y(i)
! 15          continue
!           endif
!           return
!         endif
!         if(abs(hnext).lt.hmin) pause
!      *'stepsize smaller than minimum in odeint'
!         h=hnext
! 16    continue
!       pause 'too many steps in odeint'
!       return
!       END
```

```
! c
!       SUBROUTINE bsstep(y,dydx,nv,x,htry,eps,yscal,hdid,hnext)
!       implicit none
!       INTEGER nv,NMAX,KMAXX,IMAX
!       REAL*8 eps,hdid,hnext,htry,x,dydx(nv),y(nv),yscal(nv),
!      *SAFE1,SAFE2,REDMAX,REDMIN,TINY,SCALMX
!       PARAMETER (NMAX=50,KMAXX=8,IMAX=KMAXX+1,SAFE1=.25d0,SAFE2=.7d0,
!      *REDMAX=1.d-5,REDMIN=.7d0,TINY=1.d-30,SCALMX=.1d0)
!       INTEGER i,iq,k,kk,km,kmax,kopt,nseq(IMAX)
!       REAL*8 eps1,epsold,errmax,fact,h,red,scale,work,wrkmin,xest,
!      *xnew,a(IMAX),alf(KMAXX,KMAXX),err(KMAXX),yerr(NMAX),ysav(NMAX),
!      *yseq(NMAX)
!       LOGICAL first,reduct
!       SAVE a,alf,epsold,first,kmax,kopt,nseq,xnew
!       DATA first/.true./,epsold/-1.d0/
!       DATA nseq /2,4,6,8,10,12,14,16,18/
!       if(eps.ne.epsold)then
!         hnext=-1.d29
!         xnew=-1.d29
!         eps1=SAFE1*eps
!         a(1)=nseq(1)+1
!         do 11 k=1,KMAXX
!           a(k+1)=a(k)+nseq(k+1)
! 11      continue
!         do 13 iq=2,KMAXX
!           do 12 k=1,iq-1
!             alf(k,iq)=eps1**((a(k+1)-a(iq+1))/((a(iq+1)-a(1)+1.d0)*
!      *(2.d0*k+1.d0)))
! 12        continue
! 13      continue
!         epsold=eps
!         do 14 kopt=2,KMAXX-1
!           if(a(kopt+1).gt.a(kopt)*alf(kopt-1,kopt))goto 1
! 14      continue
! 1       kmax=kopt
!       endif
!       h=htry
!       do 15 i=1,nv
!         ysav(i)=y(i)
! 15    continue
!       if(h.ne.hnext.or.x.ne.xnew)then
!         first=.true.
!         kopt=kmax
!       endif
!       reduct=.false.
! 2     do 17 k=1,kmax
!         xnew=x+h
!         if(xnew.eq.x)pause 'step size underflow in bsstep'
!         call mmid(ysav,dydx,nv,x,h,nseq(k),yseq)
!         xest=(h/nseq(k))**2
!         call pzext0(k,xest,yseq,y,yerr,nv)
!         if(k.ne.1)then
!           errmax=TINY
!           do 16 i=1,nv
```

```
!             errmax=max(errmax,abs(yerr(i)/yscal(i)))
! 16        continue
!           errmax=errmax/eps
!           km=k-1
!           err(km)=(errmax/SAFE1)**(1.d0/(2.d0*km+1.d0))
!         endif
!         if(k.ne.1.and.(k.ge.kopt-1.or.first))then
!           if(errmax.lt.1.)goto 4
!           if(k.eq.kmax.or.k.eq.kopt+1)then
!             red=SAFE2/err(km)
!             goto 3
!           else if(k.eq.kopt)then
!             if(alf(kopt-1,kopt).lt.err(km))then
!               red=1.d0/err(km)
!               goto 3
!             endif
!           else if(kopt.eq.kmax)then
!             if(alf(km,kmax-1).lt.err(km))then
!               red=alf(km,kmax-1)*SAFE2/err(km)
!               goto 3
!             endif
!           else if(alf(km,kopt).lt.err(km))then
!             red=alf(km,kopt-1)/err(km)
!             goto 3
!           endif
!         endif
! 17    continue
! 3     red=min(red,REDMIN)
!       red=max(red,REDMAX)
!       h=h*red
!       reduct=.true.
!       goto 2
! 4     x=xnew
!       hdid=h
!       first=.false.
!       wrkmin=1.d35
!       do 18 kk=1,km
!         fact=max(err(kk),SCALMX)
!         work=fact*a(kk+1)
!         if(work.lt.wrkmin)then
!           scale=fact
!           wrkmin=work
!           kopt=kk+1
!         endif
! 18    continue
!       hnext=h/scale
!       if(kopt.ge.k.and.kopt.ne.kmax.and..not.reduct)then
!         fact=max(scale/alf(kopt-1,kopt),SCALMX)
!         if(a(kopt+1)*fact.le.wrkmin)then
!           hnext=h/fact
!           kopt=kopt+1
!         endif
!       endif
!       return
```

```
!       END
! c
!       SUBROUTINE hypser(a,b,c,z,series,deriv)
!       implicit none
!       INTEGER n
!       COMPLEX*16 a,b,c,z,series,deriv,aa,bb,cc,fac,temp
!       deriv=cmplx(0.d0,0.d0,kind=kind(0d0))
!       fac=cmplx(1.d0,0.d0,kind=kind(0d0))
!       temp=fac
!       aa=a
!       bb=b
!       cc=c
!       do 11 n=1,1000
!         fac=fac*aa*bb/cc
!         deriv=deriv+fac
!         fac=fac*z/n
!         series=temp+fac
!         if (series.eq.temp) return
!         temp=series
!         aa=aa+1.d0
!         bb=bb+1.d0
!         cc=cc+1.d0
! 11    continue
!       pause 'convergence failure in hypser'
!       END
! c
!       SUBROUTINE hypdrv(s,y,dyds)
!       implicit none
!       REAL*8 s
!       COMPLEX*16 y(2),dyds(2),aa,bb,cc,z0,dz,z
!       COMMON /hypg/ aa,bb,cc,z0,dz
!       z=z0+s*dz
!       dyds(1)=y(2)*dz
!       dyds(2)=(aa*bb*y(1)-(cc-(aa+bb+1.d0)*z)*y(2))*dz/(z*(1.d0-z))
!       return
!       END
! c
!       SUBROUTINE mmid(y,dydx,nvar,xs,htot,nstep,yout)
!       implicit none
!       INTEGER nstep,nvar,NMAX
!       REAL*8 htot,xs,dydx(nvar),y(nvar),yout(nvar)
!       PARAMETER (NMAX=50)
!       INTEGER i,n
!       REAL*8 h,h2,swap,x,ym(NMAX),yn(NMAX)
!       h=htot/nstep
!       do 11 i=1,nvar
!         ym(i)=y(i)
!         yn(i)=y(i)+h*dydx(i)
! 11    continue
!       x=xs+h
!       call hypdrv(x,yn,yout)
!       h2=2.d0*h
!       do 13 n=2,nstep
!         do 12 i=1,nvar
```

```
!           swap=ym(i)+h2*yout(i)
!           ym(i)=yn(i)
!           yn(i)=swap
! 12      continue
!         x=x+h
!         call hypdrv(x,yn,yout)
! 13    continue
!       do 14 i=1,nvar
!         yout(i)=0.5d0*(ym(i)+yn(i)+h*yout(i))
! 14    continue
!       return
!       END
!!! /FB
c
      SUBROUTINE pzext0(iest,xest,yest,yz,dy,nv)
      implicit none
      INTEGER iest,nv,IMAX,NMAX
      REAL*8 xest,dy(nv),yest(nv),yz(nv)
      PARAMETER (IMAX=13,NMAX=50)
      INTEGER j,k1
      REAL*8 delta,f1,f2,q,d(NMAX),qcol(NMAX,IMAX),x(IMAX)
      SAVE qcol,x
      x(iest)=xest
      do 11 j=1,nv
        dy(j)=yest(j)
        yz(j)=yest(j)
11    continue
      if(iest.eq.1) then
        do 12 j=1,nv
          qcol(j,1)=yest(j)
12      continue
      else
        do 13 j=1,nv
          d(j)=yest(j)
13      continue
        do 15 k1=1,iest-1
          delta=1.d0/(x(iest-k1)-xest)
          f1=xest*delta
          f2=x(iest-k1)*delta
          do 14 j=1,nv
            q=qcol(j,k1)
            qcol(j,k1)=dy(j)
            delta=d(j)-q
            dy(j)=f1*delta
            d(j)=f2*delta
            yz(j)=yz(j)+dy(j)
14        continue
15      continue
        do 16 j=1,nv
          qcol(j,iest)=dy(j)
16      continue
      endif
      return
      END
```

```
c
c
        complex*16 function zdigamma(z)
        implicit none
        complex*16 z,psi,psipr1,psipr2
        call mkpsi(z,psi,psipr1,psipr2)
        zdigamma=psi
        end
c
      subroutine mkpsi(z,psi,psipr1,psipr2)
      implicit none
      complex*16 tmp,tmps2,tmps3,tmp0,tmp1,tmp2,ser0,ser1,ser2,ser3,
     .           zz,z,psi,psipr1,psipr2,off0,off1,off2,zcf,ser02,ser12,
     .           z1,z2
      real*8 cof(6),re1
      integer i
      data cof/76.18009173d0,-86.50532033d0,24.01409822d0,
     .    -1.231739516d0,.120858003d-2,-.536382d-5/
      save
      zz=z
      off0=cmplx(0.d0,0.d0,kind=kind(0d0))
      off1=cmplx(0.d0,0.d0,kind=kind(0d0))
      off2=cmplx(0.d0,0.d0,kind=kind(0d0))
    5 re1=real(zz)
      if (re1.le.0.d0) then
         off0=off0+1.d0/zz
         z1=zz*zz
         off1=off1-1.d0/z1
         z2=z1*zz
         off2=off2+2.d0/z2
         zz=zz+(1.d0,0.d0)
         goto 5
      endif
      tmp=zz+cmplx(4.5d0,0.d0,kind=kind(0d0))
      tmps2=tmp*tmp
      tmps3=tmp*tmps2
      tmp0=(zz-cmplx(0.5d0,0.d0,kind=kind(0d0)))/tmp+log(tmp)
     u     -cmplx(1.d0,0.d0,kind=kind(0d0))
      tmp1=(5.d0,0.d0)/tmps2+1.d0/tmp
      tmp2=(-10.0d0,0.d0)/tmps3-1.d0/tmps2
      ser0=cmplx(1.d0,0.d0,kind=kind(0d0))
      ser1=cmplx(0.d0,0.d0,kind=kind(0d0))
      ser2=cmplx(0.d0,0.d0,kind=kind(0d0))
      ser3=cmplx(0.d0,0.d0,kind=kind(0d0))
      do 10 i=1,6
         zcf=cof(i)/zz
         ser0=ser0+zcf
         zcf=zcf/zz
         ser1=ser1+zcf
         zcf=zcf/zz
         ser2=ser2+zcf
         zcf=zcf/zz
         ser3=ser3+zcf
         zz=zz+(1.d0,0.d0)
```

```
   10 continue
      ser1=-ser1
      ser2=2.d0*ser2
      ser3=-6.d0*ser3
      ser02=ser0*ser0
      ser12=ser1*ser1
      psi=tmp0+ser1/ser0-off0
      psipr1=tmp1+(ser2*ser0-ser12)/ser02-off1
      psipr2=tmp2+(ser3*ser02-3.d0*ser2*ser1*ser0+2.d0*ser12*ser1)
     .            /ser02/ser0-off2
      return
      end
```

⟨toppik_axial.f⟩≡

```
! WHIZARD ⟨Version⟩ ⟨Date⟩

! TOPPIK code by M. Jezabek, T. Teubner (v1.1, 1992), T. Teubner (1998)
!
! NOTE: axial part (p-wave) only
!
! FB: -commented out numerical recipes code for hypergeometric 2F1
!      included in hypgeo.f90;
!     -replaced function 'cdabs' by 'abs';
!     -replaced function 'dabs' by 'abs';
!     -replaced function 'dimag' by 'aimag';
!     -replaced function 'dcmplx(,)' by 'cmplx(,,kind=kind(0d0))';
!     -replaced function 'dreal' by 'real';
!     -replaced function 'dlog' by 'log';
!     -replaced function 'dsqrt' by 'sqrt';
!     -renamed function 'a' to 'aax'
!     -renamed function 'fretil1' to 'fretil1ax'
!     -renamed function 'fretil2' to 'fretil2ax'
!     -renamed function 'fimtil1' to 'fimtil1ax'
!     -renamed function 'fimtil2' to 'fimtil2ax'
!     -renamed function 'freal' to 'frealax'
!     -renamed function 'fim' to 'fimax'
!     -renamed subroutine 'vhat' to 'vhatax'
!     -renamed subroutine 'sae' to 'saeax'
!     -commented out many routines identically defined in 'toppik.f'
!     -modified 'tttoppikaxial' to catch unstable runs.
!
!!!
```

```
     u     xkincm,xkinca,jknflg,jgcflg,xkincv,jvflg,
     u     xim,xdi,np,xpp,xww,xdsdp,zftild)
c
c ***********************************************************************
c
c !! THIS IS NOT A PUBLIC VERSION !!
c
c !!! Only P wave result given as output!!! 9.4.1999, tt.
c
c -- Calculation of the Green function in momentum space by solving the
c     Lippmann-Schwinger equation
c     F(p) = G_0(p) + G_0(p) int_0^xcutn V(p,q) q.p/p^2 F(q) dq
c
c -- Written by Thomas Teubner, Hamburg, November 1998
c     * Based on TOPPIK Version 1.1
c        from M. Jezabek and TT, Karlsruhe, June 1992
c     * Version originally for non-constant top-width
c     * Constant width supplied here
c     * No generator included
c
c -- Use of double precision everywhere
c
c -- All masses, momenta, energies, widths in GeV
c
c -- Input parameters:
c
c    xenergy  :  E=Sqrt[s]-2*topmass
c    xtm      :  topmass (in the Pole scheme)
c    xtg      :  top-width
c    xalphas  :  alpha_s^{MSbar,n_f=5}(xscale)
c    xscale   :  soft scale  mu_{soft}
c    xcutn    :  numerical UV cutoff on all momenta
c                (UV cutoff of the Gauss-Legendre grid)
c    xcutv    :  renormalization cutoff on the
c                 delta-, the (p^2+q^2)/(p-q)^2-, and the
c                  1/r^2-[1/|p-q|]-potential:
c                 if (max(p,q).ge.xcutv) then the three potentials
c                  are set to zero in the Lippmann-Schwinger equation
c    xc0      :  0th order coefficient for the Coulomb potential,
c                 see calling example above
c    xc1      :  1st order coefficient for the Coulomb potential
c    xc2      :  2nd order coefficient for the Coulomb potential
c    xcdeltc  :  constant of the delta(r)-
c                 [= constant in momentum space-] potential
c    xcdeltl  :  constant for the additional log(q^2/mu^2)-part of the
c                 delta-potential:
c                  xcdeltc*1 + xcdeltl*log(q^2/mu^2)
c    xcfullc  :  constant of the (p^2+q^2)/(p-q)^2-potential
c    xcfulll  :  constant for the additional log(q^2/mu^2)-part of the
c                 (p^2+q^2)/(p-q)^2-potential
c    xcrm2    :  constant of the 1/r^2-[1/|p-q|]-potential
c    xkincm   :  } kinetic corrections in the 0th order Green function:
c    xkinca   :  }  G_0(p):=1/[E+iGamma_t-p^2/m_t]*(1+xkincm)+xkinca
c     !!! WATCH THE SIGN IN G_0 !!!
```

```
c    jknflg   :  flag for these kinetic corrections:
c                 0 : no kinetic corrections applied
c                 1 : kinetic corrections applied with cutoff xcutv
c                      for  xkinca  only
c                 2 : kinetic corrections applied with cutoff xcutv
c                      for  xkinca  AND  xkincm
c    jgcflg   :  flag for G_0(p) in the LS equation:
c                 0 (standard choice) : G_0(p) as given above
c                 1 (for TIPT)        : G_0(p) = G_c^{0}(p) the 0th
c                                         order Coulomb Green function
c                                         in analytical form; not for
c                                         momenta  p > 1000*topmass
c    xkincv   :  additional kinematic vertexcorrection in G_0, see below:
c    jvflg    :  flag for the additional vertexcorrection  xkincv  in the
c                 ``zeroth order'' G_0(p) in the LS-equation:
c                 0 : no correction, means  G = G_0 + G_0 int V G
c                      with G_0=1/[E+iGamma_t-p^2/m_t]*(1+xkincm)+xkinca
c                 1 : apply the correction in the LS equation as
c                      G = G_0 + xkincv*p^2/m_t^2/[E+iGamma_t-p^2/m_t] +
c                          G_0 int V G
c                     and correct the integral over Im[G(p)] to get sigma_tot
c                     from the optical theorem by the same factor.
c                     The cutoff  xcutv  is applied for these corrections.
c
c -- Output:
c
c    xim      :  R^{P wave}_{ttbar} from the imaginary part of the Green
c                 function
c    xdi      :  R^{P wave}_{ttbar} from the integral over the momentum
c                 distribution: int_0^xcutv dp p^3/m_t*|F(p,E)|^2
c    np       :  number of points used for the grid; fixed in tttoppik
c    xpp      :  1-dim array (max. 400 elements) giving the momenta of
c                 the Gauss-Legendre grid (pp(i) in the code)
c    xww      :  1-dim array (max. 400 elements) giving the corresponding
c                 Gauss-Legendre weights for the grid
c    xdsdp    :  1-dim array (max. 400 elements) giving the
c                 momentum distribution of top: d\sigma^{P wave}/dp,
c                  normalized to R,
c                  at the momenta of the Gauss-Legendre grid xpp(i)
c    zftild   :  1-dim array (max. 400 elements) of COMPLEX*16 numbers
c                 giving the vertex function K_A for the P-wave
c                 at the momenta of the grid.
c                 Then F(p)=K_A (p)*G_0(p) corresponding to G=K_V*G_0.
c
c ******************************************************************
c
c
          implicit none
          real*8
     u       pi,energy,vzero,eps,
     u       pp,
     u       tmass,tgamma,zmass,alphas,alamb5,
     u       wmass,wgamma,bmass,GFERMI,
     u       xx,critp,consde,
```

```
     u        w1,w2,sig1,sig2,const,
     u        gtpcor,etot,
     u        xenergy,xtm,xtg,xalphas,xscale,xc0,xc1,xc2,xim,xdi,
     u        xaai,xaad,xdsdp,xpp,xww,
     u        cplas,scale,c0,c1,c2,cdeltc,cdeltl,cfullc,cfulll,crm2,
     u        xcutn,dcut,xcutv,
     u        xp,xpmax,
     u        kincom,kincoa,kincov,xkincm,xkinca,xkincv,
     u        xcdeltc,xcdeltl,xcfullc,xcfulll,xcrm2
           complex*16 bb,vec,gg,a1,aax,g0,g0c,zvfct,zftild
           integer i,n,nmax,npot,np,gcflg,kinflg,jknflg,jgcflg,
     u             jvflg,vflag
           parameter (nmax=400)
           dimension pp(nmax),bb(nmax),vec(nmax),xx(nmax),gg(nmax),
     u               w1(nmax),w2(nmax),a1(nmax),
     u               xdsdp(nmax),xpp(nmax),xww(nmax),
     u               zvfct(nmax),zftild(nmax)
c
           external aax,gtpcor,g0,g0c
c
           common/ovalco/ pi, energy, vzero, eps, npot
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
           common/cplcns/cplas,scale,c0,c1,c2,
     u                   cdeltc,cdeltl,cfullc,cfulll,crm2
           common/mom/ xp,xpmax,dcut
           common/g0inf/kincom,kincoa,kincov,kinflg,gcflg,vflag
c
           pi=3.141592653589793238d0
c
c Number of points to evaluate on the integral equation
c  (<=400 and n mod 3 = 0 !!):
           n=360
           np=n
c
c For second order potential with free parameters:
c
           npot=5
c Internal accuracy for TOPPIK, the reachable limit may be smaller,
c  depending on the parameters. But increase in real accuracy only
c  in combination with large number of points.
           eps=1.d-3
c Some physical parameters:
           wgamma=2.07d0
           zmass=91.187d0
           wmass=80.33d0
           bmass=4.7d0
c
c Input:
           tmass=xtm
           energy=xenergy
           tgamma=xtg
           cplas=xalphas
           scale=xscale
```

```
          c0=xc0
          c1=xc1
          c2=xc2
          cdeltc=xcdeltc
          cdeltl=xcdeltl
          cfullc=xcfullc
          cfulll=xcfulll
          crm2=xcrm2
          kincom=xkincm
          kincoa=xkinca
          kincov=xkincv
          kinflg=jknflg
          gcflg=jgcflg
          vflag=jvflg
c
          alphas=xalphas
c
c Cut for divergent potential-terms for large momenta in the function vhatax
c  and in the integrals aax(p):
          dcut=xcutv
c
c Numerical Cutoff of all momenta (maximal momenta of the grid):
          xpmax=xcutn
          if (dcut.gt.xpmax) then
             write(*,*) ' dcut > xpmax  makes no sense! Stop.'
             stop
          endif
c
c Not needed for the fixed order potentials:
          alamb5=0.2d0
c
c      WRITE(*,*) 'INPUT TGAMMA=',TGAMMA
c Needed in subroutine GAMMAT:
          GFERMI=1.16637d-5
c          CALL GAMMAT
c          WRITE(*,*) 'CALCULATED TGAMMA=',TGAMMA
c
          etot=2.d0*tmass+energy
c
          if ((npot.eq.1).or.(npot.eq.3).or.(npot.eq.4).or.
     u         (npot.eq.5)) then
c For pure coulomb and fixed order potentials there is no delta-part:
             consde = 0.d0
          else if (npot.eq.2) then
c Initialize QCD-potential common-blocks and calculate constant multiplying
c  the delta-part of the 'qcutted' potential in momentum-space:
c            call iniphc(1)
c            call vqdelt(consde)
             write(*,*) ' Not supplied with this version. Stop.'
             stop
          else
             write (*,*) ' Potential not implemented! Stop. 1'
             stop
          endif
```

```
c Delta-part of potential is absorbed by subtracting vzero from the
c  original energy (shift from the potential to the free Hamiltonian):
           vzero = consde / (2.d0*pi)**3
c          write (*,*) 'vzero=', vzero
c
c Find x-values pp(i) and weigths w1(i) for the gaussian quadrature;
c  care about large number of points in the important intervals:
c       if (energy-vzero.le.0.d0) then
cc         call gauleg(0.d0, 1.d0, pp, w1, n/3)
cc         call gauleg(1.d0, 5.d0, pp(n/3+1), w1(n/3+1), n/3)
cc         call gauleg(0.d0, 0.2d0, pp(2*n/3+1), w1(2*n/3+1), n/3)
c          call gauleg(0.d0, 5.d0, pp, w1, n/3)
c          call gauleg(5.d0, 20.d0, pp(n/3+1), w1(n/3+1), n/3)
c          call gauleg(0.d0, 0.05d0, pp(2*n/3+1), w1(2*n/3+1), n/3)
c       else
cc Avoid numerical singular points in the inner of the intervals:
c          critp = sqrt((energy-vzero)*tmass)
c          if (critp.le.1.d0) then
cc Gauss-Legendre is symmetric => automatically principal-value prescription:
c             call gauleg(0.d0, 2.d0*critp, pp, w1, n/3)
c             call gauleg(2.d0*critp, 20.d0, pp(n/3+1),
c     u                   w1(n/3+1), n/3)
c             call gauleg(0.d0, 0.05d0, pp(2*n/3+1), w1(2*n/3+1), n/3)
c          else
cc Better behaviour at the border of the intervals:
c             call gauleg(0.d0, critp, pp, w1, n/3)
c             call gauleg(critp, 2.d0*critp, pp(n/3+1),
c     u                   w1(n/3+1), n/3)
c             call gauleg(0.d0, 1.d0/(2.d0*critp), pp(2*n/3+1),
c     u                   w1(2*n/3+1), n/3)
c          endif
c       endif
c
c Or different (simpler) method, good for V_JKT:
           if (energy.le.0.d0) then
              critp=tmass/3.d0
           else
              critp=max(tmass/3.d0,2.d0*sqrt(energy*tmass))
           endif
           call gauleg(0.d0, critp, pp, w1, 2*n/3)
           call gauleg(1.d0/xpmax, 1.d0/critp, pp(2*n/3+1),
     u                 w1(2*n/3+1), n/3)
c
c Do substitution p => 1/p for the last interval explicitly:
           do 10 i=2*n/3+1,n
              pp(i) = 1.d0/pp(i)
10         continue
c
c Reorder the arrays for the third interval:
           do 20 i=1,n/3
              xx(i) = pp(2*n/3+i)
              w2(i) = w1(2*n/3+i)
20         continue
           do 30 i=1,n/3
```

```
               pp(n-i+1) = xx(i)
               w1(n-i+1) = w2(i)
30          continue
c
c Calculate the integrals aax(p) for the given momenta pp(i)
c  and store weights and momenta for the output arrays:
            do 40 i=1,n
               a1(i) = aax(pp(i)) !!! FB: can get stuck in original Toppik!
               !!! FB: abuse 'np' as a flag to communicate unstable runs
               if ( abs(a1(i)) .gt. 1d10 ) then
                 np = -1
                 return
               endif
               xpp(i)=pp(i)
               xww(i)=w1(i)
40          continue
            do 41 i=n+1,nmax
               xpp(i)=0.d0
               xww(i)=0.d0
41          continue
c
c Solve the integral-equation by solving a system of algebraic equations:
            call saeax(pp, w1, bb, vec, a1, n)
c
c (The substitution for the integration to infinity  pp => 1/pp
c  is done already.)
            do 50 i=1,n
               zvfct(i)=bb(i)
               zftild(i)=vec(i)
               gg(i) = bb(i)*g0c(pp(i))
cc             gg(i) = (1.d0 + bb(i))*g0c(pp(i))
cc Urspruenglich anderes (Minus) VZ hier, dafuer kein Minus mehr bei der
cc  Definition des WQs ueber Im G, 2.6.1998, tt.
cc             gg(i) = - (1.d0 + bb(i))*g0c(pp(i))
50          continue
c
c Normalisation on R:
            const = 8.d0*pi/tmass**2
c
c Proove of the optical theorem for the output values of saeax:
c  Simply check if sig1 = sig2.
            sig1 = 0.d0
            sig2 = 0.d0
            xaai = 0.d0
            xaad = 0.d0
            do 60 i=1,n*2/3
c              write(*,*) 'check! p(',i,') = ',pp(i)
cvv
               if (pp(i).lt.dcut.and.vflag.eq.1) then
                  sig1 = sig1 + w1(i)*pp(i)**2*aimag(gg(i)
cc     u                 *(1.d0+kincov*(pp(i)/tmass)**2)
     u   *(1.d0+kincov*g0(pp(i))*(pp(i)/tmass)**2/g0c(pp(i)))
     u                  )
               else
```

```
                 sig1 = sig1 + w1(i)*pp(i)**2*aimag(gg(i))
              endif
              if (pp(i).lt.dcut.and.kinflg.ne.0) then
                 sig2 = sig2 + w1(i)*pp(i)**2*abs(gg(i))**2 *
     u                 tgamma*gtpcor(pp(i),etot)
     u                 *(1.d0-pp(i)**2/2.d0/tmass**2)
cc     u                  *tmass/sqrt(tmass**2+pp(i)**2)
c                 xdsdp(i)=pp(i)**2*abs(gg(i))**2 *
c     u                  tgamma*gtpcor(pp(i),etot)
c     u                  *(1.d0-pp(i)**2/2.d0/tmass**2)
c     u                  /(2.d0*pi**2)*const
              else
                 sig2 = sig2 + w1(i)*pp(i)**2*abs(gg(i))**2 *
     u                 tgamma*gtpcor(pp(i),etot)
c                 xdsdp(i)=pp(i)**2*abs(gg(i))**2 *
c     u                  tgamma*gtpcor(pp(i),etot)
c     u                  /(2.d0*pi**2)*const
              endif
              xdsdp(i)=pp(i)**4/tmass**2*abs(zftild(i)*g0c(pp(i)))**2
     u                *tgamma*gtpcor(pp(i),etot)
     u                /(2.d0*pi**2)*const
              xaai=xaai+w1(i)*pp(i)**4/tmass**2*
     u                 aimag(zftild(i)*g0c(pp(i)))
              xaad=xaad+w1(i)*pp(i)**4/tmass**2*
     u                 abs(zftild(i)*g0c(pp(i)))**2 *
     u                 tgamma*gtpcor(pp(i),etot)
c             write(*,*) 'xdsdp = ',xdsdp(i)
c             write(*,*) 'zvfct = ',zvfct(i)
c             write(*,*) 'zftild = ',zftild(i)
60         continue
c '*p**2' because of substitution p => 1/p in the integration of p**2*G(p)
c  to infinity
           do 70 i=n*2/3+1,n
c             write(*,*) 'check! p(',i,') = ',pp(i)
cvv
              if (pp(i).lt.dcut.and.vflag.eq.1) then
                 sig1 = sig1 + w1(i)*pp(i)**4*aimag(gg(i)
cc     u                  *(1.d0+kincov*(pp(i)/tmass)**2)
     u   *(1.d0+kincov*g0(pp(i))*(pp(i)/tmass)**2/g0c(pp(i)))
     u                 )
              else
                 sig1 = sig1 + w1(i)*pp(i)**4*aimag(gg(i))
              endif
              if (pp(i).lt.dcut.and.kinflg.ne.0) then
                 sig2 = sig2 + w1(i)*pp(i)**4*abs(gg(i))**2 *
     u                 tgamma*gtpcor(pp(i),etot)
     u                 *(1.d0-pp(i)**2/2.d0/tmass**2)
cc     u                  *tmass/sqrt(tmass**2+pp(i)**2)
c                 xdsdp(i)=pp(i)**2*abs(gg(i))**2 *
c     u                  tgamma*gtpcor(pp(i),etot)
c     u                  *(1.d0-pp(i)**2/2.d0/tmass**2)
c     u                  /(2.d0*pi**2)*const
              else
                 sig2 = sig2 + w1(i)*pp(i)**4*abs(gg(i))**2 *
```

```
     u                 tgamma*gtpcor(pp(i),etot)
c                xdsdp(i)=pp(i)**2*abs(gg(i))**2 *
c     u                 tgamma*gtpcor(pp(i),etot)
c     u                 /(2.d0*pi**2)*const
              endif
              xdsdp(i)=pp(i)**4/tmass**2*abs(zftild(i)*g0c(pp(i)))**2
     u                *tgamma*gtpcor(pp(i),etot)
     u                /(2.d0*pi**2)*const
              xaai=xaai+w1(i)*pp(i)**6/tmass**2*
     u                 aimag(zftild(i)*g0c(pp(i)))
              xaad=xaad+w1(i)*pp(i)**6/tmass**2*
     u                 abs(zftild(i)*g0c(pp(i)))**2 *
     u                 tgamma*gtpcor(pp(i),etot)
c             write(*,*) 'xdsdp = ',xdsdp(i)
c             write(*,*) 'zvfct = ',zvfct(i)
c             write(*,*) 'zftild = ',zftild(i)
70          continue
            do 71 i=n+1,nmax
              xdsdp(i)=0.d0
              zvfct(i)=(0.d0,0.d0)
              zftild(i)=(0.d0,0.d0)
71          continue
c
c Normalisation on R:
            sig1  = sig1 / (2.d0*pi**2) * const
            sig2  = sig2 / (2.d0*pi**2) * const
c
c The results from the momentum space approach finally are:
cc Jetzt Minus hier, 2.6.98, tt.
c           xim=-sig1
c           xdi=sig2
            xaai=-xaai / (2.d0*pi**2) * const
            xaad=xaad / (2.d0*pi**2) * const
c Output of P wave part only:
            xim=xaai
            xdi=xaad
c           write(*,*) 'vvi = ',-sig1,' .  vvd = ',sig2
c           write(*,*) 'aai = ',xim,' .  aad = ',xdi
c
        end
c
c
!!! FB:
!       complex*16 function g0(p)
! c
!           implicit none
!           real*8
!      u        tmass,tgamma,zmass,alphas,alamb5,
!      u        wmass,wgamma,bmass,GFERMI,
!      u        pi,energy,vzero,eps,
!      u        p,gtpcor
!           integer npot
!       COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
!      $ WMASS,WGAMMA,BMASS,GFERMI
```

```
!          common/ovalco/ pi, energy, vzero, eps, npot
!          external gtpcor
!          save
!          g0=1.d0/cmplx(energy-vzero-p**2/tmass,
!      u                    tgamma*gtpcor(p,2.d0*tmass+energy),
!      u                    kind=kind(0d0))
!       end
! c
!       complex*16 function g0c(p)
! c
!          implicit none
!          complex*16 hypgeo,green,zk,zi,amd2k,aa,bb,cc,zzp,zzm,
!      u                hypp,hypm,g0
!          real*8
!      u        tmass,tgamma,zmass,alphas,alamb5,
!      u        wmass,wgamma,bmass,GFERMI,
!      u        pi,energy,vzero,eps,
!      u        p,gtpcor,
!      u        kincom,kincoa,kincov,xp,xpmax,dcut
!          integer npot,kinflg,gcflg,vflag
!       COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
!      $ WMASS,WGAMMA,BMASS,GFERMI
!          common/ovalco/ pi, energy, vzero, eps, npot
!          common/g0inf/kincom,kincoa,kincov,kinflg,gcflg,vflag
!          common/mom/ xp,xpmax,dcut
!          external hypgeo,gtpcor,g0
!          save
! c
!          if (gcflg.eq.0) then
!             if (kinflg.eq.0) then
!                g0c=g0(p)
!             else if (kinflg.eq.1.and.p.lt.dcut) then
!                g0c=g0(p)*(1.d0+kincom)+kincoa
!             else if (kinflg.eq.1.and.p.ge.dcut) then
!                g0c=g0(p)*(1.d0+kincom)
!             else if (kinflg.eq.2.and.p.lt.dcut) then
!                g0c=g0(p)*(1.d0+kincom)+kincoa
!             else if (kinflg.eq.2.and.p.ge.dcut) then
!                g0c=g0(p)
!             else
!                write(*,*) ' kinflg wrong! Stop.'
!                stop
!             endif
!          else if (gcflg.eq.1) then
!             zi=(0.d0,1.d0)
!             zk=-tmass*cmplx(energy,tgamma
!      u                        *gtpcor(p,2.d0*tmass+energy),
!      u                        kind=kind(0d0))
!             zk=sqrt(zk)
!             amd2k=4.d0/3.d0*alphas*tmass/2.d0/zk
!             aa=(2.d0,0.d0)
!             bb=(1.d0,0.d0)
!             cc=2.d0-amd2k
!             zzp=(1.d0+zi*p/zk)/2.d0
```

```
!             zzm=(1.d0-zi*p/zk)/2.d0
!             if (abs(zzp).gt.20.d0) then
!                hypp=(1.d0-zzp)**(-aa)*
!      u                hypgeo(aa,cc-bb,cc,zzp/(zzp-1.d0))
!             else
!                hypp=hypgeo(aa,bb,cc,zzp)
!             endif
!             if (abs(zzm).gt.20.d0) then
!                hypm=(1.d0-zzm)**(-aa)*
!      u                hypgeo(aa,cc-bb,cc,zzm/(zzm-1.d0))
!             else
!                hypm=hypgeo(aa,bb,cc,zzm)
!             endif
!             green=-zi*tmass/(4.d0*p*zk)/(1.d0-amd2k)*(hypp-hypm)
! c VZ anders herum als in Andres Konvention, da bei ihm G_0=1/[-E-i G+p^2/m]:
!             g0c=-green
!             if (p.gt.1.d3*tmass) then
!                write(*,*) ' g0cana = ',g0c,' not reliable. Stop.'
!                stop
!             endif
!          else
!             write(*,*) ' gcflg wrong! Stop.'
!             stop
!          endif
! c
!       end
!!! /FB
c
c
        complex*16 function aax(p)
c
           implicit none
           real*8
     u        tmass,tgamma,zmass,alphas,alamb5,
     u        wmass,wgamma,bmass,GFERMI,
     u        pi, energy,vzero, eps,
     $        QCUT,QMAT1,ALR,
c     u       ,PCUT,etot
     u        p,
     u        xp,xpmax, xb1,xb2,dcut,ddcut,
     u        a1, a2, a3, a4,a5,a6,
     u        adglg1, fretil1ax, fretil2ax, fimtil1ax, fimtil2ax,
     u        ALEFVQ, gtpcor, ad8gle, buf,adglg2,
c     u        xerg,
     u        kincom,kincoa,kincov
           complex*16 zapvq1
c     u               ,ZAPVGP
           integer npot,ILFLAG,kinflg,gcflg,vflag
c
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      COMMON/PARFLG/ QCUT,QMAT1,ALR,ILFLAG
           common/ovalco/ pi, energy, vzero, eps, npot
           common/mom/ xp,xpmax,dcut
```

```
          common/g0inf/kincom,kincoa,kincov,kinflg,gcflg,vflag
c
          external adglg1, fretil1ax, fretil2ax, fimtil1ax, fimtil2ax,
     u              zapvq1, ALEFVQ, gtpcor,ad8gle,adglg2
c     u             ,zapvgp
c
          if ((npot.eq.1).or.(npot.eq.3).or.(npot.eq.4).or.
     u         (npot.eq.5)) then
c
             xp=p
             buf=0.d0
c
             a1=0.d0
             a2=0.d0
             a3=0.d0
             a4=0.d0
             a5=0.d0
             a6=0.d0
             if (gcflg.eq.0) then
                ddcut=xpmax
             else if (gcflg.eq.1) then
                ddcut=dcut
             else
                write(*,*) ' gcflg wrong! Stop.'
                stop
             endif
c
             if (2.d0*xp.lt.ddcut) then
                xb1=xp
                xb2=2.d0*xp
c
c More stable for logarithmically divergent fixed order potentials:
c
                a1=adglg1(fretil1ax, buf, xb1, eps)
                a2=adglg1(fimtil1ax, buf, xb1, eps)
c Slightly unstable:
                a3=adglg2(fretil1ax,xb1,xb2,eps)
c No good:
c               a3=adglg1(fretil1ax,xb1,xb2,eps)
c Not better:
c               call adqua(xb1,xb2,fretil1ax,xerg,eps)
c               a3=xerg
c Also not better:
c               a1=adglg1(fretil1ax, buf, xb2, eps)
c
                a4=adglg2(fimtil1ax,xb1,xb2,eps)
c               a5 = adglg2(fretil1ax, xb2, ddcut, eps)
c               a6 = adglg2(fimtil1ax, xb2, ddcut, eps)
                a5 = adglg2(fretil2ax, 1.d0/ddcut, 1.d0/xb2, eps)
                a6 = adglg2(fimtil2ax, 1.d0/ddcut, 1.d0/xb2, eps)
             else if (xp.lt.ddcut) then
                xb1=xp
                xb2=ddcut
                a1=adglg1(fretil1ax, buf, xb1, eps)
```

```
                a2=adglg1(fimtil1ax, buf, xb1, eps)
                a3=adglg2(fretil1ax,xb1,xb2,eps)
                a4=adglg2(fimtil1ax,xb1,xb2,eps)
             else if (ddcut.le.xp) then
             else
                write(*,*) ' Constellation not possible! Stop.'
                stop
             endif
c
             aax  = 1.d0/(4.d0*pi**2)*cmplx(a1+a3+a5,a2+a4+a6,
     u                   kind=kind(0d0))
c
c         else if (npot.eq.2) then
c      PCUT=QCUT
c      ETOT=ENERGY+2*TMASS
c            aax  = ZAPVGP(P,ETOT,VZERO-ENERGY,PCUT,EPS)
          else
             write (*,*) ' Potential not implemented! Stop. 2'
             stop
          endif
c
       end
c
       real*8 function fretil1ax(xk)
          implicit none
          real*8 xk, frealax
          external frealax
          fretil1ax = frealax(xk)
       end
c
       real*8 function fretil2ax(xk)
          implicit none
          real*8 xk, frealax
          external frealax
          fretil2ax = frealax(1.d0/xk) * xk**(-2)
       end
c
       real*8 function fimtil1ax(xk)
          implicit none
          real*8 xk, fimax
          external fimax
          fimtil1ax = fimax(xk)
       end
c
       real*8 function fimtil2ax(xk)
          implicit none
          real*8 xk, fimax
          external fimax
          fimtil2ax = fimax(1.d0/xk) * xk**(-2)
       end
c
       real*8 function frealax(xk)
          implicit none
          complex*16 vhatax
```

```
      real*8
     u   tmass,tgamma,zmass,alphas,alamb5,
     u   wmass,wgamma,bmass,GFERMI,
     u   pi, energy, vzero, eps,
     u   p,pmax, xk, gtpcor,dcut
      complex*16 g0,g0c
      integer npot
     COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      common/ovalco/ pi, energy, vzero, eps, npot
      common/mom/ p,pmax,dcut
      external vhatax, g0, g0c, gtpcor
c
      frealax = real(g0c(xk)*vhatax(p, xk))
   end
c
   real*8 function fimax(xk)
      implicit none
      complex*16 vhatax
      real*8
     u   tmass,tgamma,zmass,alphas,alamb5,
     u   wmass,wgamma,bmass,GFERMI,
     u   pi, energy, vzero, eps,
     u   p,pmax, xk, gtpcor,dcut
      complex*16 g0,g0c
      integer npot
     COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      common/ovalco/ pi, energy, vzero, eps, npot
      common/mom/ p,pmax,dcut
      external vhatax, g0, g0c, gtpcor
      fimax = aimag(g0c(xk)*vhatax(p, xk))
   end
c
c
   complex*16 function vhatax(p, xk)
c
      implicit none
      complex*16 zi
      real*8
     u   tmass,tgamma,zmass,alphas,alamb5,
     u   wmass,wgamma,bmass,GFERMI,
     u   pi, energy, vzero, eps,
     u   p, xk,
     u   cnspot, phiint, AD8GLE,
     u   pm, xkm,
c     u    phfqcd, ALPHEF,
     u   zeta3,cf,ca,tf,xnf,a1,a2,b0,b1,
     u   cplas,scale,c0,c1,c2,
     u   cdeltc,cdeltl,cfullc,cfulll,crm2,
     u   xkpln1st,xkpln2nd,xkpln3rd,
     u   pp,pmax,dcut
      integer npot
      parameter(zi=(0.d0,1.d0))
```

```
      parameter(zeta3=1.20205690316d0,
     u               cf=4.d0/3.d0,ca=3.d0,tf=1.d0/2.d0,
     u               xnf=5.d0)
c
      external AD8GLE
c     u            , phfqcd, ALPHEF
c
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      common/ovalco/ pi, energy, vzero, eps, npot
      common/pmaxkm/ pm, xkm
      common/mom/ pp,pmax,dcut
      common/cplcns/cplas,scale,c0,c1,c2,
     u                  cdeltc,cdeltl,cfullc,cfulll,crm2
c
      b0=11.d0-2.d0/3.d0*xnf
      b1=102.d0-38.d0/3.d0*xnf
c
      a1=31.d0/9.d0*ca-20.d0/9.d0*tf*xnf
      a2=(4343.d0/162.d0+4.d0*pi**2-pi**4/4.d0+
     u         22.d0/3.d0*zeta3)*ca**2-
     u        (1798.d0/81.d0+56.d0/3.d0*zeta3)*ca*tf*xnf-
     u        (55.d0/3.d0-16.d0*zeta3)*cf*tf*xnf+
     u        (20.d0/9.d0*tf*xnf)**2
c
      pm=p
      xkm=xk
      cnspot=-4.d0/3.d0*4.d0*pi
c
      if (p/xk.le.1.d-5.and.p.le.1.d-5) then
         xkpln1st=2.d0
         xkpln2nd=-4.d0*log(scale/xk)
         xkpln3rd=-6.d0*log(scale/xk)**2
      else if (xk/p.le.1.d-5.and.xk.le.1.d-5) then
         xkpln1st=2.d0*(xk/p)**2
         xkpln2nd=-4.d0*(xk/p)**2*log(scale/p)
         xkpln3rd=-6.d0*(xk/p)**2*log(scale/p)**2
      else
c        xkpln1st=xk/p*log(abs((p+xk)/(p-xk)))
         xkpln1st=xk/p*(log(p+xk)-log(abs(p-xk)))
         xkpln2nd=xk/p*(-1.d0)*(log(scale/(p+xk))**2-
     u                            log(scale/abs(p-xk))**2)
         xkpln3rd=xk/p*(-4.d0/3.d0)*(log(scale/(p+xk))**3-
     u                                  log(scale/abs(p-xk))**3)
      endif
c
c     if (npot.eq.2) then
c        if (p/xk.le.1.d-5.and.p.le.1.d-5) then
c           vhatax = 2.d0 * cnspot * ALPHEF(xk)
c        else if (xk/p.le.1.d-5.and.xk.le.1.d-5) then
c           vhatax = 2.d0 * cnspot * xk**2 / p**2 * ALPHEF(p)
c        else
c           phiint = cnspot * (AD8GLE(phfqcd, 0.d0, 0.3d0, 1.d-5)
c     u                          +AD8GLE(phfqcd, 0.3d0, 1.d0, 1.d-5))
```

```
c                vhatax   = xk / p * log(abs((p+xk)/(p-xk))) * phiint
c             endif
c          else
              if (npot.eq.1) then
                 c0=1.d0
                 c1=0.d0
                 c2=0.d0
              else if (npot.eq.3) then
                 c0=1.d0+alphas/(4.d0*pi)*a1
                 c1=alphas/(4.d0*pi)*b0
                 c2=0
              else if (npot.eq.4) then
                 c0=1.d0+alphas/(4.d0*pi)*a1+(alphas/(4.d0*pi))**2*a2
                 c1=alphas/(4.d0*pi)*b0+
     u             (alphas/(4.d0*pi))**2*(b1+2.d0*b0*a1)
                 c2=(alphas/(4.d0*pi))**2*b0**2
              else if (npot.eq.5) then
              else
                 write (*,*) ' Potential not implemented! Stop. 3'
                 stop
              endif
              phiint=cnspot*alphas
c
c             if ((xk+p).le.dcut) then
c                vhatax=phiint*(c0*xkpln1st+c1*xkpln2nd+c2*xkpln3rd)
c     u               -1.d0/2.d0*(1.d0+2.d0*ca/cf)
c     u                *(pi*cf*alphas)**2/tmass
c     u                *xk/p*(p+xk-abs(xk-p))
c             else if (abs(xk-p).lt.dcut) then
c                vhatax=phiint*(c0*xkpln1st+c1*xkpln2nd+c2*xkpln3rd)
c     u               -1.d0/2.d0*(1.d0+2.d0*ca/cf)
c     u                *(pi*cf*alphas)**2/tmass
c     u                *xk/p*(dcut-abs(xk-p))
c             else if (dcut.le.abs(xk-p)) then
c                vhatax=phiint*(c0*xkpln1st+c1*xkpln2nd+c2*xkpln3rd)
c             else
c                write(*,*) ' Not possible! Stop.'
c                stop
c             endif
c
              if (max(xk,p).lt.dcut) then
c Coulomb + first + second order corrections:
                 vhatax=phiint*(c0*xkpln1st+c1*xkpln2nd+c2*xkpln3rd)
c All other potentials:
     u               +cdeltc*2.d0*xk**2
     u               +cdeltl*xk/p/2.d0*(
     u                (p+xk)**2*(log(((p+xk)/scale)**2)-1.d0)-
     u                (p-xk)**2*(log(((p-xk)/scale)**2)-1.d0))
     u               +cfullc*(p**2+xk**2)*xkpln1st
     u               +cfulll*(p**2+xk**2)*xk/p/4.d0*
     u                  (log(((p+xk)/scale)**2)**2-
     u                   log(((p-xk)/scale)**2)**2)
     u               +crm2*xk/p*(p+xk-abs(xk-p))
              else
```

```
                vhatax=phiint*(c0*xkpln1st+c1*xkpln2nd+c2*xkpln3rd)
             endif
c          endif
c
        end
c
c
        complex*16 function vhhat(p, xk)
c
           implicit none
           complex*16 zi
           real*8
     u        tmass,tgamma,zmass,alphas,alamb5,
     u        wmass,wgamma,bmass,GFERMI,
     u        pi, energy, vzero, eps,
     u        p, xk,
     u        cnspot, phiint, AD8GLE,
     u        pm, xkm,
     u        zeta3,cf,ca,tf,xnf,a1,a2,b0,b1,
     u        cplas,scale,c0,c1,c2,
     u        cdeltc,cdeltl,cfullc,cfulll,crm2,
     u        xkpln1st,
     u        pp,pmax,dcut
           integer npot
           parameter(zi=(0.d0,1.d0))
           parameter(zeta3=1.20205690316d0,
     u               cf=4.d0/3.d0,ca=3.d0,tf=1.d0/2.d0,
     u               xnf=5.d0)
c
           external AD8GLE
c
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
           common/ovalco/ pi, energy, vzero, eps, npot
           common/pmaxkm/ pm, xkm
           common/mom/ pp,pmax,dcut
           common/cplcns/cplas,scale,c0,c1,c2,
     u                   cdeltc,cdeltl,cfullc,cfulll,crm2
c
           b0=11.d0-2.d0/3.d0*xnf
           b1=102.d0-38.d0/3.d0*xnf
c
           a1=31.d0/9.d0*ca-20.d0/9.d0*tf*xnf
           a2=(4343.d0/162.d0+4.d0*pi**2-pi**4/4.d0+
     u         22.d0/3.d0*zeta3)*ca**2-
     u        (1798.d0/81.d0+56.d0/3.d0*zeta3)*ca*tf*xnf-
     u        (55.d0/3.d0-16.d0*zeta3)*cf*tf*xnf+
     u        (20.d0/9.d0*tf*xnf)**2
c
           pm=p
           xkm=xk
           cnspot=-4.d0/3.d0*4.d0*pi
c
             if (npot.eq.1) then
```

```
                 c0=1.d0
                 c1=0.d0
                 c2=0.d0
              else if (npot.eq.3) then
                 c0=1.d0+alphas/(4.d0*pi)*a1
                 c1=alphas/(4.d0*pi)*b0
                 c2=0
              else if (npot.eq.4) then
                 c0=1.d0+alphas/(4.d0*pi)*a1+(alphas/(4.d0*pi))**2*a2
                 c1=alphas/(4.d0*pi)*b0+
     u             (alphas/(4.d0*pi))**2*(b1+2.d0*b0*a1)
                 c2=(alphas/(4.d0*pi))**2*b0**2
              else if (npot.eq.5) then
              else
                 write (*,*) ' Potential not implemented! Stop. 4'
                 stop
              endif
              phiint=cnspot*alphas
c
c Pure Coulomb in first order only:
              xkpln1st=-(xk/p)**2*(1.d0+(xk**2+p**2)/(2.d0*xk*p)*
     u                  (log(abs(p-xk))-log(p+xk)))
                 vhhat=phiint*c0*xkpln1st
c
        end
c
c
!!! FB:
!       real*8 function gtpcor(topp,etot)
!       real*8 topp,etot,
!      u         tmass,tgamma,zmass,alphas,alamb5,
!      u         wmass,wgamma,bmass,GFERMI
!       COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
!      $ WMASS,WGAMMA,BMASS,GFERMI
!       gtpcor=1.d0
!       end
!!! /FB
c
c
c --- Routines for solving linear equations and matrix inversion (complex) ---
c
        subroutine saeax(pp, w1, bb, vec, a1, n)
c
           implicit none
           complex*16 vhatax,vhhat
           real*8
     u        tmass,tgamma,zmass,alphas,alamb5,
     u        wmass,wgamma,bmass,GFERMI,
     u        pi, energy, vzero, eps,
     u        d, d1, pp, w1, gtpcor,
     u        xp,xpmax,dcut,kincom,kincoa,kincov
           complex*16 aax, a1, bb, vec, ff, kk, cw, svw, g0, g0c
           integer i, j, npot, n, nmax, indx,kinflg,gcflg,vflag
           parameter (nmax=400)
```

```
      dimension bb(nmax),vec(nmax),ff(nmax,nmax),kk(nmax,nmax),
     u               pp(nmax),w1(nmax),indx(nmax),cw(nmax),a1(nmax)
c
      COMMON/PHCONS/TMASS,TGAMMA,ZMASS,ALPHAS,ALAMB5,
     $ WMASS,WGAMMA,BMASS,GFERMI
      common/ovalco/ pi, energy, vzero, eps, npot
      common/mom/ xp,xpmax,dcut
      common/g0inf/kincom,kincoa,kincov,kinflg,gcflg,vflag
c
      external aax, vhatax, gtpcor, g0, g0c, vhhat
c
      do 10 i=1,n*2/3
         cw(i) = w1(i) / (4.d0*pi**2) * g0c(pp(i))
c        cw(i) = w1(i) / (4.d0*pi**2 *
c     u            (cmplx(energy-vzero, tgamma*
c     u             gtpcor(pp(i),2.d0*tmass+energy),
c     u                kind=kind(0d0))-pp(i)**2/tmass))
10    continue
      do 20 i=n*2/3+1,n
         cw(i) = w1(i) / (4.d0*pi**2) * g0c(pp(i)) * pp(i)**2
c        cw(i) = w1(i) / (4.d0*pi**2 *
c     u      (cmplx(energy-vzero, tgamma*
c     u       gtpcor(pp(i),2.d0*tmass+energy),kind=kind(0d0)) /
c     u       pp(i)**2 - 1.d0/tmass))
20    continue
c
      do 30 i=1,n
cc       bb(i) = a1(i)
cvv
         if (pp(i).lt.dcut.and.vflag.eq.1) then
c           bb(i) = cmplx(1.d0+kincov*(pp(i)/tmass)**2,0.d0,
c     u                kind=kind(0d0))
            bb(i)=1.d0+kincov*
     u                  g0(pp(i))*(pp(i)/tmass)**2/g0c(pp(i))
         else
            bb(i) = (1.d0,0.d0)
         endif
c
c Without extra kinematic corrections:
         vec(i)=(1.d0,0.d0)
c
         svw = (0.d0,0.d0)
         do 40 j=1,n
            if (i.ne.j) then
               ff(i,j) = - vhatax(pp(i),pp(j)) * cw(j)
               kk(i,j) = - vhhat(pp(i),pp(j)) * cw(j)
               svw = svw + ff(i,j)
            endif
40       continue
         ff(i,i) = 1.d0 - a1(i) - svw
         kk(i,i) = ff(i,i)
30    continue
c
      call zldcmp(ff, n, nmax, indx, d)
```

```
          call zldcmp(kk, n, nmax, indx, d1)
          call zlbksb(ff, n, nmax, indx, bb)
          call zlbksb(kk, n, nmax, indx, vec)
c
        end
c
c
!!! FB:
!       SUBROUTINE ZLBKSB(A,N,NP,INDX,B)
! C complex version of lubksb
!       IMPLICIT NONE
!       INTEGER I, II, INDX, J, LL, N, NP
!       COMPLEX*16 A, B, SUM
!       DIMENSION A(NP,NP),INDX(N),B(N)
!       II=0
!       DO 12 I=1,N
!         LL=INDX(I)
!         SUM=B(LL)
!         B(LL)=B(I)
!         IF (II.NE.0)THEN
!           DO 11 J=II,I-1
!             SUM=SUM-A(I,J)*B(J)
! 11        CONTINUE
!         ELSE IF (SUM.NE.(0.D0,0.D0)) THEN
!           II=I
!         ENDIF
!         B(I)=SUM
! 12    CONTINUE
!       DO 14 I=N,1,-1
!         SUM=B(I)
!         IF(I.LT.N)THEN
!           DO 13 J=I+1,N
!             SUM=SUM-A(I,J)*B(J)
! 13        CONTINUE
!         ENDIF
!         B(I)=SUM/A(I,I)
! 14    CONTINUE
!       RETURN
!       END
! c
!       SUBROUTINE ZLDCMP(A,N,NP,INDX,D)
! C complex version of ludcmp
!       IMPLICIT NONE
!       INTEGER I, IMAX, INDX, J, K, N, NP, NMAX
!       REAL*8 AAMAX, D, TINY, VV
!       COMPLEX*16 A, DUM, SUM
!       PARAMETER (NMAX=400)
!       DIMENSION A(NP,NP), INDX(N), VV(NMAX)
! c
!       tiny=1.d-5
! c
!       D=1.D0
!       DO 12 I=1,N
!         AAMAX=0.D0
```

```
!         DO 11 J=1,N
!           IF (ABS(A(I,J)).GT.AAMAX) AAMAX=ABS(A(I,J))
! 11      CONTINUE
!         IF (AAMAX.EQ.0.D0) PAUSE 'Singular matrix.'
!         VV(I)=1.D0/AAMAX
! 12    CONTINUE
!       DO 19 J=1,N
!         IF (J.GT.1) THEN
!           DO 14 I=1,J-1
!             SUM=A(I,J)
!             IF (I.GT.1)THEN
!               DO 13 K=1,I-1
!                 SUM=SUM-A(I,K)*A(K,J)
! 13            CONTINUE
!               A(I,J)=SUM
!             ENDIF
! 14        CONTINUE
!         ENDIF
!         AAMAX=0.D0
!         DO 16 I=J,N
!           SUM=A(I,J)
!           IF (J.GT.1)THEN
!             DO 15 K=1,J-1
!               SUM=SUM-A(I,K)*A(K,J)
! 15          CONTINUE
!             A(I,J)=SUM
!           ENDIF
!           DUM=VV(I)*ABS(SUM)
!           IF (ABS(DUM).GE.AAMAX) THEN
!             IMAX=I
!             AAMAX=DUM
!           ENDIF
! 16      CONTINUE
!         IF (J.NE.IMAX) THEN
!           DO 17 K=1,N
!             DUM=A(IMAX,K)
!             A(IMAX,K)=A(J,K)
!             A(J,K)=DUM
! 17        CONTINUE
!           D=-D
!           VV(IMAX)=VV(J)
!         ENDIF
!         INDX(J)=IMAX
!         IF (J.NE.N) THEN
!           IF (A(J,J).EQ.(0.D0,0.D0)) A(J,J)=cmplx(TINY, 0.d0,
!      u                    kind=kind(0d0))
!           DUM=1.D0/A(J,J)
!           DO 18 I=J+1,N
!             A(I,J)=A(I,J)*DUM
! 18        CONTINUE
!         ENDIF
! 19    CONTINUE
!       IF(A(N,N).EQ.(0.D0,0.D0)) A(N,N)=cmplx(TINY, 0.d0,
!      u                    kind=kind(0d0))
```

```
!       RETURN
!       END
! C
! C
! C *** TOOLS ***
! C
! C
! C     ******* ROUTINES FOR GAUSSIAN INTEGRATIONS
! C
! C
!       SUBROUTINE GAULEG(X1,X2,X,W,N)
! C
! C     Given the lower and upper limits of integration X1 and X2
! C     and given N, this routine returns arrays X(N) and W(N)
! C     containing the abscissas and weights of the Gauss-Legendre
! C     N-point quadrature formula
! C
!       IMPLICIT REAL*8 (A-H,O-Z)
!       REAL*8 X1,X2,X(N),W(N)
!       PARAMETER (EPS=3.D-14)
!       save
!       M=(N+1)/2
!       XM=0.5D0*(X2+X1)
!       XL=0.5D0*(X2-X1)
!       DO 12 I=1,M
!         Z=DCOS(3.14159265358979323 8D0*(I-.25D0)/(N+.5D0))
! 1       CONTINUE
!           P1=1.D0
!           P2=0.D0
!           DO 11 J=1,N
!             P3=P2
!             P2=P1
!             P1=((2.D0*J-1.D0)*Z*P2-(J-1.D0)*P3)/J
! 11        CONTINUE
!           PP=N*(Z*P1-P2)/(Z*Z-1.D0)
!           Z1=Z
!           Z=Z1-P1/PP
!         IF(DABS(Z-Z1).GT.EPS)GO TO 1
!         X(I)=XM-XL*Z
!         X(N+1-I)=XM+XL*Z
!         W(I)=2.D0*XL/((1.D0-Z*Z)*PP*PP)
!         W(N+1-I)=W(I)
! 12    CONTINUE
!       RETURN
!       END
! C
! C
!       DOUBLE PRECISION FUNCTION AD8GLE(F,A,B,EPS)
!       implicit double precision (a-h,o-z)
!       EXTERNAL F
!       DIMENSION W(12),X(12)
! c      SAVE W, X
!       SAVE
! C
```

```
! C     ****************************************************************
! C
! C     ADAPTIVE GAUSSIAN QUADRATURE.
! C
! C     AD8GLE IS SET EQUAL TO THE APPROXIMATE VALUE OF THE INTEGRAL OF
! C     THE FUNCTION F OVER THE INTERVAL (A,B), WITH ACCURACY PARAMETER
! C     EPS.
! C
! C     ****************************************************************
! C
!      DATA W / 0.10122 85362 90376 25915 25313 543D0,
!     $         0.22238 10344 53374 47054 43559 944D0,
!     $         0.31370 66458 77887 28733 79622 020D0,
!     $         0.36268 37833 78361 98296 51504 493D0,
!     $         0.27152 45941 17540 94851 78057 246D-1,
!     $         0.62253 52393 86478 92862 84383 699D-1,
!     $         0.95158 51168 24927 84809 92510 760D-1,
!     $         0.12462 89712 55533 87205 24762 822D0,
!     $         0.14959 59888 16576 73208 15017 305D0,
!     $         0.16915 65193 95002 53818 93120 790D0,
!     $         0.18260 34150 44923 58886 67636 680D0,
!     $         0.18945 06104 55068 49628 53967 232D0/
! C
!      DATA X / 0.96028 98564 97536 23168 35608 686D0,
!     $         0.79666 64774 13626 73959 15539 365D0,
!     $         0.52553 24099 16328 98581 77390 492D0,
!     $         0.18343 46424 95649 80493 94761 424D0,
!     $         0.98940 09349 91649 93259 61541 735D0,
!     $         0.94457 50230 73232 57607 79884 155D0,
!     $         0.86563 12023 87831 74388 04678 977D0,
!     $         0.75540 44083 55003 03389 51011 948D0,
!     $         0.61787 62444 02643 74844 66717 640D0,
!     $         0.45801 67776 57227 38634 24194 430D0,
!     $         0.28160 35507 79258 91323 04605 015D0,
!     $         0.95012 50983 76374 40185 31933 543D-1/
! C
! C     ****************************************************************
! C
!      GAUSS=0.0D0
!      AD8GLE=GAUSS
!      IF(B.EQ.A) RETURN
!      CONST=EPS/(B-A)
!      BB=A
! C
! C  COMPUTATIONAL LOOP.
!    1 AA=BB
!      BB=B
!    2    C1=0.5D0*(BB+AA)
!         C2=0.5D0*(BB-AA)
!         S8=0.0D0
!         DO 3 I=1,4
!            U=C2*X(I)
!            S8=S8+W(I)*(F(C1+U)+F(C1-U))
!    3    CONTINUE
```

```
!          S8=C2*S8
!          S16=0.0D0
!          DO 4 I=5,12
!             U=C2*X(I)
!             S16=S16+W(I)*(F(C1+U)+F(C1-U))
!     4    CONTINUE
!          S16=C2*S16
!          IF( ABS(S16-S8) .LE. EPS*(abs(s8)+ABS(S16))*0.5D0 ) GO TO 5
!          BB=C1
!          IF( 1.D0+ABS(CONST*C2) .NE. 1.D0) GO TO 2
!       AD8GLE=0.0D0
!       write(*,*)'too high accuracy required in function ad8gle!'
!       RETURN
!     5 GAUSS=GAUSS+S16
!       IF(BB.NE.B) GO TO 1
!       AD8GLE=GAUSS
!       RETURN
!       END
! C
! C
!       DOUBLE PRECISION FUNCTION ADGLG1(F,A,B,EPS)
!       IMPLICIT REAL*8 (A-H,O-Z)
!       EXTERNAL F,AD8GLE,adqua
!       DIMENSION W(6),X(6),xx(6)
! c      SAVE W, XX, NUM
!       SAVE
! C
! C     ****************************************************************
! C
! C     ADAPTIVE GAUSSIAN QUADRATURE.
! C     For x->b   f(x) = O (ln^k (b-x) )
! C     A - lower limit, B - upper limit (integrable singularity)
! C     AD8GLE IS SET EQUAL TO THE APPROXIMATE VALUE OF THE INTEGRAL OF
! C     THE FUNCTION F OVER THE INTERVAL (A,B), WITH ACCURACY PARAMETER
! C     EPS.
! C
! C     ****************************************************************
!       DATA W / 4.58964 673950d-1,
!      $         4.17000 830772d-1,
!      $         1.13373 382074d-1,
!      $         1.03991 974531d-2,
!      $         2.61017 202815d-4,
!      $         8.98547 906430d-7/
! C
!       DATA X / 0.22284 66041 79d0,
!      $         1.18893 21016 73d0,
!      $         2.99273 63260 59d0,
!      $         5.77514 35691 05d0,
!      $         9.83746 74183 83d0,
!      $        15.98287 39806 02d0/
!       DATA NUM/0/
!       IF(NUM.eq.0d0) then
!       do 1 ix=1,6
!   1   xx(ix)= EXP(-x(ix))
```

```
!       ENDIF
!       num=num+1
!       sum=0d0
!       c=b-a
!       sum6=0d0
!       do 10 in=1,6
!  10   sum6= sum6+ w(in)*f(b-c*xx(in))
!       sum6=sum6*c
!       a1=a
!  15   a2= (a1+b)/2
!       c=b-a2
!       sumn=0d0
!       do 20 in=1,6
!  20   sumn= sumn+ w(in)*f(b-c*xx(in)) !!! FB: f(b) = NaN !
!       sumn=sumn*c
! ctt
! c      call adqua(a1,a2,f,sum1,eps)
! c      sum1=sum1+sum
!       sum1=AD8GLE(F,A1,A2,eps)+sum
!       IF(ABS( (sum+sum6)/(sum1+sumn)-1d0 ).lt.EPS) THEN
! ctt
! c      call adqua(a,a2,f,sum2,eps)
!         sum2=AD8GLE(F,A,A2,eps)
!         IF(ABS( (sum2+sumn)/(sum1+sumn)-1d0 ).gt.EPS) THEN
!            sum=sum2
!            a1=a2
!            sum6=sumn
!            goto 15
!         ENDIF
!         ADGLG1= SUM1+SUMN
!         RETURN
!       ELSE
!         sum=sum1
!         a1=a2
!         sum6=sumn
!         goto 15
!       ENDIF
!       END
! C
!       DOUBLE PRECISION FUNCTION ADGLG2(F,A,B,EPS)
!       IMPLICIT REAL*8 (A-H,O-Z)
!       EXTERNAL F,AD8GLE
!       DIMENSION W(6),X(6),xx(6)
! c      SAVE W,XX,NUM
!       SAVE
! C
! C     ****************************************************************
! C
! C     ADAPTIVE GAUSSIAN QUADRATURE.
! C     For x->A   f(x) = O (ln^k (x-a) )
! C     A - lower limit  (integrable singularity), B - upper limit
! C     AD8GLE IS SET EQUAL TO THE APPROXIMATE VALUE OF THE INTEGRAL OF
! C     THE FUNCTION F OVER THE INTERVAL (A,B), WITH ACCURACY PARAMETER
! C     EPS.
```

```
! C
! C     ****************************************************************
!       DATA W / 4.58964 673950d-1,
!      $         4.17000 830772d-1,
!      $         1.13373 382074d-1,
!      $         1.03991 974531d-2,
!      $         2.61017 202815d-4,
!      $         8.98547 906430d-7/
! C
!       DATA X / 0.22284 66041 79d0,
!      $         1.18893 21016 73d0,
!      $         2.99273 63260 59d0,
!      $         5.77514 35691 05d0,
!      $         9.83746 74183 83d0,
!      $        15.98287 39806 02d0/
!       DATA NUM/0/
!       IF(NUM.eq.0d0) then
!       do 1 ix=1,6
!   1   xx(ix)= EXP(-x(ix))
!       ENDIF
!       num=num+1
!       sum=0d0
!       c=b-a
!       sum6=0d0
!       do 10 in=1,6
!  10   sum6= sum6+ w(in)*f(A+c*xx(in))
!       sum6=sum6*c
!       b1=b
!  15   b2= (a+b1)/2
!       c=b2-a
!       sumn=0d0
!       do 20 in=1,6
!  20   sumn= sumn+ w(in)*f(a+c*xx(in)) !!! FB: f(a) = NaN !
!       sumn=sumn*c
!       sum1=AD8GLE(F,b2,b1,eps)+sum
!       IF(ABS( (sum+sum6)/(sum1+sumn)-1d0 ).lt.EPS) THEN
!          sum2=AD8GLE(F,b2,b,eps)
!          IF(ABS( (sum2+sumn)/(sum1+sumn)-1d0 ).gt.EPS) THEN
!             sum=sum2
!             b1=b2
!             sum6=sumn
!             goto 15
!          ENDIF
!          ADGLG2= SUM1+SUMN
!          RETURN
!       ELSE
!          sum=sum1
!          b1=b2
!          sum6=sumn
!          goto 15
!       ENDIF
!       END
! C
! C
```

```
! C-----------------------------------------------------------------
! C INTEGRATION ROUTINE ADQUA written by M. Jezabek           ------
! C-----------------------------------------------------------------
! C
!       SUBROUTINE ADQUA(XL,XU,F,Y,ACC)
! C
! C     ADAPTIVE GAUSS-LEGENDRE + SIMPSON'S RULE QUADRATURE
! C     XL - LOWER LIMIT, XU - UPPER LIMIT, F - FUNCTION TO INTEGRATE
! C     Y - INTEGRAL
! C     ACC - ACCURACY (IF .LE. 0.  ACC=1.D-6)
! c     ****** new constants,  1 error removed, Oct '92
! C
! C     CALLS: SIMPSA
! C
! C     PARAMETERS: NSUB > NO OF SUBDIVISION LEVELS IN GAUSS INTEGRATION
! C          100*2**IMAX > NO OF POINTS IN SIMPSON INTEGRATION
! C
!       IMPLICIT REAL*8 (A-H,O-Z)
!       EXTERNAL F
!       DIMENSION VAL(25,2), BOUND(25,2,2), LEV(25),SING(25,3)
!       DIMENSION W8(4),X8(4)
!       DATA W8
!      $/0.101228536290376D0, 0.222381034453374D0, 0.313706645877887D0,
!      $ 0.362683783378362D0/
!       DATA X8
!      $/0.960289856497536D0, 0.796666477413627D0, 0.525532409916329D0,
!      $ 0.183434642495650D0/
!       save
! C
!       IF(ACC.LE.0.D0) ACC=1.D-6
!       NSUB=24
!       NSG=25
!       NSC=0
!       A=XL
!       B=XU
!       C1=0.5d0*(A+B)
!       C2=C1-A
!       S8=0d0
!       DO 1 I=1,4
!       U=X8(I)*C2
!     1 S8=S8+W8(I)*(F(C1+U)+F(C1-U))
!       S8=S8*C2
!       XM=(XL+XU)/2.d0
!       BOUND(1,1,1)=XL
!       BOUND(1,1,2)=XM
!       BOUND(1,2,1)=XM
!       BOUND(1,2,2)=XU
!       NC=1
!       DO 3 IX=1,2
!       A=BOUND(NC,IX,1)
!       B=BOUND(NC,IX,2)
!       C1=0.5d0*(A+B)
!       C2=C1-A
!       VAL(NC,IX)=0.d0
```

```
!       DO 2 I=1,4
!       U=X8(I)*C2
!     2 VAL(NC,IX)=VAL(NC,IX)+W8(I)*(F(C1+U)+F(C1-U))
!     3 VAL(NC,IX)=VAL(NC,IX)*C2
!       S16=VAL(NC,1)+VAL(NC,2)
!       IF(DABS(S8-S16).GT.ACC*DABS(S16)) GOTO 4
!       Y=S16
!       RETURN
!     4 DO 5 I=1,NSUB
!     5 LEV(I)=0
!       NC1= NC+1
!    11 XM=(BOUND(NC,1,1)+BOUND(NC,1,2))/2.d0
!       BOUND(NC1,1,1)=BOUND(NC,1,1)
!       BOUND(NC1,1,2)=XM
!       BOUND(NC1,2,1)=XM
!       BOUND(NC1,2,2)=BOUND(NC,1,2)
!       DO 13 IX=1,2
!       A=BOUND(NC1,IX,1)
!       B=BOUND(NC1,IX,2)
!       C1=0.5d0*(A+B)
!       C2=C1-A
!       VAL(NC1,IX)=0.d0
!       DO 12 I=1,4
!       U=X8(I)*C2
!    12 VAL(NC1,IX)=VAL(NC1,IX)+W8(I)*(F(C1+U)+F(C1-U))
!    13 VAL(NC1,IX)=VAL(NC1,IX)*C2
!       S16=VAL(NC1,1)+VAL(NC1,2)
!       S8=VAL(NC,1)
!       IF(DABS(S8-S16).LE.ACC*DABS(S16)) GOTO 20
!       NC=NC1
!       NC1= NC+1
!       IF(NC1.LE.NSUB) GOTO 11
! C     NC=NSUB   USE SIMPSON'S RULE
!       NSC=NSC+1
!       IF(NSC.LE.NSG) GOTO 15
!       WRITE(*,911)
!   911 FORMAT(1X,'ADQUA: TOO MANY SINGULARITIES')
!       STOP
!    15 SING(NSC,1)=BOUND(NC,1,1)
!       SING(NSC,2)=BOUND(NC,2,2)
!       SING(NSC,3)=S16
!       S16=0.d0
!       NC=NC-1
!    20 VAL(NC,1)= S16
!   121 LEV(NC)=1
!    21 XM=(BOUND(NC,2,1)+BOUND(NC,2,2))/2.d0
!       BOUND(NC1,1,1)=BOUND(NC,2,1)
!       BOUND(NC1,1,2)=XM
!       BOUND(NC1,2,1)=XM
!       BOUND(NC1,2,2)=BOUND(NC,2,2)
!       DO 23 IX=1,2
!       A=BOUND(NC1,IX,1)
!       B=BOUND(NC1,IX,2)
!       C1=0.5d0*(A+B)
```

```
!       C2=C1-A
!       VAL(NC1,IX)=0.d0
!       DO 22 I=1,4
!       U=X8(I)*C2
!    22 VAL(NC1,IX)=VAL(NC1,IX)+W8(I)*(F(C1+U)+F(C1-U))
!    23 VAL(NC1,IX)=VAL(NC1,IX)*C2
!       S16=VAL(NC1,1)+VAL(NC1,2)
!       S8=VAL(NC,2)
!       IF(DABS(S8-S16).LE.ACC*DABS(S16)) GOTO 40
!       NC=NC+1
!       NC1=NC+1
!       IF(NC1.LE.NSUB) GOTO 11
! C     NC=NSUB   USE SIMPSON'S RULE
!       NSC=NSC+1
!       IF(NSC.LE.NSG) GOTO 35
!       WRITE(*,911)
!       STOP
!    35 SING(NSC,1)=BOUND(NC,1,1)
!       SING(NSC,2)=BOUND(NC,2,2)
!       SING(NSC,3)=S16
!       S16=0.d0
!       NC=NC-1
!    40 VAL(NC,2)= S16
!    45 IF(NC.GT.1) GOTO 50
!       Y1=VAL(1,1)+VAL(1,2)
!       GOTO 100
!    50 NC0=NC-1
!       IF(LEV(NC0).EQ.0) IX=1
!       IF(LEV(NC0).EQ.1) IX=2
!       LEV(NC)=0
!       NC1=NC
!       VAL(NC0,IX)=VAL(NC,1)+VAL(NC,2)
!       NC=NC0
!       IF(IX.EQ.1) GOTO 121
!       GOTO 45
!   100 CONTINUE
!       IF(NSC.GT.0) GOTO 101
!       Y=Y1
!       RETURN
!   101 FSUM=0.d0
!       DO 102 IK=1,NSC
!   102 FSUM=FSUM+DABS(SING(IK,3))
!       ACCR=ACC*DMAX1(FSUM,DABS(Y1))/FSUM/10.d0
!       DO 104 IK=1,NSC
!   104 CALL SIMPSA(SING(IK,1),SING(IK,2),F,SING(IK,3),ACCR)
!       DO 106 IK=1,NSC
!   106 Y1=Y1+SING(IK,3)
!       Y=Y1
!       RETURN
!       END
! C
!       SUBROUTINE SIMPSA(A,B,F,F0,ACC)
! C     SIMPSON'S ADAPTIVE QUADRATURE
!       IMPLICIT REAL*8 (A-H,O-Z)
```

```
!       save
!       EXTERNAL F
!       IMAX=5
!       N0=100
!       H=(B-A)/N0
!       N02=N0/2
!       S2=0.d0
!       IC=1
!       S0=F(A)+F(B)
!       DO 5 K=1,N02
!     5 S2=S2+F(A+2.d0*K*H)
!     7 S1=0.d0
!       DO 10 K=1,N02
!    10 S1=S1+F(A+(2.d0*K-1.d0)*H)
!       Y=H/3.d0*(S0+4.d0*S1+2.d0*S2)
!       IF(DABS(F0/Y-1.d0).GT.ACC) GOTO 20
!       RETURN
!    20 N02=N0
!       N0=2*N0
!       S2=S1+S2
!       H=H/2.d0
!       IF(IC.GT.IMAX) GOTO 30
!       F0=Y
!       IC=IC+1
!       GOTO 7
!    30 ACC0=DABS(Y/F0-1.d0)
!       WRITE(*,900) A,B,ACC0
!       STOP
!   900 FORMAT(1H ,'SIMPSA: TOO HIGH ACCURACY REQUIRED'/
!      /1X,   29HSINGULARITY IN THE INTERVAL  ,D20.12,1X,D20.12/
!      /1X,   29HACCURACY ACHIEVED            ,D20.12)
!       END
! C
! C
! C  ******* matrix-inversion-routines
! C
!       SUBROUTINE LUDCMP(A,N,NP,INDX,D)
!       IMPLICIT REAL*8(A-H,O-Z)
!       PARAMETER (NMAX=100,TINY=1.0E-20)
!       DIMENSION A(NP,NP),INDX(N),VV(NMAX)
!       D=1.
!       DO 12 I=1,N
!         AAMAX=0.
!         DO 11 J=1,N
!           IF (ABS(A(I,J)).GT.AAMAX) AAMAX=ABS(A(I,J))
! 11      CONTINUE
!         IF (AAMAX.EQ.0.) PAUSE 'Singular matrix.'
!         VV(I)=1./AAMAX
! 12    CONTINUE
!       DO 19 J=1,N
!         IF (J.GT.1) THEN
!           DO 14 I=1,J-1
!             SUM=A(I,J)
!             IF (I.GT.1)THEN
```

```
!               DO 13 K=1,I-1
!                 SUM=SUM-A(I,K)*A(K,J)
! 13            CONTINUE
!               A(I,J)=SUM
!             ENDIF
! 14        CONTINUE
!         ENDIF
!         AAMAX=0.
!         DO 16 I=J,N
!           SUM=A(I,J)
!           IF (J.GT.1)THEN
!             DO 15 K=1,J-1
!               SUM=SUM-A(I,K)*A(K,J)
! 15          CONTINUE
!             A(I,J)=SUM
!           ENDIF
!           DUM=VV(I)*ABS(SUM)
!           IF (DUM.GE.AAMAX) THEN
!             IMAX=I
!             AAMAX=DUM
!           ENDIF
! 16      CONTINUE
!         IF (J.NE.IMAX)THEN
!           DO 17 K=1,N
!             DUM=A(IMAX,K)
!             A(IMAX,K)=A(J,K)
!             A(J,K)=DUM
! 17        CONTINUE
!           D=-D
!           VV(IMAX)=VV(J)
!         ENDIF
!         INDX(J)=IMAX
!         IF(J.NE.N)THEN
!           IF(A(J,J).EQ.0.)A(J,J)=TINY
!           DUM=1./A(J,J)
!           DO 18 I=J+1,N
!             A(I,J)=A(I,J)*DUM
! 18        CONTINUE
!         ENDIF
! 19    CONTINUE
!       IF(A(N,N).EQ.0.)A(N,N)=TINY
!       RETURN
!       END
! c
!       SUBROUTINE LUBKSB(A,N,NP,INDX,B)
!       IMPLICIT REAL*8(A-H,O-Z)
!       DIMENSION A(NP,NP),INDX(N),B(N)
!       II=0
!       DO 12 I=1,N
!         LL=INDX(I)
!         SUM=B(LL)
!         B(LL)=B(I)
!         IF (II.NE.0)THEN
!           DO 11 J=II,I-1
```

```
!             SUM=SUM-A(I,J)*B(J)
! 11        CONTINUE
!         ELSE IF (SUM.NE.0.) THEN
!           II=I
!         ENDIF
!         B(I)=SUM
! 12    CONTINUE
!       DO 14 I=N,1,-1
!         SUM=B(I)
!         IF(I.LT.N)THEN
!           DO 13 J=I+1,N
!             SUM=SUM-A(I,J)*B(J)
! 13        CONTINUE
!         ENDIF
!         B(I)=SUM/A(I,I)
! 14    CONTINUE
!       RETURN
!       END
! C
! C
! C     *******  RANDOM NUMBER GENERATORS
! C
! C
!       FUNCTION RANF(DUMMY)
! C
! C   RANDOM NUMBER FUNCTION TAKEN FROM KNUTH
! C   (SEMINUMERICAL ALGORITHMS).
! C   METHOD IS X(N)=MOD(X(N-55)-X(N-24),1/FMODUL)
! C   NO PROVISION YET FOR CONTROL OVER THE SEED NUMBER.
! C
! C   RANF GIVES ONE RANDOM NUMBER BETWEEN 0 AND 1.
! C   IRN55 GENERATES 55 RANDOM NUMBERS BETWEEN 0 AND 1/FMODUL.
! C   IN55  INITIALIZES THE 55 NUMBERS AND WARMS UP THE SEQUENCE.
! C
!       PARAMETER (FMODUL=1.E-09)
!       SAVE /CIRN55/
!       COMMON /CIRN55/NCALL,MCALL,IA(55)
!       INTEGER IA
!       CALL RANDAT
!       IF( NCALL.EQ.0 ) THEN
!           CALL IN55 ( IA,234612947 )
!           MCALL = 55
!           NCALL = 1
!       ENDIF
!       IF ( MCALL.EQ.0 ) THEN
!           CALL IRN55(IA)
!           MCALL=55
!       ENDIF
!       RANF=IA(MCALL)*FMODUL
!       MCALL=MCALL-1
!       RETURN
!       END
! C
!       SUBROUTINE RANDAT
```

```
! C
! C  INITIALISES THE NUMBER NCALL TO 0 TO FLAG THE FIRST CALL
! C  OF THE RANDOM NUMBER GENERATOR
! C
! C      SAVE /CIRN55/
! C      SAVE FIRST
!       SAVE
!       COMMON /CIRN55/NCALL,MCALL,IA(55)
!       INTEGER IA
!       LOGICAL FIRST
!       DATA FIRST /.TRUE./
!       IF(FIRST)THEN
!          FIRST=.FALSE.
!          NCALL=0
!       ENDIF
!       RETURN
!       END
! C
!       SUBROUTINE IN55(IA,IX)
!       PARAMETER (MODULO=1000000000)
!       INTEGER IA(55)
! C
!       IA(55)=IX
!       J=IX
!       K=1
!       DO 10 I=1,54
!          II=MOD(21*I,55)
!          IA(II)=K
!          K=J-K
!          IF(K.LT.0)K=K+MODULO
!          J=IA(II)
! 10    CONTINUE
!       DO 20 I=1,10
!          CALL IRN55(IA)
! 20    CONTINUE
!       RETURN
!       END
! C
!       SUBROUTINE IRN55(IA)
!       PARAMETER (MODULO=1000000000)
!       INTEGER IA(55)
!       DO 10 I=1,24
!          J=IA(I)-IA(I+31)
!          IF(J.LT.0)J=J+MODULO
!          IA(I)=J
! 10    CONTINUE
!       DO 20 I=25,55
!          J=IA(I)-IA(I-24)
!          IF(J.LT.0)J=J+MODULO
!          IA(I)=J
! 20    CONTINUE
!       RETURN
!       END
! C
```

```
! C
!       FUNCTION RAN2(IDUM)
! C     ******************
!       REAL RDM(31)
!       DATA  IWARM/0/
! C
!       IF (IDUM.LT.0.OR.IWARM.EQ.0) THEN
! C INITIALIZATION OR REINITIALISATION
!       IWARM=1
!       IA1=         1279
!       IC1=       351762
!       M1=       1664557
!       IA2=         2011
!       IC2=       221592
!       M2=       1048583
!       IA3=        15091
!       IC3=         6171
!       M3=         29201
!       IX1=MOD(-IDUM,M1)
!       IX1=MOD(IA1*IX1+IC1,M1)
!       IX2=MOD(IX1,M2)
!       IX1=MOD(IA1*IX1+IC1,M1)
!       IX3=MOD(IX1,M3)
!       RM1=1./FLOAT(M1)
!       RM2=1./FLOAT(M2)
!       DO 10 J=1,31
!       IX1=MOD(IA1*IX1+IC1,M1)
!       IX2=MOD(IA2*IX2+IC2,M2)
! 10    RDM(J)=(FLOAT(IX1)+FLOAT(IX2)*RM2)*RM1
!       ENDIF
! C
! C GENERATE NEXT NUMBER IN SEQUENCE
!       IF(IWARM.EQ.0) GOTO 901
!       IX1=MOD(IA1*IX1+IC1,M1)
!       IX2=MOD(IA2*IX2+IC2,M2)
!       IX3=MOD(IA3*IX3+IC3,M3)
!       J=1+(31*IX3)/M3
!       RAN2=RDM(J)
!       RDM(J)=(FLOAT(IX1)+FLOAT(IX2)*RM2)*RM1
!       RETURN
! 901   PRINT 9010
! 9010  FORMAT('   RAN2: LACK OF ITINIALISATION')
!       STOP
!       END
! C
! C
! C     *******    SPECIAL FUNCTIONS
! C
! C
!       DOUBLE PRECISION FUNCTION DILOG(X)
! C
! C     SPENCE'S DILOGARITHM IN DOUBLE PRECISION
! C
!       IMPLICIT REAL*8 (A-H,O-Z)
```

```
!       Z=-1.644934066848226
!       IF(X .LT.-1.0) GO TO 1
!       IF(X .LE. 0.5) GO TO 2
!       IF(X .EQ. 1.0) GO TO 3
!       IF(X .LE. 2.0) GO TO 4
!       Z=3.289868133696453
!     1 T=1.0/X
!       S=-0.5
!       Z=Z-0.5*DLOG(DABS(X))**2
!       GO TO 5
!     2 T=X
!       S=0.5
!       Z=0.
!       GO TO 5
!     3 DILOG=1.644934066848226
!       RETURN
!     4 T=1.0-X
!       S=-0.5
!       Z=1.644934066848226-DLOG(X)*DLOG(DABS(T))
!     5 Y=2.666666666666667*T+0.666666666666667
!       B=      0.00000 00000 00001
!       A=Y*B  +0.00000 00000 00004
!       B=Y*A-B+0.00000 00000 00011
!       A=Y*B-A+0.00000 00000 00037
!       B=Y*A-B+0.00000 00000 00121
!       A=Y*B-A+0.00000 00000 00398
!       B=Y*A-B+0.00000 00000 01312
!       A=Y*B-A+0.00000 00000 04342
!       B=Y*A-B+0.00000 00000 14437
!       A=Y*B-A+0.00000 00000 48274
!       B=Y*A-B+0.00000 00001 62421
!       A=Y*B-A+0.00000 00005 50291
!       B=Y*A-B+0.00000 00018 79117
!       A=Y*B-A+0.00000 00064 74338
!       B=Y*A-B+0.00000 00225 36705
!       A=Y*B-A+0.00000 00793 87055
!       B=Y*A-B+0.00000 02835 75385
!       A=Y*B-A+0.00000 10299 04264
!       B=Y*A-B+0.00000 38163 29463
!       A=Y*B-A+0.00001 44963 00557
!       B=Y*A-B+0.00005 68178 22718
!       A=Y*B-A+0.00023 20021 96094
!       B=Y*A-B+0.00100 16274 96164
!       A=Y*B-A+0.00468 63619 59447
!       B=Y*A-B+0.02487 93229 24228
!       A=Y*B-A+0.16607 30329 27855
!       A=Y*A-B+1.93506 43008 69969
!       DILOG=S*T*(A-B)+Z
!       RETURN
!       END
! C
! c Everything for hypergeometric function F_{2,1},
! c  taken from Numerical Recipes.
! c
```

```
!       FUNCTION hypgeo(a,b,c,z)
!       implicit none
!       COMPLEX*16 hypgeo,a,b,c,z,hp
!       REAL*8 EPS
! cttt
! c      PARAMETER (EPS=1.d-6)
!       PARAMETER (EPS=1.d-8)
!       INTEGER kmax,nbad,nok
!       COMPLEX*16 z0,dz,aa,bb,cc,y(2)
! cu    needs bsstep,hypser,odeint,mmid,hypdrv
!       COMMON /hypg/ aa,bb,cc,z0,dz
!       COMMON /path1/ kmax
!       kmax=0
!       if (real(z)**2+aimag(z)**2.le.0.25d0) then
!         call hypser(a,b,c,z,hp,y(2))
!         hypgeo=hp
!         return
!       else if (real(z).lt.0.d0) then
!         z0=cmplx(-0.5d0,0.d0,kind=kind(0d0))
!       else if (real(z).le.1.d0) then
!         z0=cmplx(0.5d0,0.d0,kind=kind(0d0))
!       else
!         z0=cmplx(0.d0,sign(0.5d0,aimag(z)),kind=kind(0d0))
!       endif
!       aa=a
!       bb=b
!       cc=c
!       dz=z-z0
!       call hypser(aa,bb,cc,z0,y(1),y(2))
!       call odeint(y,4,0.d0,1.d0,EPS,.1d0,.00001d0,nok,nbad)
!       hypgeo=y(1)
!       return
!       END
! c
!       SUBROUTINE odeint(ystart,nvar,x1,x2,eps,h1,hmin,nok,nbad)
!       implicit none
!       INTEGER nbad,nok,nvar,KMAXX,MAXSTP,NMAX
!       REAL*8 eps,h1,hmin,x1,x2,ystart(nvar),TINY
!       PARAMETER (MAXSTP=10000,NMAX=50,KMAXX=200,TINY=1.d-30)
!       INTEGER i,kmax,kount,nstp
!       REAL*8 dxsav,h,hdid,hnext,x,xsav,dydx(NMAX),xp(KMAXX),y(NMAX),
!      *yp(NMAX,KMAXX),yscal(NMAX)
!       COMMON /path/ dxsav,xp,yp,kount
!       COMMON /path1/ kmax
!       x=x1
!       h=sign(h1,x2-x1)
!       nok=0
!       nbad=0
!       kount=0
!       do 11 i=1,nvar
!         y(i)=ystart(i)
! 11    continue
!       if (kmax.gt.0) xsav=x-2.d0*dxsav
!       do 16 nstp=1,MAXSTP
```

```
!         call hypdrv(x,y,dydx)
!         do 12 i=1,nvar
!           yscal(i)=abs(y(i))+abs(h*dydx(i))+TINY
! 12      continue
!         if(kmax.gt.0)then
!           if(abs(x-xsav).gt.abs(dxsav)) then
!             if(kount.lt.kmax-1)then
!               kount=kount+1
!               xp(kount)=x
!               do 13 i=1,nvar
!                 yp(i,kount)=y(i)
! 13            continue
!               xsav=x
!             endif
!           endif
!         endif
!         if((x+h-x2)*(x+h-x1).gt.0.d0) h=x2-x
!         call bsstep(y,dydx,nvar,x,h,eps,yscal,hdid,hnext)
!         if(hdid.eq.h)then
!           nok=nok+1
!         else
!           nbad=nbad+1
!         endif
!         if((x-x2)*(x2-x1).ge.0.d0)then
!           do 14 i=1,nvar
!             ystart(i)=y(i)
! 14        continue
!           if(kmax.ne.0)then
!             kount=kount+1
!             xp(kount)=x
!             do 15 i=1,nvar
!               yp(i,kount)=y(i)
! 15          continue
!           endif
!           return
!         endif
!         if(abs(hnext).lt.hmin) pause
!      *'stepsize smaller than minimum in odeint'
!         h=hnext
! 16    continue
!       pause 'too many steps in odeint'
!       return
!       END
! c
!       SUBROUTINE bsstep(y,dydx,nv,x,htry,eps,yscal,hdid,hnext)
!       implicit none
!       INTEGER nv,NMAX,KMAXX,IMAX
!       REAL*8 eps,hdid,hnext,htry,x,dydx(nv),y(nv),yscal(nv),
!      *SAFE1,SAFE2,REDMAX,REDMIN,TINY,SCALMX
!       PARAMETER (NMAX=50,KMAXX=8,IMAX=KMAXX+1,SAFE1=.25d0,SAFE2=.7d0,
!      *REDMAX=1.d-5,REDMIN=.7d0,TINY=1.d-30,SCALMX=.1d0)
!       INTEGER i,iq,k,kk,km,kmax,kopt,nseq(IMAX)
!       REAL*8 eps1,epsold,errmax,fact,h,red,scale,work,wrkmin,xest,
!      *xnew,a(IMAX),alf(KMAXX,KMAXX),err(KMAXX),yerr(NMAX),ysav(NMAX),
```

```
!      *yseq(NMAX)
!       LOGICAL first,reduct
!       SAVE a,alf,epsold,first,kmax,kopt,nseq,xnew
!       DATA first/.true./,epsold/-1.d0/
!       DATA nseq /2,4,6,8,10,12,14,16,18/
!       if(eps.ne.epsold)then
!         hnext=-1.d29
!         xnew=-1.d29
!         eps1=SAFE1*eps
!         a(1)=nseq(1)+1
!         do 11 k=1,KMAXX
!           a(k+1)=a(k)+nseq(k+1)
! 11      continue
!         do 13 iq=2,KMAXX
!           do 12 k=1,iq-1
!             alf(k,iq)=eps1**((a(k+1)-a(iq+1))/((a(iq+1)-a(1)+1.d0)*
!      *(2.d0*k+1.d0)))
! 12        continue
! 13      continue
!         epsold=eps
!         do 14 kopt=2,KMAXX-1
!           if(a(kopt+1).gt.a(kopt)*alf(kopt-1,kopt))goto 1
! 14      continue
! 1       kmax=kopt
!       endif
!       h=htry
!       do 15 i=1,nv
!         ysav(i)=y(i)
! 15    continue
!       if(h.ne.hnext.or.x.ne.xnew)then
!         first=.true.
!         kopt=kmax
!       endif
!       reduct=.false.
! 2     do 17 k=1,kmax
!         xnew=x+h
!         if(xnew.eq.x)pause 'step size underflow in bsstep'
!         call mmid(ysav,dydx,nv,x,h,nseq(k),yseq)
!         xest=(h/nseq(k))**2
!         call pzext0(k,xest,yseq,y,yerr,nv)
!         if(k.ne.1)then
!           errmax=TINY
!           do 16 i=1,nv
!             errmax=max(errmax,abs(yerr(i)/yscal(i)))
! 16        continue
!           errmax=errmax/eps
!           km=k-1
!           err(km)=(errmax/SAFE1)**(1.d0/(2.d0*km+1.d0))
!         endif
!         if(k.ne.1.and.(k.ge.kopt-1.or.first))then
!           if(errmax.lt.1.)goto 4
!           if(k.eq.kmax.or.k.eq.kopt+1)then
!             red=SAFE2/err(km)
!             goto 3
```

```
!           else if(k.eq.kopt)then
!             if(alf(kopt-1,kopt).lt.err(km))then
!               red=1.d0/err(km)
!               goto 3
!             endif
!           else if(kopt.eq.kmax)then
!             if(alf(km,kmax-1).lt.err(km))then
!               red=alf(km,kmax-1)*SAFE2/err(km)
!               goto 3
!             endif
!           else if(alf(km,kopt).lt.err(km))then
!             red=alf(km,kopt-1)/err(km)
!             goto 3
!           endif
!         endif
! 17    continue
! 3     red=min(red,REDMIN)
!       red=max(red,REDMAX)
!       h=h*red
!       reduct=.true.
!       goto 2
! 4     x=xnew
!       hdid=h
!       first=.false.
!       wrkmin=1.d35
!       do 18 kk=1,km
!         fact=max(err(kk),SCALMX)
!         work=fact*a(kk+1)
!         if(work.lt.wrkmin)then
!           scale=fact
!           wrkmin=work
!           kopt=kk+1
!         endif
! 18    continue
!       hnext=h/scale
!       if(kopt.ge.k.and.kopt.ne.kmax.and..not.reduct)then
!         fact=max(scale/alf(kopt-1,kopt),SCALMX)
!         if(a(kopt+1)*fact.le.wrkmin)then
!           hnext=h/fact
!           kopt=kopt+1
!         endif
!       endif
!       return
!       END
! c
!       SUBROUTINE hypser(a,b,c,z,series,deriv)
!       implicit none
!       INTEGER n
!       COMPLEX*16 a,b,c,z,series,deriv,aa,bb,cc,fac,temp
!       deriv=cmplx(0.d0,0.d0,kind=kind(0d0))
!       fac=cmplx(1.d0,0.d0,kind=kind(0d0))
!       temp=fac
!       aa=a
!       bb=b
```

```
!       cc=c
!       do 11 n=1,1000
!         fac=fac*aa*bb/cc
!         deriv=deriv+fac
!         fac=fac*z/n
!         series=temp+fac
!         if (series.eq.temp) return
!         temp=series
!         aa=aa+1.d0
!         bb=bb+1.d0
!         cc=cc+1.d0
! 11    continue
!       pause 'convergence failure in hypser'
!       END
! c
!       SUBROUTINE hypdrv(s,y,dyds)
!       implicit none
!       REAL*8 s
!       COMPLEX*16 y(2),dyds(2),aa,bb,cc,z0,dz,z
!       COMMON /hypg/ aa,bb,cc,z0,dz
!       z=z0+s*dz
!       dyds(1)=y(2)*dz
!       dyds(2)=(aa*bb*y(1)-(cc-(aa+bb+1.d0)*z)*y(2))*dz/(z*(1.d0-z))
!       return
!       END
! c
!       SUBROUTINE mmid(y,dydx,nvar,xs,htot,nstep,yout)
!       implicit none
!       INTEGER nstep,nvar,NMAX
!       REAL*8 htot,xs,dydx(nvar),y(nvar),yout(nvar)
!       PARAMETER (NMAX=50)
!       INTEGER i,n
!       REAL*8 h,h2,swap,x,ym(NMAX),yn(NMAX)
!       h=htot/nstep
!       do 11 i=1,nvar
!         ym(i)=y(i)
!         yn(i)=y(i)+h*dydx(i)
! 11    continue
!       x=xs+h
!       call hypdrv(x,yn,yout)
!       h2=2.d0*h
!       do 13 n=2,nstep
!         do 12 i=1,nvar
!           swap=ym(i)+h2*yout(i)
!           ym(i)=yn(i)
!           yn(i)=swap
! 12      continue
!         x=x+h
!         call hypdrv(x,yn,yout)
! 13    continue
!       do 14 i=1,nvar
!         yout(i)=0.5d0*(ym(i)+yn(i)+h*yout(i))
! 14    continue
!       return
```

```
!       END
! c
!       SUBROUTINE pzext0(iest,xest,yest,yz,dy,nv)
!       implicit none
!       INTEGER iest,nv,IMAX,NMAX
!       REAL*8 xest,dy(nv),yest(nv),yz(nv)
!       PARAMETER (IMAX=13,NMAX=50)
!       INTEGER j,k1
!       REAL*8 delta,f1,f2,q,d(NMAX),qcol(NMAX,IMAX),x(IMAX)
!       SAVE qcol,x
!       x(iest)=xest
!       do 11 j=1,nv
!         dy(j)=yest(j)
!         yz(j)=yest(j)
! 11    continue
!       if(iest.eq.1) then
!         do 12 j=1,nv
!           qcol(j,1)=yest(j)
! 12      continue
!       else
!         do 13 j=1,nv
!           d(j)=yest(j)
! 13      continue
!         do 15 k1=1,iest-1
!           delta=1.d0/(x(iest-k1)-xest)
!           f1=xest*delta
!           f2=x(iest-k1)*delta
!           do 14 j=1,nv
!             q=qcol(j,k1)
!             qcol(j,k1)=dy(j)
!             delta=d(j)-q
!             dy(j)=f1*delta
!             d(j)=f2*delta
!             yz(j)=yz(j)+dy(j)
! 14        continue
! 15      continue
!         do 16 j=1,nv
!           qcol(j,iest)=dy(j)
! 16      continue
!       endif
!       return
!       END
! c
!!! /FB
```

⟨ttv_formfactors.f90⟩≡

⟨*File header*⟩

```
module ttv_formfactors

  use kinds
  use constants
  use numeric_utils
  use physics_defs, only: CF, CA, TR
  use sm_physics
```

```
  use lorentz
  use interpolation
  use nr_tools
  use io_units, only: free_unit, given_output_unit
  use string_utils
  use iso_varying_string, string_t => varying_string
  use system_dependencies
  use, intrinsic :: iso_fortran_env !NODEP!
  use diagnostics
  use solver, only: solver_function_t, solve_qgaus
⟨Standard module head⟩
  save

⟨ttv formfactors: public⟩

⟨ttv formfactors: parameters⟩

⟨ttv formfactors: types⟩

⟨ttv formfactors: global variables⟩

⟨ttv formfactors: interfaces⟩

contains

⟨ttv formfactors: procedures⟩

end module ttv_formfactors
```

⟨*ttv formfactors: public*⟩≡

```
  public :: onshell_projection_t
```

⟨*ttv formfactors: types*⟩≡

```
  type :: onshell_projection_t
     logical :: production
     logical :: decay
     logical :: width
     logical :: boost_decay
  contains
   ⟨ttv formfactors: onshell projection: TBP⟩
  end type onshell_projection_t
```

⟨*ttv formfactors: onshell projection: TBP*⟩≡

```
  procedure :: debug_write => onshell_projection_debug_write
```

⟨*ttv formfactors: procedures*⟩≡

```
  subroutine onshell_projection_debug_write (onshell_projection)
    class(onshell_projection_t), intent(in) :: onshell_projection
    call msg_debug (D_THRESHOLD, "onshell_projection%production", &
         onshell_projection%production)
    call msg_debug (D_THRESHOLD, "onshell_projection%decay", &
         onshell_projection%decay)
    call msg_debug (D_THRESHOLD, "onshell_projection%width", &
         onshell_projection%width)
    call msg_debug (D_THRESHOLD, "onshell_projection%boost_decay", &
```

```
        onshell_projection%boost_decay)
    end subroutine onshell_projection_debug_write
```

⟨*ttv formfactors: onshell projection: TBP*⟩+≡

```
    procedure :: set_all => onshell_projection_set_all
```

⟨*ttv formfactors: procedures*⟩+≡

```
    pure subroutine onshell_projection_set_all (onshell_projection, flag)
      class(onshell_projection_t), intent(inout) :: onshell_projection
      logical, intent(in) :: flag
      onshell_projection%production = flag
      onshell_projection%decay = flag
    end subroutine onshell_projection_set_all
```

⟨*ttv formfactors: onshell projection: TBP*⟩+≡

```
    procedure :: active => onshell_projection_active
```

⟨*ttv formfactors: procedures*⟩+≡

```
    pure function onshell_projection_active (onshell_projection) result (active)
      logical :: active
      class(onshell_projection_t), intent(in) :: onshell_projection
      active = onshell_projection%production .or. &
           onshell_projection%decay
    end function onshell_projection_active
```

⟨*ttv formfactors: types*⟩+≡

```
    type :: helicity_approximation_t
      logical :: simple = .false.
      logical :: extra = .false.
      logical :: ultra = .false.
    contains
```

⟨*ttv formfactors: helicity approximation: TBP*⟩

```
    end type helicity_approximation_t
```

⟨*ttv formfactors: public*⟩+≡

```
    public :: settings_t
```

⟨*ttv formfactors: types*⟩+≡

```
    type :: settings_t
       ! look what is set by initialized_parameters, bundle them in a class and rename to initialize
       logical :: initialized_parameters
       ! this belongs to init_threshold_phase_space_grid in phase_space_grid_t
       logical :: initialized_ps
       ! this belongs to the ff_grid_t, its usefulness is doubtful
       logical :: initialized_ff
       logical :: mpole_dynamic
       integer :: offshell_strategy
       logical :: factorized_computation
       logical :: interference
       logical :: only_interference_term
       logical :: nlo
       logical :: no_nlo_width_in_signal_propagators
       logical :: force_minus_one
```

```
      logical :: flip_relative_sign
      integer :: sel_hel_top = 0
      integer :: sel_hel_topbar = 0
      logical :: Z_disabled
      type(onshell_projection_t) :: onshell_projection
      type(helicity_approximation_t) :: helicity_approximation
   contains
    ⟨ttv formfactors: settings: TBP⟩
   end type settings_t
```

⟨*ttv formfactors: settings: TBP*⟩≡

```
   procedure :: setup_flags => settings_setup_flags
```

⟨*ttv formfactors: procedures*⟩+≡

```
   ! TODO: (bcn 2016-03-21) break this up into a part regarding the
   ! FF grid and a part regarding the settings
   subroutine settings_setup_flags (settings, ff_in, offshell_strategy_in, &
            top_helicity_selection)
     class(settings_t), intent(inout) :: settings
     integer, intent(in) :: ff_in, offshell_strategy_in, top_helicity_selection
     logical :: bit_top, bit_topbar
     !!! RESUMMED_SWITCHOFF = - 2
     !!! MATCHED = -1, &
     SWITCHOFF_RESUMMED                 = ff_in < 0
     TOPPIK_RESUMMED                    = ff_in <= 1
     settings%nlo = btest(offshell_strategy_in, 0)
     settings%factorized_computation = btest(offshell_strategy_in, 1)
     settings%interference = btest(offshell_strategy_in, 2)
     call settings%onshell_projection%set_all(btest(offshell_strategy_in, 3))
     settings%no_nlo_width_in_signal_propagators = btest(offshell_strategy_in, 4)
     settings%helicity_approximation%simple = btest(offshell_strategy_in, 5)
     if (.not. settings%onshell_projection%active ()) then
        settings%onshell_projection%production = btest(offshell_strategy_in, 6)
        settings%onshell_projection%decay = btest(offshell_strategy_in, 7)
     end if
     settings%onshell_projection%width = .not. btest(offshell_strategy_in, 8)
     settings%onshell_projection%boost_decay = btest(offshell_strategy_in, 9)
     settings%helicity_approximation%extra = btest(offshell_strategy_in, 10)
     settings%force_minus_one = btest(offshell_strategy_in, 11)
     settings%flip_relative_sign = btest(offshell_strategy_in, 12)
     if (top_helicity_selection > -1) then
        settings%helicity_approximation%ultra = .true.
        bit_top = btest (top_helicity_selection, 0)
        bit_topbar = btest (top_helicity_selection, 1)
        if (bit_top) then
           settings%sel_hel_top = 1
        else
           settings%sel_hel_top = -1
        end if
        if (bit_topbar) then
           settings%sel_hel_topbar = 1
        else
           settings%sel_hel_topbar = -1
        end if
```

```
    end if
    settings%only_interference_term = btest(offshell_strategy_in, 14)
    settings%Z_disabled = btest(offshell_strategy_in, 15)
    if (ff_in == MATCHED .or. ff_in == MATCHED_NOTSOHARD) then
       settings%onshell_projection%width = .true.
       settings%onshell_projection%production = .true.
       settings%onshell_projection%decay = .true.
       settings%factorized_computation = .true.
       settings%interference = .true.
       settings%onshell_projection%boost_decay = .true.
    end if
    call msg_debug (D_THRESHOLD, "SWITCHOFF_RESUMMED", SWITCHOFF_RESUMMED)
    call msg_debug (D_THRESHOLD, "TOPPIK_RESUMMED", TOPPIK_RESUMMED)
    if (debug_active (D_THRESHOLD)) &
         call settings%write ()
  end subroutine settings_setup_flags
```

⟨*ttv formfactors: settings: TBP*⟩+≡

```
  procedure :: write => settings_write
```

⟨*ttv formfactors: procedures*⟩+≡

```
  subroutine settings_write (settings, unit)
    class(settings_t), intent(in) :: settings
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, '(A,L1)') "settings%helicity_approximation%simple = ", &
         settings%helicity_approximation%simple
    write (u, '(A,L1)') "settings%helicity_approximation%extra = ", &
         settings%helicity_approximation%extra
    write (u, '(A,L1)') "settings%helicity_approximation%ultra = ", &
         settings%helicity_approximation%ultra
    write (u, '(A,L1)') "settings%initialized_parameters = ", &
         settings%initialized_parameters
    write (u, '(A,L1)') "settings%initialized_ps = ", &
         settings%initialized_ps
    write (u, '(A,L1)') "settings%initialized_ff = ", &
         settings%initialized_ff
    write (u, '(A,L1)') "settings%mpole_dynamic = ", &
         settings%mpole_dynamic
    write (u, '(A,I5)') "settings%offshell_strategy = ", &
         settings%offshell_strategy
    write (u, '(A,L1)') "settings%factorized_computation = ", &
         settings%factorized_computation
    write (u, '(A,L1)') "settings%interference = ", settings%interference
    write (u, '(A,L1)') "settings%only_interference_term = ", &
         settings%only_interference_term
    write (u, '(A,L1)') "settings%Z_disabled = ", &
         settings%Z_disabled
    write (u, '(A,L1)') "settings%nlo = ", settings%nlo
    write (u, '(A,L1)') "settings%no_nlo_width_in_signal_propagators = ", &
         settings%no_nlo_width_in_signal_propagators
    write (u, '(A,L1)') "settings%force_minus_one = ", settings%force_minus_one
    write (u, '(A,L1)') "settings%flip_relative_sign = ", settings%flip_relative_sign
```

```
    call settings%onshell_projection%debug_write ()
  end subroutine settings_write
```

⟨*ttv formfactors: settings: TBP*⟩+≡

```
  procedure :: use_nlo_width => settings_use_nlo_width
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure function settings_use_nlo_width (settings, ff) result (nlo)
    logical :: nlo
    class(settings_t), intent(in) :: settings
    integer, intent(in) :: ff
    nlo = settings%nlo
  end function settings_use_nlo_width
```

⟨*ttv formfactors: public*⟩+≡

```
  public :: formfactor_t
```

⟨*ttv formfactors: types*⟩+≡

```
  type :: formfactor_t
     logical :: active
  contains
   ⟨ttv formfactors: formfactor: TBP⟩
  end type formfactor_t
```

⟨*ttv formfactors: formfactor: TBP*⟩≡

```
  procedure :: activate => formfactor_activate
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure subroutine formfactor_activate (formfactor)
    class(formfactor_t), intent(inout) :: formfactor
    formfactor%active = .true.
  end subroutine formfactor_activate
```

⟨*ttv formfactors: formfactor: TBP*⟩+≡

```
  procedure :: disable => formfactor_disable
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure subroutine formfactor_disable (formfactor)
    class(formfactor_t), intent(inout) :: formfactor
    formfactor%active = .false.
  end subroutine formfactor_disable
```

This function actually returns $\tilde{F}$, i.e. $F - 1$.

⟨*ttv formfactors: formfactor: TBP*⟩+≡

```
  procedure :: compute => formfactor_compute
```

⟨*ttv formfactors: procedures*⟩+≡

```
  function formfactor_compute (formfactor, ps, vec_type, FF_mode) result (FF)
    complex(default) :: FF
    class(formfactor_t), intent(in) :: formfactor
    type(phase_space_point_t), intent(in) :: ps
    integer, intent(in) :: vec_type, FF_mode
    real(default) :: f
```

```
    if (threshold%settings%initialized_parameters .and. formfactor%active) then
       select case (FF_mode)
       case (MATCHED, MATCHED_NOTSOHARD, RESUMMED, RESUMMED_SWITCHOFF)
          FF = resummed_formfactor (ps, vec_type) - one
       case (MATCHED_EXPANDED)
          f = f_switch_off (v_matching (ps%sqrts, GAM_M1S))
          FF = - expanded_formfactor (f * AS_HARD, f * AS_HARD, ps, vec_type) &
               + resummed_formfactor (ps, vec_type)
       case (MATCHED_EXPANDED_NOTSOHARD)
          f = f_switch_off (v_matching (ps%sqrts, GAM_M1S))
          FF = - expanded_formfactor (f * alphas_notsohard (ps%sqrts), f * &
               alphas_notsohard (ps%sqrts), ps, vec_type) &
               + resummed_formfactor (ps, vec_type)
       case (EXPANDED_HARD)
          FF = expanded_formfactor (AS_HARD, AS_HARD, ps, vec_type) - one
       case (EXPANDED_NOTSOHARD)
          FF = expanded_formfactor (alphas_notsohard (ps%sqrts), &
               alphas_notsohard (ps%sqrts), ps, vec_type) - one
       case (EXPANDED_SOFT)
          FF = expanded_formfactor (AS_HARD, alphas_soft (ps%sqrts), ps, &
               vec_type) - one
       case (EXPANDED_SOFT_SWITCHOFF)
          f = f_switch_off (v_matching (ps%sqrts, GAM_M1S))
          FF = expanded_formfactor (f * AS_HARD, &
               f * alphas_soft (ps%sqrts), ps, vec_type) - one
       case (RESUMMED_ANALYTIC_LL)
          FF = formfactor_LL_analytic (alphas_soft (ps%sqrts), ps%sqrts, &
               ps%p, vec_type) - one
       case (TREE)
          FF = zero
       case default
          FF = zero
       end select
    else
       FF = zero
    end if
    if (debug2_active (D_THRESHOLD)) then
       call update_global_sqrts_dependent_variables (ps%sqrts)
       call msg_debug2 (D_THRESHOLD, "threshold%settings%initialized_parameters", threshold%settin
       call msg_debug2 (D_THRESHOLD, "formfactor%active", formfactor%active)
       call msg_debug2 (D_THRESHOLD, "FF_mode", FF_mode)
       call msg_debug2 (D_THRESHOLD, "FF", FF)
       call msg_debug2 (D_THRESHOLD, "v", sqrts_to_v (ps%sqrts, GAM))
       call msg_debug2 (D_THRESHOLD, "vec_type", vec_type)
       call ps%write ()
    end if
  end function formfactor_compute
```

⟨*ttv formfactors: public*⟩+≡

```
  public :: width_t
```

⟨*ttv formfactors: types*⟩+≡

```
  type :: width_t
    real(default) :: aem
```

```
    real(default) :: sw
    real(default) :: mw
    real(default) :: mb
    real(default) :: vtb
    real(default) :: gam_inv
  contains
   ⟨ttv formfactors: width: TBP⟩
  end type width_t
```

⟨*ttv formfactors: width: TBP*⟩≡

```
  procedure :: init => width_init
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure subroutine width_init (width, aemi, sw, mw, mb, vtb, gam_inv)
    class(width_t), intent(inout) :: width
    real(default), intent(in) :: aemi, sw, mw, mb, vtb, gam_inv
    width%aem = one / aemi
    width%sw = sw
    width%mw = mw
    width%mb = mb
    width%vtb = vtb
    width%gam_inv = gam_inv
  end subroutine width_init
```

⟨*ttv formfactors: width: TBP*⟩+≡

```
  procedure :: compute => width_compute
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure function width_compute (width, top_mass, sqrts, initial) result (gamma)
    real(default) :: gamma
    class(width_t), intent(in) :: width
    real(default), intent(in) :: top_mass, sqrts
    logical, intent(in), optional :: initial
    real(default) :: alphas
    logical :: ini
    ini = .false.;  if (present (initial))  ini = initial
    if (ini) then
       alphas = AS_HARD
    else
       alphas = alphas_notsohard (sqrts)
    end if
    if (threshold%settings%nlo) then
       gamma = top_width_sm_qcd_nlo_jk (width%aem, width%sw, top_mass, &
            width%mw, width%mb, alphas) + width%gam_inv
    else
       gamma = top_width_sm_lo (width%aem, width%sw, width%vtb, top_mass, &
            width%mw, width%mb) + width%gam_inv
    end if
  end function width_compute
```

Use singleton pattern instead of global variables. At least shows where the variables are from.

⟨*ttv formfactors: public*⟩+≡

```
  public :: threshold
```

⟨*ttv formfactors: global variables*⟩≡
```
type(threshold_t) :: threshold
```

⟨*ttv formfactors: public*⟩+≡
```
public :: threshold_t
```

⟨*ttv formfactors: types*⟩+≡
```
type :: threshold_t
   type(settings_t) :: settings
   type(formfactor_t) :: formfactor
   type(width_t) :: width
contains
 ⟨ttv formfactors: threshold: TBP⟩
end type threshold_t
```

⟨*ttv formfactors: parameters*⟩≡
```
integer, parameter :: VECTOR = 1
integer, parameter :: AXIAL = 2
integer, parameter, public :: MATCHED_EXPANDED_NOTSOHARD = -5, &
                              MATCHED_NOTSOHARD = -4, &
                              MATCHED_EXPANDED = - 3, &
                              RESUMMED_SWITCHOFF = - 2, &
                              MATCHED = -1, &
                              RESUMMED = 1, &
                              EXPANDED_HARD = 4, &
                              EXPANDED_SOFT = 5, &
                              EXPANDED_SOFT_SWITCHOFF = 6, &
                              RESUMMED_ANALYTIC_LL = 7, &
                              EXPANDED_NOTSOHARD = 8, &
                              TREE = 9
real(default), parameter :: NF = 5.0_default

real(default), parameter :: z3 = 1.2020569031595942853997381615 1_default
real(default), parameter :: A1 = 31./9.*CA - 20./9.*TR*NF
real(default), parameter :: A2 = (4343./162. + 4.*pi**2 - pi**4/4. + &
     22./3.*z3)*CA**2 - (1798./81. + 56./3.*z3)*CA*TR*NF - &
     (55./3. - 16.*z3)*CF*TR*NF + (20./9.*TR*NF)**2
complex(default), parameter :: ieps = imago*tiny_10
```

`gam_m1s` is only used for the scale nustar

⟨*ttv formfactors: public*⟩+≡
```
public :: GAM, GAM_M1S
```

⟨*ttv formfactors: global variables*⟩+≡
```
real(default) :: M1S, GAM, GAM_M1S
integer :: NRQCD_ORDER
real(default) :: MTPOLE = - one
real(default) :: mtpole_init
real(default) :: RESCALE_H, MU_HARD, AS_HARD
real(default) :: AS_MZ, MASS_Z
real(default) :: MU_USOFT, AS_USOFT
```

`NUSTAR_FIXED` is normally not used

⟨*ttv formfactors: public*⟩+≡

```
public :: AS_SOFT
public :: AS_LL_SOFT
public :: AS_USOFT
public :: AS_HARD
public :: SWITCHOFF_RESUMMED
public :: TOPPIK_RESUMMED
```

⟨*ttv formfactors: global variables*⟩+≡

```
real(default) :: RESCALE_F, MU_SOFT, AS_SOFT, AS_LL_SOFT, NUSTAR_FIXED
logical :: NUSTAR_DYNAMIC, SWITCHOFF_RESUMMED, TOPPIK_RESUMMED
real(default) :: B0
real(default) :: B1

real(default), dimension(2) :: aa2, aa3, aa4, aa5, aa8, aa0
character(len=200) :: parameters_ref
type(nr_spline_t) :: ff_p_spline
real(default) :: v1, v2

integer :: POINTS_SQ, POINTS_P, POINTS_P0, n_q
real(default), dimension(:), allocatable :: sq_grid, p_grid, p0_grid, q_grid
complex(default), dimension(:,:,:,:), allocatable :: ff_grid
complex(single), dimension(:,:,:,:,:), allocatable :: Vmatrix
```

Explicit range and step size of the sqrts-grid relative to 2*M1S:

⟨*ttv formfactors: global variables*⟩+≡

```
real(default) :: sqrts_min, sqrts_max, sqrts_it
```

⟨*ttv formfactors: interfaces*⟩≡

```
interface char
  module procedure int_to_char, real_to_char, complex_to_char, logical_to_char
end interface char
```

⟨*ttv formfactors: public*⟩+≡

```
public :: m1s_to_mpole
```

⟨*ttv formfactors: types*⟩+≡

```
type, public :: phase_space_point_t
  real(default) :: p2 = 0, k2 = 0, q2 = 0
  real(default) :: sqrts = 0, p = 0, p0 = 0
  real(default) :: mpole = 0, en = 0
  logical :: inside_grid = .false., onshell = .false.
contains
⟨ttv formfactors: phase space point: TBP⟩
end type phase_space_point_t
```

⟨*ttv formfactors: phase space point: TBP*⟩≡

```
procedure :: init => phase_space_point_init_rel
```

⟨*ttv formfactors: procedures*⟩+≡

```
pure subroutine phase_space_point_init_rel (ps_point, p2, k2, q2, m)
  class(phase_space_point_t), intent(inout) :: ps_point
  real(default), intent(in) :: p2
  real(default), intent(in) :: k2
```

```
    real(default), intent(in) :: q2
    real(default), intent(in), optional :: m
    ps_point%p2 = p2
    ps_point%k2 = k2
    ps_point%q2 = q2
    call rel_to_nonrel (p2, k2, q2, ps_point%sqrts, ps_point%p, ps_point%p0)
    ps_point%mpole = m1s_to_mpole (ps_point%sqrts)
    ps_point%en = sqrts_to_en (ps_point%sqrts)
    ps_point%inside_grid = sqrts_within_range (ps_point%sqrts)
    if ( present(m) ) ps_point%onshell = ps_point%is_onshell (m)
  end subroutine phase_space_point_init_rel
```

⟨*ttv formfactors: phase space point: TBP*⟩+≡

```
  procedure :: init_nonrel => phase_space_point_init_nonrel
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure subroutine phase_space_point_init_nonrel (ps_point, sqrts, p, p0, m)
    class(phase_space_point_t), intent(inout) :: ps_point
    real(default), intent(in) :: sqrts
    real(default), intent(in) :: p
    real(default), intent(in) :: p0
    real(default), intent(in), optional :: m
    ps_point%sqrts = sqrts
    ps_point%p = p
    ps_point%p0 = p0
    call nonrel_to_rel (sqrts, p, p0, ps_point%p2, ps_point%k2, ps_point%q2)
    ps_point%mpole = m1s_to_mpole (sqrts)
    ps_point%en = sqrts_to_en (sqrts, ps_point%mpole)
    ps_point%inside_grid = sqrts_within_range (sqrts)
    if ( present(m) ) ps_point%onshell = ps_point%is_onshell (m)
  end subroutine phase_space_point_init_nonrel
```

⟨*ttv formfactors: procedures*⟩+≡

```
  !!! convert squared 4-momenta into sqrts, p0 = E_top-sqrts/2 and abs. 3-momentum p
  pure subroutine rel_to_nonrel (p2, k2, q2, sqrts, p, p0)
    real(default), intent(in) :: p2
    real(default), intent(in) :: k2
    real(default), intent(in) :: q2
    real(default), intent(out) :: sqrts
    real(default), intent(out) :: p
    real(default), intent(out) :: p0
    sqrts = sqrt(q2)
    p0 = abs(p2 - k2) / (2. * sqrts)
    p = sqrt (0.5_default * (- p2 - k2 + sqrts**2/2. + 2.* p0**2))
  end subroutine rel_to_nonrel
```

⟨*ttv formfactors: procedures*⟩+≡

```
  !!! convert sqrts, p0 = E_top-sqrts/2 and abs. 3-momentum p into squared 4-momenta
  pure subroutine nonrel_to_rel (sqrts, p, p0, p2, k2, q2)
    real(default), intent(in) :: sqrts
    real(default), intent(in) :: p
    real(default), intent(in) :: p0
    real(default), intent(out) :: p2
```

```
    real(default), intent(out) :: k2
    real(default), intent(out) :: q2
    p2 = (sqrts/2.+p0)**2 - p**2
    k2 = (sqrts/2.-p0)**2 - p**2
    q2 = sqrts**2
  end subroutine nonrel_to_rel
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure function complex_m2 (m, w) result (m2c)
    real(default), intent(in) :: m
    real(default), intent(in) :: w
    complex(default) :: m2c
    m2c = m**2 - imago*m*w
  end function complex_m2
```

⟨*ttv formfactors: phase space point: TBP*⟩+≡

```
  procedure :: is_onshell => phase_space_point_is_onshell
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure function phase_space_point_is_onshell (ps_point, m) result (flag)
    logical :: flag
    class(phase_space_point_t), intent(in) :: ps_point
    real(default), intent(in) :: m
    flag = nearly_equal (ps_point%p2 , m**2, rel_smallness=1E-5_default) .and. &
         nearly_equal (ps_point%k2 , m**2, rel_smallness=1E-5_default)
  end function phase_space_point_is_onshell
```

⟨*ttv formfactors: phase space point: TBP*⟩+≡

```
  procedure :: write => phase_space_point_write
```

⟨*ttv formfactors: procedures*⟩+≡

```
  subroutine phase_space_point_write (psp, unit)
    class(phase_space_point_t), intent(in) :: psp
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, '(A)') char ("p2 = " // str (psp%p2))
    write (u, '(A)') char ("k2 = " // str (psp%k2))
    write (u, '(A)') char ("q2 = " // str (psp%q2))
    write (u, '(A)') char ("sqrts = " // str (psp%sqrts))
    write (u, '(A)') char ("p = " // str (psp%p))
    write (u, '(A)') char ("p0 = " // str (psp%p0))
    write (u, '(A)') char ("mpole = " // str (psp%mpole))
    write (u, '(A)') char ("en = " // str (psp%en))
    write (u, '(A)') char ("inside_grid = " // str (psp%inside_grid))
    write (u, '(A)') char ("onshell = " // str (psp%onshell))
  end subroutine phase_space_point_write
```

⟨*ttv formfactors: types*⟩+≡

```
  type, extends (solver_function_t) :: p0_q_integrand_t
    real(default) :: a = 0
    type(phase_space_point_t) :: ps
    integer :: i = 0
```

```
  contains
    procedure :: update => p0_q_integrand_update
    procedure :: evaluate => p0_q_integrand_evaluate
  end type p0_q_integrand_t
```

⟨*ttv formfactors: procedures*⟩+≡

```
  function set_nrqcd_order (nrqcd_order_in) result (nrqcdorder)
    integer :: nrqcdorder
    real(default), intent(in) :: nrqcd_order_in
    nrqcdorder = 1
    if ( int(nrqcd_order_in) > nrqcdorder ) then
      call msg_warning ("reset to highest available NRQCD_ORDER = " // char(nrqcdorder))
    else
      nrqcdorder = int(nrqcd_order_in)
    end if
  end function set_nrqcd_order
```

⟨*ttv formfactors: public*⟩+≡

```
  public :: init_parameters
```

⟨*ttv formfactors: procedures*⟩+≡

```
  subroutine init_parameters (mpole_out, gam_out, m1s_in, Vtb, gam_inv, &
         aemi, sw, az, mz, mw, mb, h_in, f_in, nrqcd_order_in, ff_in, &
         offshell_strategy_in, v1_in, v2_in, scan_sqrts_min, &
         scan_sqrts_max, scan_sqrts_stepsize, mpole_fixed, top_helicity_selection)
    real(default), intent(out) :: mpole_out
    real(default), intent(out) :: gam_out
    real(default), intent(in) :: m1s_in
    real(default), intent(in) :: Vtb
    real(default), intent(in) :: gam_inv
    real(default), intent(in) :: aemi
    real(default), intent(in) :: sw
    real(default), intent(in) :: az
    real(default), intent(in) :: mz
    real(default), intent(in) :: mw
    real(default), intent(in) :: mb
    real(default), intent(in) :: h_in
    real(default), intent(in) :: f_in
    real(default), intent(in) :: nrqcd_order_in
    real(default), intent(in) :: ff_in
    real(default), intent(in) :: offshell_strategy_in
    real(default), intent(in) :: v1_in
    real(default), intent(in) :: v2_in
    real(default), intent(in) :: scan_sqrts_min
    real(default), intent(in) :: scan_sqrts_max
    real(default), intent(in) :: scan_sqrts_stepsize
    logical, intent(in) :: mpole_fixed
    real(default), intent(in) :: top_helicity_selection
    if (debug_active (D_THRESHOLD))  call show_input()
    threshold%settings%initialized_parameters = .false.
    M1S = m1s_in
    threshold%settings%mpole_dynamic = .not. mpole_fixed
    threshold%settings%offshell_strategy = int (offshell_strategy_in)
    call threshold%settings%setup_flags (int(ff_in), &
```

```
        threshold%settings%offshell_strategy, &
        int (top_helicity_selection))
    NRQCD_ORDER = set_nrqcd_order (nrqcd_order_in)
    v1 = v1_in
    v2 = v2_in
    sqrts_min = scan_sqrts_min
    sqrts_max = scan_sqrts_max
    sqrts_it = scan_sqrts_stepsize
    !!! global hard parameters incl. hard alphas used in all form factors
    RESCALE_H = h_in
    MU_HARD   = M1S * RESCALE_H
    AS_MZ     = az
    MASS_Z    = mz
    AS_HARD   = running_as (MU_HARD, az, mz, 2, NF)
    call threshold%width%init (aemi, sw, mw, mb, vtb, gam_inv)
    GAM_M1S = threshold%width%compute (M1S, zero, initial=.true.)
    call compute_global_auxiliary_numbers ()
    !!! soft parameters incl. mtpole
    !!! (depend on sqrts: initialize with sqrts ~ 2*M1S)
    NUSTAR_FIXED = - one
    NUSTAR_DYNAMIC = NUSTAR_FIXED  < zero
    RESCALE_F = f_in
    call update_global_sqrts_dependent_variables (2. * M1S)
    mtpole_init = MTPOLE
    mpole_out = mtpole_init
    gam_out = GAM
    threshold%settings%initialized_parameters = .true.
  contains
    ⟨ttv formfactors: init parameters: subroutines⟩
  end subroutine init_parameters
```

⟨*ttv formfactors: init parameters: subroutines*⟩≡

```
  subroutine show_input()
    call msg_debug (D_THRESHOLD, "init_parameters")
    call msg_debug (D_THRESHOLD, "m1s_in", m1s_in)
    call msg_debug (D_THRESHOLD, "Vtb", Vtb)
    call msg_debug (D_THRESHOLD, "gam_inv", gam_inv)
    call msg_debug (D_THRESHOLD, "aemi", aemi)
    call msg_debug (D_THRESHOLD, "sw", sw)
    call msg_debug (D_THRESHOLD, "az", az)
    call msg_debug (D_THRESHOLD, "mz", mz)
    call msg_debug (D_THRESHOLD, "mw", mw)
    call msg_debug (D_THRESHOLD, "mb", mb)
    call msg_debug (D_THRESHOLD, "h_in", h_in)
    call msg_debug (D_THRESHOLD, "f_in", f_in)
    call msg_debug (D_THRESHOLD, "nrqcd_order_in", nrqcd_order_in)
    call msg_debug (D_THRESHOLD, "ff_in", ff_in)
    call msg_debug (D_THRESHOLD, "offshell_strategy_in", offshell_strategy_in)
    call msg_debug (D_THRESHOLD, "top_helicity_selection", top_helicity_selection)
    call msg_debug (D_THRESHOLD, "v1_in", v1_in)
    call msg_debug (D_THRESHOLD, "v2_in", v2_in)
    call msg_debug (D_THRESHOLD, "scan_sqrts_min", scan_sqrts_min)
    call msg_debug (D_THRESHOLD, "scan_sqrts_max", scan_sqrts_max)
    call msg_debug (D_THRESHOLD, "scan_sqrts_stepsize", scan_sqrts_stepsize)
```

```
    call msg_debug (D_THRESHOLD, "AS_HARD", AS_HARD)
  end subroutine show_input
```

⟨*ttv formfactors: procedures*⟩+≡

```
  subroutine compute_global_auxiliary_numbers ()
    !!! auxiliary numbers needed later
    !!! current coefficients Ai(S,L,J), cf. arXiv:hep-ph/0609151, Eqs. (63)-(64)
    !!! 3S1 coefficients (s-wave, vector current)
    B0 = coeff_b0(NF) * (4.*pi)
    B1 = coeff_b1(NF) * (4.*pi)**2
    aa2(1) = (CF*(CA*CF*(9.*CA - 100.*CF) - &
              B0*(26.*CA**2 + 19.*CA*CF - 32.*CF**2)))/(26.*B0**2 *CA)
    aa3(1) = CF**2/( B0**2 *(6.*B0 - 13.*CA)*(B0 - 2.*CA)) * &
              (CA**2 *(9.*CA - 100.*CF) + B0*CA*(74.*CF - CA*16.) - &
              6.*B0**2 *(2.*CF - CA))
    aa4(1) = (24.*CF**2 * (11.*CA - 3.*B0)*(5.*CA + 8.*CF)) / &
              (13.*CA*(6.*B0 - 13.*CA)**2)
    aa5(1) =  (CF**2 * (CA*(15.-28) + B0*5.))/(6.*(B0-2.*CA)**2)
    aa8(1) = zero
    aa0(1) = -((8.*CF*(CA + CF)*(CA + 2.*CF))/(3.*B0**2))
    !!! 3P1 coefficients (p-wave, axial vector current)
    aa2(2) = -1./3. * (CF*(CA+2.*CF)/B0 - CF**2/(4.*B0) )
    aa3(2) =  zero
    aa4(2) =  zero
    aa5(2) =  1./3. * CF**2/(4.*(B0-2.*CA))
    aa8(2) = -1./3. * CF**2/(B0-CA)
    aa0(2) = -1./3. * 8.*CA*CF*(CA+4.*CF)/(3.*B0**2)
  end subroutine compute_global_auxiliary_numbers
```

⟨*ttv formfactors: public*⟩+≡

```
  public :: init_threshold_grids
```

⟨*ttv formfactors: procedures*⟩+≡

```
  subroutine init_threshold_grids (test)
    real(default), intent(in) :: test
    if (debug_active (D_THRESHOLD)) then
       call msg_debug (D_THRESHOLD, "init_threshold_grids")
       call msg_debug (D_THRESHOLD, "TOPPIK_RESUMMED", TOPPIK_RESUMMED)
    end if
    if (test > zero) then
      call msg_message ("TESTING ONLY: Skip threshold initialization and use tree-level SM.")
      return
    end if
    if (.not. threshold%settings%initialized_parameters) call msg_fatal ("init_threshold_grid: par
    !!! !!! !!! MAC OS X and BSD don't load the global module with parameter values stored
    !!! if (parameters_ref == parameters_string ()) return
    call dealloc_grids ()
    if (TOPPIK_RESUMMED) call init_formfactor_grid ()
    parameters_ref = parameters_string ()
  end subroutine init_threshold_grids
```

⟨*ttv formfactors: procedures*⟩+≡

```
  !!! LL/NLL resummation of nonrelativistic Coulomb potential
```

```
pure function resummed_formfactor (ps, vec_type) result (c)
  type(phase_space_point_t), intent(in) :: ps
  integer, intent(in) :: vec_type
  complex(default) :: c
  c = one
  if (.not. threshold%settings%initialized_ff .or. .not. ps%inside_grid) return
  call interpolate_linear (sq_grid, p_grid, ff_grid(:,:,1,vec_type), ps%sqrts, ps%p, c)
end function resummed_formfactor
```

⟨*ttv formfactors: procedures*⟩+≡

```
!!! leading nonrelativistic O(alphas^1) contribution (-> expansion of resummation)
function expanded_formfactor (alphas_hard, alphas_soft, ps, vec_type) result (FF)
  complex(default) :: FF
  real(default), intent(in) :: alphas_hard, alphas_soft
  type(phase_space_point_t), intent(in) :: ps
  integer, intent(in) :: vec_type
  real(default) :: shift_from_hard_current
  complex(default) :: v, contrib_from_potential
  FF = one
  if (.not. threshold%settings%initialized_parameters .or. vec_type == AXIAL) return
  call update_global_sqrts_dependent_variables (ps%sqrts)
  v = sqrts_to_v (ps%sqrts, GAM)
  if (NRQCD_ORDER == 1) then
     shift_from_hard_current = - two * CF / pi
  else
     shift_from_hard_current = zero
  end if
  if (ps%onshell) then
     contrib_from_potential = CF * ps%mpole * Pi / (4 * ps%p)
  else
     contrib_from_potential = imago * CF * ps%mpole * &
          log ((ps%p + ps%mpole * v) / &
               (-ps%p + ps%mpole * v) + ieps) / (two * ps%p)
  end if
  FF = one + alphas_soft * contrib_from_potential + &
       alphas_hard * shift_from_hard_current
end function expanded_formfactor
```

⟨*ttv formfactors: procedures*⟩+≡

```
subroutine init_formfactor_grid ()
  type(string_t) :: ff_file
  call msg_debug (D_THRESHOLD, "init_formfactor_grid")
  threshold%settings%initialized_ff = .false.
  ff_file = "SM_tt_threshold.grid"
  call msg_message ()
  call msg_message ("%%%
```

```
      call msg_message (" by M. Jezabek, T. Teubner.")
    end if
    call msg_message ("%%%
```

```
    if (st > 0)  call msg_fatal ("close " // char(ff_file) // ": iostat = " // char(st))
    threshold%settings%initialized_ps = .true.
    threshold%settings%initialized_ff = .true.
  end subroutine read_formfactor_grid
```

⟨*ttv formfactors: procedures*⟩+≡

```
  subroutine write_formfactor_grid (ff_file)
    type(string_t), intent(in) :: ff_file
    integer :: u, st
    if (.not. threshold%settings%initialized_ff) then
      call msg_warning ("write_formfactor_grid: no grids initialized!")
      return
    end if
    u = free_unit ()
    open (unit=u, status='replace', file=char(ff_file), form='unformatted', iostat=st)
    if (st /= 0)  call msg_fatal ("open " // char(ff_file) // ": iostat = " // char(st))
    write (u) parameters_string ()
    write (u) shape(ff_grid)
    write (u) sq_grid
    write (u) p_grid
    write (u) cmplx(ff_grid, kind=single)
    close (u, iostat=st)
    if (st > 0)  call msg_fatal ("close " // char(ff_file) // ": iostat = " // char(st))
  end subroutine write_formfactor_grid
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure function parameters_string () result (str)
    character(len(parameters_ref)) :: str
    str = char(M1S) // " " // char(GAM_M1S) // " " // char(NRQCD_ORDER) &
           // " " // char(RESCALE_H) &
           // " " // char(RESCALE_F) &
           //  " " // char(sqrts_min) &
           // " " // char(sqrts_max) // " " // char(sqrts_it)
  end function parameters_string
```

⟨*ttv formfactors: procedures*⟩+≡

```
  subroutine update_global_sqrts_dependent_variables (sqrts)
    real(default), intent(in) :: sqrts
    real(default) :: nu_soft, f
    logical :: only_once_for_fixed_nu, already_done
    real(default), save :: last_sqrts = - one
    call msg_debug (D_THRESHOLD, "update_global_sqrts_dependent_variables")
    call msg_debug (D_THRESHOLD, "sqrts", sqrts)
    call msg_debug (D_THRESHOLD, "last_sqrts", last_sqrts)
    already_done = threshold%settings%initialized_parameters .and. &
         nearly_equal (sqrts, last_sqrts, rel_smallness=1E-6_default)
    call msg_debug (D_THRESHOLD, "already_done", already_done)
    only_once_for_fixed_nu = .not. NUSTAR_DYNAMIC .and. MTPOLE > zero
    call msg_debug (D_THRESHOLD, "only_once_for_fixed_nu", only_once_for_fixed_nu)
    if (only_once_for_fixed_nu .or. already_done) return
    last_sqrts = sqrts
    nu_soft = RESCALE_F * nustar (sqrts)
    MU_SOFT = M1S * RESCALE_H * nu_soft
```

```
    MU_USOFT = M1S * RESCALE_H * nu_soft**2
    AS_SOFT = running_as (MU_SOFT, AS_HARD, MU_HARD, NRQCD_ORDER, NF)
    AS_LL_SOFT = running_as (MU_SOFT, AS_HARD, MU_HARD, 0, NF)
    AS_USOFT = running_as (MU_USOFT, AS_HARD, MU_HARD, 0, NF) !!! LL here
    if (SWITCHOFF_RESUMMED) then
       f = f_switch_off (v_matching (sqrts, GAM_M1S))
       AS_SOFT = AS_SOFT * f
       AS_LL_SOFT = AS_LL_SOFT * f
       AS_USOFT = AS_USOFT * f
    end if
    MTPOLE = m1s_to_mpole (sqrts)
    GAM = threshold%width%compute (MTPOLE, sqrts)
    call msg_debug (D_THRESHOLD, "GAM", GAM)
    call msg_debug (D_THRESHOLD, "nu_soft", nu_soft)
    call msg_debug (D_THRESHOLD, "MTPOLE", MTPOLE)
    call msg_debug (D_THRESHOLD, "AS_SOFT", AS_SOFT)
    call msg_debug (D_THRESHOLD, "AS_LL_SOFT", AS_LL_SOFT)
    call msg_debug (D_THRESHOLD, "AS_USOFT", AS_USOFT)
  end subroutine update_global_sqrts_dependent_variables

  !!! Coulomb potential coefficients needed by TOPPIK
  pure function xc (a_soft, i_xc) result (xci)
    real(default), intent(in) :: a_soft
    integer, intent(in) :: i_xc
    real(default) :: xci
    xci = zero
    select case (i_xc)
      case (0)
        xci = one
        if ( NRQCD_ORDER>0 ) xci = xci + a_soft/(4.*pi) * A1
        if ( NRQCD_ORDER>1 ) xci = xci + (a_soft/(4.*pi))**2 * A2
      case (1)
        if ( NRQCD_ORDER>0 ) xci = xci + a_soft/(4.*pi) * B0
        if ( NRQCD_ORDER>1 ) xci = xci + (a_soft/(4.*pi))**2 * (B1 + 2*B0*A1)
      case (2)
        if ( NRQCD_ORDER>1 ) xci = xci + (a_soft/(4.*pi))**2 * B0**2
      case default
        return
    end select
  end function xc
```

⟨*ttv formfactors: procedures*⟩+≡

```
  function current_coeff (a_hard, a_soft, a_usoft, i) result (coeff)
    real(default), intent(in) :: a_hard, a_soft, a_usoft
    integer, intent(in) :: i
    real(default) :: coeff
    real(default) :: matching_c, c1
    real(default) :: z, w
    call msg_debug (D_THRESHOLD, "current_coeff")
    coeff = one
    if (NRQCD_ORDER == 0) return
    z = a_soft / a_hard
    w = a_usoft / a_soft
    !!! hard s/p-wave 1-loop matching coefficients, cf. arXiv:hep-ph/0604072
```

```
    select case (i)
      case (1)
        matching_c = one - 2.*(CF/pi) * a_hard
      case (2)
        matching_c = one -    (CF/pi) * a_hard
     case default
        call msg_fatal ("current_coeff: unknown coeff i = " // char(i))
    end select
    !!! current coefficient c1, cf. arXiv:hep-ph/0609151, Eq. (62)
    c1 = exp( a_hard * pi * ( aa2(i)*(1.-z) + aa3(i)*log(z) + &
         aa4(i)*(1.-z**(1.-13.*CA/(6.*B0))) + aa5(i)*(1.-z**(1.-2.*CA/B0)) + &
         aa8(i)*(1.-z**(1.-CA/B0)) + aa0(i)*(z-1.-log(w)/w) ))
    coeff = matching_c * c1
  end function current_coeff
```

⟨*ttv formfactors: public*⟩+≡

```
  public :: v_matching
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure function v_matching (sqrts, gamma) result (v)
    real(default) :: v
    real(default), intent(in) :: sqrts, gamma
    v = abs (sqrts_to_v_1S (sqrts, gamma))
  end function v_matching
```

Smooth transition from `f1` to `f2` between `v1` and `v2` (simplest polynom).

⟨*ttv formfactors: public*⟩+≡

```
  public :: f_switch_off
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure function f_switch_off (v) result (fval)
    real(default), intent(in) :: v
    real(default) :: fval
    real(default) :: vm, f1, f2, x
    f1 = one
    f2 = zero + tiny_10
    vm = (v1+v2) / 2.
    if ( v < v1 ) then
      fval = f1
    else if (v < v2) then
       x = (v - v1) / (v2 - v1)
       fval = 1 - x**2 * (3 - 2 * x)
    else
      fval = f2
    end if
  end function f_switch_off
```

⟨*ttv formfactors: procedures*⟩+≡

```
  function formfactor_LL_analytic (a_soft, sqrts, p, vec_type) result (c)
    real(default), intent(in) :: a_soft
    real(default), intent(in) :: sqrts
    real(default), intent(in) :: p
    integer, intent(in) :: vec_type
```

```
    complex(default) :: c
    real(default) :: en
    c = one
    if (.not. threshold%settings%initialized_parameters) return
    call update_global_sqrts_dependent_variables (sqrts)
    en = sqrts_to_en (sqrts, MTPOLE)
    select case (vec_type)
      case (1)
        c = G0p (CF*a_soft, en, p, MTPOLE, GAM) / G0p_tree (en, p, MTPOLE, GAM)
      case (2)
        c = G0p_ax (CF*a_soft, en, p, MTPOLE, GAM) / G0p_tree (en, p, MTPOLE, GAM)
      case default
        call msg_fatal ("unknown ttZ/ttA vertex component, vec_type = " // char(vec_type))
    end select
  end function formfactor_LL_analytic
```

⟨*ttv formfactors: procedures*⟩+≡

```
  !!! Max's LL nonrelativistic threshold Green's function
  function G0p (a, en, p, m, w) result (c)
    real(default), intent(in) :: a
    real(default), intent(in) :: en
    real(default), intent(in) :: p
    real(default), intent(in) :: m
    real(default), intent(in) :: w
    complex(default) :: c
    complex(default) :: k, ipk, la, z1, z2
    complex(default) :: one, two, cc, dd
    k   = sqrt( -m*en -imago*m*w )
    ipk = imago * p / k
    la  = a * m / 2. / k
    one = cmplx (1., kind=default)
    two = cmplx (2., kind=default)
    cc  = 2. - la
    dd  = ( 1. + ipk ) / 2.
    z1  = nr_hypgeo (two, one, cc, dd)
    dd  = ( 1. - ipk ) / 2.
    z2  = nr_hypgeo (two, one, cc, dd)
    c   = - imago * m / (4.*p*k) / (1.-la) * ( z1 - z2 )
  end function G0p
```

⟨*ttv formfactors: procedures*⟩+≡

```
  !!! tree level version: a_soft -> 0
  pure function G0p_tree (en, p, m, w) result (c)
    real(default), intent(in) :: en
    real(default), intent(in) :: p
    real(default), intent(in) :: m
    real(default), intent(in) :: w
    complex(default) :: c
    c = m / (p**2 - m*(en+imago*w))
  end function G0p_tree
```

⟨*ttv formfactors: procedures*⟩+≡

```
  !!! Peter Poier's LL nonrelativistic axial threshold Green's function
```

```
function G0p_ax (a, en, p, m, w) result (c)
  real(default), intent(in) :: a
  real(default), intent(in) :: en
  real(default), intent(in) :: p
  real(default), intent(in) :: m
  real(default), intent(in) :: w
  complex(default) :: c
  complex(default) :: k, ipk, la, z1, z2, z3, z4
  complex(default) :: zero, two, three, cc, ddp, ddm
  k   = sqrt( -m*en -imago*m*w )
  ipk = imago * p / k
  la  = a * m / 2. / k
  zero = cmplx (0., kind=default)
  two = cmplx (2., kind=default)
  three = cmplx (3., kind=default)
  cc  = 1. - la
  ddp = ( 1. + ipk ) / 2.
  ddm = ( 1. - ipk ) / 2.
  z1  = nr_hypgeo (zero, two, cc, ddp)
  z2  = nr_hypgeo (zero, two, cc, ddm)
  cc  = 2. - la
  z3  = nr_hypgeo (zero, three, cc, ddm)
  z4  = nr_hypgeo (zero, three, cc, ddp)
  c   = m / 2. / p**3 * ( 2.*p + imago*k*(1.-la)*(z1-z2) + imago*k*(z3-z4) )
end function G0p_ax
```

⟨*ttv formfactors: procedures*⟩+≡

```
pure function nustar (sqrts) result (nu)
  real(default), intent(in) :: sqrts
  real(default) :: nu
  real(default), parameter :: nustar_offset = 0.05_default
  complex(default) :: arg
  if (NUSTAR_DYNAMIC) then
    !!! from [arXiv:1309.6323], Eq. (3.2) (other definitions possible)
    arg = ( sqrts - 2.*M1S + imago*GAM_M1S ) / M1S
    nu  = nustar_offset + abs(sqrt(arg))
  else
    nu  = NUSTAR_FIXED
  end if
end function nustar
```

We recompute `alpha_soft` for form factors that do not call `update_global_parameters` (it is called in the scan for the (N)LL grid).

⟨*ttv formfactors: procedures*⟩+≡

```
pure function alphas_soft (sqrts) result (a_soft)
  real(default) :: a_soft
  real(default), intent(in) :: sqrts
  real(default) :: mu_soft, nusoft
  nusoft = RESCALE_F * nustar (sqrts)
  mu_soft = RESCALE_H * M1S * nusoft
  a_soft = running_as (mu_soft, AS_HARD, MU_HARD, NRQCD_ORDER, NF)
end function alphas_soft
```

⟨*ttv formfactors: public*⟩+≡

```
public :: alphas_notsohard
```

⟨*ttv formfactors: procedures*⟩+≡

```
pure function alphas_notsohard (sqrts) result (a_soft)
  real(default) :: a_soft
  real(default), intent(in) :: sqrts
  real(default) :: mu_notsohard
  ! complex(default) :: v
  ! v = sqrts_to_v_1S (sqrts, GAM_M1S)
  ! mu_notsohard = RESCALE_H * M1S * sqrt(abs(v))
  mu_notsohard = RESCALE_H * M1S * sqrt(nustar (sqrts))
  a_soft = running_as (mu_notsohard, AS_MZ, MASS_Z, 2, NF)
end function alphas_notsohard
```

⟨*ttv formfactors: procedures*⟩+≡

```
pure function m1s_to_mpole (sqrts) result (mpole)
  real(default), intent(in) :: sqrts
  real(default) :: mpole
  mpole = mtpole_init
  if (threshold%settings%mpole_dynamic) then
     mpole = M1S * ( 1. + deltaM(sqrts) )
  else
     mpole = M1S
  end if
end function m1s_to_mpole
```

⟨*ttv formfactors: procedures*⟩+≡

```
!pure
!function mpole_to_M1S (mpole, sqrts, nl) result (m)
  !real(default), intent(in) :: mpole
  !real(default), intent(in) :: sqrts
  !integer, intent(in) :: nl
  !real(default) :: m
  !m = mpole * ( 1. - deltaM(sqrts, nl) )
!end function mpole_to_M1S
```

⟨*ttv formfactors: procedures*⟩+≡

```
pure function deltaM (sqrts) result (del)
  real(default), intent(in) :: sqrts
  real(default) :: del
  real(default) :: ac
  ac  = CF * alphas_soft (sqrts)
  del = ac**2 / 8.
  if (NRQCD_ORDER > 0) then
    del = del + ac**3 / (8. * pi * CF) * &
          (B0 * (log (RESCALE_H * RESCALE_F * nustar (sqrts) / ac) + one) + A1 / 2.)
  end if
end function deltaM
```

⟨*ttv formfactors: procedures*⟩+≡

```
pure function sqrts_within_range (sqrts) result (flag)
```

```
    real(default), intent(in) :: sqrts
    logical :: flag
    flag = ( sqrts >= sqrts_min - tiny_07 .and. sqrts <= sqrts_max + tiny_07 )
  end function
```

⟨*ttv formfactors: procedures*⟩+≡

```
  ! The mapping is such that even for min=max, we get three points:
  ! min - it , min, min + it
  pure function sqrts_iter (i_sq) result (sqrts)
    integer, intent(in) :: i_sq
    real(default) :: sqrts
    sqrts = sqrts_min - sqrts_it + &
            (sqrts_max - sqrts_min + two * sqrts_it) * &
            real(i_sq - 1) / real(POINTS_SQ - 1)
  end function sqrts_iter
```

⟨*ttv formfactors: procedures*⟩+≡

```
  function scan_formfactor_over_p_LL_analytic (a_soft, sqrts, vec_type) result (ff_analytic)
    real(default), intent(in) :: a_soft
    real(default), intent(in) :: sqrts
    integer, intent(in) :: vec_type
    complex(default), dimension(POINTS_P) :: ff_analytic
    integer :: i_p
    ff_analytic = [(formfactor_LL_analytic (a_soft, sqrts, p_grid(i_p), vec_type), i_p=1, POINTS_P
  end function scan_formfactor_over_p_LL_analytic
```

⟨*ttv formfactors: procedures*⟩+≡

```
  !!! tttoppik wrapper
  subroutine scan_formfactor_over_p_TOPPIK (a_soft, sqrts, vec_type, p_grid_out, mpole_in, ff_topp
    real(default), intent(in) :: a_soft
    real(default), intent(in) :: sqrts
    integer, intent(in) :: vec_type
    real(default), dimension(POINTS_P), intent(out), optional :: p_grid_out
    real(default), intent(in), optional :: mpole_in
    complex(default), dimension(POINTS_P), optional :: ff_toppik
    integer :: i_p
    real(default) :: mpole, alphas_hard, f
    real(default), dimension(POINTS_P) :: p_toppik
    type(nr_spline_t) :: toppik_spline
    real*8 :: xenergy, xtm, xtg, xalphas, xscale, xc0, xc1, xc2, xim, xdi, &
        xcutn, xcutv, xkincm, xkinca, xkincv, xcdeltc, &
        xcdeltl, xcfullc, xcfulll, xcrm2
    integer, parameter :: nmax=400
    real*8 :: xdsdp(nmax), xpp(nmax), xww(nmax)
    complex*16 :: zff(nmax)
    integer :: np, jknflg, jgcflg, jvflg
    call msg_debug (D_THRESHOLD, "scan_formfactor_over_p_TOPPIK")
    if (POINTS_P > nmax-40) call msg_fatal ("TOPPIK: POINTS_P must be <=" // char(nmax-40))
    call msg_debug (D_THRESHOLD, "POINTS_P", POINTS_P)
    if (present (ff_toppik))  ff_toppik = zero
    mpole = MTPOLE;  if (present (mpole_in)) mpole = mpole_in
    xenergy = sqrts_to_en (sqrts, MTPOLE)
    xtm     = mpole
```

```
xtg     = GAM
xalphas = a_soft
xscale  = MU_SOFT
xcutn   = 175.E6
xcutv   = 175.E6
xc0     = xc (a_soft, 0)
xc1     = xc (a_soft, 1)
xc2     = xc (a_soft, 2)
xcdeltc = 0.
xcdeltl = 0.
xcfullc = 0.
xcfulll = 0.
xcrm2   = 0.
xkincm  = 0.
xkinca  = 0.
jknflg  = 0
jgcflg  = 0
xkincv  = 0.
jvflg   = 0
select case (vec_type)
  case (VECTOR)
     call msg_debug (D_THRESHOLD, "calling tttoppik")
     call tttoppik &
            (xenergy,xtm,xtg,xalphas,xscale,xcutn,xcutv,xc0,xc1,xc2, &
             xcdeltc,xcdeltl,xcfullc,xcfulll,xcrm2,xkincm,xkinca,jknflg, &
             jgcflg, xkincv,jvflg,xim,xdi,np,xpp,xww,xdsdp,zff)
  case (AXIAL)
     call msg_debug (D_THRESHOLD, "calling tttoppikaxial")
     call tttoppikaxial &
            (xenergy,xtm,xtg,xalphas,xscale,xcutn,xcutv,xc0,xc1,xc2, &
             xcdeltc,xcdeltl,xcfullc,xcfulll,xcrm2,xkincm,xkinca,jknflg, &
             jgcflg, xkincv,jvflg,xim,xdi,np,xpp,xww,xdsdp,zff)
     !!! 1st ~10 TOPPIK p-wave entries are ff_unstable: discard them
     zff(1:10) = [(zff(11), i_p=1, 10)]
  case default
     call msg_fatal ("unknown ttZ/ttA vertex component, vec_type = " // char(vec_type))
end select
if (present (p_grid_out)) p_grid_out = xpp(1:POINTS_P)
if (.not. present (ff_toppik)) return
!!! keep track of TOPPIK instabilities and try to repair later
if (np < 0) then
  ff_toppik(1) = 2.d30
  if (debug_active (D_THRESHOLD)) then
     call msg_warning ("caught TOPPIK instability at sqrts = " // char(sqrts))
  end if
  return
end if
p_toppik = xpp(1:POINTS_P)
ff_toppik = zff(1:POINTS_P)
!!! TOPPIK output p-grid scales with en above ~ 4 GeV:
!!! interpolate for global sqrts/p grid
if (.not. nearly_equal (p_toppik(42), p_grid(42), rel_smallness=1E-6_default)) then
  call toppik_spline%init (p_toppik, ff_toppik)
  ff_toppik(2:POINTS_P) = [(toppik_spline%interpolate (p_grid(i_p)), i_p=2, POINTS_P)]
```

```
      call toppik_spline%dealloc ()
    end if
    !!! TOPPIK output includes tree level ~ 1, a_soft @ LL in current coefficient!
    if (SWITCHOFF_RESUMMED) then
       f = f_switch_off (v_matching (sqrts, GAM_M1S))
       alphas_hard = AS_HARD * f
    else
       alphas_hard = AS_HARD
    end if
    ff_toppik = ff_toppik * current_coeff (alphas_hard, AS_LL_SOFT, AS_USOFT, vec_type)
    call msg_debug (D_THRESHOLD, &
         "current_coeff (alphas_hard, AS_LL_SOFT, AS_USOFT, vec_type)", &
         current_coeff (alphas_hard, AS_LL_SOFT, AS_USOFT, vec_type))
  end subroutine scan_formfactor_over_p_TOPPIK
```

⟨*ttv formfactors: procedures*⟩+≡

```
  function scan_formfactor_over_p (sqrts, vec_type) result (ff)
    real(default), intent(in) :: sqrts
    integer, intent(in) :: vec_type
    complex(default), dimension(POINTS_P) :: ff
    call msg_debug (D_THRESHOLD, "scan_formfactor_over_p")
    select case (NRQCD_ORDER)
      case (0)
       ! ff = scan_formfactor_over_p_LL_analytic (AS_SOFT, sqrts, vec_type)
        call scan_formfactor_over_p_TOPPIK (AS_SOFT, sqrts, vec_type, ff_toppik=ff)
      case (1)
        call scan_formfactor_over_p_TOPPIK (AS_SOFT, sqrts, vec_type, ff_toppik=ff)
      case default
        call msg_fatal ("NRQCD_ORDER = " // char(NRQCD_ORDER))
    end select
  end function scan_formfactor_over_p
```

⟨*ttv formfactors: procedures*⟩+≡

```
  subroutine scan_formfactor_over_phase_space_grid ()
    integer :: i_sq, vec_type, unstable_loop
    logical, dimension(:,:), allocatable :: ff_unstable
    real(default) :: t1, t2, t3, t_toppik, t_p0_dep
    call msg_debug (D_THRESHOLD, "scan_formfactor_over_phase_space_grid")
    allocate (ff_grid(POINTS_SQ,POINTS_P,POINTS_P0,2))
    allocate (ff_unstable(POINTS_SQ,2))
    t_toppik = zero
    t_p0_dep = zero
    write (msg_buffer, "(3(A,F7.3,1X),A)") "Scanning from ", &
         sqrts_min - sqrts_it, "GeV to ", &
         sqrts_max + sqrts_it, "GeV in steps of ", sqrts_it, "GeV"
    call msg_message ()
    ENERGY_SCAN: do i_sq = 1, POINTS_SQ
      if (signal_is_pending ())  return
      call update_global_sqrts_dependent_variables (sq_grid(i_sq))
      !!! vector and axial vector
      do vec_type = VECTOR, AXIAL
        call cpu_time (t1)
        unstable_loop = 0
```

```
        UNTIL_STABLE: do
           ff_grid(i_sq,:,1,vec_type) = scan_formfactor_over_p (sq_grid(i_sq), vec_type)
           ff_unstable(i_sq,vec_type) = abs(ff_grid(i_sq,1,1,vec_type)) > 1.d30
           unstable_loop = unstable_loop + 1
           if (ff_unstable(i_sq,vec_type) .and. unstable_loop < 10) then
              cycle
           else
              exit
           end if
        end do UNTIL_STABLE
        call cpu_time (t2)
        !!!  include p0 dependence by an integration over the p0-independent FF
        call cpu_time (t3)
        t_toppik = t_toppik + t2 - t1
        t_p0_dep = t_p0_dep + t3 - t2
      end do
      call msg_show_progress (i_sq, POINTS_SQ)
    end do ENERGY_SCAN
    if (debug_active (D_THRESHOLD)) then
       print *, "time for TOPPIK call:   ", t2 - t1, " seconds."
       print *, "time for p0 dependence: ", t3 - t2, " seconds."
    end if
    if (any (ff_unstable))  call handle_TOPPIK_instabilities (ff_grid, ff_unstable)
    if (allocated(Vmatrix))  deallocate(Vmatrix)
    if (allocated(q_grid))  deallocate(q_grid)
    threshold%settings%initialized_ff = .true.
  end subroutine scan_formfactor_over_phase_space_grid
```

⟨*ttv formfactors: procedures*⟩+≡

```
  subroutine init_threshold_phase_space_grid ()
    integer :: i_sq
    call msg_debug (D_THRESHOLD, "init_threshold_phase_space_grid")
    POINTS_SQ = int ((sqrts_max - sqrts_min) / sqrts_it + tiny_07) + 3
    call msg_debug (D_THRESHOLD, "Number of sqrts grid points: POINTS_SQ", POINTS_SQ)
    call msg_debug (D_THRESHOLD, "sqrts_max", sqrts_max)
    call msg_debug (D_THRESHOLD, "sqrts_min", sqrts_min)
    call msg_debug (D_THRESHOLD, "sqrts_it", sqrts_it)
    allocate (sq_grid(POINTS_SQ))
    sq_grid = [(sqrts_iter (i_sq), i_sq=1, POINTS_SQ)]
    POINTS_P = 360
    allocate (p_grid(POINTS_P))
    p_grid = p_grid_from_TOPPIK ()
    POINTS_P0 = 1
    threshold%settings%initialized_ps = .true.
  end subroutine init_threshold_phase_space_grid
```

⟨*ttv formfactors: procedures*⟩+≡

```
  subroutine init_p0_grid (p_in, n)
    real(default), dimension(:), allocatable, intent(in) :: p_in
    integer, intent(in) :: n
    call msg_debug (D_THRESHOLD, "init_p0_grid")
    call msg_debug (D_THRESHOLD, "n", n)
    call msg_debug (D_THRESHOLD, "size(p_in)", size(p_in))
```

```
    if (.not. allocated (p_in))  call msg_fatal ("init_p0_grid: p_in not allocated!")
    if (allocated (p0_grid))  deallocate (p0_grid)
    allocate (p0_grid(n))
    p0_grid(1) = zero
    p0_grid(2:n) = p_in(1:n-1)
  end subroutine init_p0_grid
```

⟨*ttv formfactors: procedures*⟩+≡

```
  !!! Andre's procedure to refine an existing grid
  pure subroutine finer_grid (gr, fgr, n_in)
    real(default), dimension(:), intent(in) :: gr
    real(default), dimension(:), allocatable, intent(inout) :: fgr
    integer, intent(in), optional :: n_in
    integer :: n, i, j
    real(default), dimension(:), allocatable :: igr
    n = 4
    if ( present(n_in) ) n = n_in
    allocate( igr(n) )
    if ( allocated(fgr) ) deallocate( fgr )
    allocate( fgr(n*(size(gr)-1)+1) )
    do i=1, size(gr)-1
      do j=0, n-1
        igr(j+1) = gr(i) + real(j)*(gr(i+1)-gr(i))/real(n)
      end do
      fgr((i-1)*n+1:i*n) = igr
    end do
    fgr(size(fgr)) = gr(size(gr))
    deallocate( igr )
  end subroutine finer_grid
```

⟨*ttv formfactors: procedures*⟩+≡

```
  subroutine dealloc_grids ()
    if ( allocated(sq_grid) ) deallocate( sq_grid )
    if ( allocated( p_grid) ) deallocate(  p_grid )
    if ( allocated(p0_grid) ) deallocate( p0_grid )
    if ( allocated(ff_grid) ) deallocate( ff_grid )
    threshold%settings%initialized_ps = .false.
    threshold%settings%initialized_ff = .false.
  end subroutine dealloc_grids
```

⟨*ttv formfactors: procedures*⟩+≡

```
  subroutine trim_p_grid (n_p_new)
    integer, intent(in) :: n_p_new
    real(default), dimension(n_p_new) :: p_save
    complex(default), dimension(POINTS_SQ,n_p_new,POINTS_P0,2) :: ff_save
    if (n_p_new > POINTS_P) then
      call msg_fatal ("trim_p_grid: new size larger than old size.")
      return
    end if
    p_save = p_grid(1:n_p_new)
    ff_save = ff_grid(:,1:n_p_new,:,:)
    deallocate( p_grid, ff_grid )
    allocate( p_grid(n_p_new), ff_grid(POINTS_SQ,n_p_new,POINTS_P0,2) )
```

```
    p_grid = p_save
    ff_grid = ff_save
  end subroutine trim_p_grid
```

⟨*ttv formfactors: procedures*⟩+≡

```
  !!! try to repair TOPPIK instabilities by interpolation of adjacent sq_grid points
  subroutine handle_TOPPIK_instabilities (ff, nan)
    complex(default), dimension(:,:,:,:), intent(inout) :: ff
    logical, dimension(:,:), intent(in) :: nan
    integer :: i, i_sq, n_nan
    logical :: interrupt
    n_nan = sum (merge ([(1, i=1, 2*POINTS_SQ)], &
         [(0, i=1, 2*POINTS_SQ)], reshape (nan, [2*POINTS_SQ])) )
    interrupt = n_nan > 3
    do i = 1, 2
      if (interrupt ) exit
      if (.not. any (nan(:,i))) cycle
      do i_sq = 2, POINTS_SQ - 1
        if (.not. nan(i_sq,i)) cycle
        if (nan(i_sq+1,i) .or. nan(i_sq-1,i)) then
          interrupt = .true.
          exit
        end if
        ff(i_sq,:,:,i) = (ff(i_sq-1,:,:,i) + ff(i_sq+1,:,:,i)) / two
      end do
    end do
    if (.not. interrupt) return
    call msg_fatal ("Too many TOPPIK instabilities! Check your parameter setup " &
                    // "or slightly vary the scales sh and/or sf.")
  end subroutine handle_TOPPIK_instabilities
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure function sqrts_to_v (sqrts, gamma) result (v)
    complex(default) :: v
    real(default), intent(in) :: sqrts, gamma
    real(default) :: m
    m = m1s_to_mpole (sqrts)
    v = sqrt ((sqrts - two * m + imago * gamma) / m)
  end function sqrts_to_v
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure function sqrts_to_v_1S (sqrts, gamma) result (v)
    complex(default) :: v
    real(default), intent(in) :: sqrts, gamma
    v = sqrt ((sqrts - two * M1S + imago * gamma) / M1S)
  end function sqrts_to_v_1S
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure function v_to_sqrts (v) result (sqrts)
    real(default), intent(in) :: v
    real(default) :: sqrts
    real(default) :: m
```

```
    m = mtpole_init
    sqrts = 2.*m + m*v**2
  end function v_to_sqrts
```

⟨*ttv formfactors: procedures*⟩+≡

```
  !!! -q^2 times the Coulomb potential V at LO resp. NLO
  function minus_q2_V (a, q, p, p0r, vec_type) result (v)
    real(default), intent(in) :: a
    real(default), intent(in) :: q
    real(default), intent(in) :: p
    real(default), intent(in) :: p0r
    integer, intent(in) :: vec_type
    complex(default) :: p0, log_mppp, log_mmpm, log_mu_s, v
    p0 = abs(p0r) + ieps
    log_mppp = log( (p-p0+q) * (p+p0+q) )
    log_mmpm = log( (p-p0-q) * (p+p0-q) )
    select case (vec_type)
      case (1)
        select case (NRQCD_ORDER)
          case (0)
            v = CF*a * 2.*pi*(log_mppp-log_mmpm) * q/p
          case (1)
            log_mu_s = 2.*log(MU_SOFT)
            v = CF*a * (2.*(4.*pi+A1*a)*(log_mppp-log_mmpm) &
                      + B0*a*((log_mmpm-log_mu_s)**2-(log_mppp-log_mu_s)**2)) * q/(4.*p)
          case default
            call msg_fatal ("NRQCD_ORDER = " // char(NRQCD_ORDER))
        end select
      case (2)
        !!! not implemented yet
        v = zero
      case default
        call msg_fatal ("unknown ttZ/ttA vertex component, vec_type = " // char(vec_type))
    end select
  end function minus_q2_V
```

⟨*ttv formfactors: procedures*⟩+≡

```
  !!! compute support points (~> q-grid) for numerical integration: trim p-grid and
  !!! merge with singular points of integrand: q = p, |p-p0|, p+p0, sqrt(mpole*E)
  subroutine compute_support_points (en, i_p, i_p0, n_trim)
    real(default), intent(in) :: en
    integer, intent(in) :: i_p
    integer, intent(in) :: i_p0
    integer, intent(in) :: n_trim
    real(default) :: p, p0
    real(default), dimension(4) :: sing_vals
    integer :: n_sing, i_q
    if (mod (POINTS_P, n_trim) /= 0) call msg_fatal ("trim p-grid for q-integration: POINTS_P = "
                                  // char(POINTS_P) // " and n_trim = " // char(n_trim))
    n_q = POINTS_P / n_trim + merge(0,1,n_trim==1)
    p = p_grid(i_p)
    p0 = p0_grid(i_p0)
    n_sing = 0
```

```
    if ( i_p /= 1 .and. mod(i_p,n_trim) /= 0 ) then
      n_sing = n_sing+1
      sing_vals(n_sing) = p
    end if
    if ( i_p0 /= 1 ) then
      n_sing = n_sing+1
      sing_vals(n_sing) = p0 + p
      if ( i_p0 /= i_p+1 ) then
        n_sing = n_sing+1
        sing_vals(n_sing) = abs( p0 - p )
      end if
    end if
    if ( en > 0. ) then
      n_sing = n_sing+1
      sing_vals(n_sing) = sqrt( MTPOLE * en )
    end if
    if ( allocated(q_grid) ) deallocate( q_grid )
    allocate( q_grid(n_q+n_sing) )
    q_grid(1) = p_grid(1)
    q_grid(2:n_q) = [(p_grid(i_q), i_q=max(n_trim,2), POINTS_P, n_trim)]
    if (n_sing > 0 ) q_grid(n_q+1:n_q+n_sing) = sing_vals(1:n_sing)
    call nr_sort (q_grid)
  end subroutine compute_support_points
```

⟨*ttv formfactors: procedures*⟩+≡

```
  !!! cf. arXiv:hep-ph/9503238, validated against arXiv:hep-ph/0008171
  pure function formfactor_ttv_relativistic_nlo (alphas, ps, J0) result (c)
    real(default), intent(in) :: alphas
    type(phase_space_point_t), intent(in) :: ps
    complex(default), intent(in) :: J0
    complex(default) :: c
    real(default) :: p2, k2, q2, kp, pq, kq
    complex(default) :: D2, chi, ln1, ln2, L1, L2, z, S, m2, m
    complex(default) :: JA, JB, JC, JD, JE, IA, IB, IC, ID, IE
    complex(default) :: CCmsbar
    complex(default) :: dF1, dF2, dM1, dM2
    complex(default), dimension(12) :: P1
    complex(default), parameter :: ximo = zero
    p2 = ps%p2
    k2 = ps%k2
    q2 = ps%q2
    m2 =  complex_m2 (ps%mpole, GAM)
    !!! kinematic abbreviations
    kp = 0.5_default * (-q2 + p2 + k2)
    pq = 0.5_default * ( k2 - p2 - q2)
    kq = 0.5_default * (-p2 + k2 + q2)
    D2 = kp**2 - k2*p2
    chi = p2*k2*q2 + 2.*m2*((p2 + k2)*kp - 2.*p2*k2) + m2**2 * q2
    ln1 = log( (1. - p2/m2)*(1,0) + ieps )
    ln2 = log( (1. - k2/m2)*(1,0) + ieps )
    L1 = (1. - m2/p2) * ln1
    L2 = (1. - m2/k2) * ln2
    z = sqrt( (1.-4.*m2/q2)*(1,0) )
    S = 0.5_default * z * log( (z+1.)/(z-1.) + ieps )
```

```
    m = sqrt(m2)

    !!! loop integrals in terms of J0
    JA = 1./D2 * (J0/2.*(-m2*pq - p2*kq) + kp*L2 - p2*L1 - 2.*pq*S)
    JB = 1./D2 * (J0/2.*( m2*kq + k2*pq) + kp*L1 - k2*L2 + 2.*kq*S)
    JC = 1/(4.*D2) * (2.*p2 + 2*kp*m2/k2 - 4.*kp*S + 2.*kp*(1. - m2/k2)*L2 + &
            (2.*kp*(p2 - m2) + 3.*p2*(m2 - k2))*JA + p2*(m2 - p2)*JB)
    JD = 1./(4.*D2) * (2.*kp*((k2 - m2)*JA + (p2 - m2)*JB - 1.) - k2*(2.*m2/k2 &
            - 2.*S + (1. - m2/k2)*L2 + (p2 - m2)*JA) - p2*(-2.*S + (1. - &
            m2/p2)*L1 + (k2 - m2)*JB))
    JE = 1./(4.*D2) * (2.*k2 + 2*kp*m2/p2 - 4.*kp*S + 2.*kp*(1. - m2/p2)*L1 + &
            (2.*kp*(k2 - m2) + 3.*k2*(m2 - p2))*JB + k2*(m2 - k2)*JA)
    IA = 1./D2 * (-(kq/2.)*J0 - 2.*q2/chi *((m2 - p2)*k2 - (m2 - k2)*kp)*S + &
            1./(m2 - p2)*(p2 - kp + p2*q2/chi *(k2 - m2)*(m2 + kp))*L1 + &
            k2*q2/chi *(m2 + kp)*L2)
    IB = 1./D2 * ( (pq/2.)*J0 - 2.*q2/chi *((m2 - k2)*p2 - (m2 - p2)*kp)*S + &
            1./(m2 - k2)*(k2 - kp + k2*q2/chi *(p2 - m2)*(m2 + kp))*L2 + &
            p2*q2/chi *(m2 + kp)*L1)
    IC = 1./(4.*D2) * (2.*p2*J0 - 4.*kp/k2*(1. + m2/(k2 - m2)*L2) + (2.*kp - &
            3.*p2)*JA - p2*JB + (-2.*kp*(m2 - p2) + 3.*p2*(m2 - k2))*IA + &
            p2*(m2 - p2)*IB)
    ID = 1./(4.*D2) * (-2.*kp*J0 + 2.*(1. + m2/(k2 - m2)*L2) + 2.*(1. + &
            m2/(p2 - m2)*L1) + (2.*kp - k2)*JA + (2.*kp - p2)*JB + (k2*(m2 - &
            p2) - 2.*kp*(m2 - k2))*IA + (p2*(m2 - k2) - 2.*kp*(m2 - p2))*IB)
    IE = 1./(4.*D2) * (2.*k2*J0 - 4.*kp/p2*(1. + m2/(p2 - m2)*L1) + (2.*kp - &
            3.*k2)*JB - k2*JA + (-2.*kp*(m2 - k2) + 3.*k2*(m2 - p2))*IB + &
            k2*(m2 - k2)*IA)

    !!! divergent part ~ 1/epsilon: depends on subtraction scheme
    CCmsbar = -2.0_default * log(RESCALE_H)

    ! real top mass in the loop numerators
!    m2 = cmplx(real(m2), kind=default)
!    m  = sqrt(m2)

    !!! quark self energies
    dF1 = - (ximo+1.) * (CCmsbar + (1.+m2/p2)*(1.-L1))
    dF2 = - (ximo+1.) * (CCmsbar + (1.+m2/k2)*(1.-L2))
    dM1 = m/p2 * ( (ximo+1.)*(1.+m2/p2*ln1) - 3.*ln1 )
    dM2 = m/k2 * ( (ximo+1.)*(1.+m2/k2*ln2) - 3.*ln2 )

    !!! coefficient list: vertex function Gamma_mu (k,p) = sum_i( Vi_mu * Pi )
    P1(1)  =  2.*JA - 2.*JC + ximo*(m2*IC + p2*ID)
    P1(2)  =  2.*JB - 2.*JE + ximo*(k2*ID + m2*IE)
    P1(3)  = -2.*J0 + 2.*JA + 2.*JB - 2.*JD + ximo*(-J0/2. - k2/2.*IC - &
                kp*ID + m2*ID + p2/2.*IE + JA)
    P1(4)  = -2.*JD + ximo*(k2*IC + m2*ID - JA)
    P1(5)  = J0 - JA - JB + ximo*(J0/4. + k2/4.*IC + kp/2.*ID + p2/4.*IE - &
                1./2.*JA - 1./2.*JB)
    P1(6)  = -m2*J0 - k2*JA - p2*JB + k2/2.*JC + kp*JD + p2/2.*JE + &
                (1./2. + CCmsbar - 2.*S) &
                + ximo*(-m2*J0/4. - m2/4.*k2*IC - m2/2.*kp*ID - m2/4.*p2*IE &
                - k2/2.*JA - p2/2.*JB + (CCmsbar + 2.))
    P1(7)  =  2.*m*J0 - 4.*m*JA + ximo*m*(J0/2. - 2.*kp*IC + k2/2.*IC - &
```

```
                p2*ID - kp*ID - p2/2.*IE - JA)
    P1(8)  =  2.*m*J0 - 4.*m*JB + ximo*m*(J0/2. + k2/2.*IC - kp*ID + k2*ID - &
                p2/2.*IE - JB)
    P1(9)  =  ximo*m*(ID + IE)
    P1(10) =  ximo*m*(ID + IC)
    P1(11) =  ximo*m*( p2*ID + kp*IC + p2/2.*IE - k2/2.*IC) + dM2
                                !!! self energy contribution: ~ gamma_mu.k_slash = V11
    P1(12) =  ximo*m*(-k2*ID - kp*IE + p2/2.*IE - k2/2.*IC) + dM1
                                !!! self energy contribution: ~ gamma_mu.p_slash = V12

    !!! leading form factor: V6 = gamma_mu, V5 = gamma_mu.k_slash.p_slash ~> -m^2*gamma_mu
    c = one + alphas * CF / (4.*pi) * ( P1(6) - m2*P1(5) &
                !!! self energy contributions ~ gamma^mu
                + dF1 + dF2 + m*( dM1 + dM2 ) )
                !!! on-shell subtraction: UV divergence cancels
!                + 0.5_default*( dF1 + dF2 + m*( dM1 + dM2 ) )
  end function formfactor_ttv_relativistic_nlo
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure function sqrts_to_en (sqrts, mpole_in) result (en)
    real(default), intent(in) :: sqrts
    real(default), intent(in), optional :: mpole_in
    real(default) :: mpole, en
    if (present (mpole_in)) then
      mpole = mpole_in
    else
      mpole = m1s_to_mpole (sqrts)
    end if
    en = sqrts - two * mpole
  end function sqrts_to_en
```

⟨*ttv formfactors: procedures*⟩+≡

```
  function p_grid_from_TOPPIK (mpole_in) result (p_toppik)
    real(default), intent(in), optional :: mpole_in
    real(default), dimension(POINTS_P) :: p_toppik
    real(default) :: mpole
    call msg_debug (D_THRESHOLD, "p_grid_from_TOPPIK")
    mpole = MTPOLE;  if (present (mpole_in))  mpole = mpole_in
    call scan_formfactor_over_p_TOPPIK &
                (alphas_soft(2. * M1S), 2. * M1S, 1, p_toppik, mpole)
    if (.not. strictly_monotonous (p_toppik)) &
      call msg_fatal ("p_grid NOT strictly monotonous!")
  end function p_grid_from_TOPPIK
```

⟨*ttv formfactors: procedures*⟩+≡

```
  pure function int_to_char (i) result (c)
    integer, intent(in) :: i
    character(len=len(trim(int2fixed(i)))) :: c
    c = int2char (i)
  end function int_to_char
```

⟨*ttv formfactors: procedures*⟩+≡

```
pure function real_to_char (r) result (c)
  real(default), intent(in) :: r
  character(len=len(trim(real2fixed(r)))) :: c
  c = real2char (r)
end function real_to_char
```

⟨*ttv formfactors: procedures*⟩+≡

```
pure function complex_to_char (z) result (c)
  complex(default), intent(in) :: z
  character(len=len(trim(real2fixed(real(z))))+len(trim(real2fixed(aimag(z))))+5) :: c
  character(len=len(trim(real2fixed(real(z))))) :: re
  character(len=len(trim(real2fixed(aimag(z))))) :: im
  re = real_to_char (real(z))
  im = real_to_char (aimag(z))
  if (nearly_equal (aimag(z), zero)) then
    c = re
  else
    c = re // " + " // im // "*I"
  end if
end function complex_to_char
```

⟨*ttv formfactors: procedures*⟩+≡

```
pure function logical_to_char (l) result (c)
  logical, intent(in) :: l
  character(len=1) :: c
  write (c, '(l1)') l
end function logical_to_char
```

⟨*ttv formfactors: procedures*⟩+≡

```
subroutine p0_q_integrand_update (solver_f, a, ps, i)
  class(p0_q_integrand_t), intent(inout) :: solver_f
  real(default), intent(in) :: a
  type(phase_space_point_t), intent(in) :: ps
  integer, intent(in) :: i
  solver_f%a = a
  solver_f%ps = ps
  solver_f%i = i
end subroutine p0_q_integrand_update
```

⟨*ttv formfactors: procedures*⟩+≡

```
function p0_q_integrand_evaluate (solver_f, x) result (f)
  complex(default) :: f
  class(p0_q_integrand_t), intent(in) :: solver_f
  real(default), intent(in) :: x
  f = G0p_tree (solver_f%ps%en, x, solver_f%ps%mpole, GAM) &
        * minus_q2_V (solver_f%a, x, solver_f%ps%p, solver_f%ps%p0, solver_f%i) &
        * ff_p_spline%interpolate (x)
end function p0_q_integrand_evaluate
```

⟨*ttv formfactors: procedures*⟩+≡

```
subroutine get_rest_frame (p1_in, p2_in, p1_out, p2_out)
  type(vector4_t), intent(in) :: p1_in, p2_in
```

```
  type(vector4_t), intent(out) :: p1_out, p2_out
  type(lorentz_transformation_t) :: L
  L = inverse (boost (p1_in + p2_in, (p1_in + p2_in)**1))
  p1_out = L * p1_in; p2_out = L * p2_in
end subroutine get_rest_frame

function shift_momentum (p_in, E, p) result (p_out)
  type(vector4_t) :: p_out
  type(vector4_t), intent(in) :: p_in
  real(default), intent(in) :: E, p
  type(vector3_t) :: vec
  vec = p_in%p(1:3) / space_part_norm (p_in)
  p_out = vector4_moving (E, p * vec)
end function shift_momentum

subroutine evaluate_one_to_two_splitting_threshold (p_origin, &
    p1_in, p2_in, p1_out, p2_out, msq_in, jac)
  type(vector4_t), intent(in) :: p_origin
  type(vector4_t), intent(in) :: p1_in, p2_in
  type(vector4_t), intent(inout) :: p1_out, p2_out
  real(default), intent(in), optional :: msq_in
  real(default), intent(inout), optional :: jac
  type(lorentz_transformation_t) :: L
  type(vector4_t) :: p1_rest, p2_rest
  real(default) :: msq, msq1, msq2
  real(default) :: m
  real(default) :: E1, E2, E_max
  real(default) :: p, lda
  real(default), parameter :: E_offset = 0.001_default
  !!! (TODO-cw-2016-10-13) Find a better way to get masses
  real(default), parameter :: mb = 4.2_default
  real(default), parameter :: mw = 80.419_default

  call get_rest_frame (p1_in, p2_in, p1_rest, p2_rest)

  msq = p_origin**2; m = sqrt(msq)
  msq1 = p1_in**2
  msq2 = m * (m - two * p1_rest%p(0))
  E1 = (msq + msq1 - msq2) / (two * m)
  E_max = (msq - (mb + mw)**2) / (two * m)
  E_max = E_max - E_offset
  if (E1 > E_max) then
     E1 = E_max
     msq2 = m * (m - two * E_max)
  end if

  lda = lambda (msq, msq1, msq2)
  if (lda < zero) call msg_fatal &
       ("Threshold Splitting: lambda < 0 encountered! Use a higher offset.")
  p = sqrt(lda) / (two * m)

  E1 = sqrt (msq1 + p**2)
  E2 = sqrt (msq2 + p**2)
```

```
    p1_out = shift_momentum (p1_rest, E1, p)
    p2_out = shift_momentum (p2_rest, E2, p)

    L = boost (p_origin, p_origin**1)
    p1_out = L  * p1_out
    p2_out = L  * p2_out
  end subroutine evaluate_one_to_two_splitting_threshold
```

⟨*ttv formfactors: public*⟩+≡

```
  public :: generate_on_shell_decay_threshold
```

⟨*ttv formfactors: procedures*⟩+≡

```
  subroutine generate_on_shell_decay_threshold (p_decay, p_top, p_decay_onshell)
    !!! Gluon must be on first position in this array
    type(vector4_t), intent(in), dimension(:) :: p_decay
    type(vector4_t), intent(inout) :: p_top
    type(vector4_t), intent(inout), dimension(:) :: p_decay_onshell
    procedure(evaluate_one_to_two_splitting_special), pointer :: ppointer
    ppointer => evaluate_one_to_two_splitting_threshold
    call generate_on_shell_decay (p_top, p_decay, p_decay_onshell, 1, &
         evaluate_special = ppointer)
  end subroutine generate_on_shell_decay_threshold
```

29.0.1 Unit tests

Test module, followed by the corresponding implementation module.

⟨ttv_formfactors_ut.f90⟩≡

```
  ⟨File header⟩

  module ttv_formfactors_ut
    use unit_tests
    use ttv_formfactors_uti

  ⟨Standard module head⟩

  ⟨ttv formfactors: public test⟩

  contains

  ⟨ttv formfactors: test driver⟩

  end module ttv_formfactors_ut
```

⟨ttv_formfactors_uti.f90⟩≡

```
  ⟨File header⟩

  module ttv_formfactors_uti

  ⟨Use kinds⟩
    use constants
    use ttv_formfactors
    use diagnostics
    use sm_physics, only: running_as
```

```
  use numeric_utils
```

⟨*Standard module head*⟩

⟨*ttv formfactors: test declarations*⟩

```
contains
```

⟨*ttv formfactors: tests*⟩

```
end module ttv_formfactors_uti
```

API: driver for the unit tests below.

⟨*ttv formfactors: public test*⟩≡

```
  public ::ttv_formfactors_test
```

⟨*ttv formfactors: test driver*⟩≡

```
  subroutine ttv_formfactors_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```

⟨*ttv formfactors: execute tests*⟩

```
  end subroutine ttv_formfactors_test
```

Basic setup

⟨*ttv formfactors: execute tests*⟩≡

```
  call test(ttv_formfactors_1, "ttv_formfactors_1", &
            "Basic setup", u, results)
```

⟨*ttv formfactors: test declarations*⟩≡

```
  public :: ttv_formfactors_1
```

⟨*ttv formfactors: tests*⟩≡

```
  subroutine ttv_formfactors_1 (u)
    integer, intent(in) :: u
    real(default) :: m1s, Vtb, wt_inv, alphaemi, sw, alphas_mz, mz, &
         mw, mb, sh, sf, NRQCD_ORDER, FF, offshell_strategy, v1, v2, &
         scan_sqrts_max, sqrts, scan_sqrts_min, scan_sqrts_stepsize, &
         test, gam_out, mpole
    type(formfactor_t) :: formfactor
    type(phase_space_point_t) :: ps
    logical :: mpole_fixed
    integer :: top_helicity_selection
    write (u, "(A)")  "* Test output: ttv_formfactors_1"
    write (u, "(A)")  "*   Purpose: Basic setup"
    write (u, "(A)")

    m1s = 172.0_default
    Vtb = one
    wt_inv = zero
    alphaemi = 125.0_default
    alphas_mz = 0.118_default
    mz = 91.1876_default
    mw = 80.399_default
```

```
  sw = sqrt(one - mw**2 / mz**2)
  mb = 4.2_default
  sh = one
  sf = one
  NRQCD_ORDER = one
  FF = MATCHED
  offshell_strategy = 0
  top_helicity_selection = -1
  v1 = 0.3_default
  v2 = 0.5_default
  sqrts = 2 * m1s + 0.01_default
  scan_sqrts_min = sqrts
  scan_sqrts_max = sqrts
  scan_sqrts_stepsize = 0.1_default
  mpole_fixed = .true.
  test = - one
⟨(re)start grid⟩
  call threshold%formfactor%activate ()

  write (u, "(A)") "Check that the mass is fixed"
  call ps%init (m1s**2, m1s**2, sqrts**2, mpole)
  call assert_equal (u, m1s_to_mpole (350.0_default), m1s, &
       "m1s_to_mpole (350.0_default) == m1s")
  call assert_equal (u, m1s_to_mpole (550.0_default), m1s, &
       "m1s_to_mpole (550.0_default) == m1s")
  write (u, "(A)") ""

  write (u, "(A)") "Check that the mass is not fixed"
  mpole_fixed = .false.
⟨(re)start grid⟩
  call assert (u, m1s_to_mpole (350.0_default) > m1s + 0.1_default, &
       "m1s_to_mpole (350.0_default) > m1s")
  write (u, "(A)")

  !!! care: the formfactor contains the tree level that we usually subtract again
  write (u, "(A)") "Check low energy behavior"
  mpole_fixed = .true.
⟨(re)start grid⟩
  call assert_equal (u, f_switch_off (v_matching (ps%sqrts, GAM_M1S)), one, "f_switch_off (v_mat
  call formfactor%disable ()
  call assert_equal (u, &
       abs(formfactor%compute (ps, 1, 1)), &
       zero, &
       "disabled formfactor should return zero")
  call formfactor%activate ()
  call assert_equal (u, &
       formfactor%compute (ps, 1, EXPANDED_SOFT_SWITCHOFF), &
       formfactor%compute (ps, 1, EXPANDED_SOFT), &
       "switchoff function should do nothing here")
  write (u, "(A)") ""

  write (u, "(A)") "Check high energy behavior"
  sqrts = 500.0_default
  scan_sqrts_min = sqrts
```

```
    scan_sqrts_max = sqrts
  ⟨(re)start grid⟩
    ! For simplicity we test on-shell back-to-back tops
    call ps%init (m1s**2, m1s**2, sqrts**2, mpole)
    call assert_equal (u, f_switch_off (v_matching (ps%sqrts, GAM_M1S)), tiny_10, &
         "f_switch_off (v_matching (ps%sqrts, GAM_M1S))")
    call assert (u, &
         abs (formfactor%compute (ps, 1, EXPANDED_HARD)) > &
         abs (formfactor%compute (ps, 1, RESUMMED)), &
         "expansion with hard alphas should be larger " // &
         "than resummed (with switchoff)")
    call assert_equal (u, &
         abs (formfactor%compute (ps, 1, RESUMMED)), zero, &
         "resummed (with switchoff) should be zero", abs_smallness=tiny_10)
    call assert_equal (u, &
         abs (formfactor%compute (ps, 1, EXPANDED_SOFT_SWITCHOFF)), zero, &
         "expanded (with switchoff) should be zero", abs_smallness=tiny_10)
    write (u, "(A)") ""

    write (u, "(A)") "Check global variables"
    call assert_equal (u, AS_HARD, &
         running_as (m1s, alphas_mz, mz, 2, 5.0_default), "hard alphas")
    call assert_equal (u, AS_SOFT, zero, "soft alphas", abs_smallness=tiny_10)
    call assert_equal (u, AS_USOFT, zero, "ultrasoft alphas", abs_smallness=tiny_10)
    call assert_equal (u, AS_LL_SOFT, zero, "LL soft alphas", abs_smallness=tiny_10)

    write (u, "(A)")  "* Test output end: ttv_formfactors_1"
  end subroutine ttv_formfactors_1
```

⟨*(re)start grid*⟩≡

```
  call init_parameters &
       (mpole, gam_out, m1s, Vtb, wt_inv, &
        alphaemi, sw, alphas_mz, mz, mw, &
        mb, sh, sf, NRQCD_ORDER, FF, offshell_strategy, &
        v1, v2, scan_sqrts_min, scan_sqrts_max, &
        scan_sqrts_stepsize, mpole_fixed, real(top_helicity_selection, default))
  call init_threshold_grids (test)
```

Test flags

⟨*ttv formfactors: execute tests*⟩+≡

```
  call test(ttv_formfactors_2, "ttv_formfactors_2", &
            "Test flags", u, results)
```

⟨*ttv formfactors: test declarations*⟩+≡

```
  public :: ttv_formfactors_2
```

⟨*ttv formfactors: tests*⟩+≡

```
  subroutine ttv_formfactors_2 (u)
    integer, intent(in) :: u
    write (u, "(A)")  "* Test output: ttv_formfactors_2"
    write (u, "(A)")  "*   Purpose: Test flags"
    write (u, "(A)")

    write (u, "(A)") "RESUMMED_SWITCHOFF + NLO"
```

```
call threshold%settings%setup_flags (-2, 1, -1)
call assert (u, SWITCHOFF_RESUMMED, "SWITCHOFF_RESUMMED")
call assert (u, TOPPIK_RESUMMED, "TOPPIK_RESUMMED")
call assert (u, threshold%settings%nlo, "threshold%settings%nlo")
call assert (u, .not. threshold%settings%factorized_computation, &
     ".not. threshold%settings%factorized_computation")
call assert (u, .not. threshold%settings%interference, &
     ".not. threshold%settings%interference")
call assert (u, .not. threshold%settings%no_nlo_width_in_signal_propagators, &
     ".not. threshold%settings%no_nlo_width_in_signal_propagators")

write (u, "(A)") "MATCHED + FACTORIZATION"
call threshold%settings%setup_flags (-1, 0+2, -1)
call assert (u, .not. threshold%settings%nlo, ".not. threshold%settings%nlo")
call assert (u, TOPPIK_RESUMMED, "TOPPIK_RESUMMED")
call assert (u, threshold%settings%factorized_computation, &
     "threshold%settings%factorized_computation")

write (u, "(A)") "RESUMMED + INTERFERENCE"
call threshold%settings%setup_flags (1, 0+0+4, -1)
call assert (u, .not. SWITCHOFF_RESUMMED, ".not. SWITCHOFF_RESUMMED")
call assert (u, TOPPIK_RESUMMED, "TOPPIK_RESUMMED")
call assert (u, .not. threshold%settings%nlo, ".not. threshold%settings%nlo")
call assert (u, .not. threshold%settings%factorized_computation, &
     ".not. threshold%settings%factorized_computation")
call assert (u, threshold%settings%interference, "threshold%settings%interference")

write (u, "(A)") "EXPANDED_HARD"
call threshold%settings%setup_flags (4, 0+2+4, -1)
call assert (u, .not. SWITCHOFF_RESUMMED, ".not. SWITCHOFF_RESUMMED")
call assert (u, .not. TOPPIK_RESUMMED, ".not. TOPPIK_RESUMMED")
call assert (u, .not. threshold%settings%nlo, ".not. threshold%settings%nlo")
call assert (u, threshold%settings%factorized_computation, &
     "threshold%settings%factorized_computation")
call assert (u, threshold%settings%interference, "threshold%settings%interference")

write (u, "(A)") "EXPANDED_SOFT"
call threshold%settings%setup_flags (5, 1+2+4, -1)
call assert (u, .not. SWITCHOFF_RESUMMED, ".not. SWITCHOFF_RESUMMED")
call assert (u, .not. TOPPIK_RESUMMED, ".not. TOPPIK_RESUMMED")
call assert (u, threshold%settings%nlo, "threshold%settings%nlo")
call assert (u, threshold%settings%factorized_computation, &
     "threshold%settings%factorized_computation")
call assert (u, threshold%settings%interference, &
     "threshold%settings%interference")

write (u, "(A)") "EXPANDED_SOFT_SWITCHOFF"
call threshold%settings%setup_flags (6, 0+0+0+8, -1)
call assert (u, .not. SWITCHOFF_RESUMMED, "SWITCHOFF_RESUMMED")
call assert (u, .not. TOPPIK_RESUMMED, ".not. TOPPIK_RESUMMED")
call assert (u, .not. threshold%settings%nlo, "threshold%settings%nlo")
call assert (u, .not. threshold%settings%factorized_computation, &
     "threshold%settings%factorized_computation")
call assert (u, .not. threshold%settings%interference, &
```

```
     "threshold%settings%interference")

write (u, "(A)") "RESUMMED_ANALYTIC_LL"
call threshold%settings%setup_flags (7, 0+0+4+8, -1)
call assert (u, .not. SWITCHOFF_RESUMMED, "SWITCHOFF_RESUMMED")
call assert (u, .not. TOPPIK_RESUMMED, ".not. TOPPIK_RESUMMED")
call assert (u, .not. threshold%settings%nlo, "threshold%settings%nlo")
call assert (u, .not. threshold%settings%factorized_computation, &
     "threshold%settings%factorized_computation")
call assert (u, threshold%settings%interference, "threshold%settings%interference")
call assert (u, threshold%settings%onshell_projection%production, &
     "threshold%settings%onshell_projection%production")

write (u, "(A)") "EXPANDED_SOFT_HARD"
call threshold%settings%setup_flags (8, 0+2+0+128, -1)
call assert (u, .not. SWITCHOFF_RESUMMED, "SWITCHOFF_RESUMMED")
call assert (u, .not. TOPPIK_RESUMMED, ".not. TOPPIK_RESUMMED")
call assert (u, .not. threshold%settings%nlo, "threshold%settings%nlo")
call assert (u, threshold%settings%factorized_computation, &
     "threshold%settings%factorized_computation")
call assert (u, .not. threshold%settings%interference, "threshold%settings%interference")
call assert (u, .not. threshold%settings%onshell_projection%production, &
     "threshold%settings%onshell_projection%production")
call assert (u, threshold%settings%onshell_projection%decay, &
     "threshold%settings%onshell_projection%decay")

write (u, "(A)") "EXTRA_TREE"
call threshold%settings%setup_flags (9, 1+0+0+16+64, -1)
call assert (u, .not. SWITCHOFF_RESUMMED, "SWITCHOFF_RESUMMED")
call assert (u, .not. TOPPIK_RESUMMED, ".not. TOPPIK_RESUMMED")
call assert (u, threshold%settings%nlo, "threshold%settings%nlo")
call assert (u, .not. threshold%settings%factorized_computation, &
     "threshold%settings%factorized_computation")
call assert (u, .not. threshold%settings%interference, "threshold%settings%interference")
call assert (u, threshold%settings%onshell_projection%production, &
     "threshold%settings%onshell_projection%production")
call assert (u, .not. threshold%settings%onshell_projection%decay, &
     "threshold%settings%onshell_projection%decay")
call assert (u, threshold%settings%no_nlo_width_in_signal_propagators, &
     "threshold%settings%no_nlo_width_in_signal_propagators")

write (u, "(A)") "test projection of width"
call threshold%settings%setup_flags (9, 0+0+0+0+256, -1)
call assert (u, .not. threshold%settings%onshell_projection%production, &
     "threshold%settings%onshell_projection%production")
call assert (u, .not. threshold%settings%onshell_projection%decay, &
     "threshold%settings%onshell_projection%decay")
call assert (u, .not. threshold%settings%onshell_projection%width, &
     "threshold%settings%onshell_projection%width")

write (u, "(A)") "test boost of decay momenta"
call threshold%settings%setup_flags (9, 512, -1)
call msg_debug (D_THRESHOLD, &
     "threshold%settings%onshell_projection%boost_decay", &
```

```
      threshold%settings%onshell_projection%boost_decay)
  call threshold%settings%setup_flags (9, 0, -1)
  call msg_debug (D_THRESHOLD, &
      ".not. threshold%settings%onshell_projection%boost_decay", &
      .not. threshold%settings%onshell_projection%boost_decay)

  write (u, "(A)") "test helicity approximations"
  call threshold%settings%setup_flags (9, 32, -1)
  call assert (u, threshold%settings%helicity_approximation%simple, &
      "threshold%settings%helicity_approximation%simple")
  call assert (u, .not. threshold%settings%helicity_approximation%extra, &
      ".not. threshold%settings%helicity_approximation%extra")
  call assert (u, .not. threshold%settings%helicity_approximation%ultra, &
      ".not. threshold%settings%helicity_approximation%ultra")
  call threshold%settings%setup_flags (9, 1024, -1)
  call assert (u, .not. threshold%settings%helicity_approximation%simple, &
      ".not. threshold%settings%helicity_approximation%simple")
  call assert (u, threshold%settings%helicity_approximation%extra, &
      "threshold%settings%helicity_approximation%extra")

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: ttv_formfactors_2"
end subroutine ttv_formfactors_2
```

Chapter 30

Integration and Process Objects

This is the central part of the WHIZARD package. It provides the functionality for evaluating structure functions, kinematics and matrix elements, integration and event generation. It combines the various parts that deal with those tasks individually and organizes the data transfer between them.

subevt_expr This enables process observables as (abstract) expressions, to be evaluated for each process call.

parton_states A parton_state_t object represents an elementary partonic interaction. There are two versions: one for the isolated elementary process, one for the elementary process convoluted with the structure-function chain. The parton state is an effective state. It needs not coincide with the seed-kinematics state which is used in evaluating phase space.

process Here, all pieces are combined for the purpose of evaluating the elementary processes. The whole algorithm is coded in terms of abstract data types as defined in the appropriate modules: prc_core for matrix-element evaluation, prc_core_def for the associated configuration and

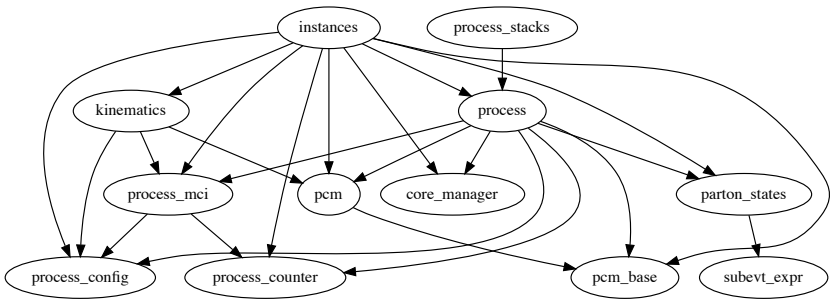

Figure 30.1: Module dependencies in src/process_integration.

driver, `sf_base` for beams and structure-functions, `phs_base` for phase space, and `mci_base` for integration and event generation.

process_config

process_counter Very simple object for statistics

process_mci

pcm

kinematics

instances While the above modules set up all static information, the instances have the changing event data. There are term and process instances but no component instances.

process_stacks Process stacks collect process objects.

We combine here hard interactions, phase space, and (for scatterings) structure functions and interfaces them to the integration module.

The process object implements the combination of a fixed beam and structure-function setup with a number of elementary processes. The latter are called process components. The process object represents an entity which is supposedly observable. It should be meaningful to talk about the cross section of a process.

The individual components of a process are, technically, processes themselves, but they may have unphysical cross sections which have to be added for a physical result. Process components may be exclusive tree-level elementary processes, dipole subtraction term, loop corrections, etc.

The beam and structure function setup is common to all process components. Thus, there is only one instance of this part.

The process may be a scattering process or a decay process. In the latter case, there are no structure functions, and the beam setup consists of a single particle. Otherwise, the two classes are treated on the same footing.

Once a sampling point has been chosen, a process determines a set of partons with a correlated density matrix of quantum numbers. In general, each sampling point will generate, for each process component, one or more distinct parton configurations. This is the `computed` state. The computed state is the subject of the multi-channel integration algorithm.

For NLO computations, it is necessary to project the computed states onto another set of parton configurations (e.g., by recombining certain pairs). This is the `observed` state. When computing partonic observables, the information is taken from the observed state.

For the purpose of event generation, we will later select one parton configuration from the observed state and collapse the correlated quantum state. This configuration is then dressed by applying parton shower, decays and hadronization. The decay chain, in particular, combines a scattering process with possible subsequent decay processes on the parton level, which are full-fledged process objects themselves.

30.1 Process observables

We define an abstract subevt_expr_t object as an extension of the subevt_t type. The object contains a local variable list, variable instances (as targets for pointers in the variable list), and evaluation trees. The evaluation trees reference both the variables and the subevt.

There are two instances of the abstract type: one for process instances, one for physical events. Both have a common logical expression selection which determines whether the object passes user-defined cuts.

The intention is that we fill the subevt_t base object and compute the variables once we have evaluated a kinematical phase space point (or a complete event). We then evaluate the expressions and can use the results in further calculations.

The process_expr_t extension contains furthermore scale and weight expressions. The event_expr_t extension contains a reweighting-factor expression and a logical expression for event analysis. In practice, we will link the variable list of the event_obs object to the variable list of the currently active process_obs object, such that the process variables are available to both objects. Event variables are meaningful only for physical events.

Note that there are unit tests, but they are deferred to the expr_tests module.

⟨subevt_expr.f90⟩≡

```
⟨File header⟩
module subevt_expr

⟨Use kinds⟩
⟨Use strings⟩
  use constants, only: zero, one
  use io_units
  use format_utils, only: write_separator
  use diagnostics
  use lorentz
  use subevents
  use variables
  use flavors
  use quantum_numbers
  use interactions
  use particles
  use expr_base

⟨Standard module head⟩

⟨Subevt expr: public⟩

⟨Subevt expr: types⟩

⟨Subevt expr: interfaces⟩

contains

⟨Subevt expr: procedures⟩
```

```
end module subevt_expr
```

30.1.1 Abstract base type

⟨*Subevt expr: types*⟩≡

```
type, extends (subevt_t), abstract :: subevt_expr_t
   logical :: subevt_filled = .false.
   type(var_list_t) :: var_list
   real(default) :: sqrts_hat = 0
   integer :: n_in = 0
   integer :: n_out = 0
   integer :: n_tot = 0
   logical :: has_selection = .false.
   class(expr_t), allocatable :: selection
 contains
 ⟨Subevt expr: subevt expr: TBP⟩
end type subevt_expr_t
```

Output: Base and extended version. We already have a `write` routine for the `subevt_t` parent type.

⟨*Subevt expr: subevt expr: TBP*⟩≡

```
procedure :: base_write => subevt_expr_write
```

⟨*Subevt expr: procedures*⟩≡

```
subroutine subevt_expr_write (object, unit, pacified)
  class(subevt_expr_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacified
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Local variables:"
  call write_separator (u)
  call var_list_write (object%var_list, u, follow_link=.false., &
       pacified = pacified)
  call write_separator (u)
  if (object%subevt_filled) then
     call object%subevt_t%write (u, pacified = pacified)
     if (object%has_selection) then
        call write_separator (u)
        write (u, "(1x,A)")  "Selection expression:"
        call write_separator (u)
        call object%selection%write (u)
     end if
  else
     write (u, "(1x,A)")  "subevt: [undefined]"
  end if
end subroutine subevt_expr_write
```

Finalizer.

⟨*Subevt expr: subevt expr: TBP*⟩+≡

```
procedure (subevt_expr_final), deferred :: final
procedure :: base_final => subevt_expr_final
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine subevt_expr_final (object)
  class(subevt_expr_t), intent(inout) :: object
  call object%var_list%final ()
  if (object%has_selection) then
     call object%selection%final ()
  end if
end subroutine subevt_expr_final
```

30.1.2 Initialization

Initialization: define local variables and establish pointers.

The common variables are `sqrts` (the nominal beam energy, fixed), `sqrts_hat` (the actual energy), `n_in`, `n_out`, and `n_tot` for the `subevt`. With the exception of `sqrts`, all are implemented as pointers to subobjects.

⟨*Subevt expr: subevt expr: TBP*⟩+≡

```
procedure (subevt_expr_setup_vars), deferred :: setup_vars
procedure :: base_setup_vars => subevt_expr_setup_vars
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine subevt_expr_setup_vars (expr, sqrts)
  class(subevt_expr_t), intent(inout), target :: expr
  real(default), intent(in) :: sqrts
  call expr%var_list%final ()
  call var_list_append_real (expr%var_list, &
       var_str ("sqrts"), sqrts, &
       locked = .true., verbose = .false., intrinsic = .true.)
  call var_list_append_real_ptr (expr%var_list, &
       var_str ("sqrts_hat"), expr%sqrts_hat, &
       is_known = expr%subevt_filled, &
       locked = .true., verbose = .false., intrinsic = .true.)
  call var_list_append_int_ptr (expr%var_list, &
       var_str ("n_in"), expr%n_in, &
       is_known = expr%subevt_filled, &
       locked = .true., verbose = .false., intrinsic = .true.)
  call var_list_append_int_ptr (expr%var_list, &
       var_str ("n_out"), expr%n_out, &
       is_known = expr%subevt_filled, &
       locked = .true., verbose = .false., intrinsic = .true.)
  call var_list_append_int_ptr (expr%var_list, &
       var_str ("n_tot"), expr%n_tot, &
       is_known = expr%subevt_filled, &
       locked = .true., verbose = .false., intrinsic = .true.)
end subroutine subevt_expr_setup_vars
```

Append the subevent expr (its base-type core) itself to the variable list, if it is not yet present.

⟨*Subevt expr: subevt expr: TBP*⟩+≡

```
procedure :: setup_var_self => subevt_expr_setup_var_self
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine subevt_expr_setup_var_self (expr)
  class(subevt_expr_t), intent(inout), target :: expr
  if (.not. expr%var_list%contains (var_str ("@evt"))) then
     call var_list_append_subevt_ptr &
          (expr%var_list, &
          var_str ("@evt"), expr%subevt_t, &
          is_known = expr%subevt_filled, &
          locked = .true., verbose = .false., intrinsic=.true.)
  end if
end subroutine subevt_expr_setup_var_self
```

Link a variable list to the local one. This could be done event by event, but before evaluating expressions.

⟨*Subevt expr: subevt expr: TBP*⟩+≡

```
procedure :: link_var_list => subevt_expr_link_var_list
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine subevt_expr_link_var_list (expr, var_list)
  class(subevt_expr_t), intent(inout) :: expr
  type(var_list_t), intent(in), target :: var_list
  call expr%var_list%link (var_list)
end subroutine subevt_expr_link_var_list
```

Compile the selection expression. If there is no expression, the build method won't allocate the expression object.

⟨*Subevt expr: subevt expr: TBP*⟩+≡

```
procedure :: setup_selection => subevt_expr_setup_selection
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine subevt_expr_setup_selection (expr, ef_cuts)
  class(subevt_expr_t), intent(inout), target :: expr
  class(expr_factory_t), intent(in) :: ef_cuts
  call ef_cuts%build (expr%selection)
  if (allocated (expr%selection)) then
     call expr%setup_var_self ()
     call expr%selection%setup_lexpr (expr%var_list)
     expr%has_selection = .true.
  end if
end subroutine subevt_expr_setup_selection
```

30.1.3 Evaluation

Reset to initial state, i.e., mark the `subevt` as invalid.

⟨*Subevt expr: subevt expr: TBP*⟩+≡

```
procedure :: reset => subevt_expr_reset
procedure :: base_reset => subevt_expr_reset
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine subevt_expr_reset (expr)
  class(subevt_expr_t), intent(inout) :: expr
  expr%subevt_filled = .false.
```

```
end subroutine subevt_expr_reset
```

Evaluate the selection expression and return the result. There is also a deferred version: this should evaluate the remaining expressions if the event has passed.

⟨*Subevt expr: subevt expr: TBP*⟩+≡

```
procedure :: base_evaluate => subevt_expr_evaluate
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine subevt_expr_evaluate (expr, passed)
  class(subevt_expr_t), intent(inout) :: expr
  logical, intent(out) :: passed
  if (expr%has_selection) then
     call expr%selection%evaluate ()
     if (expr%selection%is_known ()) then
        passed = expr%selection%get_log ()
     else
        call msg_error ("Evaluate selection expression: result undefined")
        passed = .false.
     end if
  else
     passed = .true.
  end if
end subroutine subevt_expr_evaluate
```

30.1.4 Implementation for partonic events

This implementation contains the expressions that we can evaluate for the partonic process during integration.

⟨*Subevt expr: public*⟩≡

```
public :: parton_expr_t
```

⟨*Subevt expr: types*⟩+≡

```
type, extends (subevt_expr_t) :: parton_expr_t
   integer, dimension(:), allocatable :: i_beam
   integer, dimension(:), allocatable :: i_in
   integer, dimension(:), allocatable :: i_out
   logical :: has_scale = .false.
   logical :: has_fac_scale = .false.
   logical :: has_ren_scale = .false.
   logical :: has_weight = .false.
   class(expr_t), allocatable :: scale
   class(expr_t), allocatable :: fac_scale
   class(expr_t), allocatable :: ren_scale
   class(expr_t), allocatable :: weight
 contains
 ⟨Subevt expr: parton expr: TBP⟩
end type parton_expr_t
```

Finalizer.

⟨*Subevt expr: parton expr: TBP*⟩≡

```
procedure :: final => parton_expr_final
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine parton_expr_final (object)
  class(parton_expr_t), intent(inout) :: object
  call object%base_final ()
  if (object%has_scale) then
     call object%scale%final ()
  end if
  if (object%has_fac_scale) then
     call object%fac_scale%final ()
  end if
  if (object%has_ren_scale) then
     call object%ren_scale%final ()
  end if
  if (object%has_weight) then
     call object%weight%final ()
  end if
end subroutine parton_expr_final
```

Output: continue writing the active expressions, after the common selection expression.

Note: the `prefix` argument is declared in the `write` method of the `subevt_t` base type. Here, it is unused.

⟨*Subevt expr: parton expr: TBP*⟩+≡

```
procedure :: write => parton_expr_write
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine parton_expr_write (object, unit, prefix, pacified)
  class(parton_expr_t), intent(in) :: object
  integer, intent(in), optional :: unit
  character(*), intent(in), optional :: prefix
  logical, intent(in), optional :: pacified
  integer :: u
  u = given_output_unit (unit)
  call object%base_write (u, pacified = pacified)
  if (object%subevt_filled) then
     if (object%has_scale) then
        call write_separator (u)
        write (u, "(1x,A)")  "Scale expression:"
        call write_separator (u)
        call object%scale%write (u)
     end if
     if (object%has_fac_scale) then
        call write_separator (u)
        write (u, "(1x,A)")  "Factorization scale expression:"
        call write_separator (u)
        call object%fac_scale%write (u)
     end if
     if (object%has_ren_scale) then
        call write_separator (u)
        write (u, "(1x,A)")  "Renormalization scale expression:"
        call write_separator (u)
        call object%ren_scale%write (u)
     end if
     if (object%has_weight) then
```

```
          call write_separator (u)
          write (u, "(1x,A)")  "Weight expression:"
          call write_separator (u)
          call object%weight%write (u)
       end if
    end if
  end subroutine parton_expr_write
```

Define variables.

⟨*Subevt expr: parton expr: TBP*⟩+≡

```
  procedure :: setup_vars => parton_expr_setup_vars
```

⟨*Subevt expr: procedures*⟩+≡

```
  subroutine parton_expr_setup_vars (expr, sqrts)
    class(parton_expr_t), intent(inout), target :: expr
    real(default), intent(in) :: sqrts
    call expr%base_setup_vars (sqrts)
  end subroutine parton_expr_setup_vars
```

Compile the scale expressions. If a pointer is disassociated, there is no expression.

⟨*Subevt expr: parton expr: TBP*⟩+≡

```
  procedure :: setup_scale => parton_expr_setup_scale
  procedure :: setup_fac_scale => parton_expr_setup_fac_scale
  procedure :: setup_ren_scale => parton_expr_setup_ren_scale
```

⟨*Subevt expr: procedures*⟩+≡

```
  subroutine parton_expr_setup_scale (expr, ef_scale)
    class(parton_expr_t), intent(inout), target :: expr
    class(expr_factory_t), intent(in) :: ef_scale
    call ef_scale%build (expr%scale)
    if (allocated (expr%scale)) then
       call expr%setup_var_self ()
       call expr%scale%setup_expr (expr%var_list)
       expr%has_scale = .true.
    end if
  end subroutine parton_expr_setup_scale

  subroutine parton_expr_setup_fac_scale (expr, ef_fac_scale)
    class(parton_expr_t), intent(inout), target :: expr
    class(expr_factory_t), intent(in) :: ef_fac_scale
    call ef_fac_scale%build (expr%fac_scale)
    if (allocated (expr%fac_scale)) then
       call expr%setup_var_self ()
       call expr%fac_scale%setup_expr (expr%var_list)
       expr%has_fac_scale = .true.
    end if
  end subroutine parton_expr_setup_fac_scale

  subroutine parton_expr_setup_ren_scale (expr, ef_ren_scale)
    class(parton_expr_t), intent(inout), target :: expr
    class(expr_factory_t), intent(in) :: ef_ren_scale
    call ef_ren_scale%build (expr%ren_scale)
    if (allocated (expr%ren_scale)) then
```

```
      call expr%setup_var_self ()
      call expr%ren_scale%setup_expr (expr%var_list)
      expr%has_ren_scale = .true.
    end if
  end subroutine parton_expr_setup_ren_scale
```

Compile the weight expression.

⟨*Subevt expr: parton expr: TBP*⟩+≡

```
  procedure :: setup_weight => parton_expr_setup_weight
```

⟨*Subevt expr: procedures*⟩+≡

```
  subroutine parton_expr_setup_weight (expr, ef_weight)
    class(parton_expr_t), intent(inout), target :: expr
    class(expr_factory_t), intent(in) :: ef_weight
    call ef_weight%build (expr%weight)
    if (allocated (expr%weight)) then
      call expr%setup_var_self ()
      call expr%weight%setup_expr (expr%var_list)
      expr%has_weight = .true.
    end if
  end subroutine parton_expr_setup_weight
```

Filling the partonic state consists of two parts. The first routine prepares the subevt without assigning momenta. It takes the particles from an `interaction_t`. It needs the indices and flavors for the beam, incoming, and outgoing particles.

We can assume that the particle content of the subevt does not change. Therefore, we set the event variables `n_in`, `n_out`, `n_tot` already in this initialization step.

⟨*Subevt expr: parton expr: TBP*⟩+≡

```
  procedure :: setup_subevt => parton_expr_setup_subevt
```

⟨*Subevt expr: procedures*⟩+≡

```
  subroutine parton_expr_setup_subevt (expr, int, &
      i_beam, i_in, i_out, f_beam, f_in, f_out)
    class(parton_expr_t), intent(inout) :: expr
    type(interaction_t), intent(in), target :: int
    integer, dimension(:), intent(in) :: i_beam, i_in, i_out
    type(flavor_t), dimension(:), intent(in) :: f_beam, f_in, f_out
    allocate (expr%i_beam (size (i_beam)))
    allocate (expr%i_in (size (i_in)))
    allocate (expr%i_out (size (i_out)))
    expr%i_beam = i_beam
    expr%i_in = i_in
    expr%i_out = i_out
    call interaction_to_subevt (int, &
        expr%i_beam, expr%i_in, expr%i_out, expr%subevt_t)
    call subevt_set_pdg_beam     (expr%subevt_t, f_beam%get_pdg ())
    call subevt_set_pdg_incoming (expr%subevt_t, f_in%get_pdg ())
    call subevt_set_pdg_outgoing (expr%subevt_t, f_out%get_pdg ())
    call subevt_set_p2_beam     (expr%subevt_t, f_beam%get_mass () ** 2)
    call subevt_set_p2_incoming (expr%subevt_t, f_in%get_mass ()   ** 2)
    call subevt_set_p2_outgoing (expr%subevt_t, f_out%get_mass ()  ** 2)
    expr%n_in  = size (i_in)
```

```
    expr%n_out = size (i_out)
    expr%n_tot = expr%n_in + expr%n_out
  end subroutine parton_expr_setup_subevt
```

Transfer PDG codes, masses (initalization) and momenta to a predefined subevent. We use the flavor assignment of the first branch in the interaction state matrix. Only incoming and outgoing particles are transferred. Switch momentum sign for incoming particles.

⟨*Subevt expr: interfaces*⟩≡

```
  interface interaction_momenta_to_subevt
     module procedure interaction_momenta_to_subevt_id
     module procedure interaction_momenta_to_subevt_tr
  end interface
```

⟨*Subevt expr: procedures*⟩+≡

```
  subroutine interaction_to_subevt (int, j_beam, j_in, j_out, subevt)
    type(interaction_t), intent(in), target :: int
    integer, dimension(:), intent(in) :: j_beam, j_in, j_out
    type(subevt_t), intent(out) :: subevt
    type(flavor_t), dimension(:), allocatable :: flv
    integer :: n_beam, n_in, n_out, i, j
    allocate (flv (int%get_n_tot ()))
    flv = quantum_numbers_get_flavor (int%get_quantum_numbers (1))
    n_beam = size (j_beam)
    n_in = size (j_in)
    n_out = size (j_out)
    call subevt_init (subevt, n_beam + n_in + n_out)
    do i = 1, n_beam
       j = j_beam(i)
       call subevt_set_beam (subevt, i, &
            flv(j)%get_pdg (), &
            vector4_null, &
            flv(j)%get_mass () ** 2)
    end do
    do i = 1, n_in
       j = j_in(i)
       call subevt_set_incoming (subevt, n_beam + i, &
            flv(j)%get_pdg (), &
            vector4_null, &
            flv(j)%get_mass () ** 2)
    end do
    do i = 1, n_out
       j = j_out(i)
       call subevt_set_outgoing (subevt, n_beam + n_in + i, &
            flv(j)%get_pdg (), &
            vector4_null, &
            flv(j)%get_mass () ** 2)
    end do
  end subroutine interaction_to_subevt

  subroutine interaction_momenta_to_subevt_id (int, j_beam, j_in, j_out, subevt)
    type(interaction_t), intent(in) :: int
    integer, dimension(:), intent(in) :: j_beam, j_in, j_out
```

```
    type(subevt_t), intent(inout) :: subevt
    call subevt_set_p_beam (subevt, - int%get_momenta (j_beam))
    call subevt_set_p_incoming (subevt, - int%get_momenta (j_in))
    call subevt_set_p_outgoing (subevt, int%get_momenta (j_out))
  end subroutine interaction_momenta_to_subevt_id

  subroutine interaction_momenta_to_subevt_tr &
       (int, j_beam, j_in, j_out, lt, subevt)
    type(interaction_t), intent(in) :: int
    integer, dimension(:), intent(in) :: j_beam, j_in, j_out
    type(subevt_t), intent(inout) :: subevt
    type(lorentz_transformation_t), intent(in) :: lt
    call subevt_set_p_beam &
         (subevt, - lt * int%get_momenta (j_beam))
    call subevt_set_p_incoming &
         (subevt, - lt * int%get_momenta (j_in))
    call subevt_set_p_outgoing &
         (subevt, lt * int%get_momenta (j_out))
  end subroutine interaction_momenta_to_subevt_tr
```

The second part takes the momenta from the interaction object and thus completes the subevt. The partonic energy can then be computed.

⟨*Subevt expr: parton expr: TBP*⟩+≡

```
  procedure :: fill_subevt => parton_expr_fill_subevt
```

⟨*Subevt expr: procedures*⟩+≡

```
  subroutine parton_expr_fill_subevt (expr, int)
    class(parton_expr_t), intent(inout) :: expr
    type(interaction_t), intent(in), target :: int
    call interaction_momenta_to_subevt (int, &
         expr%i_beam, expr%i_in, expr%i_out, expr%subevt_t)
    expr%sqrts_hat = subevt_get_sqrts_hat (expr%subevt_t)
    expr%subevt_filled = .true.
  end subroutine parton_expr_fill_subevt
```

Evaluate, if the event passes the selection. For absent expressions we take default values.

⟨*Subevt expr: parton expr: TBP*⟩+≡

```
  procedure :: evaluate => parton_expr_evaluate
```

⟨*Subevt expr: procedures*⟩+≡

```
  subroutine parton_expr_evaluate &
       (expr, passed, scale, fac_scale, ren_scale, weight, scale_forced, force_evaluation)
    class(parton_expr_t), intent(inout) :: expr
    logical, intent(out) :: passed
    real(default), intent(out) :: scale
    real(default), intent(out) :: fac_scale
    real(default), intent(out) :: ren_scale
    real(default), intent(out) :: weight
    real(default), intent(in), allocatable, optional :: scale_forced
    logical, intent(in), optional :: force_evaluation
    logical :: force_scale, force_eval
    force_scale = .false.; force_eval = .false.
    if (present (scale_forced))  force_scale = allocated (scale_forced)
```

```
if (present (force_evaluation)) force_eval = force_evaluation
call expr%base_evaluate (passed)
if (passed .or. force_eval) then
   if (force_scale) then
      scale = scale_forced
   else if (expr%has_scale) then
      call expr%scale%evaluate ()
      if (expr%scale%is_known ()) then
         scale = expr%scale%get_real ()
      else
         call msg_error ("Evaluate scale expression: result undefined")
         scale = zero
      end if
   else
      scale = expr%sqrts_hat
   end if
   if (force_scale) then
      fac_scale = scale_forced
   else if (expr%has_fac_scale) then
      call expr%fac_scale%evaluate ()
      if (expr%fac_scale%is_known ()) then
         fac_scale = expr%fac_scale%get_real ()
      else
         call msg_error ("Evaluate factorization scale expression: &
              &result undefined")
         fac_scale = zero
      end if
   else
      fac_scale = scale
   end if
   if (force_scale) then
      ren_scale = scale_forced
   else if (expr%has_ren_scale) then
      call expr%ren_scale%evaluate ()
      if (expr%ren_scale%is_known ()) then
         ren_scale = expr%ren_scale%get_real ()
      else
         call msg_error ("Evaluate renormalization scale expression: &
              &result undefined")
         ren_scale = zero
      end if
   else
      ren_scale = scale
   end if
   if (expr%has_weight) then
      call expr%weight%evaluate ()
      if (expr%weight%is_known ()) then
         weight = expr%weight%get_real ()
      else
         call msg_error ("Evaluate weight expression: result undefined")
         weight = zero
      end if
   else
      weight = one
```

```
      end if
    else
      weight = zero
    end if
  end subroutine parton_expr_evaluate
```

Return the beam/incoming parton indices.

⟨*Subevt expr: parton expr: TBP*⟩+≡

```
  procedure :: get_beam_index => parton_expr_get_beam_index
  procedure :: get_in_index => parton_expr_get_in_index
```

⟨*Subevt expr: procedures*⟩+≡

```
  subroutine parton_expr_get_beam_index (expr, i_beam)
    class(parton_expr_t), intent(in) :: expr
    integer, dimension(:), intent(out) :: i_beam
    i_beam = expr%i_beam
  end subroutine parton_expr_get_beam_index

  subroutine parton_expr_get_in_index (expr, i_in)
    class(parton_expr_t), intent(in) :: expr
    integer, dimension(:), intent(out) :: i_in
    i_in = expr%i_in
  end subroutine parton_expr_get_in_index
```

30.1.5 Implementation for full events

This implementation contains the expressions that we can evaluate for the full event. It also contains data that pertain to the event, suitable for communication with external event formats. These data simultaneously serve as pointer targets for the variable lists hidden in the expressions (eval trees).

Squared matrix element and weight values: when reading events from file, the `ref` value is the number in the file, while the `prc` value is the number that we calculate from the momenta in the file, possibly with different parameters. When generating events the first time, or if we do not recalculate, the numbers should coincide. Furthermore, the array of `alt` values is copied from an array of alternative event records. These values should represent calculated values.

⟨*Subevt expr: public*⟩+≡

```
  public :: event_expr_t
```

⟨*Subevt expr: types*⟩+≡

```
  type, extends (subevt_expr_t) :: event_expr_t
     logical :: has_reweight = .false.
     logical :: has_analysis = .false.
     class(expr_t), allocatable :: reweight
     class(expr_t), allocatable :: analysis
     logical :: has_id = .false.
     type(string_t) :: id
     logical :: has_num_id = .false.
     integer :: num_id = 0
     logical :: has_index = .false.
     integer :: index = 0
     logical :: has_sqme_ref = .false.
```

```
    real(default) :: sqme_ref = 0
    logical :: has_sqme_prc = .false.
    real(default) :: sqme_prc = 0
    logical :: has_weight_ref = .false.
    real(default) :: weight_ref = 0
    logical :: has_weight_prc = .false.
    real(default) :: weight_prc = 0
    logical :: has_excess_prc = .false.
    real(default) :: excess_prc = 0
    integer :: n_alt = 0
    logical :: has_sqme_alt = .false.
    real(default), dimension(:), allocatable :: sqme_alt
    logical :: has_weight_alt = .false.
    real(default), dimension(:), allocatable :: weight_alt
  contains
  ⟨Subevt expr: event expr: TBP⟩
end type event_expr_t
```

Finalizer for the expressions.

⟨*Subevt expr: event expr: TBP*⟩≡

```
procedure :: final => event_expr_final
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine event_expr_final (object)
  class(event_expr_t), intent(inout) :: object
  call object%base_final ()
  if (object%has_reweight) then
     call object%reweight%final ()
  end if
  if (object%has_analysis) then
     call object%analysis%final ()
  end if
end subroutine event_expr_final
```

Output: continue writing the active expressions, after the common selection expression.

Note: the `prefix` argument is declared in the `write` method of the `subevt_t` base type. Here, it is unused.

⟨*Subevt expr: event expr: TBP*⟩+≡

```
procedure :: write => event_expr_write
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine event_expr_write (object, unit, prefix, pacified)
  class(event_expr_t), intent(in) :: object
  integer, intent(in), optional :: unit
  character(*), intent(in), optional :: prefix
  logical, intent(in), optional :: pacified
  integer :: u
  u = given_output_unit (unit)
  call object%base_write (u, pacified = pacified)
  if (object%subevt_filled) then
     if (object%has_reweight) then
        call write_separator (u)
        write (u, "(1x,A)")  "Reweighting expression:"
```

```
         call write_separator (u)
         call object%reweight%write (u)
      end if
      if (object%has_analysis) then
         call write_separator (u)
         write (u, "(1x,A)")  "Analysis expression:"
         call write_separator (u)
         call object%analysis%write (u)
      end if
   end if
end subroutine event_expr_write
```

Initializer. This is required only for the `sqme_alt` and `weight_alt` arrays.

⟨*Subevt expr: event expr: TBP*⟩+≡

```
procedure :: init => event_expr_init
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine event_expr_init (expr, n_alt)
  class(event_expr_t), intent(out) :: expr
  integer, intent(in), optional :: n_alt
  if (present (n_alt)) then
     expr%n_alt = n_alt
     allocate (expr%sqme_alt (n_alt), source = 0._default)
     allocate (expr%weight_alt (n_alt), source = 0._default)
  end if
end subroutine event_expr_init
```

Define variables. We have the variables of the base type plus specific variables for full events. There is the event index.

⟨*Subevt expr: event expr: TBP*⟩+≡

```
procedure :: setup_vars => event_expr_setup_vars
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine event_expr_setup_vars (expr, sqrts)
  class(event_expr_t), intent(inout), target :: expr
  real(default), intent(in) :: sqrts
  call expr%base_setup_vars (sqrts)
  call var_list_append_string_ptr (expr%var_list, &
       var_str ("$process_id"), expr%id, &
       is_known = expr%has_id, &
       locked = .true., verbose = .false., intrinsic = .true.)
  call var_list_append_int_ptr (expr%var_list, &
       var_str ("process_num_id"), expr%num_id, &
       is_known = expr%has_num_id, &
       locked = .true., verbose = .false., intrinsic = .true.)
  call var_list_append_real_ptr (expr%var_list, &
       var_str ("sqme"), expr%sqme_prc, &
       is_known = expr%has_sqme_prc, &
       locked = .true., verbose = .false., intrinsic = .true.)
  call var_list_append_real_ptr (expr%var_list, &
       var_str ("sqme_ref"), expr%sqme_ref, &
       is_known = expr%has_sqme_ref, &
       locked = .true., verbose = .false., intrinsic = .true.)
  call var_list_append_int_ptr (expr%var_list, &
```

```
        var_str ("event_index"), expr%index, &
        is_known = expr%has_index, &
        locked = .true., verbose = .false., intrinsic = .true.)
    call var_list_append_real_ptr (expr%var_list, &
        var_str ("event_weight"), expr%weight_prc, &
        is_known = expr%has_weight_prc, &
        locked = .true., verbose = .false., intrinsic = .true.)
    call var_list_append_real_ptr (expr%var_list, &
        var_str ("event_weight_ref"), expr%weight_ref, &
        is_known = expr%has_weight_ref, &
        locked = .true., verbose = .false., intrinsic = .true.)
    call var_list_append_real_ptr (expr%var_list, &
        var_str ("event_excess"), expr%excess_prc, &
        is_known = expr%has_excess_prc, &
        locked = .true., verbose = .false., intrinsic = .true.)
  end subroutine event_expr_setup_vars
```

Compile the analysis expression. If the pointer is disassociated, there is no expression.

⟨*Subevt expr: event expr: TBP*⟩+≡

```
  procedure :: setup_analysis => event_expr_setup_analysis
```

⟨*Subevt expr: procedures*⟩+≡

```
  subroutine event_expr_setup_analysis (expr, ef_analysis)
    class(event_expr_t), intent(inout), target :: expr
    class(expr_factory_t), intent(in) :: ef_analysis
    call ef_analysis%build (expr%analysis)
    if (allocated (expr%analysis)) then
       call expr%setup_var_self ()
       call expr%analysis%setup_lexpr (expr%var_list)
       expr%has_analysis = .true.
    end if
  end subroutine event_expr_setup_analysis
```

Compile the reweight expression.

⟨*Subevt expr: event expr: TBP*⟩+≡

```
  procedure :: setup_reweight => event_expr_setup_reweight
```

⟨*Subevt expr: procedures*⟩+≡

```
  subroutine event_expr_setup_reweight (expr, ef_reweight)
    class(event_expr_t), intent(inout), target :: expr
    class(expr_factory_t), intent(in) :: ef_reweight
    call ef_reweight%build (expr%reweight)
    if (allocated (expr%reweight)) then
       call expr%setup_var_self ()
       call expr%reweight%setup_expr (expr%var_list)
       expr%has_reweight = .true.
    end if
  end subroutine event_expr_setup_reweight
```

Store the string or numeric process ID. This should be done during initialization.

⟨*Subevt expr: event expr: TBP*⟩+≡

```
  procedure :: set_process_id => event_expr_set_process_id
  procedure :: set_process_num_id => event_expr_set_process_num_id
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine event_expr_set_process_id (expr, id)
  class(event_expr_t), intent(inout) :: expr
  type(string_t), intent(in) :: id
  expr%id = id
  expr%has_id = .true.
end subroutine event_expr_set_process_id

subroutine event_expr_set_process_num_id (expr, num_id)
  class(event_expr_t), intent(inout) :: expr
  integer, intent(in) :: num_id
  expr%num_id = num_id
  expr%has_num_id = .true.
end subroutine event_expr_set_process_num_id
```

Reset / set the data that pertain to a particular event.

⟨*Subevt expr: event expr: TBP*⟩+≡

```
procedure :: reset => event_expr_reset
procedure :: set => event_expr_set
```

⟨*Subevt expr: procedures*⟩+≡

```
subroutine event_expr_reset (expr)
  class(event_expr_t), intent(inout) :: expr
  call expr%base_reset ()
  expr%has_sqme_ref = .false.
  expr%has_sqme_prc = .false.
  expr%has_sqme_alt = .false.
  expr%has_weight_ref = .false.
  expr%has_weight_prc = .false.
  expr%has_weight_alt = .false.
  expr%has_excess_prc = .false.
end subroutine event_expr_reset

subroutine event_expr_set (expr, &
     weight_ref, weight_prc, weight_alt, &
     excess_prc, &
     sqme_ref, sqme_prc, sqme_alt)
  class(event_expr_t), intent(inout) :: expr
  real(default), intent(in), optional :: weight_ref, weight_prc
  real(default), intent(in), optional :: excess_prc
  real(default), intent(in), optional :: sqme_ref, sqme_prc
  real(default), dimension(:), intent(in), optional :: sqme_alt, weight_alt
  if (present (sqme_ref)) then
     expr%has_sqme_ref = .true.
     expr%sqme_ref = sqme_ref
  end if
  if (present (sqme_prc)) then
     expr%has_sqme_prc = .true.
     expr%sqme_prc = sqme_prc
  end if
  if (present (sqme_alt)) then
     expr%has_sqme_alt = .true.
     expr%sqme_alt = sqme_alt
  end if
```

```
    if (present (weight_ref)) then
       expr%has_weight_ref = .true.
       expr%weight_ref = weight_ref
    end if
    if (present (weight_prc)) then
       expr%has_weight_prc = .true.
       expr%weight_prc = weight_prc
    end if
    if (present (weight_alt)) then
       expr%has_weight_alt = .true.
       expr%weight_alt = weight_alt
    end if
    if (present (excess_prc)) then
       expr%has_excess_prc = .true.
       expr%excess_prc = excess_prc
    end if
  end subroutine event_expr_set
```

Fill the event expression: take the particle data and kinematics from a `particle_set` object.

We allow the particle content to change for each event. Therefore, we set the event variables each time.

Also increment the event index; initialize it if necessary.

⟨*Subevt expr: event expr: TBP*⟩+≡

```
  procedure :: fill_subevt => event_expr_fill_subevt
```

⟨*Subevt expr: procedures*⟩+≡

```
  subroutine event_expr_fill_subevt (expr, particle_set)
    class(event_expr_t), intent(inout) :: expr
    type(particle_set_t), intent(in) :: particle_set
    call particle_set%to_subevt (expr%subevt_t)
    expr%sqrts_hat = subevt_get_sqrts_hat (expr%subevt_t)
    expr%n_in  = subevt_get_n_in  (expr%subevt_t)
    expr%n_out = subevt_get_n_out (expr%subevt_t)
    expr%n_tot = expr%n_in + expr%n_out
    expr%subevt_filled = .true.
    if (expr%has_index) then
       expr%index = expr%index + 1
    else
       expr%index = 1
       expr%has_index = .true.
    end if
  end subroutine event_expr_fill_subevt
```

Evaluate, if the event passes the selection. For absent expressions we take default values.

⟨*Subevt expr: event expr: TBP*⟩+≡

```
  procedure :: evaluate => event_expr_evaluate
```

⟨*Subevt expr: procedures*⟩+≡

```
  subroutine event_expr_evaluate (expr, passed, reweight, analysis_flag)
    class(event_expr_t), intent(inout) :: expr
    logical, intent(out) :: passed
```

```
    real(default), intent(out) :: reweight
    logical, intent(out) :: analysis_flag
    call expr%base_evaluate (passed)
    if (passed) then
       if (expr%has_reweight) then
          call expr%reweight%evaluate ()
          if (expr%reweight%is_known ()) then
             reweight = expr%reweight%get_real ()
          else
             call msg_error ("Evaluate reweight expression: &
                  &result undefined")
             reweight = 0
          end if
       else
          reweight = 1
       end if
       if (expr%has_analysis) then
          call expr%analysis%evaluate ()
          if (expr%analysis%is_known ()) then
             analysis_flag = expr%analysis%get_log ()
          else
             call msg_error ("Evaluate analysis expression: &
                  &result undefined")
             analysis_flag = .false.
          end if
       else
          analysis_flag = .true.
       end if
    end if
  end subroutine event_expr_evaluate
```

30.2 Parton states

A `parton_state_t` object contains the effective kinematics and dynamics of an elementary partonic interaction, with or without the beam/structure function state included. The type is abstract and has two distinct extensions. The `isolated_state_t` extension describes the isolated elementary interaction where the `int_eff` subobject contains the complex transition amplitude, exclusive in all quantum numbers. The particle content and kinematics describe the effective partonic state. The `connected_state_t` extension contains the partonic `subevt` and the expressions for cuts and scales which use it.

In the isolated state, the effective partonic interaction may either be identical to the hard interaction, in which case it is just a pointer to the latter. Or it may involve a rearrangement of partons, in which case we allocate it explicitly and flag this by `int_is_allocated`.

The `trace` evaluator contains the absolute square of the effective transition amplitude matrix, summed over final states. It is also summed over initial states, depending on the the beam setup allows. The result is used for integration.

The `matrix` evaluator is the counterpart of `trace` which is kept exclusive in all observable quantum numbers. The `flows` evaluator is furthermore exclusive

in colors, but neglecting all color interference. The `matrix` and `flows` evaluators are filled only for sampling points that become part of physical events.

Note: It would be natural to make the evaluators allocatable. However, this causes memory corruption in gfortran 4.6.3. The extra `has_XXX` flags indicate whether evaluators are active, instead.

This module contains no unit tests. The tests are covered by the `processes` module below.

⟨`parton_states.f90`⟩≡

```
⟨File header⟩
module parton_states

⟨Use kinds⟩
  use io_units
  use format_utils, only: write_separator
  use diagnostics
  use lorentz
  use subevents
  use variables
  use expr_base
  use model_data
  use flavors
  use helicities
  use colors
  use quantum_numbers
  use state_matrices
  use polarizations
  use interactions
  use evaluators

  use beams
  use sf_base
  use process_constants
  use prc_core
  use subevt_expr

⟨Standard module head⟩

⟨Parton states: public⟩

⟨Parton states: types⟩

contains

⟨Parton states: procedures⟩

end module parton_states
```

30.2.1 Abstract base type

The common part are the evaluators, one for the trace (summed over all quantum numbers), one for the transition matrix (summed only over unobservable quantum numbers), and one for the flow distribution (transition matrix without

interferences, exclusive in color flow).

⟨*Parton states: types*⟩≡
```
type, abstract :: parton_state_t
   logical :: has_trace = .false.
   logical :: has_matrix = .false.
   logical :: has_flows = .false.
   type(evaluator_t) :: trace
   type(evaluator_t) :: matrix
   type(evaluator_t) :: flows
 contains
 ⟨Parton states: parton state: TBP⟩
end type parton_state_t
```

The `isolated_state_t` extension contains the `sf_chain_eff` object and the (hard) effective interaction `int_eff`, separately, both implemented as a pointer. The evaluators (trace, matrix, flows) apply to the hard interaction only.

If the effective interaction differs from the hard interaction, the pointer is allocated explicitly. Analogously for `sf_chain_eff`.

⟨*Parton states: public*⟩≡
```
public :: isolated_state_t
```

⟨*Parton states: types*⟩+≡
```
type, extends (parton_state_t) :: isolated_state_t
   logical :: sf_chain_is_allocated = .false.
   type(sf_chain_instance_t), pointer :: sf_chain_eff => null ()
   logical :: int_is_allocated = .false.
   type(interaction_t), pointer :: int_eff => null ()
 contains
 ⟨Parton states: isolated state: TBP⟩
end type isolated_state_t
```

The `connected_state_t` extension contains all data that enable the evaluation of observables for the effective connected state. The evaluators connect the (effective) structure-function chain and hard interaction that were kept separate in the `isolated_state_t`.

The `flows_sf` evaluator is an extended copy of the structure-function

The `expr` subobject consists of the `subevt`, a simple event record, expressions for cuts etc. which refer to this record, and a `var_list` which contains event-specific variables, linked to the process variable list. Variables used within the expressions are looked up in `var_list`.

⟨*Parton states: public*⟩+≡
```
public :: connected_state_t
```

⟨*Parton states: types*⟩+≡
```
type, extends (parton_state_t) :: connected_state_t
   type(state_flv_content_t) :: state_flv
   logical :: has_flows_sf = .false.
   type(evaluator_t) :: flows_sf
   logical :: has_expr = .false.
   type(parton_expr_t) :: expr
 contains
 ⟨Parton states: connected state: TBP⟩
end type connected_state_t
```

Output: each evaluator is written only when it is active. The `sf_chain` is only written if it is explicitly allocated.

⟨*Parton states: parton state: TBP*⟩≡

```
procedure :: write => parton_state_write
```

⟨*Parton states: procedures*⟩≡

```
subroutine parton_state_write (state, unit, testflag)
  class(parton_state_t), intent(in) :: state
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  select type (state)
  class is (isolated_state_t)
     if (state%sf_chain_is_allocated) then
        call write_separator (u)
        call state%sf_chain_eff%write (u)
     end if
     if (state%int_is_allocated) then
        call write_separator (u)
        write (u, "(1x,A)") &
             "Effective interaction:"
        call write_separator (u)
        call state%int_eff%basic_write (u, testflag = testflag)
     end if
  class is (connected_state_t)
     if (state%has_flows_sf) then
        call write_separator (u)
        write (u, "(1x,A)") &
             "Evaluator (extension of the beam evaluator &
             &with color contractions):"
        call write_separator (u)
        call state%flows_sf%write (u, testflag = testflag)
     end if
  end select
  if (state%has_trace) then
     call write_separator (u)
     write (u, "(1x,A)") &
          "Evaluator (trace of the squared transition matrix):"
     call write_separator (u)
     call state%trace%write (u, testflag = testflag)
  end if
  if (state%has_matrix) then
     call write_separator (u)
     write (u, "(1x,A)") &
          "Evaluator (squared transition matrix):"
     call write_separator (u)
     call state%matrix%write (u, testflag = testflag)
  end if
  if (state%has_flows) then
     call write_separator (u)
     write (u, "(1x,A)") &
          "Evaluator (squared color-flow matrix):"
```

```
      call write_separator (u)
      call state%flows%write (u, testflag = testflag)
   end if
   select type (state)
   class is (connected_state_t)
      if (state%has_expr) then
         call write_separator (u)
         call state%expr%write (u)
      end if
   end select
end subroutine parton_state_write
```

Finalize interaction and evaluators, but only if allocated.

⟨*Parton states: parton state: TBP*⟩+≡

```
procedure :: final => parton_state_final
```

⟨*Parton states: procedures*⟩+≡

```
subroutine parton_state_final (state)
   class(parton_state_t), intent(inout) :: state
   if (state%has_flows) then
      call state%flows%final ()
      state%has_flows = .false.
   end if
   if (state%has_matrix) then
      call state%matrix%final ()
      state%has_matrix = .false.
   end if
   if (state%has_trace) then
      call state%trace%final ()
      state%has_trace = .false.
   end if
   select type (state)
   class is (connected_state_t)
      if (state%has_flows_sf) then
         call state%flows_sf%final ()
         state%has_flows_sf = .false.
      end if
      call state%expr%final ()
   class is (isolated_state_t)
      if (state%int_is_allocated) then
         call state%int_eff%final ()
         deallocate (state%int_eff)
         state%int_is_allocated = .false.
      end if
      if (state%sf_chain_is_allocated) then
         call state%sf_chain_eff%final ()
      end if
   end select
end subroutine parton_state_final
```

30.2.2 Common Initialization

Initialize the isolated parton state. In this version, the effective structure-function chain `sf_chain_eff` and the effective interaction `int_eff` both are trivial pointers to the seed structure-function chain and to the hard interaction, respectively.

⟨*Parton states: isolated state: TBP*⟩≡
```
procedure :: init => isolated_state_init
```

⟨*Parton states: procedures*⟩+≡
```
subroutine isolated_state_init (state, sf_chain, int)
  class(isolated_state_t), intent(out) :: state
  type(sf_chain_instance_t), intent(in), target :: sf_chain
  type(interaction_t), intent(in), target :: int
  state%sf_chain_eff => sf_chain
  state%int_eff => int
end subroutine isolated_state_init
```

30.2.3 Evaluator initialization: isolated state

Create an evaluator for the trace of the squared transition matrix. The trace goes over all outgoing quantum numbers. Whether we trace over incoming quantum numbers other than color, depends on the given `qn_mask_in`.

There are two options: explicitly computing the color factor table (`use_cf` false; `nc` defined), or taking the color factor table from the hard matrix element data.

⟨*Parton states: isolated state: TBP*⟩+≡
```
procedure :: setup_square_trace => isolated_state_setup_square_trace
```

⟨*Parton states: procedures*⟩+≡
```
subroutine isolated_state_setup_square_trace (state, core, &
     qn_mask_in, col, keep_fs_flavor)
  class(isolated_state_t), intent(inout), target :: state
  class(prc_core_t), intent(in) :: core
  type(quantum_numbers_mask_t), intent(in), dimension(:) :: qn_mask_in
  !!! Actually need allocatable attribute here fore once because col might
  !!! enter the subroutine non-allocated.
  integer, intent(in), dimension(:), allocatable :: col
  logical, intent(in) :: keep_fs_flavor
  type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
  associate (data => core%data)
    allocate (qn_mask (data%n_in + data%n_out))
    qn_mask( : data%n_in) = &
            quantum_numbers_mask (.false., .true., .false.) &
            .or. qn_mask_in
    qn_mask(data%n_in + 1 : ) = &
            quantum_numbers_mask (.not. keep_fs_flavor, .true., .true.)
    if (core%use_color_factors) then
       call state%trace%init_square (state%int_eff, qn_mask, &
            col_flow_index = data%cf_index, &
            col_factor = data%color_factors, &
            col_index_hi = col, &
            nc = core%nc)
```

```
      else
         call state%trace%init_square (state%int_eff, qn_mask, nc = core%nc)
      end if
    end associate
    state%has_trace = .true.
  end subroutine isolated_state_setup_square_trace
```

Setup an identity-evaluator for the trace. This implies that `me` is considered to be a squared amplitude, as for example for BLHA matrix elements.

⟨*Parton states: isolated state: TBP*⟩+≡

```
  procedure :: setup_identity_trace => isolated_state_setup_identity_trace
```

⟨*Parton states: procedures*⟩+≡

```
  subroutine isolated_state_setup_identity_trace (state, core, qn_mask_in, &
      keep_fs_flavors, keep_colors)
    class(isolated_state_t), intent(inout), target :: state
    class(prc_core_t), intent(in) :: core
    type(quantum_numbers_mask_t), intent(in), dimension(:) :: qn_mask_in
    logical, intent(in), optional :: keep_fs_flavors, keep_colors
    type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
    logical :: fs_flv_flag, col_flag
    fs_flv_flag = .true.; col_flag = .true.
    if (present(keep_fs_flavors)) fs_flv_flag = .not. keep_fs_flavors
    if (present(keep_colors)) col_flag = .not. keep_colors
    associate (data => core%data)
       allocate (qn_mask (data%n_in + data%n_out))
       qn_mask( : data%n_in) = &
          quantum_numbers_mask (.false., col_flag, .false.) .or. qn_mask_in
       qn_mask(data%n_in + 1 : ) = &
          quantum_numbers_mask (fs_flv_flag, col_flag, .true.)
    end associate
    call state%int_eff%set_mask (qn_mask)
    call state%trace%init_identity (state%int_eff)
    state%has_trace = .true.
  end subroutine isolated_state_setup_identity_trace
```

Setup the evaluator for the transition matrix, exclusive in helicities where this is requested.

For all unstable final-state particles we keep polarization according to the applicable decay options. If the process is a decay itself, this applies also to the initial state.

For all polarized final-state particles, we keep polarization including off-diagonal entries. We drop helicity completely for unpolarized final-state particles.

For the initial state, if the particle has not been handled yet, we apply the provided `qn_mask_in` which communicates the beam properties.

⟨*Parton states: isolated state: TBP*⟩+≡

```
  procedure :: setup_square_matrix => isolated_state_setup_square_matrix
```

⟨*Parton states: procedures*⟩+≡

```
  subroutine isolated_state_setup_square_matrix &
       (state, core, model, qn_mask_in, col)
    class(isolated_state_t), intent(inout), target :: state
```

```
    class(prc_core_t), intent(in) :: core
    class(model_data_t), intent(in), target :: model
    type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
    integer, dimension(:), intent(in) :: col
    type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
    type(flavor_t), dimension(:), allocatable :: flv
    integer :: i
    logical :: helmask, helmask_hd
    associate (data => core%data)
      allocate (qn_mask (data%n_in + data%n_out))
      allocate (flv (data%n_flv))
      do i = 1, data%n_in + data%n_out
         call flv%init (data%flv_state(i,:), model)
         if ((data%n_in == 1 .or. i > data%n_in) &
              .and. any (.not. flv%is_stable ())) then
            helmask = all (flv%decays_isotropically ())
            helmask_hd = all (flv%decays_diagonal ())
            qn_mask(i) = quantum_numbers_mask (.false., .true., helmask, &
                 mask_hd = helmask_hd)
         else if (i > data%n_in) then
            helmask = all (.not. flv%is_polarized ())
            qn_mask(i) = quantum_numbers_mask (.false., .true., helmask)
         else
            qn_mask(i) = quantum_numbers_mask (.false., .true., .false.) &
              .or. qn_mask_in(i)
         end if
      end do
      if (core%use_color_factors) then
         call state%matrix%init_square (state%int_eff, qn_mask, &
              col_flow_index = data%cf_index, &
              col_factor = data%color_factors, &
              col_index_hi = col, &
              nc = core%nc)
      else
         call state%matrix%init_square (state%int_eff, &
              qn_mask, &
              nc = core%nc)
      end if
    end associate
    state%has_matrix = .true.
  end subroutine isolated_state_setup_square_matrix
```

This procedure initializes the evaluator that computes the contributions to color flows, neglecting color interference. The incoming-particle mask can be used to sum over incoming flavor.

Helicity handling: see above.

⟨*Parton states: isolated state: TBP*⟩+≡

```
  procedure :: setup_square_flows => isolated_state_setup_square_flows
```

⟨*Parton states: procedures*⟩+≡

```
  subroutine isolated_state_setup_square_flows (state, core, model, qn_mask_in)
    class(isolated_state_t), intent(inout), target :: state
    class(prc_core_t), intent(in) :: core
    class(model_data_t), intent(in), target :: model
```

```
  type(quantum_numbers_mask_t), dimension(:), intent(in) :: qn_mask_in
  type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
  type(flavor_t), dimension(:), allocatable :: flv
  integer :: i
  logical :: helmask, helmask_hd
  associate (data => core%data)
    allocate (qn_mask (data%n_in + data%n_out))
    allocate (flv (data%n_flv))
    do i = 1, data%n_in + data%n_out
       call flv%init (data%flv_state(i,:), model)
       if ((data%n_in == 1 .or. i > data%n_in) &
            .and. any (.not. flv%is_stable ())) then
          helmask = all (flv%decays_isotropically ())
          helmask_hd = all (flv%decays_diagonal ())
          qn_mask(i) = quantum_numbers_mask (.false., .false., helmask, &
               mask_hd = helmask_hd)
       else if (i > data%n_in) then
          helmask = all (.not. flv%is_polarized ())
          qn_mask(i) = quantum_numbers_mask (.false., .false., helmask)
       else
          qn_mask(i) = quantum_numbers_mask (.false., .false., .false.) &
            .or. qn_mask_in(i)
       end if
    end do
    call state%flows%init_square (state%int_eff, qn_mask, &
         expand_color_flows = .true.)
  end associate
  state%has_flows = .true.
end subroutine isolated_state_setup_square_flows
```

30.2.4 Evaluator initialization: connected state

Setup a trace evaluator as a product of two evaluators (incoming state, effective interaction). In the result, all quantum numbers are summed over.

If the optional `int` interaction is provided, use this for the first factor in the convolution. Otherwise, use the final interaction of the stored `sf_chain`.

The `resonant` flag applies if we want to construct a decay chain. The resonance property can propagate to the final event output.

⟨*Parton states: connected state: TBP*⟩≡

```
procedure :: setup_connected_trace => connected_state_setup_connected_trace
```

⟨*Parton states: procedures*⟩+≡

```
subroutine connected_state_setup_connected_trace &
     (state, isolated, int, resonant, undo_helicities, n_sub, &
      keep_fs_flavors)
  class(connected_state_t), intent(inout), target :: state
  type(isolated_state_t), intent(in), target :: isolated
  type(interaction_t), intent(in), optional, target :: int
  logical, intent(in), optional :: resonant
  logical, intent(in), optional :: undo_helicities
  integer, intent(in), optional :: n_sub
  logical, intent(in), optional :: keep_fs_flavors
```

```
    type(quantum_numbers_mask_t) :: mask
    type(interaction_t), pointer :: src_int
    logical :: reduce, fs_flv_flag
    reduce = .false.; fs_flv_flag = .true.
    if (present (undo_helicities)) reduce = undo_helicities
    if (present (keep_fs_flavors)) fs_flv_flag = .not. keep_fs_flavors
    mask = quantum_numbers_mask (fs_flv_flag, .true., .true.)
    if (present (int)) then
       src_int => int
    else
       src_int => isolated%sf_chain_eff%get_out_int_ptr ()
    end if
    if (present (n_sub)) then
       if (n_sub > 0) call interaction_declare_subtraction (src_int, n_sub)
    end if
    if (reduce) call undo_qn_hel (src_int, mask, src_int%get_n_tot (), &
       isolated%sf_chain_eff%config%n_strfun == 0)

    call state%trace%init_product (src_int, isolated%trace, &
          qn_mask_conn = mask, &
          qn_mask_rest = mask, &
          connections_are_resonant = resonant)
    state%has_trace = .true.
  contains
    subroutine undo_qn_hel (int_in, mask, n_tot, replace_sf)
      type(interaction_t), intent(inout) :: int_in
      type(quantum_numbers_mask_t), intent(in) :: mask
      integer, intent(in) :: n_tot
      logical, intent(in) :: replace_sf
      type(quantum_numbers_mask_t), dimension(n_tot) :: mask_in
      mask_in = mask
      call int_in%set_mask (mask_in)
      if (replace_sf) call int_in%set_matrix_element (cmplx (1, 0, default))
    end subroutine undo_qn_hel
  end subroutine connected_state_setup_connected_trace
```

Setup a matrix evaluator as a product of two evaluators (incoming state, effective interation). In the intermediate state, color and helicity is summed over. In the final state, we keep the quantum numbers which are present in the original evaluators.

⟨*Parton states: connected state: TBP*⟩+≡

```
  procedure :: setup_connected_matrix => connected_state_setup_connected_matrix
```

⟨*Parton states: procedures*⟩+≡

```
  subroutine connected_state_setup_connected_matrix &
       (state, isolated, int, resonant, qn_filter_conn)
    class(connected_state_t), intent(inout), target :: state
    type(isolated_state_t), intent(in), target :: isolated
    type(interaction_t), intent(in), optional, target :: int
    logical, intent(in), optional :: resonant
    type(quantum_numbers_t), intent(in), optional :: qn_filter_conn
    type(quantum_numbers_mask_t) :: mask
    type(interaction_t), pointer :: src_int
    mask = quantum_numbers_mask (.false., .true., .true.)
```

```
    if (present (int)) then
       src_int => int
    else
       src_int => isolated%sf_chain_eff%get_out_int_ptr ()
    end if
    call state%matrix%init_product &
         (src_int, isolated%matrix, mask, &
          qn_filter_conn = qn_filter_conn, &
          connections_are_resonant = resonant)
    state%has_matrix = .true.
  end subroutine connected_state_setup_connected_matrix
```

Setup a matrix evaluator as a product of two evaluators (incoming state, effective interation). In the intermediate state, only helicity is summed over. In the final state, we keep the quantum numbers which are present in the original evaluators.

If the optional `int` interaction is provided, use this for the first factor in the convolution. Otherwise, use the final interaction of the stored `sf_chain`, after creating an intermediate interaction that includes a correlated color state. We assume that for a caller-provided `int`, this is not necessary.

⟨*Parton states: connected state: TBP*⟩+≡

```
  procedure :: setup_connected_flows => connected_state_setup_connected_flows
```

⟨*Parton states: procedures*⟩+≡

```
  subroutine connected_state_setup_connected_flows &
       (state, isolated, int, resonant, qn_filter_conn)
    class(connected_state_t), intent(inout), target :: state
    type(isolated_state_t), intent(in), target :: isolated
    type(interaction_t), intent(in), optional, target :: int
    logical, intent(in), optional :: resonant
    type(quantum_numbers_t), intent(in), optional :: qn_filter_conn
    type(quantum_numbers_mask_t) :: mask
    type(interaction_t), pointer :: src_int
    mask = quantum_numbers_mask (.false., .false., .true.)
    if (present (int)) then
       src_int => int
    else
       src_int => isolated%sf_chain_eff%get_out_int_ptr ()
       call state%flows_sf%init_color_contractions (src_int)
       state%has_flows_sf = .true.
       src_int => state%flows_sf%interaction_t
    end if
    call state%flows%init_product (src_int, isolated%flows, mask, &
          qn_filter_conn = qn_filter_conn, &
          connections_are_resonant = resonant)
    state%has_flows = .true.
  end subroutine connected_state_setup_connected_flows
```

Determine and store the flavor content for the connected state. This queries the `matrix` evaluator component, which should hold the requested flavor information.

⟨*Parton states: connected state: TBP*⟩+≡

```
  procedure :: setup_state_flv => connected_state_setup_state_flv
```

⟨*Parton states: procedures*⟩+≡

```
subroutine connected_state_setup_state_flv (state, n_out_hard)
  class(connected_state_t), intent(inout), target :: state
  integer, intent(in) :: n_out_hard
  call interaction_get_flv_content &
       (state%matrix%interaction_t, state%state_flv, n_out_hard)
end subroutine connected_state_setup_state_flv
```

Return the current flavor state object.

⟨*Parton states: connected state: TBP*⟩+≡

```
procedure :: get_state_flv => connected_state_get_state_flv
```

⟨*Parton states: procedures*⟩+≡

```
function connected_state_get_state_flv (state) result (state_flv)
  class(connected_state_t), intent(in) :: state
  type(state_flv_content_t) :: state_flv
  state_flv = state%state_flv
end function connected_state_get_state_flv
```

30.2.5 Cuts and expressions

Set up the `subevt` that corresponds to the connected interaction. The index arrays refer to the interaction.

We assign the particles as follows: the beam particles are the first two (decay process: one) entries in the trace evaluator. The incoming partons are identified by their link to the outgoing partons of the structure-function chain. The outgoing partons are those of the trace evaluator, which include radiated partons during the structure-function chain.

⟨*Parton states: connected state: TBP*⟩+≡

```
procedure :: setup_subevt => connected_state_setup_subevt
```

⟨*Parton states: procedures*⟩+≡

```
subroutine connected_state_setup_subevt (state, sf_chain, f_beam, f_in, f_out)
  class(connected_state_t), intent(inout), target :: state
  type(sf_chain_instance_t), intent(in), target :: sf_chain
  type(flavor_t), dimension(:), intent(in) :: f_beam, f_in, f_out
  integer :: n_beam, n_in, n_out, n_vir, n_tot, i, j
  integer, dimension(:), allocatable :: i_beam, i_in, i_out
  integer :: sf_out_i
  type(interaction_t), pointer :: sf_int
  sf_int => sf_chain%get_out_int_ptr ()
  n_beam = size (f_beam)
  n_in = size (f_in)
  n_out = size (f_out)
  n_vir = state%trace%get_n_vir ()
  n_tot = state%trace%get_n_tot ()
  allocate (i_beam (n_beam), i_in (n_in), i_out (n_out))
  i_beam = [(i, i = 1, n_beam)]
  do j = 1, n_in
     sf_out_i = sf_chain%get_out_i (j)
     i_in(j) = interaction_find_link &
          (state%trace%interaction_t, sf_int, sf_out_i)
```

```
    end do
    i_out = [(i, i = n_vir + 1, n_tot)]
    call state%expr%setup_subevt (state%trace%interaction_t, &
         i_beam, i_in, i_out, f_beam, f_in, f_out)
    state%has_expr = .true.
  end subroutine connected_state_setup_subevt
```

Initialize the variable list specific for this state/term. We insert event variables (`sqrts_hat`) and link the process variable list. The variable list acquires pointers to subobjects of `state`, which must therefore have a `target` attribute.

⟨*Parton states: connected state: TBP*⟩+≡

```
  procedure :: setup_var_list => connected_state_setup_var_list
```

⟨*Parton states: procedures*⟩+≡

```
  subroutine connected_state_setup_var_list (state, process_var_list, beam_data)
    class(connected_state_t), intent(inout), target :: state
    type(var_list_t), intent(in), target :: process_var_list
    type(beam_data_t), intent(in) :: beam_data
    call state%expr%setup_vars (beam_data%get_sqrts ())
    call state%expr%link_var_list (process_var_list)
  end subroutine connected_state_setup_var_list
```

Allocate the cut expression etc.

⟨*Parton states: connected state: TBP*⟩+≡

```
  procedure :: setup_cuts => connected_state_setup_cuts
  procedure :: setup_scale => connected_state_setup_scale
  procedure :: setup_fac_scale => connected_state_setup_fac_scale
  procedure :: setup_ren_scale => connected_state_setup_ren_scale
  procedure :: setup_weight => connected_state_setup_weight
```

⟨*Parton states: procedures*⟩+≡

```
  subroutine connected_state_setup_cuts (state, ef_cuts)
    class(connected_state_t), intent(inout), target :: state
    class(expr_factory_t), intent(in) :: ef_cuts
    call state%expr%setup_selection (ef_cuts)
  end subroutine connected_state_setup_cuts

  subroutine connected_state_setup_scale (state, ef_scale)
    class(connected_state_t), intent(inout), target :: state
    class(expr_factory_t), intent(in) :: ef_scale
    call state%expr%setup_scale (ef_scale)
  end subroutine connected_state_setup_scale

  subroutine connected_state_setup_fac_scale (state, ef_fac_scale)
    class(connected_state_t), intent(inout), target :: state
    class(expr_factory_t), intent(in) :: ef_fac_scale
    call state%expr%setup_fac_scale (ef_fac_scale)
  end subroutine connected_state_setup_fac_scale

  subroutine connected_state_setup_ren_scale (state, ef_ren_scale)
    class(connected_state_t), intent(inout), target :: state
    class(expr_factory_t), intent(in) :: ef_ren_scale
    call state%expr%setup_ren_scale (ef_ren_scale)
  end subroutine connected_state_setup_ren_scale
```

```
subroutine connected_state_setup_weight (state, ef_weight)
  class(connected_state_t), intent(inout), target :: state
  class(expr_factory_t), intent(in) :: ef_weight
  call state%expr%setup_weight (ef_weight)
end subroutine connected_state_setup_weight
```

Reset the expression object: invalidate the subevt.

⟨*Parton states: connected state: TBP*⟩+≡

```
procedure :: reset_expressions => connected_state_reset_expressions
```

⟨*Parton states: procedures*⟩+≡

```
subroutine connected_state_reset_expressions (state)
  class(connected_state_t), intent(inout) :: state
  if (state%has_expr)  call state%expr%reset ()
end subroutine connected_state_reset_expressions
```

30.2.6 Evaluation

Transfer momenta to the trace evaluator and fill the `subevt` with this effective kinematics, if applicable.

Note: we may want to apply a boost for the `subevt`.

⟨*Parton states: parton state: TBP*⟩+≡

```
procedure :: receive_kinematics => parton_state_receive_kinematics
```

⟨*Parton states: procedures*⟩+≡

```
subroutine parton_state_receive_kinematics (state)
  class(parton_state_t), intent(inout), target :: state
  if (state%has_trace) then
     call state%trace%receive_momenta ()
     select type (state)
     class is (connected_state_t)
        if (state%has_expr) then
           call state%expr%fill_subevt (state%trace%interaction_t)
        end if
     end select
  end if
end subroutine parton_state_receive_kinematics
```

Recover kinematics: We assume that the trace evaluator is filled with momenta. Send those momenta back to the sources, then fill the variables and subevent as above.

The incoming momenta of the connected state are not connected to the isolated state but to the beam interaction. Therefore, the incoming momenta within the isolated state do not become defined, yet. Instead, we reconstruct the beam (and ISR) momentum configuration.

⟨*Parton states: parton state: TBP*⟩+≡

```
procedure :: send_kinematics => parton_state_send_kinematics
```

⟨*Parton states: procedures*⟩+≡

```
subroutine parton_state_send_kinematics (state)
  class(parton_state_t), intent(inout), target :: state
  if (state%has_trace) then
     call interaction_send_momenta (state%trace%interaction_t)
     select type (state)
     class is (connected_state_t)
        call state%expr%fill_subevt (state%trace%interaction_t)
     end select
  end if
end subroutine parton_state_send_kinematics
```

Evaluate the expressions. The routine evaluates first the cut expression. If the event passes, it evaluates the other expressions. Where no expressions are defined, default values are inserted.

⟨*Parton states: connected state: TBP*⟩+≡

```
procedure :: evaluate_expressions => connected_state_evaluate_expressions
```

⟨*Parton states: procedures*⟩+≡

```
subroutine connected_state_evaluate_expressions (state, passed, &
     scale, fac_scale, ren_scale, weight, scale_forced, force_evaluation)
  class(connected_state_t), intent(inout) :: state
  logical, intent(out) :: passed
  real(default), intent(out) :: scale, fac_scale, ren_scale, weight
  real(default), intent(in), allocatable, optional :: scale_forced
  logical, intent(in), optional :: force_evaluation
  if (state%has_expr) then
     call state%expr%evaluate (passed, scale, fac_scale, ren_scale, weight, &
          scale_forced, force_evaluation)
  end if
end subroutine connected_state_evaluate_expressions
```

Evaluate the structure-function chain, if it is allocated explicitly. The argument is the factorization scale.

If the chain is merely a pointer, the chain should already be evaluated at this point.

⟨*Parton states: isolated state: TBP*⟩+≡

```
procedure :: evaluate_sf_chain => isolated_state_evaluate_sf_chain
```

⟨*Parton states: procedures*⟩+≡

```
subroutine isolated_state_evaluate_sf_chain (state, fac_scale)
  class(isolated_state_t), intent(inout) :: state
  real(default), intent(in) :: fac_scale
  if (state%sf_chain_is_allocated) call state%sf_chain_eff%evaluate (fac_scale)
end subroutine isolated_state_evaluate_sf_chain
```

Evaluate the trace.

⟨*Parton states: parton state: TBP*⟩+≡

```
procedure :: evaluate_trace => parton_state_evaluate_trace
```

⟨*Parton states: procedures*⟩+≡

```
subroutine parton_state_evaluate_trace (state)
  class(parton_state_t), intent(inout) :: state
```

```
    if (state%has_trace) call state%trace%evaluate ()
  end subroutine parton_state_evaluate_trace
```

⟨*Parton states: parton state: TBP*⟩+≡

```
  procedure :: evaluate_matrix => parton_state_evaluate_matrix
```

⟨*Parton states: procedures*⟩+≡

```
  subroutine parton_state_evaluate_matrix (state)
    class(parton_state_t), intent(inout) :: state
    if (state%has_matrix) call state%matrix%evaluate ()
  end subroutine parton_state_evaluate_matrix
```

Evaluate the extra evaluators that we need for physical events.

⟨*Parton states: parton state: TBP*⟩+≡

```
  procedure :: evaluate_event_data => parton_state_evaluate_event_data
```

⟨*Parton states: procedures*⟩+≡

```
  subroutine parton_state_evaluate_event_data (state, only_momenta)
    class(parton_state_t), intent(inout) :: state
    logical, intent(in), optional :: only_momenta
    logical :: only_mom
    only_mom = .false.; if (present (only_momenta)) only_mom = only_momenta
    select type (state)
    type is (connected_state_t)
       if (state%has_flows_sf) then
          call state%flows_sf%receive_momenta ()
          if (.not. only_mom) call state%flows_sf%evaluate ()
       end if
    end select
    if (state%has_matrix) then
       call state%matrix%receive_momenta ()
       if (.not. only_mom) call state%matrix%evaluate ()
    end if
    if (state%has_flows) then
       call state%flows%receive_momenta ()
       if (.not. only_mom) call state%flows%evaluate ()
    end if
  end subroutine parton_state_evaluate_event_data
```

Normalize the helicity density matrix by its trace, i.e., factor out the trace and put it into an overall normalization factor. The trace and flow evaluators are unchanged.

⟨*Parton states: parton state: TBP*⟩+≡

```
  procedure :: normalize_matrix_by_trace => &
       parton_state_normalize_matrix_by_trace
```

⟨*Parton states: procedures*⟩+≡

```
  subroutine parton_state_normalize_matrix_by_trace (state)
    class(parton_state_t), intent(inout) :: state
    if (state%has_matrix) call state%matrix%normalize_by_trace ()
  end subroutine parton_state_normalize_matrix_by_trace
```

30.2.7 Accessing the state

Three functions return a pointer to the event-relevant interactions.

⟨*Parton states: parton state: TBP*⟩+≡

```
procedure :: get_trace_int_ptr => parton_state_get_trace_int_ptr
procedure :: get_matrix_int_ptr => parton_state_get_matrix_int_ptr
procedure :: get_flows_int_ptr => parton_state_get_flows_int_ptr
```

⟨*Parton states: procedures*⟩+≡

```
function parton_state_get_trace_int_ptr (state) result (ptr)
  class(parton_state_t), intent(in), target :: state
  type(interaction_t), pointer :: ptr
  if (state%has_trace) then
     ptr => state%trace%interaction_t
  else
     ptr => null ()
  end if
end function parton_state_get_trace_int_ptr

function parton_state_get_matrix_int_ptr (state) result (ptr)
  class(parton_state_t), intent(in), target :: state
  type(interaction_t), pointer :: ptr
  if (state%has_matrix) then
     ptr => state%matrix%interaction_t
  else
     ptr => null ()
  end if
end function parton_state_get_matrix_int_ptr

function parton_state_get_flows_int_ptr (state) result (ptr)
  class(parton_state_t), intent(in), target :: state
  type(interaction_t), pointer :: ptr
  if (state%has_flows) then
     ptr => state%flows%interaction_t
  else
     ptr => null ()
  end if
end function parton_state_get_flows_int_ptr
```

Return the indices of the beam particles and the outgoing particles within the trace (and thus, matrix and flows) evaluator, respectively.

⟨*Parton states: connected state: TBP*⟩+≡

```
procedure :: get_beam_index => connected_state_get_beam_index
procedure :: get_in_index => connected_state_get_in_index
```

⟨*Parton states: procedures*⟩+≡

```
subroutine connected_state_get_beam_index (state, i_beam)
  class(connected_state_t), intent(in) :: state
  integer, dimension(:), intent(out) :: i_beam
  call state%expr%get_beam_index (i_beam)
end subroutine connected_state_get_beam_index

subroutine connected_state_get_in_index (state, i_in)
  class(connected_state_t), intent(in) :: state
  integer, dimension(:), intent(out) :: i_in
```

```
      call state%expr%get_in_index (i_in)
    end subroutine connected_state_get_in_index
```

⟨*Parton states: public*⟩+≡

```
    public :: refill_evaluator
```

⟨*Parton states: procedures*⟩+≡

```
    subroutine refill_evaluator (sqme, qn, flv_index, evaluator)
      complex(default), intent(in), dimension(:) :: sqme
      type(quantum_numbers_t), intent(in), dimension(:,:) :: qn
      integer, intent(in), dimension(:), optional :: flv_index
      type(evaluator_t), intent(inout) :: evaluator
      integer :: i, i_flv
      do i = 1, size (sqme)
         if (present (flv_index)) then
            i_flv = flv_index(i)
         else
            i_flv = i
         end if
         call evaluator%add_to_matrix_element (qn(:,i_flv), sqme(i), &
              match_only_flavor = .true.)
      end do
    end subroutine refill_evaluator
```

Return the number of outgoing (hard) particles for the state.

⟨*Parton states: parton state: TBP*⟩+≡

```
    procedure :: get_n_out => parton_state_get_n_out
```

⟨*Parton states: procedures*⟩+≡

```
    function parton_state_get_n_out (state) result (n)
      class(parton_state_t), intent(in), target :: state
      integer :: n
      n = state%trace%get_n_out ()
    end function parton_state_get_n_out
```

30.2.8 Unit tests

⟨parton_states_ut.f90⟩≡

```
  ⟨File header⟩

  module parton_states_ut
    use unit_tests
    use parton_states_uti

  ⟨Standard module head⟩

  ⟨Parton states: public test⟩

  contains

  ⟨Parton states: test driver⟩

  end module parton_states_ut
```

⟨parton_states_uti.f90⟩≡
⟨*File header*⟩

```
module parton_states_uti
```

⟨*Use kinds*⟩
⟨*Use strings*⟩

```
  use constants, only: zero
  use numeric_utils
  use flavors
  use colors
  use helicities
  use quantum_numbers
  use sf_base, only: sf_chain_instance_t
  use state_matrices, only: state_matrix_t
  use prc_template_me, only: prc_template_me_t
  use interactions, only: interaction_t
  use models, only: model_t, create_test_model
  use parton_states
```

⟨*Standard module head*⟩

⟨*Parton states: test declarations*⟩

```
contains
```

⟨*Parton states: tests*⟩

```
end module parton_states_uti
```

⟨*Parton states: public test*⟩≡

```
  public :: parton_states_test
```

⟨*Parton states: test driver*⟩≡

```
  subroutine parton_states_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```

⟨*Parton states: execute tests*⟩

```
  end subroutine parton_states_test
```

Test a simple isolated state

⟨*Parton states: execute tests*⟩≡

```
  call test (parton_states_1, "parton_states_1", &
       "Create a 2 -> 2 isolated state and compute trace", &
       u, results)
```

⟨*Parton states: test declarations*⟩≡

```
  public :: parton_states_1
```

⟨*Parton states: tests*⟩≡

```
  subroutine parton_states_1 (u)
    integer, intent(in) :: u
    type(state_matrix_t), allocatable :: state
    type(flavor_t), dimension(2) :: flv_in
```

```
type(flavor_t), dimension(2) :: flv_out1, flv_out2
type(flavor_t), dimension(4) :: flv_tot
type(helicity_t), dimension(4) :: hel
type(color_t), dimension(4) :: col
integer :: h1, h2, h3, h4
integer :: f
integer :: i
type(quantum_numbers_t), dimension(4) :: qn
type(prc_template_me_t) :: core
type(sf_chain_instance_t), target :: sf_chain
type(interaction_t), target :: int
type(isolated_state_t) :: isolated_state
integer :: n_states = 0
integer, dimension(:), allocatable :: col_flow_index
type(quantum_numbers_mask_t), dimension(2) :: qn_mask
integer, dimension(8) :: i_allowed_states
complex(default), dimension(8) :: me
complex(default) :: me_check_tot, me_check_1, me_check_2, me2
logical :: tmp1, tmp2
type(model_t), pointer :: test_model => null ()

write (u, "(A)") "* Test output: parton_states_1"
write (u, "(A)") "* Purpose: Test the standard parton states"
write (u, "(A)")

call flv_in%init ([11, -11])
call flv_out1%init ([1, -1])
call flv_out2%init ([2, -2])

write (u, "(A)") "* Using incoming flavors: "
call flavor_write_array (flv_in, u)
write (u, "(A)") "* Two outgoing flavor structures: "
call flavor_write_array (flv_out1, u)
call flavor_write_array (flv_out2, u)


write (u, "(A)") "* Initialize state matrix"
allocate (state)
call state%init ()

write (u, "(A)") "* Fill state matrix"
call col(3)%init ([1])
call col(4)%init ([-1])
do f = 1, 2
   do h1 = -1, 1, 2
      do h2 = -1, 1, 2
         do h3 = -1, 1, 2
            do h4 = -1, 1, 2
               n_states = n_states + 1
               call hel%init ([h1, h2, h3, h4], [h1, h2, h3, h4])
               if (f == 1) then
                  flv_tot = [flv_in, flv_out1]
               else
                  flv_tot = [flv_in, flv_out2]
```

```
               end if
               call qn%init (flv_tot, col, hel)
               call state%add_state (qn)
            end do
         end do
      end do
   end do
end do

!!! Two flavors, one color flow, 2 x 2 x 2 x 2 helicity configurations
!!! -> 32 states.
write (u, "(A)")
write (u, "(A,I2)") "* Generated number of states: ", n_states

call state%freeze ()

!!! Indices of the helicity configurations which are non-zero
i_allowed_states = [6, 7, 10, 11, 22, 23, 26, 27]
me = [cmplx (-1.89448E-5_default,  9.94456E-7_default, default), &
      cmplx (-8.37887E-2_default,  4.30842E-3_default, default), &
      cmplx (-1.99997E-1_default, -1.01985E-2_default, default), &
      cmplx ( 1.79717E-5_default,  9.27038E-7_default, default), &
      cmplx (-1.74859E-5_default,  8.78819E-7_default, default), &
      cmplx ( 1.67577E-1_default, -8.61683E-3_default, default), &
      cmplx ( 2.41331E-1_default,  1.23306E-2_default, default), &
      cmplx (-3.59435E-5_default, -1.85407E-6_default, default)]
me_check_tot = cmplx (zero, zero, default)
me_check_1 = cmplx (zero, zero, default)
me_check_2 = cmplx (zero, zero, default)
do i = 1, 8
   me2 = me(i) * conjg (me(i))
   me_check_tot = me_check_tot + me2
   if (i < 5) then
      me_check_1 = me_check_1 + me2
   else
      me_check_2 = me_check_2 + me2
   end if
   call state%set_matrix_element (i_allowed_states(i), me(i))
end do

!!! Don't forget the color factor
me_check_tot = 3._default * me_check_tot
me_check_1 = 3._default * me_check_1
me_check_2 = 3._default * me_check_2
write (u, "(A)")

write (u, "(A)") "* Setup interaction"
call int%basic_init (2, 0, 2, set_relations = .true.)
call int%set_state_matrix (state)

core%data%n_in = 2; core%data%n_out = 2
core%data%n_flv = 2
allocate (core%data%flv_state (4, 2))
core%data%flv_state (1, :) = [11, 11]
```

```
    core%data%flv_state (2, :) = [-11, -11]
    core%data%flv_state (3, :) = [1, 2]
    core%data%flv_state (4, :) = [-1, -2]
    core%use_color_factors = .false.
    core%nc = 3

    write (u, "(A)") "* Init isolated state"
    call isolated_state%init (sf_chain, int)
    !!! There is only one color flow.
    allocate (col_flow_index (n_states)); col_flow_index = 1
    call qn_mask%init (.false., .false., .true., mask_cg = .false.)
    write (u, "(A)") "* Give a trace to the isolated state"
    call isolated_state%setup_square_trace (core, qn_mask, col_flow_index, .false.)
    call isolated_state%evaluate_trace ()
    write (u, "(A)")
    write (u, "(A)", advance = "no") "* Squared matrix element correct: "
    write (u, "(L1)") nearly_equal (me_check_tot, &
       isolated_state%trace%get_matrix_element (1), rel_smallness = 0.00001_default)


    write (u, "(A)") "* Give a matrix to the isolated state"
    call create_test_model (var_str ("SM"), test_model)
    call isolated_state%setup_square_matrix (core, test_model, qn_mask, col_flow_index)
    call isolated_state%evaluate_matrix ()

    write (u, "(A)") "* Sub-matrixelements correct: "
    tmp1 = nearly_equal (me_check_1, &
       isolated_state%matrix%get_matrix_element (1), rel_smallness = 0.00001_default)
    tmp2 = nearly_equal (me_check_2, &
       isolated_state%matrix%get_matrix_element (2), rel_smallness = 0.00001_default)
    write (u, "(A,L1,A,L1)") "* 1: ", tmp1, ", 2: ", tmp2

    write (u, "(A)") "* Test output end: parton_states_1"
  end subroutine parton_states_1
```

30.3 Process component management

This module contains tools for managing and combining process components and matrix-element code and values, acting at a level below the actual process definition.

30.3.1 Abstract base type

The types introduced here are abstract base types.

⟨pcm_base.f90⟩≡
  ⟨*File header*⟩

```
  module pcm_base

    use diagnostics
```

```
  use prc_core_def
  use prc_core

⟨Standard module head⟩

⟨PCM base: public⟩

⟨PCM base: types⟩

⟨PCM base: interfaces⟩

contains

⟨PCM base: procedures⟩

end module pcm_base
```

This object may hold process and method-specific data, and it should allocate the corresponding manager instance.

⟨*PCM base: public*⟩≡
```
  public :: pcm_t
```

⟨*PCM base: types*⟩≡
```
  type, abstract :: pcm_t
     logical :: initialized = .false.
  contains
  ⟨PCM base: pcm: TBP⟩
  end type pcm_t
```

The factory method. We use the `inout` intent, so calling this again is an error.

⟨*PCM base: pcm: TBP*⟩≡
```
  procedure(pcm_allocate_instance), deferred :: allocate_instance
```

⟨*PCM base: interfaces*⟩≡
```
  abstract interface
     subroutine pcm_allocate_instance (pcm, instance)
       import
       class(pcm_t), intent(in) :: pcm
       class(pcm_instance_t), intent(inout), allocatable :: instance
     end subroutine pcm_allocate_instance
  end interface
```

⟨*PCM base: pcm: TBP*⟩+≡
```
  procedure(pcm_is_nlo), deferred :: is_nlo
```

⟨*PCM base: interfaces*⟩+≡
```
  abstract interface
     function pcm_is_nlo (pcm) result (is_nlo)
        import
        logical :: is_nlo
        class(pcm_t), intent(in) :: pcm
     end function pcm_is_nlo
  end interface
```

⟨*PCM base: pcm: TBP*⟩+≡
```
procedure(pcm_final), deferred :: final
```

⟨*PCM base: interfaces*⟩+≡
```
abstract interface
   subroutine pcm_final (pcm)
      import
      class(pcm_t), intent(inout) :: pcm
   end subroutine pcm_final
end interface
```

30.3.2 Manager instance

This object deals with the actual (squared) matrix element values.

⟨*PCM base: public*⟩+≡
```
public :: pcm_instance_t
```

⟨*PCM base: types*⟩+≡
```
type, abstract :: pcm_instance_t
  class(pcm_t), pointer :: config => null ()
contains
⟨PCM base: pcm instance: TBP⟩
end type pcm_instance_t
```

⟨*PCM base: pcm instance: TBP*⟩≡
```
procedure(pcm_instance_final), deferred :: final
```

⟨*PCM base: interfaces*⟩+≡
```
abstract interface
   subroutine pcm_instance_final (pcm_instance)
      import
      class(pcm_instance_t), intent(inout) :: pcm_instance
   end subroutine pcm_instance_final
end interface
```

⟨*PCM base: pcm instance: TBP*⟩+≡
```
procedure(pcm_instance_is_valid), deferred :: is_valid
```

⟨*PCM base: interfaces*⟩+≡
```
abstract interface
   function pcm_instance_is_valid (pcm_instance) result (valid)
      import
      logical :: valid
      class(pcm_instance_t), intent(in) :: pcm_instance
   end function pcm_instance_is_valid
end interface
```

⟨*PCM base: pcm instance: TBP*⟩+≡
```
procedure :: link_config => pcm_instance_link_config
```

⟨*PCM base: procedures*⟩≡
```
subroutine pcm_instance_link_config (pcm_instance, config)
   class(pcm_instance_t), intent(inout) :: pcm_instance
   class(pcm_t), intent(in), target :: config
   pcm_instance%config => config
end subroutine pcm_instance_link_config
```

30.4 Manager configuration

This object holds information about the cores used by the components and allocates the corresponding manager instance.

⟨core_manager.f90⟩≡
```
⟨File header⟩

module core_manager

  use format_utils, only: write_integer_array, write_separator
  use physics_defs, only: BORN, NLO_REAL
  use io_units
  use prc_core

⟨Standard module head⟩

⟨Core manager: public⟩

⟨Core manager: parameters⟩

⟨Core manager: types⟩

contains

⟨Core manager: procedures⟩

end module core_manager
```

Auxilliary container for polymorphic cores.

⟨*Core manager: types*⟩≡
```
type :: generic_core_t
  class(prc_core_t), allocatable :: core
end type  generic_core_t
```

⟨*Core manager: parameters*⟩≡
```
integer, parameter, public :: N_MAX_CORES = 100
```

⟨*Core manager: public*⟩≡
```
public :: core_manager_t
```

⟨*Core manager: types*⟩+≡
```
type :: core_manager_t
  integer, dimension(N_MAX_CORES) :: i_component_to_i_core = 0
  integer, dimension(N_MAX_CORES) :: i_core = 0
```

```
    logical, dimension(N_MAX_CORES) :: sub = .false.
    character(32), dimension(N_MAX_CORES) :: md5s = ""
    integer, dimension(N_MAX_CORES) :: nlo_type
    integer :: n_cores = 0
    integer, dimension(:), allocatable :: i_core_to_first_i_component
    type(generic_core_t), dimension(:), allocatable :: cores
    integer :: current_index = 1
  contains
  ⟨Core manager: cm: TBP⟩
  end type core_manager_t
```

⟨*Core manager: cm: TBP*⟩≡

```
  procedure :: register_new => cm_register_new
```

⟨*Core manager: procedures*⟩≡

```
  subroutine cm_register_new (cm, nlo_type, i_component, md5sum)
    class(core_manager_t), intent(inout) :: cm
    integer, intent(in) :: nlo_type, i_component
    character(32), intent(in) :: md5sum
    cm%nlo_type(cm%current_index) = nlo_type
    cm%md5s(cm%current_index) = md5sum
    cm%i_component_to_i_core(i_component) = cm%current_index
    cm%i_core(cm%current_index) = cm%current_index
    cm%current_index = cm%current_index + 1
  end subroutine cm_register_new
```

⟨*Core manager: cm: TBP*⟩+≡

```
  procedure :: register_existing => cm_register_existing
```

⟨*Core manager: procedures*⟩+≡

```
  subroutine cm_register_existing (cm, i_existing, i_component)
    class(core_manager_t), intent(inout) :: cm
    integer, intent(in) :: i_existing, i_component
    integer :: i_core
    i_core = cm%i_component_to_i_core(i_existing)
    cm%i_component_to_i_core(i_component) = i_core
  end subroutine cm_register_existing
```

⟨*Core manager: cm: TBP*⟩+≡

```
  procedure :: allocate_core_array => cm_allocate_core_array
```

⟨*Core manager: procedures*⟩+≡

```
  subroutine cm_allocate_core_array (cm)
    class(core_manager_t), intent(inout) :: cm
    cm%n_cores = count (cm%i_core > 0)
    allocate (cm%cores (cm%n_cores))
  end subroutine cm_allocate_core_array
```

⟨*Core manager: cm: TBP*⟩+≡

```
  procedure :: create_i_core_to_first_i_component &
     => core_manager_create_i_core_to_first_i_component
```

⟨*Core manager: procedures*⟩+≡

```
subroutine core_manager_create_i_core_to_first_i_component (cm, n_components)
  class(core_manager_t), intent(inout) :: cm
  integer, intent(in) :: n_components
  integer :: i, i_core
  allocate (cm%i_core_to_first_i_component (cm%n_cores))
  cm%i_core_to_first_i_component = 0
  do i = 1, n_components
     if (.not. any (cm%i_core_to_first_i_component == i)) then
        i_core = cm%i_component_to_i_core (i)
        cm%i_core_to_first_i_component(i_core) = i
     end if
  end do
end subroutine core_manager_create_i_core_to_first_i_component
```

⟨*Core manager: cm: TBP*⟩+≡

```
procedure :: allocate_core => cm_allocate_core
```

⟨*Core manager: procedures*⟩+≡

```
subroutine cm_allocate_core (cm, i_core, core_template)
  class(core_manager_t), intent(inout) :: cm
  integer, intent(in) :: i_core
  class(prc_core_t), intent(in) :: core_template
  allocate (cm%cores(i_core)%core, source = core_template)
end subroutine cm_allocate_core
```

⟨*Core manager: cm: TBP*⟩+≡

```
procedure :: get_core => cm_get_core
```

⟨*Core manager: procedures*⟩+≡

```
function cm_get_core (cm, i_core) result (core)
  class(prc_core_t), pointer :: core
  class(core_manager_t), intent(in), target :: cm
  integer, intent(in) :: i_core
  core => cm%cores(i_core)%core
end function cm_get_core
```

⟨*Core manager: cm: TBP*⟩+≡

```
procedure :: get_subtraction_core => cm_get_subtraction_core
```

⟨*Core manager: procedures*⟩+≡

```
function cm_get_subtraction_core (cm) result (core)
  class(prc_core_t), pointer :: core
  class(core_manager_t), intent(in), target :: cm
  integer :: i
  core => null ()
  do i = 1, cm%n_cores
     if (cm%sub(i)) then
        core => cm%cores(i)%core
        exit
     end if
  end do
end function cm_get_subtraction_core
```

⟨*Core manager: cm: TBP*⟩+≡

```
procedure :: get_flv_states => cm_get_flv_states
```

⟨*Core manager: procedures*⟩+≡

```
pure subroutine cm_get_flv_states (cm, flv_born, flv_real, n_in)
  class(core_manager_t), intent(in) :: cm
  integer, dimension(:,:), allocatable, intent(out) :: flv_born, flv_real
  integer, intent(out) :: n_in
  integer :: i
  do i = 1, cm%n_cores
     if (cm%nlo_type(i) == BORN) then
        if (.not. allocated (flv_born)) &
           allocate (flv_born (size (cm%cores(i)%core%data%flv_state, 1), &
              size (cm%cores(i)%core%data%flv_state, 2)))
        flv_born = cm%cores(i)%core%data%flv_state
        n_in = cm%cores(i)%core%data%n_in
     else if (cm%nlo_type(i) == NLO_REAL) then
        if (.not. allocated (flv_real)) &
           allocate (flv_real (size (cm%cores(i)%core%data%flv_state, 1), &
              size (cm%cores(i)%core%data%flv_state, 2)))
        flv_real = cm%cores(i)%core%data%flv_state
        n_in = cm%cores(i)%core%data%n_in
     end if
  end do
end subroutine cm_get_flv_states
```

⟨*Core manager: cm: TBP*⟩+≡

```
procedure :: core_is_radiation => cm_core_is_radiation
```

⟨*Core manager: procedures*⟩+≡

```
elemental function cm_core_is_radiation (cm, i_core) result (is_rad)
  logical :: is_rad
  class(core_manager_t), intent(in) :: cm
  integer, intent(in) :: i_core
  is_rad = cm%nlo_type(i_core) == NLO_REAL .and. .not. cm%sub(i_core)
end function cm_core_is_radiation
```

Asterisks denote the subtraction components

⟨*Core manager: cm: TBP*⟩+≡

```
procedure :: write => cm_write
```

⟨*Core manager: procedures*⟩+≡

```
subroutine cm_write (cm, unit)
  class(core_manager_t), intent(in) :: cm
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  write (u,"(1x,A,I1,A)") "component manager instance with ", cm%n_cores, " cores: "
  write (u,"(1x,A)") "(* denotes subtraction cores)"
  write (u,"(1x,A,L1)") 'Cores allocated? ', allocated (cm%cores)
  do i = 1, cm%n_cores
     write (u,"(1x,A,I1,A)", advance = "no") "Core nr. ", i, ": "
     if (cm%sub(i)) write (u,"(A)", advance = "no") "*"
     call cm%cores(i)%core%write_name (u)
```

```
    end do
    call write_separator (u, 1)
    write (u,"(1x,A)") "i_component -> i_core: "
    do i = 1, N_MAX_CORES
       if (cm%i_component_to_i_core(i) > 0) then
          write (u, "(I0, A)", advance = "no") cm%i_component_to_i_core(i), ", "
       else
          write (u, "(A)") ""
          exit
       end if
    end do
    call write_separator (u, 1)
    if (allocated (cm%i_core_to_first_i_component)) then
       write (u, "(1x,A)") "i_core -> i_component_first: "
       call write_integer_array (cm%i_core_to_first_i_component, &
          unit = u, n_max = cm%n_cores)
    else
       write (u, "(1X,A)") "cm%i_core_to_first_i_component: Not allocated."
    end if
    call write_separator (u, 1)
    write (u,"(1x,A)") "nlo type -> i_core: "
    call write_integer_array (cm%i_core, unit = u, n_max = cm%n_cores)
    call write_separator (u, 1)
    write (u, "(1x,A)") "Md5 sums: "
    do i = 1, cm%n_cores
       write (u, "(A,A)") cm%md5s(i)
    end do
    call write_separator (u, 1)
  end subroutine cm_write
```

⟨*Core manager: cm: TBP*⟩+≡

```
  procedure :: final => cm_final
```

⟨*Core manager: procedures*⟩+≡

```
  subroutine cm_final (cm)
    class(core_manager_t), intent(inout) :: cm
    cm%i_component_to_i_core = 0
    cm%n_cores = 0
    cm%i_core = 0
    cm%nlo_type = 0
    cm%md5s = ""
    cm%current_index = 1
    if (allocated (cm%cores)) deallocate (cm%cores)
    if (allocated (cm%i_core_to_first_i_component)) &
       deallocate (cm%i_core_to_first_i_component)
  end subroutine cm_final
```

30.5 The process object

⟨process.f90⟩≡

⟨*File header*⟩

```
module process

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_utils, only: write_separator
  use constants
  use diagnostics
  use numeric_utils
  use lorentz
  use cputime
  use md5
  use rng_base
  use os_interface
  use sm_qcd
  use integration_results
  use mci_base
  use flavors
  use model_data
  use models
  use physics_defs
  use process_libraries
  use process_constants
  use particles
  use variables
  use beam_structures
  use beams
  use interactions
  use pdg_arrays
  use expr_base
  use sf_base
  use sf_mappings
  use resonances, only: resonance_history_t

  use prc_test_core, only: test_t
  use prc_core, only: prc_core_t
  use prc_user_defined, only: prc_user_defined_base_t
  use prc_recola, only: prc_recola_t
  use blha_olp_interfaces, only: prc_blha_t, blha_template_t
  use prc_threshold, only: prc_threshold_t

  use phs_base
  use phs_wood, only: phs_wood_config_t
  use phs_wood, only: EXTENSION_NONE, EXTENSION_DEFAULT, EXTENSION_DGLAP
  use phs_fks, only: phs_fks_config_t, get_filtered_resonance_histories
  use blha_config, only: blha_master_t
  use nlo_data, only: FKS_DEFAULT, FKS_RESONANCES
  use nlo_data, only: fks_template_t

  use parton_states, only: connected_state_t
  use pcm_base
  use pcm
  use process_counter
```

```
  use core_manager
  use process_config
  use process_mci
```

⟨*Standard module head*⟩

⟨*Process: public*⟩

⟨*Process: types*⟩

```
contains
```

⟨*Process: procedures*⟩

```
end module process
```

A process object is the workspace for the process instance. After initialization, its contents are filled by integration passes which shape the integration grids and compute cross sections. Processes are set up initially from user-level configuration data. After calculating integrals and thus developing integration grid data, the program may use a process object or a copy of it for the purpose of generating events.

The process object consists of several subobjects with their specific purposes. The corresponding types are defined below. (Technically, the subobject type definitions have to come before the process type definition, but with NOWEB magic we reverse this order here.)

The `type` determines whether we are considering a decay or a scattering process.

The `meta` object describes the process and its environment. All contents become fixed when the object is initialized.

The `config` object holds physical and technical configuration data that have been obtained during process initialization, and which are common to all process components.

The individual process components are configured in the `component` objects. These objects contain more configuration parameters and workspace, as needed for the specific process variant.

The `term` objects describe parton configurations which are technically used as phase-space points. Each process component may split into several terms with distinct kinematics and particle content. Furthermore, each term may project on a different physical state, e.g., by particle recombination. The `term` object provides the framework for this projection, for applying cuts, weight, and thus completing the process calculation.

The `beam_config` object describes the incoming particles, either the decay mother or the scattering beams. It also contains the structure-function information.

The `mci_entry` objects configure a MC input parameter set and integrator, each. The number of parameters depends on the process component and on the beam and structure-function setup.

The `pcm` component is the process-component manager. This polymorphic object manages and hides the details of dealing with NLO processes where several components have to be combined in a non-trivial way. It also acts as an

abstract factory for the corresponding object in `process_instance`, which does the actual work for this matter.

⟨*Process: public*⟩≡
```
public :: process_t
```

⟨*Process: types*⟩≡
```
type :: process_t
   private
   type(process_metadata_t) :: &
        meta
   type(process_config_data_t) :: &
        config
   type(process_counter_t) :: &
        counter
   type(process_component_t), dimension(:), allocatable :: &
        component
   type(process_term_t), dimension(:), allocatable :: &
        term
   type(process_beam_config_t) :: &
        beam_config
   type(process_mci_entry_t), dimension(:), allocatable :: &
        mci_entry
   class(pcm_t), allocatable :: &
        pcm
   type(core_manager_t) :: cm
   logical, dimension(:), allocatable :: component_selected
 contains
 ⟨Process: process: TBP⟩
end type process_t
```

This procedure is an important debugging and inspection tool; it is not used during normal operation. The process object is written to a file (identified by unit, which may also be standard output). Optional flags determine whether we show everything or just the interesting parts.

⟨*Process: process: TBP*⟩≡
```
procedure :: write => process_write
```

⟨*Process: procedures*⟩≡
```
subroutine process_write (process, screen, unit, &
     show_all, show_var_list, &
     show_os_data, &
     show_rng_factory, show_model, show_expressions, &
     show_sfchain, &
     show_equivalences, show_history, show_histories, &
     show_forest, show_x, &
     show_subevt, show_evaluators, pacify)
  class(process_t), intent(in) :: process
  logical, intent(in) :: screen
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: show_all
  logical, intent(in), optional :: show_var_list
  logical, intent(in), optional :: show_os_data
  logical, intent(in), optional :: show_rng_factory
  logical, intent(in), optional :: show_model, show_expressions
```

```
logical, intent(in), optional :: show_sfchain
logical, intent(in), optional :: show_equivalences
logical, intent(in), optional :: show_history, show_histories
logical, intent(in), optional :: show_forest, show_x
logical, intent(in), optional :: show_subevt, show_evaluators
logical, intent(in), optional :: pacify
logical :: all
logical :: var_list
logical :: counters
logical :: os_data
logical :: rng_factory, model, expressions
integer :: u, i
u = given_output_unit (unit)
if (present (show_all)) then
   all = show_all
else
   all = .false.
end if
var_list = .false.
counters = .true.
os_data = .false.
model = .false.
rng_factory = .true.
expressions = .false.
if (present (show_var_list)) then
   all = .false.; var_list = show_var_list
end if
if (present (show_os_data)) then
   all = .false.; os_data = show_os_data
end if
if (present (show_rng_factory)) then
   all = .false.; rng_factory = show_rng_factory
end if
if (present (show_model)) then
   all = .false.; model = show_model
end if
if (present (show_expressions)) then
   all = .false.; expressions = show_expressions
end if
if (all) then
   var_list = .true.
   rng_factory = .true.
   model = .true.
   expressions = .true.
end if
if (screen) then
   write (msg_buffer, "(A)")  repeat ("-", 72)
   call msg_message ()
else
   call write_separator (u, 2)
end if
call process%meta%write (u, var_list, screen)
if (process%meta%type == PRC_UNKNOWN) then
   call write_separator (u, 2)
```

```
      return
   else
      if (.not. screen)  call write_separator (u)
   end if
   if (screen)  return
   call process%config%write &
        (u, counters, os_data, rng_factory, model, expressions)
   call write_separator (u, 2)
   if (allocated (process%component)) then
      write (u, "(1x,A)") "Process component configuration:"
      do i = 1, size (process%component)
         call write_separator (u)
         call process%component(i)%write (u)
      end do
   else
      write (u, "(1x,A)") "Process component configuration: [undefined]"
   end if
   call write_separator (u, 2)
   if (allocated (process%term)) then
      write (u, "(1x,A)") "Process term configuration:"
      do i = 1, size (process%term)
         call write_separator (u)
         call process%term(i)%write (u)
      end do
   else
      write (u, "(1x,A)") "Process term configuration: [undefined]"
   end if
   call write_separator (u, 2)
   call process%beam_config%write (u)
   call write_separator (u, 2)
   if (allocated (process%mci_entry)) then
      write (u, "(1x,A)") "Multi-channel integrator configurations:"
      do i = 1, size (process%mci_entry)
         call write_separator (u)
         write (u, "(1x,A,I0,A)")  "MCI #", i, ":"
         call process%mci_entry(i)%write (u, pacify)
      end do
   end if
   call write_separator (u, 2)
end subroutine process_write
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: write_meta => process_write_meta
```

⟨*Process: procedures*⟩+≡

```
subroutine process_write_meta (process, unit, testflag)
   class(process_t), intent(in) :: process
   integer, intent(in), optional :: unit
   logical, intent(in), optional :: testflag
   integer :: u, i
   u = given_output_unit (unit)
   select case (process%meta%type)
   case (PRC_UNKNOWN)
      write (u, "(1x,A)") "Process instance [undefined]"
```

```
      return
   case (PRC_DECAY)
      write (u, "(1x,A)", advance="no") "Process instance [decay]:"
   case (PRC_SCATTERING)
      write (u, "(1x,A)", advance="no") "Process instance [scattering]:"
   case default
      call msg_bug ("process_instance_write: undefined process type")
   end select
   write (u, "(1x,A,A,A)") "'", char (process%meta%id), "'"
   write (u, "(3x,A,A,A)") "Run ID = '", char (process%meta%run_id), "'"
   if (allocated (process%meta%component_id)) then
      write (u, "(3x,A)")  "Process components:"
      do i = 1, size (process%meta%component_id)
         if (process%component_selected (i)) then
            write (u, "(3x,'*')", advance="no")
         else
            write (u, "(4x)", advance="no")
         end if
         write (u, "(1x,I0,9A)")  i, ": '", &
              char (process%meta%component_id (i)), "':   ", &
              char (process%meta%component_description (i))
      end do
   end if
end subroutine process_write_meta
```

Screen output. Write a short account of the process configuration and the current results. The verbose version lists the components, the short version just the results.

⟨*Process: process: TBP*⟩+≡

```
procedure :: show => process_show
```

⟨*Process: procedures*⟩+≡

```
subroutine process_show (object, unit, verbose)
  class(process_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u
  logical :: verb
  u = given_output_unit (unit)
  verb = .true.;  if (present (verbose)) verb = verbose
  if (verb) then
     call object%meta%show (u, object%config%model%get_name ())
     select case (object%meta%type)
     case (PRC_DECAY)
        write (u, "(2x,A)", advance="no")  "Computed width ="
     case (PRC_SCATTERING)
        write (u, "(2x,A)", advance="no")  "Computed cross section ="
     case default;  return
     end select
  else
     write (u, "(A)", advance="no") char (object%meta%id)
     select case (object%meta%num_id)
     case (0)
        write (u, "(':')")
```

```
      case default
         write (u, "(1x,'(',I0,')',':')") object%meta%num_id
      end select
      write (u, "(2x)", advance="no")
   end if
   write (u, "(ES14.7,1x,'+-',ES9.2)", advance="no") &
        object%get_integral_tot (), object%get_error_tot ()
   select case (object%meta%type)
   case (PRC_DECAY)
      write (u, "(1x,A)")  "GeV"
   case (PRC_SCATTERING)
      write (u, "(1x,A)")  "fb"
   case default
      write (u, *)
   end select
end subroutine process_show
```

Finalizer. Explicitly iterate over all subobjects that may contain allocated pointers.

⟨*Process: process: TBP*⟩+≡

```
procedure :: final => process_final
```

⟨*Process: procedures*⟩+≡

```
subroutine process_final (process)
   class(process_t), intent(inout) :: process
   integer :: i
   call process%meta%final ()
   call process%config%final ()
   if (allocated (process%component)) then
      do i = 1, size (process%component)
         call process%component(i)%final ()
      end do
   end if
   if (allocated (process%term)) then
      do i = 1, size (process%term)
         call process%term(i)%final ()
      end do
   end if
   call process%beam_config%final ()
   if (allocated (process%mci_entry)) then
      do i = 1, size (process%mci_entry)
         call process%mci_entry(i)%final ()
      end do
   end if
   call process%cm%final ()
   if (allocated (process%pcm)) then
      call process%pcm%final ()
      deallocate (process%pcm)
   end if
end subroutine process_final
```

Process setup

Initialize a process. We need a process library, the name of the process, and a run ID. The model is imported via its pointer, and the original pointer freed.

⟨*Process: process: TBP*⟩+≡

```
procedure :: init => process_init
```

⟨*Process: procedures*⟩+≡

```
subroutine process_init (process, proc_id, run_id, &
     lib, os_data, qcd, rng_factory, model)
  class(process_t), intent(out) :: process
  type(string_t), intent(in) :: proc_id
  type(string_t), intent(in) :: run_id
  type(process_library_t), intent(in), target :: lib
  type(os_data_t), intent(in) :: os_data
  type(qcd_t), intent(in) :: qcd
  class(rng_factory_t), intent(inout), allocatable :: rng_factory
  class(model_data_t), intent(inout), pointer :: model
  if (.not. lib%is_active ()) then
     call msg_bug ("Process init: inactive library not handled yet")
  end if
  if (.not. lib%contains (proc_id)) then
     call msg_fatal ("Process library doesn't contain process '" &
          // char (proc_id) // "'")
     return
  end if
  associate (meta => process%meta)
    call meta%init (proc_id, run_id, lib)
    call process%config%init &
         (meta, os_data, qcd, rng_factory, model)
    allocate (process%component (meta%n_components))
    allocate (process%component_selected (meta%n_components))
    process%component_selected = .false.
  end associate
  if (.not. lib%get_nlo_process (proc_id)) then
     allocate (pcm_default_t :: process%pcm)
  else
     allocate (pcm_nlo_t :: process%pcm)
  end if
end subroutine process_init
```

Store a snapshot of the common variable list.

⟨*Process: process: TBP*⟩+≡

```
procedure :: set_var_list => process_set_var_list
```

⟨*Process: procedures*⟩+≡

```
subroutine process_set_var_list (process, var_list)
  class(process_t), intent(inout) :: process
  type(var_list_t), intent(in) :: var_list
  call var_list_init_snapshot &
       (process%meta%var_list, var_list, follow_link=.true.)
end subroutine process_set_var_list
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: core_manager_register => process_core_manager_register
procedure :: core_manager_register_default => process_core_manager_register_default
procedure :: core_manager_register_sub => process_core_manager_register_sub
```

⟨*Process: procedures*⟩+≡

```
subroutine process_core_manager_register (process, &
   nlo_type, i_component, type_string)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: nlo_type, i_component
  type(string_t), intent(in), optional :: type_string
  if (nlo_type == NLO_SUBTRACTION) then
     call process%core_manager_register_sub (nlo_type, i_component, type_string)
  else
     call process%core_manager_register_default (nlo_type, i_component, type_string)
  end if
end subroutine process_core_manager_register

subroutine process_core_manager_register_sub (process, nlo_type, i_component, type_string)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: nlo_type, i_component
  type(string_t), intent(in), optional :: type_string
  character(32) :: md5sum
  integer :: i
  md5sum = process%get_md5sum_constants (i_component, type_string, nlo_type)
  if (any (process%cm%md5s == md5sum)) then
     do i = 1, N_MAX_CORES
        if (process%cm%i_core(i) == 0) exit
        if (md5sum == process%cm%md5s(i)) then
          process%cm%sub(i) = .true.
       end if
     end do
  else
     process%cm%sub(process%cm%current_index) = .true.
     call process%cm%register_new (nlo_type, i_component, md5sum)
  end if
end subroutine process_core_manager_register_sub

subroutine process_core_manager_register_default &
   (process, nlo_type, i_component, type_string)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: nlo_type, i_component
  type(string_t), intent(in), optional :: type_string
  character(32) :: md5sum
  integer :: i
  logical :: check
  md5sum = process%get_md5sum_constants (i_component, type_string, nlo_type)
  check = .false.
  associate (cm => process%cm)
     if (.not. any (cm%md5s == md5sum)) then
        call cm%register_new (nlo_type, i_component, md5sum)
     else
        do i = 1, N_MAX_CORES
           if (cm%md5s(i) == md5sum) then
              call cm%register_existing (i, i_component)
```

```
                check = .true.
                exit
             end if
          end do
          if (.not. check) call msg_fatal ("Register core: Inconsistency encountered!")
       end if
    end associate
  end subroutine process_core_manager_register_default
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: allocate_cm_arrays => process_allocate_cm_arrays
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_allocate_cm_arrays (process, n_components)
    class(process_t), intent(inout) :: process
    integer, intent(in) :: n_components
    call process%cm%allocate_core_array ()
    call process%cm%create_i_core_to_first_i_component (n_components)
  end subroutine process_allocate_cm_arrays
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: allocate_core => process_allocate_core
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_allocate_core (process, i_core, core_template)
    class(process_t), intent(inout) :: process
    integer, intent(in) :: i_core
    class(prc_core_t), intent(in) :: core_template
    call process%cm%allocate_core (i_core, core_template)
  end subroutine process_allocate_core
```

Initialize the process components, one by one, using a template for the process core object. The template is taken only for allocating the correct type; the contents are set by extracting the process entry from the library.

⟨*Process: process: TBP*⟩+≡

```
  procedure :: init_component => process_init_component
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_init_component &
       (process, index, active, mci_template, phs_config_template)
    class(process_t), intent(inout), target :: process
    integer, intent(in) :: index
    logical, intent(in) :: active
    class(mci_t), intent(in), allocatable :: mci_template
    class(phs_config_t), intent(in), allocatable :: phs_config_template
    type(process_constants_t) :: data
    call process%meta%lib%fill_constants (process%meta%id, index, data)
    associate (component => process%component(index))
       call component%init (index, &
            process%meta, process%config, &
            active, data, &
            mci_template, phs_config_template)
       if (.not. component%active .and. &
            component%config%get_nlo_type () /= NLO_SUBTRACTION) &
```

```
            call process%meta%deactivate_component(index)
      end associate
    end subroutine process_init_component
```

Determine the process terms for each process component.

⟨*Process: process: TBP*⟩+≡

```
    procedure :: setup_terms => process_setup_terms
```

⟨*Process: procedures*⟩+≡

```
    subroutine process_setup_terms (process, with_beams)
      class(process_t), intent(inout) :: process
      logical, intent(in), optional :: with_beams
      class(model_data_t), pointer :: model
      integer :: i, j, k, i_term
      integer, dimension(:), allocatable :: n_entry
      integer :: n_components, n_tot
      integer :: i_sub = 0
      type(string_t) :: subtraction_method
      class(prc_core_t), pointer :: core => null ()
      logical :: setup_subtraction_component
      model => process%config%model
      n_components = process%meta%n_components
      allocate (n_entry (n_components), source = 0)
      do i = 1, n_components
         associate (component => process%component(i))
           if (component%active) then
              n_entry(i) = 1
              if (component%get_nlo_type () == NLO_REAL) then
                 select type (pcm => process%pcm)
                 type is (pcm_nlo_t)
                    if (component%component_type /= COMP_REAL_FIN) &
                         n_entry(i) = n_entry(i) + pcm%region_data%get_n_phs ()
                 end select
              end if
           end if
         end associate
      end do
      n_tot = sum (n_entry)
      allocate (process%term (n_tot))
      k = 0
      if (process%is_nlo_calculation ()) then
         i_sub = process%component(1)%config%get_associated_subtraction ()
         subtraction_method = process%component(i_sub)%config%get_me_method ()
      end if

      do i = 1, n_components
         associate (component => process%component(i))
           if (.not. component%active)  cycle
             allocate (component%i_term (n_entry(i)))
             do j = 1, n_entry(i)
                setup_subtraction_component = &
                     (component%get_nlo_type () == NLO_REAL &
                      .and. component%component_type /= COMP_REAL_FIN &
                      .and. j == n_entry(i))
```

```
            i_term = k + j
            component%i_term(j) = i_term
            process%term(i_term)%i_sub = k + n_entry(i)
            process%term(i_term)%i_core = set_i_core (i, component%get_nlo_type (), &
                 setup_subtraction_component, component%config%get_def_type_string ())
            if (process%term(i_term)%i_core == 0) call msg_fatal ("Core not found!")
            core => process%get_core_term (i_term)
            if (i_sub > 0) then
               select type (pcm => process%pcm)
               type is (pcm_nlo_t)
                  call process%term(i_term)%init (i_term, i, j, core, model, &
                       nlo_type = component%config%get_nlo_type (), &
                       use_beam_pol = with_beams, &
                       subtraction_method = subtraction_method)
               class default
                  call process%term(i_term)%init (i_term, i, j, core, model, &
                       nlo_type = component%config%get_nlo_type (), &
                       use_beam_pol = with_beams, &
                       subtraction_method = subtraction_method)
               end select
            else
               call process%term(i_term)%init (i_term, i, j, core, model, &
                    nlo_type = component%config%get_nlo_type (), &
                    use_beam_pol = with_beams)
            end if
         end do
      end associate
      k = k + n_entry(i)
   end do
   process%config%n_terms = n_tot
contains
   function set_i_core (i_component, nlo_type, sub, type_string) result (i_core)
      integer :: i_core
      integer, intent(in) :: i_component, nlo_type
      logical, intent(in) :: sub
      type(string_t), intent(in) :: type_string
      character(32) :: md5sum
      integer :: index
      i_core = 0
      md5sum = process%get_md5sum_constants (i_component, type_string, nlo_type)
      do index = 1, N_MAX_CORES
         if (sub) then
            if (process%cm%sub(index)) then
               i_core = index
               exit
            end if
         else
            i_core = process%cm%i_core(index)
            if (process%cm%md5s(index) == md5sum) then
               i_core = index
               exit
            end if
         end if
      end do
```

```
    end function set_i_core
  end subroutine process_setup_terms
```

Initialize the beam setup. This is the trivial version where the incoming state of the matrix element coincides with the initial state of the process. For a scattering process, we need the c.m. energy, all other variables are set to their default values (no polarization, lab frame and c.m. frame coincide, etc.)

We assume that all components consistently describe a scattering process, i.e., two incoming particles.

Note: The current layout of the `beam_data_t` record requires that the flavor for each beam is unique. For processes with multiple flavors in the initial state, one has to set up beams explicitly. This restriction could be removed by extending the code in the `beams` module.

⟨*Process: process: TBP*⟩+≡

```
  procedure :: setup_beams_sqrts => process_setup_beams_sqrts
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_setup_beams_sqrts (process, sqrts, beam_structure, i_core)
    class(process_t), intent(inout) :: process
    real(default), intent(in) :: sqrts
    type(beam_structure_t), intent(in), optional :: beam_structure
    integer, intent(in), optional :: i_core
    type(pdg_array_t), dimension(:,:), allocatable :: pdg_in
    integer, dimension(2) :: pdg_scattering
    type(flavor_t), dimension(2) :: flv_in
    integer :: i, i0, ic
    class(prc_core_t), pointer :: core => null ()
    allocate (pdg_in (2, process%meta%n_components))
    i0 = 0
    do i = 1, process%meta%n_components
       if (process%component(i)%active) then
          if (present (i_core)) then
             ic = i_core
          else
             ic = process%cm%i_component_to_i_core (i)
          end if
          core => process%cm%get_core (ic)
          pdg_in(:,i) = core%data%get_pdg_in ()
          if (i0 == 0)  i0 = i
       end if
    end do
    do i = 1, process%meta%n_components
       if (.not. process%component(i)%active) then
          pdg_in(:,i) = pdg_in(:,i0)
       end if
    end do
    if (all (pdg_array_get_length (pdg_in) == 1) .and. &
         all (pdg_in(1,:) == pdg_in(1,i0)) .and. &
         all (pdg_in(2,:) == pdg_in(2,i0))) then
       pdg_scattering = pdg_array_get (pdg_in(:,i0), 1)
       call flv_in%init (pdg_scattering, process%config%model)
       call process%beam_config%init_scattering (flv_in, sqrts, beam_structure)
    else
```

```
      call msg_fatal ("Setting up process '" // char (process%meta%id) // "':", &
          [var_str ("   ---------------------------------------------"), &
           var_str ("Inconsistent initial state. This happens if either "), &
           var_str ("several processes with non-matching initial states "), &
           var_str ("have been added, or for a single process with an "), &
           var_str ("initial state flavor sum. In that case, please set beams "), &
           var_str ("explicitly [singling out a flavor / structure function.]")])
    end if
  end subroutine process_setup_beams_sqrts
```

This is the version that applies to decay processes. The energy is the particle mass, hence no extra argument.

⟨*Process: process: TBP*⟩+≡

```
  procedure :: setup_beams_decay => process_setup_beams_decay
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_setup_beams_decay (process, rest_frame, beam_structure, i_core)
    class(process_t), intent(inout) :: process
    logical, intent(in), optional :: rest_frame
    type(beam_structure_t), intent(in), optional :: beam_structure
    integer, intent(in), optional :: i_core
    type(pdg_array_t), dimension(:,:), allocatable :: pdg_in
    integer, dimension(1) :: pdg_decay
    type(flavor_t), dimension(1) :: flv_in
    integer :: i, i0, ic
    class(prc_core_t), pointer :: core => null ()
    allocate (pdg_in (1, process%meta%n_components))
    i0 = 0
    do i = 1, process%meta%n_components
       if (process%component(i)%active) then
          if (present (i_core)) then
             ic = i_core
          else
             ic = process%cm%i_component_to_i_core (i)
          end if
          core => process%cm%get_core (ic)
          pdg_in(:,i) = core%data%get_pdg_in ()
          if (i0 == 0)  i0 = i
       end if
    end do
    do i = 1, process%meta%n_components
       if (.not. process%component(i)%active) then
          pdg_in(:,i) = pdg_in(:,i0)
       end if
    end do
    if (all (pdg_array_get_length (pdg_in) == 1) &
         .and. all (pdg_in(1,:) == pdg_in(1,i0))) then
       pdg_decay = pdg_array_get (pdg_in(:,i0), 1)
       call flv_in%init (pdg_decay, process%config%model)
       call process%beam_config%init_decay (flv_in, rest_frame, beam_structure)
    else
       call msg_fatal ("Setting up decay '" &
            // char (process%meta%id) // "': decaying particle not unique")
    end if
```

```
end subroutine process_setup_beams_decay
```

We have to make sure that the masses of the various flavors in a given position in the particle string coincide.

⟨*Process: process: TBP*⟩+≡

```
procedure :: check_masses => process_check_masses
```

⟨*Process: procedures*⟩+≡

```
subroutine process_check_masses (process)
     class(process_t), intent(in) :: process
     type(flavor_t), dimension(:), allocatable :: flv
     real(default), dimension(:), allocatable :: mass
     !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
     integer :: i, j, k
     integer :: i_component
     class(prc_core_t), pointer :: core => null ()
     do i = 1, process%get_n_terms ()
        i_component = process%term(i)%i_component
        if (.not. process%component(i_component)%active)  cycle
        core => process%get_core_term (i)
        associate (data => core%data)
          allocate (flv (data%n_flv), mass (data%n_flv))
          do j = 1, data%n_in + data%n_out
             call flv%init (data%flv_state(j,:), process%config%model)
             !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
             do k = 1, data%n_flv
                mass(k) = flv(k)%get_mass ()
             end do
             if (any (.not. nearly_equal(mass, mass(1)))) then
                call msg_fatal ("Process '" // char (process%meta%id) // "': " &
                     // "mass values in flavor combination do not coincide. ")
             end if
          end do
          deallocate (flv, mass)
        end associate
     end do
 end subroutine process_check_masses
```

For some structure functions we need to get the list of initial state flavors. This is a two-dimensional array. The first index is the beam index, the second index is the component index. Each array element is itself a PDG array object, which consists of the list of incoming PDG values for this beam and component.

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_pdg_in => process_get_pdg_in
```

⟨*Process: procedures*⟩+≡

```
subroutine process_get_pdg_in (process, pdg_in)
  class(process_t), intent(in) :: process
  type(pdg_array_t), dimension(:,:), allocatable, intent(out) :: pdg_in
  integer :: i, i_core
  class(prc_core_t), pointer :: core => null ()
  allocate (pdg_in (process%config%n_in, process%meta%n_components))
  do i = 1, process%meta%n_components
```

```
      if (process%component(i)%active) then
         i_core = process%cm%i_component_to_i_core (i)
         core => process%cm%get_core (i_core)
         pdg_in(:,i) = core%data%get_pdg_in ()
      end if
   end do
   core => null ()
end subroutine process_get_pdg_in
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_phs_config => process_get_phs_config
```

⟨*Process: procedures*⟩+≡

```
function process_get_phs_config (process, i_component) result (phs_config)
   class(phs_config_t), pointer :: phs_config
   class(process_t), intent(in), target :: process
   integer, intent(in) :: i_component
   if (allocated (process%component)) then
      phs_config => process%component(i_component)%phs_config
   else
      phs_config => null ()
   end if
end function process_get_phs_config
```

Initialize from a complete beam setup. If the beam setup doesn't apply directly to the process, choose a fallback option as a straight scattering or decay process.

⟨*Process: process: TBP*⟩+≡

```
procedure :: setup_beams_beam_structure => process_setup_beams_beam_structure
```

⟨*Process: procedures*⟩+≡

```
subroutine process_setup_beams_beam_structure &
     (process, beam_structure, sqrts, decay_rest_frame)
   class(process_t), intent(inout) :: process
   type(beam_structure_t), intent(in) :: beam_structure
   real(default), intent(in) :: sqrts
   logical, intent(in), optional :: decay_rest_frame
   integer :: n_in
   logical :: applies
   n_in = process%get_n_in ()
   call beam_structure%check_against_n_in (process%get_n_in (), applies)
   if (applies) then
      call process%beam_config%init_beam_structure &
           (beam_structure, sqrts, process%get_model_ptr (), decay_rest_frame)
   else if (n_in == 2) then
      call process%setup_beams_sqrts (sqrts, beam_structure)
   else
      call process%setup_beams_decay (decay_rest_frame, beam_structure)
   end if
end subroutine process_setup_beams_beam_structure
```

Notify the user about beam setup.

⟨*Process: process: TBP*⟩+≡

```
procedure :: beams_startup_message => process_beams_startup_message
```

⟨*Process: procedures*⟩+≡

```
subroutine process_beams_startup_message (process, unit, beam_structure)
  class(process_t), intent(in) :: process
  integer, intent(in), optional :: unit
  type(beam_structure_t), intent(in), optional :: beam_structure
  call process%beam_config%startup_message (unit, beam_structure)
end subroutine process_beams_startup_message
```

We complete the kinematics configuration after the beam setup, but before we configure the chain of structure functions. The reason is that we need the total energy `sqrts` for the kinematics, but the structure-function setup requires the number of channels, which depends on the kinematics configuration. For instance, the kinematics module may return the need for parameterizing an s-channel resonance.

⟨*Process: process: TBP*⟩+≡

```
procedure :: configure_phs => process_configure_phs
```

⟨*Process: procedures*⟩+≡

```
subroutine process_configure_phs (process, rebuild, ignore_mismatch, &
   combined_integration)
  class(process_t), intent(inout) :: process
  logical, intent(in), optional :: rebuild
  logical, intent(in), optional :: ignore_mismatch
  logical, intent(in), optional :: combined_integration
  real(default) :: sqrts
  integer :: i, i_born
  class(phs_config_t), pointer :: phs_config_born
  sqrts = process%get_sqrts ()
  do i = 1, process%meta%n_components
     associate (component => process%component(i))
       if (component%active) then
          select type (pcm => process%pcm)
          type is (pcm_default_t)
             call component%configure_phs (sqrts, process%beam_config, &
                  rebuild, ignore_mismatch)
          class is (pcm_nlo_t)
             select case (component%config%get_nlo_type ())
             case (BORN, NLO_VIRTUAL, NLO_SUBTRACTION)
                call component%configure_phs (sqrts, process%beam_config, &
                     rebuild, ignore_mismatch)
                call check_and_extend_phs (component)
             case (NLO_REAL, NLO_MISMATCH, NLO_DGLAP)
                i_born = component%config%get_associated_born ()
                call check_and_extend_phs (component)
                call process%component(i_born)%get_phs_config (phs_config_born)
                select type (config => component%phs_config)
                type is (phs_fks_config_t)
                   select type (phs_config_born)
                   type is (phs_wood_config_t)
                      config%md5sum_born_config = phs_config_born%md5sum_phs_config
                      call config%set_born_config (phs_config_born)
                      call config%set_mode (component%config%get_nlo_type ())
                   end select
                end select
```

```
                 call component%configure_phs (sqrts, &
                      process%beam_config, rebuild, ignore_mismatch)
              end select
           class default
              call msg_bug ("process_configure_phs: unsupported PCM type")
           end select
        end if
     end associate
  end do
contains
  subroutine check_and_extend_phs (component)
    type(process_component_t), intent(inout) :: component
    logical :: requires_dglap_random_number
    if (combined_integration) then
       requires_dglap_random_number = any (process%component%get_nlo_type () == NLO_DGLAP)
       select type (phs_config => component%phs_config)
       class is (phs_wood_config_t)
          if (requires_dglap_random_number) then
             call phs_config%set_extension_mode (EXTENSION_DGLAP)
          else
             call phs_config%set_extension_mode (EXTENSION_DEFAULT)
          end if
          call phs_config%increase_n_par ()
       end select
    end if
  end subroutine check_and_extend_phs
end subroutine process_configure_phs
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: print_phs_startup_message => process_print_phs_startup_message
```

⟨*Process: procedures*⟩+≡

```
subroutine process_print_phs_startup_message (process)
  class(process_t), intent(in) :: process
  integer :: i_component
  do i_component = 1, process%meta%n_components
     associate (component => process%component(i_component))
        if (component%active) then
           call component%phs_config%startup_message ()
        end if
     end associate
  end do
end subroutine process_print_phs_startup_message
```

Insert the structure-function configuration data. First allocate the storage, then insert data one by one. The third procedure declares a mapping (of the MC input parameters) for a specific channel and structure-function combination.

We take the number of channels from the corresponding entry in the `config_data` section.

Otherwise, these a simple wrapper routines. The extra level in the call tree may allow for simple addressing of multiple concurrent beam configurations, not implemented currently.

If we do not want structure functions, we simply do not call those procedures.

⟨*Process: process: TBP*⟩+≡

```
procedure :: init_sf_chain => process_init_sf_chain
generic :: set_sf_channel => set_sf_channel_single
procedure :: set_sf_channel_single => process_set_sf_channel
generic :: set_sf_channel => set_sf_channel_array
procedure :: set_sf_channel_array => process_set_sf_channel_array
```

⟨*Process: procedures*⟩+≡

```
subroutine process_init_sf_chain (process, sf_config, sf_trace_file)
  class(process_t), intent(inout) :: process
  type(sf_config_t), dimension(:), intent(in) :: sf_config
  type(string_t), intent(in), optional :: sf_trace_file
  type(string_t) :: file
  if (present (sf_trace_file)) then
     if (sf_trace_file /= "") then
        file = sf_trace_file
     else
        file = process%get_id () // "_sftrace.dat"
     end if
     call process%beam_config%init_sf_chain (sf_config, file)
  else
     call process%beam_config%init_sf_chain (sf_config)
  end if
end subroutine process_init_sf_chain

subroutine process_set_sf_channel (process, c, sf_channel)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: c
  type(sf_channel_t), intent(in) :: sf_channel
  call process%beam_config%set_sf_channel (c, sf_channel)
end subroutine process_set_sf_channel

subroutine process_set_sf_channel_array (process, sf_channel)
  class(process_t), intent(inout) :: process
  type(sf_channel_t), dimension(:), intent(in) :: sf_channel
  integer :: c
  call process%beam_config%allocate_sf_channels (size (sf_channel))
  do c = 1, size (sf_channel)
     call process%beam_config%set_sf_channel (c, sf_channel(c))
  end do
end subroutine process_set_sf_channel_array
```

Notify about the structure-function setup.

⟨*Process: process: TBP*⟩+≡

```
procedure :: sf_startup_message => process_sf_startup_message
```

⟨*Process: procedures*⟩+≡

```
subroutine process_sf_startup_message (process, sf_string, unit)
  class(process_t), intent(in) :: process
  type(string_t), intent(in) :: sf_string
  integer, intent(in), optional :: unit
  call process%beam_config%sf_startup_message (sf_string, unit)
end subroutine process_sf_startup_message
```

As soon as both the kinematics configuration and the structure-function setup are complete, we match parameterizations (channels) for both. The matching entries are (re)set in the `component` phase-space configuration, while the structure-function configuration is left intact.

⟨*Process: process: TBP*⟩+≡
```
procedure :: collect_channels => process_collect_channels
```

⟨*Process: procedures*⟩+≡
```
subroutine process_collect_channels (process, coll)
  class(process_t), intent(inout) :: process
  type(phs_channel_collection_t), intent(inout) :: coll
  integer :: i
  do i = 1, process%meta%n_components
     associate (component => process%component(i))
       if (component%active) &
          call component%collect_channels (coll)
     end associate
  end do
end subroutine process_collect_channels
```

Independently, we should be able to check if any component doesn't contain phase-space parameters. Such a process can only be integrated if there are structure functions.

⟨*Process: process: TBP*⟩+≡
```
procedure :: contains_trivial_component => process_contains_trivial_component
```

⟨*Process: procedures*⟩+≡
```
function process_contains_trivial_component (process) result (flag)
  class(process_t), intent(in) :: process
  logical :: flag
  integer :: i
  flag = .true.
  do i = 1, process%meta%n_components
     associate (component => process%component(i))
       if (component%active) then
          if (component%get_n_phs_par () == 0)  return
       end if
     end associate
  end do
  flag = .false.
end function process_contains_trivial_component
```

⟨*Process: process: TBP*⟩+≡
```
procedure :: deactivate_real_component => process_deactivate_real_component
```

⟨*Process: procedures*⟩+≡
```
subroutine process_deactivate_real_component (process)
  class(process_t), intent(inout) :: process
  type(process_component_def_t) :: config
  integer :: i
  config = process%component(1)%get_config ()
  call process%mci_entry(1)%deactivate_real_component (config)
  do i = 1, process%meta%n_components
     select case (process%component(i)%component_type)
```

```
      case (COMP_REAL)
         process%component(i)%active = .false.
      end select
   end do
end subroutine process_deactivate_real_component
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: deactivate_components => process_deactivate_components
```

⟨*Process: procedures*⟩+≡

```
subroutine process_deactivate_components (process, fixed_mci)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: fixed_mci
  integer :: i_mci, i_component
  do i_mci = 1, size (process%mci_entry)
     if (i_mci /= fixed_mci) then
        i_component = process%i_mci_to_i_component (i_mci)
        process%component(i_component)%active = .false.
     end if
  end do
end subroutine process_deactivate_components
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: i_mci_to_i_component => process_i_mci_to_i_component
```

⟨*Process: procedures*⟩+≡

```
function process_i_mci_to_i_component (process, i_mci) result (i_component)
   integer :: i_component
   class(process_t), intent(in) :: process
   integer, intent(in) :: i_mci
   integer :: i
   i_component = 0
   do i = 1, size (process%component)
      if (process%component(i)%i_mci == i_mci) then
         i_component = i
         return
      end if
   end do
end function process_i_mci_to_i_component
```

Determine the MC parameter set structure and the MCI configuration for each process component. We need data from the structure-function and phase-space setup, so those should be complete before this is called. We also make a random-number generator instance for each MCI group.

⟨*Process: process: TBP*⟩+≡

```
procedure :: setup_mci => process_setup_mci
```

⟨*Process: procedures*⟩+≡

```
subroutine process_setup_mci (process, combined_integration)
  class(process_t), intent(inout) :: process
  logical, intent(in), optional :: combined_integration
  integer :: n_mci, i_mci
```

```
  integer :: i
  logical :: uses_real_partition
  call msg_debug (D_PROCESS_INTEGRATION, "process_setup_mci")
  n_mci = 0
  do i = 1, process%meta%n_components
     associate (component => process%component(i))
        if (component%needs_mci_entry (combined_integration) .and. &
            component%config%get_nlo_type () /= NLO_SUBTRACTION) then
          n_mci = n_mci + 1
          component%i_mci = n_mci
       end if
       call msg_debug (D_PROCESS_INTEGRATION, &
            "component%component_type", component%component_type)
     end associate
  end do
  process%config%n_mci = n_mci
  if (.not. allocated (process%config%rng_factory)) &
       call msg_bug ("Process setup: rng factory not allocated")
  allocate (process%mci_entry (n_mci))
  i_mci = 0
  uses_real_partition = &
      any (process%component%component_type == COMP_REAL_FIN)
  call msg_debug (D_PROCESS_INTEGRATION, "uses_real_partition", &
       uses_real_partition)
  do i = 1, process%meta%n_components
     associate (component => process%component(i))
        if (component%needs_mci_entry (combined_integration) .and. &
            component%config%get_nlo_type () /= NLO_SUBTRACTION) then
          i_mci = i_mci + 1
          associate (mci_entry => process%mci_entry(i_mci))
            call mci_entry%set_combined_integration (combined_integration)
            if (uses_real_partition) then
               if (component%component_type == COMP_REAL_FIN) then
                  mci_entry%real_partition_type = REAL_FINITE
               else
                  mci_entry%real_partition_type = REAL_SINGULAR
               end if
            end if

            call mci_entry%init (process%meta%type, &
                 i_mci, i, component, process%beam_config%n_sfpar, &
                 process%config%rng_factory)
          end associate
        end if
     end associate
  end do
  do i_mci = 1, size (process%mci_entry)
     call process%mci_entry(i_mci)%set_parameters (process%meta%var_list)
  end do
end subroutine process_setup_mci
```

Set cuts. This is a parse node, namely the right-hand side of the `cut` assignment. When creating an instance, we compile this into an evaluation tree. The parse

node may be null.

⟨*Process: process: TBP*⟩+≡
```
procedure :: set_cuts => process_set_cuts
```

⟨*Process: procedures*⟩+≡
```
subroutine process_set_cuts (process, ef_cuts)
  class(process_t), intent(inout) :: process
  class(expr_factory_t), intent(in) :: ef_cuts
  allocate (process%config%ef_cuts, source = ef_cuts)
end subroutine process_set_cuts
```

Analogously for the other expressions.

⟨*Process: process: TBP*⟩+≡
```
procedure :: set_scale => process_set_scale
procedure :: set_fac_scale => process_set_fac_scale
procedure :: set_ren_scale => process_set_ren_scale
procedure :: set_weight => process_set_weight
```

⟨*Process: procedures*⟩+≡
```
subroutine process_set_scale (process, ef_scale)
  class(process_t), intent(inout) :: process
  class(expr_factory_t), intent(in) :: ef_scale
  allocate (process%config%ef_scale, source = ef_scale)
end subroutine process_set_scale

subroutine process_set_fac_scale (process, ef_fac_scale)
  class(process_t), intent(inout) :: process
  class(expr_factory_t), intent(in) :: ef_fac_scale
  allocate (process%config%ef_fac_scale, source = ef_fac_scale)
end subroutine process_set_fac_scale

subroutine process_set_ren_scale (process, ef_ren_scale)
  class(process_t), intent(inout) :: process
  class(expr_factory_t), intent(in) :: ef_ren_scale
  allocate (process%config%ef_ren_scale, source = ef_ren_scale)
end subroutine process_set_ren_scale

subroutine process_set_weight (process, ef_weight)
  class(process_t), intent(inout) :: process
  class(expr_factory_t), intent(in) :: ef_weight
  allocate (process%config%ef_weight, source = ef_weight)
end subroutine process_set_weight
```

MD5 sum

The MD5 sum of the process object should reflect the state completely, including integration results. It is used for checking the integrity of event files. This global checksum includes checksums for the various parts. In particular, the MCI object receives a checksum that includes the configuration of all configuration parts relevant for an individual integration. This checksum is used for checking the integrity of integration grids.

We do not need MD5 sums for the process terms, since these are generated from the component definitions.

⟨*Process: process: TBP*⟩+≡
```
procedure :: compute_md5sum => process_compute_md5sum
```

⟨*Process: procedures*⟩+≡
```
subroutine process_compute_md5sum (process)
  class(process_t), intent(inout) :: process
  integer :: i
  call process%config%compute_md5sum ()
  do i = 1, process%config%n_components
     associate (component => process%component(i))
       if (component%active) then
          call component%compute_md5sum ()
       end if
     end associate
  end do
  call process%beam_config%compute_md5sum ()
  do i = 1, process%config%n_mci
     call process%mci_entry(i)%compute_md5sum &
          (process%config, process%component, process%beam_config)
  end do
end subroutine process_compute_md5sum
```

⟨*Process: process: TBP*⟩+≡
```
procedure :: sampler_test => process_sampler_test
```

⟨*Process: procedures*⟩+≡
```
subroutine process_sampler_test (process, sampler, n_calls, i_mci)
  class(process_t), intent(inout) :: process
  class(mci_sampler_t), intent(inout) :: sampler
  integer, intent(in) :: n_calls, i_mci
  call process%mci_entry(i_mci)%sampler_test (sampler, n_calls)
end subroutine process_sampler_test
```

The finalizer should be called after all integration passes have been completed. It will, for instance, write a summary of the integration results.

`integrate_dummy` does a "dummy" integration in the sense that nothing is done but just empty integration results appended.

⟨*Process: process: TBP*⟩+≡
```
procedure :: final_integration => process_final_integration
procedure :: integrate_dummy => process_integrate_dummy
```

⟨*Process: procedures*⟩+≡
```
subroutine process_final_integration (process, i_mci)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: i_mci
  call process%mci_entry(i_mci)%final_integration ()
end subroutine process_final_integration

subroutine process_integrate_dummy (process)
  class(process_t), intent(inout) :: process
  type(integration_results_t) :: results
  integer :: u_log
```

```
    u_log = logfile_unit ()
    call results%init (process%meta%type)
    call results%display_init (process%meta%type, screen = .true., unit = u_log)
    call results%new_pass ()
    call results%record (1, 0, 0._default, 0._default, 0._default)
    call results%display_final ()
  end subroutine process_integrate_dummy
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: integrate => process_integrate
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_integrate (process, i_mci, mci_work, &
     mci_sampler, n_it, n_calls, adapt_grids, adapt_weights, final, &
     pacify, nlo_type)
    class(process_t), intent(inout) :: process
    integer, intent(in) :: i_mci
    type(mci_work_t), intent(inout) :: mci_work
    class(mci_sampler_t), intent(inout) :: mci_sampler
    integer, intent(in) :: n_it, n_calls
    logical, intent(in), optional :: adapt_grids, adapt_weights
    logical, intent(in), optional :: final
    logical, intent(in), optional :: pacify
    integer, intent(in), optional :: nlo_type
    associate (mci_entry => process%mci_entry(i_mci))
       call mci_entry%integrate (mci_work%mci, mci_sampler, n_it, n_calls, &
          adapt_grids, adapt_weights, final, pacify, &
          nlo_type = nlo_type)
       call mci_entry%results%display_pass (pacify)
    end associate
  end subroutine process_integrate
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: generate_weighted_event => process_generate_weighted_event
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_generate_weighted_event (process, i_mci, mci_work, &
     mci_sampler, keep_failed_events)
    class(process_t), intent(inout) :: process
    integer, intent(in) :: i_mci
    type(mci_work_t), intent(inout) :: mci_work
    class(mci_sampler_t), intent(inout) :: mci_sampler
    logical, intent(in) :: keep_failed_events
    associate (mci_entry => process%mci_entry(i_mci))
       call mci_entry%generate_weighted_event (mci_work%mci, &
          mci_sampler, keep_failed_events)
    end associate
  end subroutine process_generate_weighted_event
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: generate_unweighted_event => process_generate_unweighted_event
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_generate_unweighted_event (process, i_mci, &
```

```
 mci_work, mci_sampler)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: i_mci
  type(mci_work_t), intent(inout) :: mci_work
  class(mci_sampler_t), intent(inout) :: mci_sampler
  associate (mci_entry => process%mci_entry(i_mci))
     call mci_entry%generate_unweighted_event &
        (mci_work%mci, mci_sampler)
  end associate
end subroutine process_generate_unweighted_event
```

Display the final results for the sum of all components. (This is useful, obviously, only if there is more than one component.)

⟨*Process: process: TBP*⟩+≡

```
procedure :: display_summed_results => process_display_summed_results
```

⟨*Process: procedures*⟩+≡

```
subroutine process_display_summed_results (process, pacify)
  class(process_t), intent(inout) :: process
  logical, intent(in) :: pacify
  type(integration_results_t) :: results
  integer :: u_log
  u_log = logfile_unit ()
  call results%init (process%meta%type)
  call results%display_init (process%meta%type, screen = .true., unit = u_log)
  call results%new_pass ()
  call results%record (1, 0, &
       process%get_integral (), &
       process%get_error (), &
       process%get_efficiency (), suppress = pacify)
  select type (pcm => process%pcm)
  class is (pcm_nlo_t)
     !!! Check that Born integral is there
     if (process%component_can_be_integrated (1)) then
        call results%record_correction (process%get_correction (), &
             process%get_correction_error ())
     end if
  end select
  call results%display_final ()
end subroutine process_display_summed_results
```

Run LaTeX/Metapost to generate a ps/pdf file for the integration history. We (re)write the driver file – just in case it has been missed before – then we compile it.

⟨*Process: process: TBP*⟩+≡

```
procedure :: display_integration_history => &
     process_display_integration_history
```

⟨*Process: procedures*⟩+≡

```
subroutine process_display_integration_history &
     (process, i_mci, filename, os_data, eff_reset)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: i_mci
  type(string_t), intent(in) :: filename
```

```
    type(os_data_t), intent(in) :: os_data
    logical, intent(in), optional :: eff_reset
    call integration_results_write_driver &
         (process%mci_entry(i_mci)%results, filename, eff_reset)
    call integration_results_compile_driver &
         (process%mci_entry(i_mci)%results, filename, os_data)
  end subroutine process_display_integration_history
```

Write a complete logfile (with hardcoded name based on the process ID). We do not write internal data.

⟨*Process: process: TBP*⟩+≡

```
  procedure :: write_logfile => process_write_logfile
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_write_logfile (process, i_mci, filename)
    class(process_t), intent(inout) :: process
    integer, intent(in) :: i_mci
    type(string_t), intent(in) :: filename
    type(time_t) :: time
    integer :: unit, u
    unit = free_unit ()
    open (unit = unit, file = char (filename), action = "write", &
          status = "replace")
    u = given_output_unit (unit)
    write (u, "(A)")  repeat ("#", 79)
    call process%meta%write (u, .false., .false.)
    write (u, "(A)")  repeat ("#", 79)
    write (u, "(3x,A,ES17.10)")  "Integral   = ", &
         process%mci_entry(i_mci)%get_integral ()
    write (u, "(3x,A,ES17.10)")  "Error      = ", &
         process%mci_entry(i_mci)%get_error ()
    write (u, "(3x,A,ES17.10)")  "Accuracy   = ", &
         process%mci_entry(i_mci)%get_accuracy ()
    write (u, "(3x,A,ES17.10)")  "Chi2       = ", &
         process%mci_entry(i_mci)%get_chi2 ()
    write (u, "(3x,A,ES17.10)")  "Efficiency = ", &
         process%mci_entry(i_mci)%get_efficiency ()
    call process%mci_entry(i_mci)%get_time (time, 10000)
    if (time%is_known ()) then
       write (u, "(3x,A,1x,A)")  "T(10k evt) = ", char (time%to_string_dhms ())
    else
       write (u, "(3x,A)")  "T(10k evt) =  [undefined]"
    end if
    call process%mci_entry(i_mci)%results%write (u)
    write (u, "(A)")  repeat ("#", 79)
    call process%mci_entry(i_mci)%results%write_chain_weights (u)
    write (u, "(A)")  repeat ("#", 79)
    call process%mci_entry(i_mci)%counter%write (u)
    write (u, "(A)")  repeat ("#", 79)
    call process%mci_entry(i_mci)%mci%write_log_entry (u)
    write (u, "(A)")  repeat ("#", 79)
    call process%beam_config%data%write (u)
    write (u, "(A)")  repeat ("#", 79)
    if (allocated (process%config%ef_cuts)) then
```

```
      write (u, "(3x,A)") "Cut expression:"
      call process%config%ef_cuts%write (u)
   else
      write (u, "(3x,A)") "No cuts used."
   end if
   call write_separator (u)
   if (allocated (process%config%ef_scale)) then
      write (u, "(3x,A)") "Scale expression:"
      call process%config%ef_scale%write (u)
   else
      write (u, "(3x,A)") "No scale expression was given."
   end if
   call write_separator (u)
   if (allocated (process%config%ef_fac_scale)) then
      write (u, "(3x,A)") "Factorization scale expression:"
      call process%config%ef_fac_scale%write (u)
   else
      write (u, "(3x,A)") "No factorization scale expression was given."
   end if
   call write_separator (u)
   if (allocated (process%config%ef_ren_scale)) then
      write (u, "(3x,A)") "Renormalization scale expression:"
      call process%config%ef_ren_scale%write (u)
   else
      write (u, "(3x,A)") "No renormalization scale expression was given."
   end if
   call write_separator (u)
   if (allocated (process%config%ef_weight)) then
      call write_separator (u)
      write (u, "(3x,A)") "Weight expression:"
      call process%config%ef_weight%write (u)
   else
      write (u, "(3x,A)") "No weight expression was given."
   end if
   write (u, "(A)")  repeat ("#", 79)
   write (u, "(1x,A)") "Summary of quantum-number states:"
   write (u, "(1x,A)")  " + sign: allowed and contributing"
   write (u, "(1x,A)")  " no +  : switched off at runtime"
   call process%write_state_summary (u)
   write (u, "(A)")  repeat ("#", 79)
   write (u, "(A)")  "Variable list:"
   call var_list_write (process%meta%var_list, u)
   write (u, "(A)")  repeat ("#", 79)
   close (u)
end subroutine process_write_logfile
```

Display the quantum-number combinations of the process components, and their current status (allowed or switched off).

⟨*Process: process: TBP*⟩+≡

```
procedure :: write_state_summary => process_write_state_summary
```

⟨*Process: procedures*⟩+≡

```
subroutine process_write_state_summary (process, unit)
  class(process_t), intent(in) :: process
```

```
    integer, intent(in), optional :: unit
    integer :: i, i_component, u
    u = given_output_unit (unit)
    do i = 1, size (process%term)
       call write_separator (u)
       i_component = process%term(i)%i_component
       if (i_component /= 0) then
          call process%term(i)%write_state_summary &
               (process%get_core_term(i), unit)
       end if
    end do
  end subroutine process_write_state_summary
```

Prepare event generation for the specified MCI entry. This implies, in particular, checking the phase-space file.

⟨*Process: process: TBP*⟩+≡

```
  procedure :: prepare_simulation => process_prepare_simulation
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_prepare_simulation (process, i_mci)
    class(process_t), intent(inout) :: process
    integer, intent(in) :: i_mci
    call process%mci_entry(i_mci)%prepare_simulation ()
  end subroutine process_prepare_simulation
```

Retrieve process data

Tell whether integral (and error) are known.

⟨*Process: process: TBP*⟩+≡

```
  generic :: has_integral => has_integral_tot, has_integral_mci
  procedure :: has_integral_tot => process_has_integral_tot
  procedure :: has_integral_mci => process_has_integral_mci
```

⟨*Process: procedures*⟩+≡

```
  function process_has_integral_mci (process, i_mci) result (flag)
    logical :: flag
    class(process_t), intent(in) :: process
    integer, intent(in) :: i_mci
    flag = process%mci_entry(i_mci)%has_integral ()
  end function process_has_integral_mci

  function process_has_integral_tot (process) result (flag)
    logical :: flag
    class(process_t), intent(in) :: process
    integer :: i, j, i_component
    flag = .true.
    do i = 1, size (process%mci_entry)
       do j = 1, size (process%mci_entry(i)%i_component)
          i_component = process%mci_entry(i)%i_component(j)
          if (process%component_can_be_integrated (i_component)) &
             flag = flag .and. process%mci_entry(i)%has_integral ()
       end do
    end do
```

```
end function process_has_integral_tot
```

Return the current integral and error obtained by the integrator `i_mci`.

⟨*Process: process: TBP*⟩+≡

```
generic :: get_integral => get_integral_tot, get_integral_mci
generic :: get_error => get_error_tot, get_error_mci
generic :: get_efficiency => get_efficiency_tot, get_efficiency_mci
procedure :: get_integral_tot => process_get_integral_tot
procedure :: get_integral_mci => process_get_integral_mci
procedure :: get_error_tot => process_get_error_tot
procedure :: get_error_mci => process_get_error_mci
procedure :: get_efficiency_tot => process_get_efficiency_tot
procedure :: get_efficiency_mci => process_get_efficiency_mci
```

⟨*Process: procedures*⟩+≡

```
function process_get_integral_mci (process, i_mci) result (integral)
  real(default) :: integral
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_mci
  integral = process%mci_entry(i_mci)%get_integral ()
end function process_get_integral_mci

function process_get_error_mci (process, i_mci) result (error)
  real(default) :: error
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_mci
  error = process%mci_entry(i_mci)%get_error ()
end function process_get_error_mci

function process_get_efficiency_mci (process, i_mci) result (efficiency)
  real(default) :: efficiency
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_mci
  efficiency = process%mci_entry(i_mci)%get_efficiency ()
end function process_get_efficiency_mci

function process_get_integral_tot (process) result (integral)
  real(default) :: integral
  class(process_t), intent(in) :: process
  integer :: i, j, i_component
  integral = zero
  if (allocated (process%mci_entry)) then
     do i = 1, size (process%mci_entry)
        do j = 1, size (process%mci_entry(i)%i_component)
           i_component = process%mci_entry(i)%i_component(j)
           if (process%component_can_be_integrated(i_component)) &
                integral = integral + process%mci_entry(i)%get_integral ()
        end do
     end do
  end if
end function process_get_integral_tot

function process_get_error_tot (process) result (error)
  real(default) :: variance
```

```
  class(process_t), intent(in) :: process
  real(default) :: error
  integer :: i, j, i_component
  variance = zero
  if (allocated (process%mci_entry)) then
     do i = 1, size (process%mci_entry)
        do j = 1, size (process%mci_entry(i)%i_component)
           i_component = process%mci_entry(i)%i_component(j)
           if (process%component_can_be_integrated(i_component)) &
                variance = variance + process%mci_entry(i)%get_error () ** 2
        end do
     end do
  end if
  error = sqrt (variance)
end function process_get_error_tot

function process_get_efficiency_tot (process) result (efficiency)
  real(default) :: efficiency
  class(process_t), intent(in) :: process
  real(default) :: den, eff, int
  integer :: i, j, i_component
  den = zero
  if (allocated (process%mci_entry)) then
     do i = 1, size (process%mci_entry)
        do j = 1, size (process%mci_entry(i)%i_component)
           i_component = process%mci_entry(i)%i_component(j)
           if (process%component_can_be_integrated(i_component)) then
              int = process%get_integral (i)
              if (int > 0) then
                 eff = process%mci_entry(i)%get_efficiency ()
                 if (eff > 0) then
                    den = den + int / eff
                 else
                    efficiency = 0
                    return
                 end if
              end if
           end if
        end do
     end do
  end if
  if (den > 0) then
     efficiency = process%get_integral () / den
  else
     efficiency = 0
  end if
end function process_get_efficiency_tot
```

Let us call the ratio of the LO and the NLO result $\iota = I_{LO}/I_{NLO}$. Then usual error propagation gives

$$\sigma_\iota^2 = \left(\frac{\partial \iota}{\partial I_{LO}}\right)^2 \sigma_{I_{LO}}^2 + \left(\frac{\partial \iota}{\partial I_{NLO}}\right)^2 \sigma_{I_{NLO}}^2 = \frac{I_{NLO}^2 \sigma_{I_{LO}}^2}{I_{LO}^4} + \frac{\sigma_{I_{NLO}}^2}{I_{LO}^2}.$$

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_correction => process_get_correction
procedure :: get_correction_error => process_get_correction_error
```

⟨*Process: procedures*⟩+≡

```
function process_get_correction (process) result (ratio)
  real(default) :: ratio
  class(process_t), intent(in) :: process
  integer :: i_mci
  real(default) :: int_born, int_nlo
  int_nlo = zero
  int_born = process%mci_entry(1)%get_integral ()
  do i_mci = 2, size (process%mci_entry)
     if (process%component_can_be_integrated (i_mci)) &
        int_nlo = int_nlo + process%mci_entry(i_mci)%get_integral ()
  end do
  ratio = int_nlo / int_born * 100
end function process_get_correction

function process_get_correction_error (process) result (error)
  real(default) :: error
  class(process_t), intent(in) :: process
  real(default) :: int_born, sum_int_nlo
  real(default) :: err_born, err2
  integer :: i_mci
  sum_int_nlo = zero; err2 = zero
  int_born = process%mci_entry(1)%get_integral ()
  err_born = process%mci_entry(1)%get_error ()
  do i_mci = 2, size (process%mci_entry)
     if (process%component_can_be_integrated (i_mci)) then
        sum_int_nlo = sum_int_nlo + process%mci_entry(i_mci)%get_integral ()
        err2 = err2 + process%mci_entry(i_mci)%get_error()**2
     end if
  end do
  error = sqrt (err2 / int_born**2 + sum_int_nlo**2 * err_born**2 / int_born**4) * 100
end function process_get_correction_error
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: lab_is_cm_frame => process_lab_is_cm_frame
```

⟨*Process: procedures*⟩+≡

```
pure function process_lab_is_cm_frame (process) result (cm_frame)
  logical :: cm_frame
  class(process_t), intent(in) :: process
  cm_frame = process%beam_config%lab_is_cm_frame
end function process_lab_is_cm_frame
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_component_ptr => process_get_component_ptr
```

⟨*Process: procedures*⟩+≡

```
function process_get_component_ptr (process, i) result (component)
  type(process_component_t), pointer :: component
  class(process_t), intent(in), target :: process
  integer, intent(in) :: i
```

```
    component => process%component(i)
  end function process_get_component_ptr
```

⟨*Process: process: TBP*⟩+≡

```
  generic :: get_component_type => get_component_type_single
  procedure :: get_component_type_single => process_get_component_type_single
```

⟨*Process: procedures*⟩+≡

```
  elemental function process_get_component_type_single &
     (process, i_component) result (comp_type)
    integer :: comp_type
    class(process_t), intent(in) :: process
    integer, intent(in) :: i_component
    comp_type = process%component(i_component)%component_type
  end function process_get_component_type_single
```

⟨*Process: process: TBP*⟩+≡

```
  generic :: get_component_type => get_component_type_all
  procedure :: get_component_type_all => process_get_component_type_all
```

⟨*Process: procedures*⟩+≡

```
  function process_get_component_type_all &
     (process) result (comp_type)
    integer, dimension(:), allocatable :: comp_type
    class(process_t), intent(in) :: process
    allocate (comp_type (size (process%component)))
    comp_type = process%component%component_type
  end function process_get_component_type_all
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: get_component_i_terms => process_get_component_i_terms
```

⟨*Process: procedures*⟩+≡

```
  function process_get_component_i_terms (process, i_component) result (i_term)
     integer, dimension(:), allocatable :: i_term
     class(process_t), intent(in) :: process
     integer, intent(in) :: i_component
     allocate (i_term (size (process%component(i_component)%i_term)))
     i_term = process%component(i_component)%i_term
  end function process_get_component_i_terms
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: get_n_allowed_born => process_get_n_allowed_born
```

⟨*Process: procedures*⟩+≡

```
  function process_get_n_allowed_born (process, i_born) result (n_born)
    class(process_t), intent(inout) :: process
    integer, intent(in) :: i_born
    integer :: n_born
    n_born = process%term(i_born)%n_allowed

  end function process_get_n_allowed_born
```

Workaround getter. Would be better to remove this.

⟨*Process: process: TBP*⟩+≡
```
procedure :: get_pcm_ptr => process_get_pcm_ptr
```

⟨*Process: procedures*⟩+≡
```
function process_get_pcm_ptr (process) result (pcm)
  class(pcm_t), pointer :: pcm
  class(process_t), intent(in), target :: process
  pcm => process%pcm
end function process_get_pcm_ptr
```

⟨*Process: process: TBP*⟩+≡
```
procedure :: component_can_be_integrated => process_component_can_be_integrated
```

⟨*Process: procedures*⟩+≡
```
function process_component_can_be_integrated (process, i_component) &
         result (active)
  logical :: active
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_component
  logical :: combined_integration
  select type (pcm => process%pcm)
  type is (pcm_nlo_t)
     combined_integration = pcm%settings%combined_integration
  class default
     combined_integration = .false.
  end select
  associate (component => process%component(i_component))
     active = component%can_be_integrated ()
     if (combined_integration) &
          active = active .and. component%component_type <= COMP_MASTER
  end associate
end function process_component_can_be_integrated
```

⟨*Process: process: TBP*⟩+≡
```
procedure :: reset_selected_cores => process_reset_selected_cores
```

⟨*Process: procedures*⟩+≡
```
pure subroutine process_reset_selected_cores (process)
  class(process_t), intent(inout) :: process
  process%component_selected = .false.
end subroutine process_reset_selected_cores
```

⟨*Process: process: TBP*⟩+≡
```
procedure :: select_components => process_select_components
```

⟨*Process: procedures*⟩+≡
```
pure subroutine process_select_components (process, indices)
  class(process_t), intent(inout) :: process
  integer, dimension(:), intent(in) :: indices
  process%component_selected(indices) = .true.
end subroutine process_select_components
```

⟨*Process: process: TBP*⟩+≡
```
procedure :: component_is_selected => process_component_is_selected
```
⟨*Process: procedures*⟩+≡
```
pure function process_component_is_selected (process, index) result (val)
  logical :: val
  class(process_t), intent(in) :: process
  integer, intent(in) :: index
  val = process%component_selected(index)
end function process_component_is_selected
```

⟨*Process: process: TBP*⟩+≡
```
procedure :: get_coupling_powers => process_get_coupling_powers
```
⟨*Process: procedures*⟩+≡
```
pure subroutine process_get_coupling_powers (process, alpha_power, alphas_power)
  class(process_t), intent(in) :: process
  integer, intent(out) :: alpha_power, alphas_power
  call process%component(1)%config%get_coupling_powers (alpha_power, alphas_power)
end subroutine process_get_coupling_powers
```

⟨*Process: process: TBP*⟩+≡
```
procedure :: get_real_component => process_get_real_component
```
⟨*Process: procedures*⟩+≡
```
function process_get_real_component (process) result (i_real)
  integer :: i_real
  class(process_t), intent(in) :: process
  integer :: i_component
  type(process_component_def_t), pointer :: config => null ()
  logical :: comp_real_fin

  i_real = 0
  do i_component = 1, size (process%component)
     config => process%get_component_def_ptr (i_component)
     if (config%get_nlo_type () == NLO_REAL) then
        i_real = i_component
        exit
     end if
  end do
end function process_get_real_component
```

⟨*Process: process: TBP*⟩+≡
```
procedure :: extract_fixed_mci => process_extract_fixed_mci
```
⟨*Process: procedures*⟩+≡
```
function process_extract_fixed_mci (process) result (i_active)
  integer :: i_active
  class(process_t), intent(in) :: process
  integer :: i_mci, j, i_component, n_active
  call count_n_active ()
  if (n_active /= 1) i_active = 0
contains
  subroutine count_n_active ()
```

```
      n_active = 0
      do i_mci = 1, size (process%mci_entry)
         associate (mci_entry => process%mci_entry(i_mci))
            do j = 1, size (mci_entry%i_component)
               i_component = mci_entry%i_component(j)
               associate (component => process%component (i_component))
                  if (component%can_be_integrated ()) then
                     i_active = i_mci
                     n_active = n_active + 1
                  end if
               end associate
            end do
         end associate
      end do
   end subroutine count_n_active
end function process_extract_fixed_mci
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: needs_extra_code => process_needs_extra_code
```

⟨*Process: procedures*⟩+≡

```
function process_needs_extra_code (process, only_blha) result (val)
  logical :: val
  class(process_t), intent(in) :: process
  logical, intent(in), optional :: only_blha
  integer :: i
  logical :: skip_other
  type(process_component_def_t), pointer :: config => null ()
  val = .false.; skip_other = .false.
  if (present (only_blha)) skip_other = only_blha
  associate (cm => process%cm)
     do i = 1, cm%n_cores
        config => process%get_component_def_ptr &
             (cm%i_core_to_first_i_component(i))
        if (config%can_be_integrated () .or. cm%sub(i)) then
           select type (core => cm%cores(i)%core)
           type is (prc_recola_t)
              if (skip_other) cycle
              val = .true.
              exit
           class is (prc_blha_t)
              val = .true.
              exit
           class is (prc_threshold_t)
              if (skip_other) cycle
              val = .true.
              exit
           end select
        end if
     end do
  end associate

end function process_needs_extra_code
```

⟨*Process: process: TBP*⟩+≡
```
procedure :: uses_real_partition => process_uses_real_partition
```
⟨*Process: procedures*⟩+≡
```
function process_uses_real_partition (process) result (val)
   logical :: val
   class(process_t), intent(in) :: process
   val = any (process%mci_entry%real_partition_type /= REAL_FULL)
end function process_uses_real_partition
```

Return the MD5 sums that summarize the process component definitions. These values should be independent of parameters, beam details, expressions, etc. They can be used for checking the integrity of a process when reusing an old event file.

⟨*Process: process: TBP*⟩+≡
```
procedure :: get_md5sum_prc => process_get_md5sum_prc
```
⟨*Process: procedures*⟩+≡
```
function process_get_md5sum_prc (process, i_component) result (md5sum)
  character(32) :: md5sum
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_component
  if (process%component(i_component)%active) then
     md5sum = process%component(i_component)%config%get_md5sum ()
  else
     md5sum = ""
  end if
end function process_get_md5sum_prc
```

Return the MD5 sums that summarize the state of the MCI integrators. These values should encode all process data, integration and phase space configuration, etc., and the integration results. They can thus be used for checking the integrity of an event-generation setup when reusing an old event file.

⟨*Process: process: TBP*⟩+≡
```
procedure :: get_md5sum_mci => process_get_md5sum_mci
```
⟨*Process: procedures*⟩+≡
```
function process_get_md5sum_mci (process, i_mci) result (md5sum)
  character(32) :: md5sum
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_mci
  md5sum = process%mci_entry(i_mci)%get_md5sum ()
end function process_get_md5sum_mci
```

Return the MD5 sum of the process configuration. This should encode the process setup, data, and expressions, but no integration results.

⟨*Process: process: TBP*⟩+≡
```
procedure :: get_md5sum_cfg => process_get_md5sum_cfg
```
⟨*Process: procedures*⟩+≡
```
function process_get_md5sum_cfg (process) result (md5sum)
  character(32) :: md5sum
  class(process_t), intent(in) :: process
```

```
    md5sum = process%config%md5sum
  end function process_get_md5sum_cfg
```

Initialize the i_coreth core.

⟨*Process: process: TBP*⟩+≡

```
  procedure :: init_cores => process_init_cores
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_init_cores (process)
    class(process_t), intent(inout) :: process
    integer :: i_core, i_component
    type(process_component_def_t), pointer :: config
    do i_core = 1, process%get_n_cores ()
       i_component = process%cm%i_core_to_first_i_component (i_core)
       config => process%meta%lib%get_component_def_ptr (process%meta%id, i_component)
       associate (core => process%cm%cores(i_core)%core)
          call core%init (config%get_core_def_ptr (), &
               process%meta%lib, process%meta%id, i_component)
       end associate
    end do
  end subroutine process_init_cores
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: init_blha_cores => process_init_blha_cores
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_init_blha_cores (process, blha_template, var_list)
    class(process_t), intent(inout) :: process
    type(blha_template_t), intent(inout) :: blha_template
    type(var_list_t), intent(in), pointer :: var_list
    integer :: i_core, i
    integer :: n_in, n_legs, n_flv, n_hel
    do i_core = 1, process%get_n_cores ()
       call fill_blha_template (process%get_nlo_type (i_core))
       select type (core => process%cm%cores(i_core)%core)
       class is (prc_blha_t)
          select type (pcm => process%pcm)
          type is (pcm_nlo_t)
             n_in = pcm%region_data%get_n_in ()
             if (process%cm%core_is_radiation(i_core)) then
                n_legs = pcm%region_data%get_n_legs_real ()
                n_flv = pcm%region_data%get_n_flv_real ()
             else
                n_legs = pcm%region_data%get_n_legs_born ()
                n_flv = pcm%region_data%get_n_flv_born ()
             end if
          class default
             n_in = core%data%n_in
             n_legs = core%data%get_n_tot ()
             n_flv = core%data%n_flv
          end select
          n_hel = process%term(process%get_i_term (i_core))%int%get_n_in_helicities ()
          call core%init_blha (blha_template, n_in, n_legs, n_flv, n_hel)
          call core%init_driver (process%config%os_data)
```

```
      end select
      call blha_template%reset ()
    end do
  contains
    function needs_entry (me_method) result (val)
      logical :: val
      type(string_t), intent(in) :: me_method
      val = char (me_method) == 'gosam' .or. char (me_method) == 'openloops'
    end function needs_entry

    subroutine fill_blha_template (nlo_type)
      integer, intent(in) :: nlo_type
      select case (nlo_type)
      case (BORN)
         if (needs_entry (var_list%get_sval (var_str ("$born_me_method")))) &
            call blha_template%set_born ()
      case (NLO_REAL)
         if (needs_entry (var_list%get_sval (var_str ("$real_tree_me_method")))) &
            call blha_template%set_real_trees ()
      case (NLO_VIRTUAL)
         if (needs_entry (var_list%get_sval (var_str ("$loop_me_method")))) &
            call blha_template%set_loop ()
      case (NLO_SUBTRACTION)
         if (needs_entry (var_list%get_sval (var_str ("$correlation_me_method")))) then
            call blha_template%set_subtraction ()
            call blha_template%set_internal_color_correlations ()
         end if
      end select
    end subroutine fill_blha_template
  end subroutine process_init_blha_cores
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: get_n_cores => process_get_n_cores
```

⟨*Process: procedures*⟩+≡

```
  function process_get_n_cores (process) result (n)
    integer :: n
    class(process_t), intent(in) :: process
    n = process%cm%n_cores
  end function process_get_n_cores
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: get_core_manager_index => process_get_core_manager_index
```

⟨*Process: procedures*⟩+≡

```
  function process_get_core_manager_index (process, i_core) result (i)
    integer :: i
    class(process_t), intent(in) :: process
    integer, intent(in) :: i_core
    i = process%cm%i_core_to_first_i_component (i_core)
  end function process_get_core_manager_index
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: get_core_manager => process_get_core_manager
```

⟨*Process: procedures*⟩+≡

```
function process_get_core_manager (process) result (cm)
  type(core_manager_t) :: cm
  class(process_t), intent(in) :: process
  cm = process%cm
end function process_get_core_manager
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_core_manager_ptr => process_get_core_manager_ptr
```

⟨*Process: procedures*⟩+≡

```
function process_get_core_manager_ptr (process) result (cm)
  type(core_manager_t), pointer :: cm
  class(process_t), intent(in), target :: process
  cm => process%cm
end function process_get_core_manager_ptr
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_base_i_term => process_get_base_i_term
```

⟨*Process: procedures*⟩+≡

```
function process_get_base_i_term (process, i_component) result (i_term)
  integer :: i_term
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_component
  i_term = process%component(i_component)%i_term(1)
end function process_get_base_i_term
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_core_term => process_get_core_term
```

⟨*Process: procedures*⟩+≡

```
function process_get_core_term (process, i_term) result (core)
  class(prc_core_t), pointer :: core
  class(process_t), intent(in), target :: process
  integer, intent(in) :: i_term
  integer :: i_core
  i_core = process%term(i_term)%i_core
  core => process%cm%cores(i_core)%core
end function process_get_core_term
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_subtraction_core => process_get_subtraction_core
```

⟨*Process: procedures*⟩+≡

```
function process_get_subtraction_core (process) result (core)
  class(prc_core_t), pointer :: core
  class(process_t), intent(in), target :: process
  core => process%cm%get_subtraction_core ()
end function process_get_subtraction_core
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_term_ptr => process_get_term_ptr
```

⟨*Process: procedures*⟩+≡

```
function process_get_term_ptr (process, i) result (term)
  type(process_term_t), pointer :: term
  class(process_t), intent(in), target :: process
  integer, intent(in) :: i
  term => process%term(i)
end function process_get_term_ptr
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_core_nlo_type => process_get_core_nlo_type
```

⟨*Process: procedures*⟩+≡

```
function process_get_core_nlo_type (process, md5sum) result (core)
  class(prc_core_t), pointer :: core
  class(process_t), intent(in), target :: process
  character(32), intent(in) :: md5sum
  integer :: i_core
  associate (cm => process%cm)
     do i_core = 1, N_MAX_CORES
        if (cm%md5s(i_core) == md5sum) exit
     end do
     core => cm%cores(i_core)%core
  end associate
end function process_get_core_nlo_type
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_i_core_nlo_type => process_get_i_core_nlo_type
```

⟨*Process: procedures*⟩+≡

```
function process_get_i_core_nlo_type (process, nlo_type, include_sub) result (i_core)
  integer :: i_core
  class(process_t), intent(in) :: process
  integer, intent(in) :: nlo_type
  logical, intent(in), optional :: include_sub
  logical :: skip_sub
  skip_sub = .false.
  if (present (include_sub)) skip_sub = .not. include_sub
  do i_core = 1, process%cm%n_cores
     if (skip_sub) then
        if (process%cm%sub(i_core)) cycle
     end if
     if (process%cm%nlo_type (i_core) == nlo_type) return
  end do
  i_core = -1
end function process_get_i_core_nlo_type
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_i_term => process_get_i_term
```

⟨*Process: procedures*⟩+≡

```
function process_get_i_term (process, i_core) result (i_term)
  integer :: i_term
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_core
```

```
    do i_term = 1, process%get_n_terms ()
       if (process%term(i_term)%i_core == i_core) return
    end do
    i_term = -1
  end function process_get_i_term
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: set_i_mci_work => process_set_i_mci_work
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_set_i_mci_work (process, i_mci)
    class(process_t), intent(inout) :: process
    integer, intent(in) :: i_mci
    process%mci_entry(i_mci)%i_mci = i_mci
  end subroutine process_set_i_mci_work
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: get_i_mci_work => process_get_i_mci_work
```

⟨*Process: procedures*⟩+≡

```
  pure function process_get_i_mci_work (process, i_mci) result (i_mci_work)
    integer :: i_mci_work
    class(process_t), intent(in) :: process
    integer, intent(in) :: i_mci
    i_mci_work = process%mci_entry(i_mci)%i_mci
  end function process_get_i_mci_work
```

⟨*Process: process: TBP*⟩+≡

```
  generic :: component_is_active => component_is_active_single
  procedure :: component_is_active_single => process_component_is_active_single
```

⟨*Process: procedures*⟩+≡

```
  elemental function process_component_is_active_single (process, i_comp) result (val)
    logical :: val
    class(process_t), intent(in) :: process
    integer, intent(in) :: i_comp
    val = process%component(i_comp)%is_active ()
  end function process_component_is_active_single
```

⟨*Process: process: TBP*⟩+≡

```
  generic :: component_is_active => component_is_active_all
  procedure :: component_is_active_all => process_component_is_active_all
```

⟨*Process: procedures*⟩+≡

```
  pure function process_component_is_active_all (process) result (val)
    logical, dimension(:), allocatable :: val
    class(process_t), intent(in) :: process
    allocate (val (size (process%component)))
    val = process%component%is_active ()
  end function process_component_is_active_all
```

30.5.1 Default iterations

If the user does not specify the passes and iterations for integration, we should be able to give reasonable defaults. These depend on the process, therefore we implement the following procedures as methods of the process object. The algorithm is not very sophisticated yet, it may be improved by looking at the process in more detail.

We investigate only the first process component, assuming that it characterizes the complexity of the process reasonable well.

The number of passes is limited to two: one for adaption, one for integration.

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_n_pass_default => process_get_n_pass_default
procedure :: adapt_grids_default => process_adapt_grids_default
procedure :: adapt_weights_default => process_adapt_weights_default
```

⟨*Process: procedures*⟩+≡

```
function process_get_n_pass_default (process) result (n_pass)
  class(process_t), intent(in) :: process
  integer :: n_pass
  integer :: n_eff
  type(process_component_def_t), pointer :: config
  config => process%component(1)%config
  n_eff = config%get_n_tot () - 2
  select case (n_eff)
  case (1)
     n_pass = 1
  case default
     n_pass = 2
  end select
end function process_get_n_pass_default

function process_adapt_grids_default (process, pass) result (flag)
  class(process_t), intent(in) :: process
  integer, intent(in) :: pass
  logical :: flag
  integer :: n_eff
  type(process_component_def_t), pointer :: config
  config => process%component(1)%config
  n_eff = config%get_n_tot () - 2
  select case (n_eff)
  case (1)
     flag = .false.
  case default
     select case (pass)
     case (1);  flag = .true.
     case (2);  flag = .false.
     case default
        call msg_bug ("adapt grids default: impossible pass index")
     end select
  end select
end function process_adapt_grids_default

function process_adapt_weights_default (process, pass) result (flag)
  class(process_t), intent(in) :: process
```

```
    integer, intent(in) :: pass
    logical :: flag
    integer :: n_eff
    type(process_component_def_t), pointer :: config
    config => process%component(1)%config
    n_eff = config%get_n_tot () - 2
    select case (n_eff)
    case (1)
       flag = .false.
    case default
       select case (pass)
       case (1);  flag = .true.
       case (2);  flag = .false.
       case default
          call msg_bug ("adapt weights default: impossible pass index")
       end select
    end select
  end function process_adapt_weights_default
```

The number of iterations and calls per iteration depends on the number of outgoing particles.

⟨*Process: process: TBP*⟩+≡

```
  procedure :: get_n_it_default => process_get_n_it_default
  procedure :: get_n_calls_default => process_get_n_calls_default
```

⟨*Process: procedures*⟩+≡

```
  function process_get_n_it_default (process, pass) result (n_it)
    class(process_t), intent(in) :: process
    integer, intent(in) :: pass
    integer :: n_it
    integer :: n_eff
    type(process_component_def_t), pointer :: config
    config => process%component(1)%config
    n_eff = config%get_n_tot () - 2
    select case (pass)
    case (1)
       select case (n_eff)
       case (1);   n_it = 1
       case (2);   n_it = 3
       case (3);   n_it = 5
       case (4:5); n_it = 10
       case (6);   n_it = 15
       case (7:);  n_it = 20
       end select
    case (2)
       select case (n_eff)
       case (:3);   n_it = 3
       case (4:);   n_it = 5
       end select
    end select
  end function process_get_n_it_default

  function process_get_n_calls_default (process, pass) result (n_calls)
    class(process_t), intent(in) :: process
```

```
    integer, intent(in) :: pass
    integer :: n_calls
    integer :: n_eff
    type(process_component_def_t), pointer :: config
    config => process%component(1)%config
    n_eff = config%get_n_tot () - 2
    select case (pass)
    case (1)
       select case (n_eff)
       case (1);   n_calls =   100
       case (2);   n_calls =  1000
       case (3);   n_calls =  5000
       case (4);   n_calls = 10000
       case (5);   n_calls = 20000
       case (6:);  n_calls = 50000
       end select
    case (2)
       select case (n_eff)
       case (:3);  n_calls =  10000
       case (4);   n_calls =  20000
       case (5);   n_calls =  50000
       case (6);   n_calls = 100000
       case (7:);  n_calls = 200000
       end select
    end select
  end function process_get_n_calls_default
```

30.5.2 Constant process data

The following methods return basic process data that stay constant after initialization.

The process and IDs.

⟨*Process: process: TBP*⟩+≡

```
  procedure :: get_id => process_get_id
  procedure :: get_num_id => process_get_num_id
  procedure :: get_run_id => process_get_run_id
  procedure :: get_library_name => process_get_library_name
```

⟨*Process: procedures*⟩+≡

```
  function process_get_id (process) result (id)
    class(process_t), intent(in) :: process
    type(string_t) :: id
    id = process%meta%id
  end function process_get_id

  function process_get_num_id (process) result (id)
    class(process_t), intent(in) :: process
    integer :: id
    id = process%meta%num_id
  end function process_get_num_id

  function process_get_run_id (process) result (id)
    class(process_t), intent(in) :: process
```

```
    type(string_t) :: id
    id = process%meta%run_id
  end function process_get_run_id

  function process_get_library_name (process) result (id)
    class(process_t), intent(in) :: process
    type(string_t) :: id
    id = process%meta%lib%get_name ()
  end function process_get_library_name
```

The number of incoming particles.

⟨*Process: process: TBP*⟩+≡

```
  procedure :: get_n_in => process_get_n_in
```

⟨*Process: procedures*⟩+≡

```
  function process_get_n_in (process) result (n)
    class(process_t), intent(in) :: process
    integer :: n
    n = process%config%n_in
  end function process_get_n_in
```

The number of MCI data sets.

⟨*Process: process: TBP*⟩+≡

```
  procedure :: get_n_mci => process_get_n_mci
```

⟨*Process: procedures*⟩+≡

```
  function process_get_n_mci (process) result (n)
    class(process_t), intent(in) :: process
    integer :: n
    n = process%config%n_mci
  end function process_get_n_mci
```

The number of process components, total.

⟨*Process: process: TBP*⟩+≡

```
  procedure :: get_n_components => process_get_n_components
```

⟨*Process: procedures*⟩+≡

```
  function process_get_n_components (process) result (n)
    class(process_t), intent(in) :: process
    integer :: n
    n = process%meta%n_components
  end function process_get_n_components
```

The number of process terms, total.

⟨*Process: process: TBP*⟩+≡

```
  procedure :: get_n_terms => process_get_n_terms
```

⟨*Process: procedures*⟩+≡

```
  function process_get_n_terms (process) result (n)
    class(process_t), intent(in) :: process
    integer :: n
    n = process%config%n_terms
  end function process_get_n_terms
```

Return the indices of the components that belong to a specific MCI entry.

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_i_component => process_get_i_component
```

⟨*Process: procedures*⟩+≡

```
subroutine process_get_i_component (process, i_mci, i_component)
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_mci
  integer, dimension(:), intent(out), allocatable :: i_component
  associate (mci_entry => process%mci_entry(i_mci))
    allocate (i_component (size (mci_entry%i_component)))
    i_component = mci_entry%i_component
  end associate
end subroutine process_get_i_component
```

Return the ID of a specific component.

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_component_id => process_get_component_id
```

⟨*Process: procedures*⟩+≡

```
function process_get_component_id (process, i_component) result (id)
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_component
  type(string_t) :: id
  id = process%meta%component_id(i_component)
end function process_get_component_id
```

Return a pointer to the definition of a specific component.

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_component_def_ptr => process_get_component_def_ptr
```

⟨*Process: procedures*⟩+≡

```
function process_get_component_def_ptr (process, i_component) result (ptr)
  type(process_component_def_t), pointer :: ptr
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_component
  ptr => process%meta%lib%get_component_def_ptr (process%meta%id, i_component)
end function process_get_component_def_ptr
```

These procedures extract and restore (by transferring the allocation) the process core. This is useful for changing process parameters from outside this module.

⟨*Process: process: TBP*⟩+≡

```
procedure :: extract_core => process_extract_core
procedure :: restore_core => process_restore_core
```

⟨*Process: procedures*⟩+≡

```
subroutine process_extract_core (process, i_term, core)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: i_term
  class(prc_core_t), intent(inout), allocatable :: core
  integer :: i_core
  i_core = process%term(i_term)%i_core
  call move_alloc (from = process%cm%cores(i_core)%core, to = core)
end subroutine process_extract_core
```

```
subroutine process_restore_core (process, i_term, core)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: i_term
  class(prc_core_t), intent(inout), allocatable :: core
  integer :: i_core
  i_core = process%term(i_term)%i_core
  call move_alloc (from = core, to = process%cm%cores(i_core)%core)
end subroutine process_restore_core
```

The block of process constants.

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_constants => process_get_constants
```

⟨*Process: procedures*⟩+≡

```
function process_get_constants (process, i_core) result (data)
  type(process_constants_t) :: data
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_core
  data = process%cm%cores(i_core)%core%data
end function process_get_constants
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_config => process_get_config
```

⟨*Process: procedures*⟩+≡

```
function process_get_config (process) result (config)
  type(process_config_data_t) :: config
  class(process_t), intent(in) :: process
  config = process%config
end function process_get_config
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_md5sum_constants => process_get_md5sum_constants
```

⟨*Process: procedures*⟩+≡

```
function process_get_md5sum_constants (process, i_component, &
   type_string, nlo_type) result (this_md5sum)
  character(32) :: this_md5sum
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_component
  type(string_t), intent(in) :: type_string
  integer, intent(in) :: nlo_type
  type(process_constants_t) :: data
  integer :: unit
  call process%meta%lib%fill_constants (process%meta%id, i_component, data)
  unit = data%fill_unit_for_md5sum (.false.)
  write (unit, '(A)') char(type_string)
  write (unit, '(I0)') nlo_type
  rewind (unit)
  this_md5sum = md5sum (unit)
  close (unit)
end function process_get_md5sum_constants
```

Return the set of outgoing flavors that are associated with a particular term. We deduce this from the effective interaction.

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_term_flv_out => process_get_term_flv_out
```

⟨*Process: procedures*⟩+≡

```
subroutine process_get_term_flv_out (process, i_term, flv)
  class(process_t), intent(in), target :: process
  integer, intent(in) :: i_term
  type(flavor_t), dimension(:,:), allocatable, intent(out) :: flv
  type(interaction_t), pointer :: int
  int => process%term(i_term)%int_eff
  if (.not. associated (int))  int => process%term(i_term)%int
  call interaction_get_flv_out (int, flv)
end subroutine process_get_term_flv_out
```

Return true if there is any unstable particle in any of the process terms. We decide this based on the provided model instance, not the one that is stored in the process object.

⟨*Process: process: TBP*⟩+≡

```
procedure :: contains_unstable => process_contains_unstable
```

⟨*Process: procedures*⟩+≡

```
function process_contains_unstable (process, model) result (flag)
  class(process_t), intent(in) :: process
  class(model_data_t), intent(in), target :: model
  logical :: flag
  integer :: i_term
  type(flavor_t), dimension(:,:), allocatable :: flv
  flag = .false.
  do i_term = 1, process%get_n_terms ()
     call process%get_term_flv_out (i_term, flv)
     call flv%set_model (model)
     flag = .not. all (flv%is_stable ())
     deallocate (flv)
     if (flag)  return
  end do
end function process_contains_unstable
```

The nominal process energy.

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_sqrts => process_get_sqrts
```

⟨*Process: procedures*⟩+≡

```
function process_get_sqrts (process) result (sqrts)
  class(process_t), intent(in) :: process
  real(default) :: sqrts
  sqrts = process%beam_config%data%get_sqrts ()
end function process_get_sqrts
```

The beam polarization in case of simple degrees.

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_polarization => process_get_polarization
```

⟨*Process: procedures*⟩+≡

```
function process_get_polarization (process) result (pol)
  class(process_t), intent(in) :: process
  real(default), dimension(2) :: pol
  pol = process%beam_config%data%get_polarization ()
end function process_get_polarization
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_meta => process_get_meta
```

⟨*Process: procedures*⟩+≡

```
function process_get_meta (process) result (meta)
  type(process_metadata_t) :: meta
  class(process_t), intent(in) :: process
  meta = process%meta
end function process_get_meta
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: has_matrix_element => process_has_matrix_element
```

⟨*Process: procedures*⟩+≡

```
function process_has_matrix_element (process, i, is_term_index) result (active)
  logical :: active
  class(process_t), intent(in) :: process
  integer, intent(in), optional :: i
  logical, intent(in), optional :: is_term_index
  integer :: i_component
  logical :: is_term
  is_term = .false.
  if (present (i)) then
     if (present (is_term_index)) is_term = is_term_index
     if (is_term) then
        i_component = process%term(i)%i_component
     else
        i_component = i
     end if
     active = process%component(i_component)%active
  else
     active = any (process%component%active)
  end if
end function process_has_matrix_element
```

Pointer to the beam data object.

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_beam_data_ptr => process_get_beam_data_ptr
```

⟨*Process: procedures*⟩+≡

```
function process_get_beam_data_ptr (process) result (beam_data)
  class(process_t), intent(in), target :: process
  type(beam_data_t), pointer :: beam_data
  beam_data => process%beam_config%data
end function process_get_beam_data_ptr
```

⟨*Process: process: TBP*⟩+≡
```
procedure :: get_beam_config => process_get_beam_config
```
⟨*Process: procedures*⟩+≡
```
function process_get_beam_config (process) result (beam_config)
  type(process_beam_config_t) :: beam_config
  class(process_t), intent(in) :: process
  beam_config = process%beam_config
end function process_get_beam_config
```

⟨*Process: process: TBP*⟩+≡
```
procedure :: get_beam_config_ptr => process_get_beam_config_ptr
```
⟨*Process: procedures*⟩+≡
```
function process_get_beam_config_ptr (process) result (beam_config)
  type(process_beam_config_t), pointer :: beam_config
  class(process_t), intent(in), target :: process
  beam_config => process%beam_config
end function process_get_beam_config_ptr
```

Return true if lab and c.m. frame coincide for this process.

⟨*Process: process: TBP*⟩+≡
```
procedure :: cm_frame => process_cm_frame
```
⟨*Process: procedures*⟩+≡
```
function process_cm_frame (process) result (flag)
  class(process_t), intent(in), target :: process
  logical :: flag
  type(beam_data_t), pointer :: beam_data
  beam_data => process%beam_config%data
  flag = beam_data%cm_frame ()
end function process_cm_frame
```

Get the PDF set currently in use, if any.

⟨*Process: process: TBP*⟩+≡
```
procedure :: get_pdf_set => process_get_pdf_set
```
⟨*Process: procedures*⟩+≡
```
function process_get_pdf_set (process) result (pdf_set)
  class(process_t), intent(in) :: process
  integer :: pdf_set
  pdf_set = process%beam_config%get_pdf_set ()
end function process_get_pdf_set
```

Get the beam spectrum file currently in use, if any.

⟨*Process: process: TBP*⟩+≡
```
procedure :: get_beam_file => process_get_beam_file
```
⟨*Process: procedures*⟩+≡
```
function process_get_beam_file (process) result (file)
  class(process_t), intent(in) :: process
  type(string_t) :: file
  file = process%beam_config%get_beam_file ()
end function process_get_beam_file
```

Pointer to the process variable list.

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_var_list_ptr => process_get_var_list_ptr
```

⟨*Process: procedures*⟩+≡

```
function process_get_var_list_ptr (process) result (ptr)
  class(process_t), intent(in), target :: process
  type(var_list_t), pointer :: ptr
  ptr => process%meta%var_list
end function process_get_var_list_ptr
```

Pointer to the common model.

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_model_ptr => process_get_model_ptr
```

⟨*Process: procedures*⟩+≡

```
function process_get_model_ptr (process) result (ptr)
  class(process_t), intent(in) :: process
  class(model_data_t), pointer :: ptr
  ptr => process%config%model
end function process_get_model_ptr
```

Use the embedded RNG factory to spawn a new random-number generator instance. (This modifies the state of the factory.)

⟨*Process: process: TBP*⟩+≡

```
procedure :: make_rng => process_make_rng
```

⟨*Process: procedures*⟩+≡

```
subroutine process_make_rng (process, rng)
  class(process_t), intent(inout) :: process
  class(rng_t), intent(out), allocatable :: rng
  if (allocated (process%config%rng_factory)) then
     call process%config%rng_factory%make (rng)
  else
     call msg_bug ("Process: make rng: factory not allocated")
  end if
end subroutine process_make_rng
```

30.5.3 Compute an amplitude

Each process variant should allow for computing an amplitude value directly, without generating a process instance.

The process component is selected by the index `i`. The term within the process component is selected by `j`. The momentum combination is transferred as the array `p`. The function sets the specific quantum state via the indices of a flavor `f`, helicity `h`, and color `c` combination. Each index refers to the list of flavor, helicity, and color states, respectively, as stored in the process data.

Optionally, we may set factorization and renormalization scale. If unset, the partonic c.m. energy is inserted.

The function checks arguments for validity. For invalid arguments (quantum states), we return zero.

⟨*Process: process: TBP*⟩+≡

```
procedure :: compute_amplitude => process_compute_amplitude
```

⟨*Process: procedures*⟩+≡

```
function process_compute_amplitude &
     (process, i_core, i, j, p, f, h, c, fac_scale, ren_scale, alpha_qcd_forced) &
     result (amp)
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_core
  integer, intent(in) :: i, j
  type(vector4_t), dimension(:), intent(in) :: p
  integer, intent(in) :: f, h, c
  real(default), intent(in), optional :: fac_scale, ren_scale
  real(default), intent(in), allocatable, optional :: alpha_qcd_forced
  class(prc_core_t), pointer :: core => null ()
  real(default) :: fscale, rscale
  real(default), allocatable :: aqcd_forced
  complex(default) :: amp
  amp = 0
  if (0 < i .and. i <= process%meta%n_components) then
     core => process%cm%get_core(i_core)
     if (process%component(i)%active) then
        !associate (data => process%component(i)%core%data)
        associate (data => core%data)
          if (size (p) == data%n_in + data%n_out &
               .and. 0 < f .and. f <= data%n_flv &
               .and. 0 < h .and. h <= data%n_hel &
               .and. 0 < c .and. c <= data%n_col) then
             if (present (fac_scale)) then
                fscale = fac_scale
             else
                fscale = sum (p(data%n_in+1:)) ** 1
             end if
             if (present (ren_scale)) then
                rscale = ren_scale
             else
                rscale = fscale
             end if
             if (present (alpha_qcd_forced)) then
                if (allocated (alpha_qcd_forced)) &
                     allocate (aqcd_forced, source = alpha_qcd_forced)
             end if
             amp = core%compute_amplitude (j, p, f, h, c, &
                fscale, rscale, aqcd_forced)
          end if
        end associate
     else
        amp = 0
     end if
  end if
end function process_compute_amplitude
```

Sanity check for the process library. We abort the program if it has changed after process initialization.

⟨*Process: process: TBP*⟩+≡

```
procedure :: check_library_sanity => process_check_library_sanity
```

⟨*Process: procedures*⟩+≡

```
subroutine process_check_library_sanity (process)
  class(process_t), intent(in) :: process
  if (associated (process%meta%lib)) then
     if (process%meta%lib%get_update_counter () /= process%meta%lib_update_counter) then
        call msg_fatal ("Process '" // char (process%get_id ()) &
             // "': library has been recompiled after integration")
     end if
  end if
end subroutine process_check_library_sanity
```

Execute this for testing purposes, to avoid a dangling pointer in the above sanity check. Effectively switches off the sanity check.

⟨*Process: process: TBP*⟩+≡

```
procedure :: nullify_library_pointer => process_nullify_library_pointer
```

⟨*Process: procedures*⟩+≡

```
subroutine process_nullify_library_pointer (process)
  class(process_t), intent(inout) :: process
  process%meta%lib => null ()
end subroutine process_nullify_library_pointer
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: set_component_type => process_set_component_type
```

⟨*Process: procedures*⟩+≡

```
subroutine process_set_component_type (process, i_component, i_type)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: i_component, i_type
  process%component(i_component)%component_type = i_type
end subroutine process_set_component_type
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: set_counter_mci_entry => process_set_counter_mci_entry
```

⟨*Process: procedures*⟩+≡

```
subroutine process_set_counter_mci_entry (process, i_mci, counter)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: i_mci
  type(process_counter_t), intent(in) :: counter
  process%mci_entry(i_mci)%counter = counter
end subroutine process_set_counter_mci_entry
```

This is for suppression of numerical noise in the integration results stored in the `process_mci_entry` type. As the error and efficiency enter the MD5 sum, we recompute it.

⟨*Process: process: TBP*⟩+≡

```
procedure :: pacify => process_pacify
```

⟨*Process: procedures*⟩+≡

```
subroutine process_pacify (process, efficiency_reset, error_reset)
  class(process_t), intent(inout) :: process
  logical, intent(in), optional :: efficiency_reset, error_reset
  logical :: eff_reset, err_reset
  integer :: i
  eff_reset = .false.
  err_reset = .false.
  if (present (efficiency_reset))  eff_reset = efficiency_reset
  if (present (error_reset))  err_reset = error_reset
  if (allocated (process%mci_entry)) then
     do i = 1, size (process%mci_entry)
        call process%mci_entry(i)%results%pacify (efficiency_reset)
        if (allocated (process%mci_entry(i)%mci)) then
           associate (mci => process%mci_entry(i)%mci)
             if (process%mci_entry(i)%mci%error_known &
                  .and. err_reset) &
                  mci%error = 0
             if (process%mci_entry(i)%mci%efficiency_known &
                  .and. eff_reset)  &
                  mci%efficiency = 1
             call mci%pacify (efficiency_reset, error_reset)
             call mci%compute_md5sum ()
           end associate
        end if
     end do
  end if
end subroutine process_pacify
```

The following methods are used only in the unit tests; the access process internals directly that would otherwise be hidden.

⟨*Process: process: TBP*⟩+≡

```
procedure :: test_allocate_sf_channels
procedure :: test_set_component_sf_channel
procedure :: test_get_mci_ptr
```

⟨*Process: procedures*⟩+≡

```
subroutine test_allocate_sf_channels (process, n)
  class(process_t), intent(inout) :: process
  integer, intent(in) :: n
  call process%beam_config%allocate_sf_channels (n)
end subroutine test_allocate_sf_channels

subroutine test_set_component_sf_channel (process, c)
  class(process_t), intent(inout) :: process
  integer, dimension(:), intent(in) :: c
  call process%component(1)%phs_config%set_sf_channel (c)
end subroutine test_set_component_sf_channel

subroutine test_get_mci_ptr (process, mci)
  class(process_t), intent(in), target :: process
  class(mci_t), intent(out), pointer :: mci
  mci => process%mci_entry(1)%mci
end subroutine test_get_mci_ptr
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: init_mci_work => process_init_mci_work
```

⟨*Process: procedures*⟩+≡

```
subroutine process_init_mci_work (process, mci_work, i)
  class(process_t), intent(in), target :: process
  type(mci_work_t), intent(out) :: mci_work
  integer, intent(in) :: i
  call mci_work%init (process%mci_entry(i))
end subroutine process_init_mci_work
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: setup_test_cores => process_setup_test_cores
```

⟨*Process: procedures*⟩+≡

```
subroutine process_setup_test_cores (process)
  class(process_t), intent(inout) :: process
  class(prc_core_t), allocatable :: core
  allocate (test_t :: core)
  call process%core_manager_register (BORN, 1, var_str ("test_me"))
  call process%allocate_cm_arrays (1)
  call process%allocate_core (1, core)
  call process%init_cores ()
end subroutine process_setup_test_cores
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: write_cm => process_write_cm
```

⟨*Process: procedures*⟩+≡

```
subroutine process_write_cm (process, unit)
  class(process_t), intent(in) :: process
  integer, intent(in), optional :: unit
  call process%cm%write (unit)
end subroutine process_write_cm
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_connected_states => process_get_connected_states
```

⟨*Process: procedures*⟩+≡

```
function process_get_connected_states (process, i_component, &
       connected_terms) result (connected)
  type(connected_state_t), dimension(:), allocatable :: connected
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_component
  type(connected_state_t), dimension(:), intent(in) :: connected_terms
  integer :: i, i_conn
  integer :: n_conn
  n_conn = 0
  do i = 1, process%get_n_terms ()
     if (process%term(i)%i_component == i_component) then
        n_conn = n_conn + 1
     end if
  end do
```

```
    allocate (connected (n_conn))
    i_conn = 1
    do i = 1, process%get_n_terms ()
       if (process%term(i)%i_component == i_component) then
          connected (i_conn) = connected_terms(i)
          i_conn = i_conn + 1
       end if
    end do
  end function process_get_connected_states
```

30.5.4 NLO specifics

These subroutines (and the NLO specific properties they work on) could potentially be moved to pcm_nlo_t and used more generically in process_t with an appropriate interface in pcm_t

⟨*Process: process: TBP*⟩+≡

```
  procedure :: init_nlo_settings => process_init_nlo_settings
```

⟨*Process: procedures*⟩+≡

```
  subroutine process_init_nlo_settings (process, var_list, fks_template)
    class(process_t), intent(inout) :: process
    type(var_list_t), intent(in), target :: var_list
    type(fks_template_t), intent(in), optional :: fks_template
    type(string_t) :: color_method
    select type (pcm => process%pcm)
    type is (pcm_nlo_t)
       call pcm%settings%init (var_list, fks_template)
       if (debug_active (D_SUBTRACTION) .or. debug_active (D_VIRTUAL)) &
             call pcm%settings%write ()
    class default
       call msg_fatal ("Attempt to set nlo_settings with a non-NLO pcm!")
    end select
  end subroutine process_init_nlo_settings
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: get_nlo_type => process_get_nlo_type
```

⟨*Process: procedures*⟩+≡

```
  elemental function process_get_nlo_type (process, i_core) result (nlo_type)
    integer :: nlo_type
    class(process_t), intent(in) :: process
    integer, intent(in) :: i_core
    nlo_type = process%cm%nlo_type(i_core)
  end function process_get_nlo_type
```

⟨*Process: process: TBP*⟩+≡

```
  generic :: get_nlo_type_component => get_nlo_type_component_single
  procedure :: get_nlo_type_component_single => process_get_nlo_type_component_single
```

⟨*Process: procedures*⟩+≡

```
  elemental function process_get_nlo_type_component_single (process, i_component) result (val)
    integer :: val
```

```
    class(process_t), intent(in) :: process
    integer, intent(in) :: i_component
    val = process%component(i_component)%get_nlo_type ()
  end function process_get_nlo_type_component_single
```

⟨*Process: process: TBP*⟩+≡

```
  generic :: get_nlo_type_component => get_nlo_type_component_all
  procedure :: get_nlo_type_component_all => process_get_nlo_type_component_all
```

⟨*Process: procedures*⟩+≡

```
  pure function process_get_nlo_type_component_all (process) result (val)
    integer, dimension(:), allocatable :: val
    class(process_t), intent(in) :: process
    allocate (val (size (process%component)))
    val = process%component%get_nlo_type ()
  end function process_get_nlo_type_component_all
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: is_nlo_calculation => process_is_nlo_calculation
```

⟨*Process: procedures*⟩+≡

```
  function process_is_nlo_calculation (process) result (nlo)
    logical :: nlo
    class(process_t), intent(in) :: process
    select type (pcm => process%pcm)
    type is (pcm_nlo_t)
       nlo = .true.
    class default
       nlo = .false.
    end select
  end function process_is_nlo_calculation
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: is_combined_nlo_integration &
       => process_is_combined_nlo_integration
```

⟨*Process: procedures*⟩+≡

```
  function process_is_combined_nlo_integration (process) result (combined)
    logical :: combined
    class(process_t), intent(in) :: process
    select type (pcm => process%pcm)
    type is (pcm_nlo_t)
       combined = pcm%settings%combined_integration
    class default
       combined = .false.
    end select
  end function process_is_combined_nlo_integration
```

⟨*Process: process: TBP*⟩+≡

```
  procedure :: component_is_real_finite => process_component_is_real_finite
```

⟨*Process: procedures*⟩+≡

```
pure function process_component_is_real_finite (process, i_component) &
       result (val)
  logical :: val
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_component
  val = process%component(i_component)%component_type == COMP_REAL_FIN
end function process_component_is_real_finite
```

Return nlo data of a process component

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_component_nlo_type => process_get_component_nlo_type
```

⟨*Process: procedures*⟩+≡

```
elemental function process_get_component_nlo_type (process, i_component) &
         result (nlo_type)
  integer :: nlo_type
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_component
  nlo_type = process%component(i_component)%config%get_nlo_type ()
end function process_get_component_nlo_type
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_component_associated_born &
          => process_get_component_associated_born
```

⟨*Process: procedures*⟩+≡

```
function process_get_component_associated_born (process, i_component) &
         result (i_born)
  class(process_t), intent(in) :: process
  integer, intent(in) :: i_component
  integer :: i_born
  i_born = process%component(i_component)%config%get_associated_born ()
end function process_get_component_associated_born
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_associated_real_component => process_get_associated_real_component
```

⟨*Process: procedures*⟩+≡

```
function process_get_associated_real_component (process, i_component) result (i_real)
   integer :: i_real
   class(process_t), intent(in) :: process
   integer, intent(in) :: i_component
   i_real = process%component(i_component)%config%get_associated_real ()
end function process_get_associated_real_component
```

⟨*Process: process: TBP*⟩+≡

```
procedure :: get_associated_real_fin => process_get_associated_real_fin
```

⟨*Process: procedures*⟩+≡

```
elemental function process_get_associated_real_fin (process, i_component) result (i_real)
   integer :: i_real
   class(process_t), intent(in) :: process
   integer, intent(in) :: i_component
```

```
      i_real = process%component(i_component)%config%get_associated_real_fin ()
   end function process_get_associated_real_fin
```

⟨*Process: process: TBP*⟩+≡

```
   procedure :: setup_region_data => process_setup_region_data
```

⟨*Process: procedures*⟩+≡

```
   subroutine process_setup_region_data (process, i_real, data_born, data_real)
     class(process_t), intent(inout) :: process
     integer, intent(in) :: i_real
     type(process_constants_t), intent(in) :: data_born, data_real
     integer, dimension (:,:), allocatable :: flavor_born, flavor_real
     type(resonance_history_t), dimension(:), allocatable :: resonance_histories
     logical :: success
     select type (pcm => process%pcm)
     type is (pcm_nlo_t)
        call data_born%get_flv_state (flavor_born)
        call data_real%get_flv_state (flavor_real)
        call pcm%region_data%setup_fks_mappings &
             (pcm%settings%fks_template, data_born%n_in)
        select type (model => process%config%model)
        type is (model_t)
           call pcm%region_data%init (data_born%n_in, model, &
                flavor_born, flavor_real)
           associate (template => pcm%settings%fks_template)
              if (template%mapping_type == FKS_RESONANCES) then
                 select type (phs_config => process%component(i_real)%phs_config)
                 type is (phs_fks_config_t)
                    call get_filtered_resonance_histories (phs_config, &
                         data_born%n_in, flavor_born, model, template%excluded_resonances, &
                         resonance_histories, success)
                 end select
                 if (.not. success) template%mapping_type = FKS_DEFAULT
              end if
              !!! Check again, mapping_type might have changed
              if (template%mapping_type == FKS_RESONANCES) then
                 call pcm%region_data%set_resonance_mappings (resonance_histories)
                 call pcm%region_data%init_resonance_information ()
                 pcm%settings%use_resonance_mappings = .true.
              end if
           end associate
        end select
        if (pcm%settings%factorization_mode == FACTORIZATION_THRESHOLD) then
            call pcm%region_data%set_isr_pseudo_regions ()
            call pcm%region_data%split_up_interference_regions_for_threshold ()
        end if
        call pcm%region_data%compute_number_of_phase_spaces ()
        call pcm%region_data%set_i_phs_to_i_con ()
        associate (var_list => process%meta%var_list)
           call pcm%region_data%write_to_file (process%meta%id, &
                var_list%get_lval (var_str ("?vis_fks_regions")), &
                process%config%os_data)
        end associate
        if (debug_active (D_SUBTRACTION)) call pcm%region_data%check_consistency (.true.)
```

```
      end select
    end subroutine process_setup_region_data
```

⟨*Process: process: TBP*⟩+≡

```
    procedure :: setup_real_partition => process_setup_real_partition
```

⟨*Process: procedures*⟩+≡

```
    subroutine process_setup_real_partition (process, partition_scale)
      class(process_t), intent(inout) :: process
      real(default), intent(in) :: partition_scale
      select type (pcm => process%pcm)
      type is (pcm_nlo_t)
         call pcm%setup_real_partition (partition_scale)
      end select
    end subroutine process_setup_real_partition
```

⟨*Process: process: TBP*⟩+≡

```
    procedure :: check_if_threshold_method => process_check_if_threshold_method
```

⟨*Process: procedures*⟩+≡

```
    subroutine process_check_if_threshold_method (process)
      class(process_t), intent(inout) :: process
      integer :: i_core
      associate (cm => process%cm)
         do i_core = 1, cm%n_cores
            select type (core => cm%cores(i_core)%core)
            type is (prc_threshold_t)
               select type (pcm => process%pcm)
               type is (pcm_nlo_t)
                  pcm%settings%factorization_mode = FACTORIZATION_THRESHOLD
               end select
            end select
         end do
      end associate
    end subroutine process_check_if_threshold_method
```

⟨*Process: process: TBP*⟩+≡

```
    procedure :: select_i_term => process_select_i_term
```

⟨*Process: procedures*⟩+≡

```
    pure function process_select_i_term (process, i_mci) result (i_term)
      integer :: i_term
      class(process_t), intent(in) :: process
      integer, intent(in) :: i_mci
      integer :: i_component, i_sub
      i_component = process%mci_entry(i_mci)%i_component(1)
      i_term = process%component(i_component)%i_term(1)
      i_sub = process%term(i_term)%i_sub
      if (i_sub > 0) &
         i_term = process%term(i_sub)%i_term_global
    end function process_select_i_term
```

⟨*Process: process: TBP*⟩+≡

```
    procedure :: create_blha_interface => process_create_blha_interface
```

⟨*Process: procedures*⟩+≡

```
subroutine process_create_blha_interface (process, flv_born, flv_real, n_in, beam_structure)
  class(process_t), intent(inout) :: process
  integer, intent(in), dimension(:,:), allocatable :: flv_born, flv_real
  integer, intent(in) :: n_in
  type(beam_structure_t), intent(in) :: beam_structure
  integer :: alpha_power, alphas_power
  type(blha_master_t) :: blha_master
  integer :: openloops_phs_tolerance, openloops_stability_log
  logical :: use_cms, use_collier
  type(string_t) :: openloops_extra_cmd
  type(string_t) :: ew_scheme
  type(process_component_def_t), pointer :: config => null ()
  config => process%meta%lib%get_component_def_ptr (process%meta%id, 1)
  call config%get_coupling_powers (alpha_power, alphas_power)
  associate (cm => process%cm)
     associate (var_list => process%meta%var_list)
        openloops_phs_tolerance = var_list%get_ival (var_str ("openloops_phs_tolerance"))
        openloops_stability_log = var_list%get_ival (var_str ("openloops_stability_log"))
        openloops_extra_cmd = var_list%get_sval (var_str ("$openloops_extra_cmd"))
        use_cms = var_list%get_lval (var_str ("?openloops_use_cms"))
        use_collier = var_list%get_lval (var_str ("?openloops_use_collier"))
        ew_scheme = var_list%get_sval (var_str ("$blha_ew_scheme"))
        call blha_master%set_ew_scheme (ew_scheme)
     end associate
     call blha_master%set_methods (process%is_nlo_calculation (), process%meta%var_list)
     call blha_master%allocate_config_files ()
     call blha_master%setup_additional_features (openloops_phs_tolerance, &
          use_cms, &
          openloops_stability_log, &
          use_collier, &
          extra_cmd = openloops_extra_cmd, &
          beam_structure = beam_structure)
     call blha_master%generate (process%meta%id, process%config%model, &
          n_in, alpha_power, alphas_power, flv_born, flv_real)
     call blha_master%write_olp (process%meta%id)
  end associate
end subroutine process_create_blha_interface
```

Would be better to do this at the level of the writer of the core but one has to bring NLO information there. Could also be moved to the `core_manager_t`.

⟨*Process: process: TBP*⟩+≡

```
procedure :: create_and_load_extra_libraries &
   => process_create_and_load_extra_libraries
```

⟨*Process: procedures*⟩+≡

```
subroutine process_create_and_load_extra_libraries &
     (process, beam_structure, var_list, os_data)
  class(process_t), intent(inout), target :: process
  type(beam_structure_t), intent(in) :: beam_structure
  type(var_list_t), intent(in) :: var_list
  type(os_data_t), intent(in) :: os_data
  type(string_t) :: libname
  integer :: i_component, i_core
```

```
logical, dimension(process%cm%n_cores) :: loaded
logical :: give_warning
integer :: n_in
integer, dimension(:,:), allocatable :: flv_born, flv_real
type(process_component_def_t), pointer :: config => null ()

loaded = .false.
select type (pcm => process%pcm)
type is (pcm_nlo_t)
   call pcm%region_data%get_all_flv_states (flv_born, flv_real)
   n_in = pcm%region_data%get_n_in ()
class default
   i_core = process%get_i_core_nlo_type (BORN)
   associate (core => process%cm%cores(i_core)%core)
      allocate (flv_born (core%data%get_n_tot (), core%data%n_flv))
      flv_born = core%data%flv_state
      n_in = core%data%n_in
   end associate
end select
if (process%needs_extra_code (only_blha = .true.)) &
     call process%create_blha_interface (flv_born, flv_real, n_in, beam_structure)
give_warning = .false.
do i_component = 1, process%meta%n_components
   config => process%meta%lib%get_component_def_ptr (process%meta%id, i_component)
   i_core = process%get_i_core_nlo_type (config%get_nlo_type ())
   if (config%can_be_integrated () .or. &
        process%get_nlo_type (i_core) == NLO_SUBTRACTION) then
      if (.not. loaded (i_core)) then
         select type (core => process%cm%cores(i_core)%core)
         class is (prc_user_defined_base_t)
            libname = process%get_library_name ()
            if (process%cm%core_is_radiation(i_core)) then
               if (allocated (flv_real)) then
                  call core%data%set_flv_state (flv_real)
               else
                  give_warning = .true.
               end if
               call core%create_and_load_extra_libraries &
                    (flv_real, var_list, os_data, libname, &
                    process%config%model, i_core)
            else
               if (allocated (flv_born)) then
                  call core%data%set_flv_state (flv_born)
               else
                  give_warning = .true.
               end if
               call core%create_and_load_extra_libraries &
                    (flv_born, var_list, os_data, libname, &
                    process%config%model, i_core)
            end if
         end select
         loaded(i_core) = .true.
      end if
   end if
```

```
    end do
    if (give_warning) call msg_warning ("Some flavor structures ", &
         [var_str ("are not allocated. This is totally fine if "), &
          var_str ("$method = 'threshold' is used, but you should "), &
          var_str ("have a closer look if this is not the case.")])
  end subroutine process_create_and_load_extra_libraries
```

30.6 Process config

⟨process_config.f90⟩≡

```
  ⟨File header⟩

  module process_config

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use format_utils, only: write_separator
    use io_units
    use md5
    use os_interface
    use diagnostics
    use sf_base
    use sf_mappings
    use sm_qcd
    use physics_defs
    use integration_results
    use model_data
    use interactions
    use quantum_numbers
    use flavors
    use helicities
    use colors
    use rng_base
    use state_matrices
    use process_libraries
    use process_constants
    use prc_core
    use prc_user_defined
    use prc_openloops, only: prc_openloops_t
    use prc_recola, only: prc_recola_t
    use prc_threshold, only: prc_threshold_t
    use beams
    use mci_base
    use beam_structures
    use phs_base
    use variables
    use expr_base

  ⟨Standard module head⟩

  ⟨Process config: public⟩
```

⟨*Process config: parameters*⟩

⟨*Process config: types*⟩

```
contains
```

⟨*Process config: procedures*⟩

```
end module process_config
```

⟨*Process config: parameters*⟩≡

```
integer, parameter, public :: COMP_DEFAULT = 0
integer, parameter, public :: COMP_REAL_FIN = 1
integer, parameter, public :: COMP_MASTER = 2
integer, parameter, public :: COMP_VIRT = 3
integer, parameter, public :: COMP_REAL = 4
integer, parameter, public :: COMP_REAL_SING = 5
integer, parameter, public :: COMP_MISMATCH = 6
integer, parameter, public :: COMP_PDF = 7
integer, parameter, public :: COMP_SUB = 8
integer, parameter, public :: COMP_RESUM = 9
```

30.6.1 Generic configuration data

This information concerns physical and technical properties of the process. It is fixed upon initialization, using data from the process specification and the variable list.

The number `n_in` is the number of incoming beam particles, simultaneously the number of incoming partons, 1 for a decay and 2 for a scattering process. (The number of outgoing partons may depend on the process component.)

The number `n_components` is the number of components that constitute the current process.

The number `n_terms` is the number of distinct contributions to the scattering matrix that constitute the current process. Each component may generate several terms.

The number `n_mci` is the number of independent MC integration configurations that this process uses. Distinct process components that share a MCI configuration may be combined pointwise. (Nevertheless, a given MC variable set may correspond to several "nearby" kinematical configurations.) This is also the number of distinct sampling-function results that this process can generate. Process components that use distinct variable sets are added only once after an integration pass has completed.

The `model` pointer identifies the physics model and its parameters. This is a pointer to an external object.

The `rng_factory` component spawns independent random-number generators for use in integration, event generation, and event postprocessing.

Various `parse_node_t` objects are taken from the SINDARIN input. They encode expressions for evaluating cuts and scales. The workspaces for evaluating those expressions are set up in the `effective_state` subobjects. Note that these are really pointers, so the actual nodes are not stored inside the process object.

The `md5sum` is taken and used to verify the process configuration when re-reading data from file.

⟨*Process config: public*⟩≡
```
public :: process_config_data_t
```

⟨*Process config: types*⟩≡
```
type :: process_config_data_t
   integer :: n_in = 0
   integer :: n_components = 0
   integer :: n_terms = 0
   integer :: n_mci = 0
   type(os_data_t) :: os_data
   class(rng_factory_t), allocatable :: rng_factory
   type(string_t) :: model_name
   class(model_data_t), pointer :: model => null ()
   type(qcd_t) :: qcd
   class(expr_factory_t), allocatable :: ef_cuts
   class(expr_factory_t), allocatable :: ef_scale
   class(expr_factory_t), allocatable :: ef_fac_scale
   class(expr_factory_t), allocatable :: ef_ren_scale
   class(expr_factory_t), allocatable :: ef_weight
   character(32) :: md5sum = ""
 contains
 ⟨Process config: process config data: TBP⟩
end type process_config_data_t
```

Here, we may compress the expressions for cuts etc.

⟨*Process config: process config data: TBP*⟩≡
```
procedure :: write => process_config_data_write
```

⟨*Process config: procedures*⟩≡
```
subroutine process_config_data_write (config, u, &
     counters, os_data, rng_factory, model, expressions)
  class(process_config_data_t), intent(in) :: config
  integer, intent(in) :: u
  logical, intent(in) :: counters
  logical, intent(in) :: os_data
  logical, intent(in) :: rng_factory
  logical, intent(in) :: model
  logical, intent(in) :: expressions
  write (u, "(1x,A)") "Configuration data:"
  if (counters) then
     write (u, "(3x,A,I0)") "Number of incoming particles = ", &
          config%n_in
     write (u, "(3x,A,I0)") "Number of process components = ", &
          config%n_components
     write (u, "(3x,A,I0)") "Number of process terms      = ", &
          config%n_terms
     write (u, "(3x,A,I0)") "Number of MCI configurations = ", &
          config%n_mci
  end if
  if (os_data) then
     call os_data_write (config%os_data, u)
  end if
```

```
  if (associated (config%model)) then
     write (u, "(3x,A,A)")  "Model = ", char (config%model_name)
     if (model) then
        call write_separator (u)
        call config%model%write (u)
        call write_separator (u)
     end if
  else
     write (u, "(3x,A,A,A)")  "Model = ", char (config%model_name), &
          " [not associated]"
  end if
  call config%qcd%write (u, show_md5sum = .false.)
  if (rng_factory) then
     if (allocated (config%rng_factory)) then
        write (u, "(2x)", advance = "no")
        call config%rng_factory%write (u)
     end if
  end if
  call write_separator (u)
  if (expressions) then
     if (allocated (config%ef_cuts)) then
        call write_separator (u)
        write (u, "(3x,A)") "Cut expression:"
        call config%ef_cuts%write (u)
     end if
     if (allocated (config%ef_scale)) then
        call write_separator (u)
        write (u, "(3x,A)") "Scale expression:"
        call config%ef_scale%write (u)
     end if
     if (allocated (config%ef_fac_scale)) then
        call write_separator (u)
        write (u, "(3x,A)") "Factorization scale expression:"
        call config%ef_fac_scale%write (u)
     end if
     if (allocated (config%ef_ren_scale)) then
        call write_separator (u)
        write (u, "(3x,A)") "Renormalization scale expression:"
        call config%ef_ren_scale%write (u)
     end if
     if (allocated (config%ef_weight)) then
        call write_separator (u)
        write (u, "(3x,A)") "Weight expression:"
        call config%ef_weight%write (u)
     end if
  else
     call write_separator (u)
     write (u, "(3x,A)") "Expressions (cut, scales, weight): [not shown]"
  end if
  if (config%md5sum /= "") then
     call write_separator (u)
     write (u, "(3x,A,A,A)")  "MD5 sum (config)  = '", config%md5sum, "'"
  end if
end subroutine process_config_data_write
```

Initialize. We use information from the process metadata and from the process library, given the process ID. We also store the currently active OS data set.

The model that we import here should be an instance of the global model. If we create the instance outside, we can make it an extension of the basic `model_data_t`. This will allow us to synchronize the variable list with the model. Using a pointer instead of an allocatable, we can guarantee the `target` attribute.

The RNG factory object is imported by moving the allocation.

⟨*Process config: process config data: TBP*⟩+≡
```
procedure :: init => process_config_data_init
```

⟨*Process config: procedures*⟩+≡
```
subroutine process_config_data_init &
     (config, meta, os_data, qcd, rng_factory, model)
  class(process_config_data_t), intent(out) :: config
  type(process_metadata_t), intent(in) :: meta
  type(os_data_t), intent(in) :: os_data
  type(qcd_t), intent(in) :: qcd
  class(rng_factory_t), intent(inout), allocatable :: rng_factory
  class(model_data_t), intent(inout), pointer :: model
  config%n_in = meta%lib%get_n_in (meta%id)
  config%n_components = size (meta%component_id)
  config%os_data = os_data
  config%qcd = qcd
  call move_alloc (from = rng_factory, to = config%rng_factory)
  config%model_name = model%get_name ()
  config%model => model
  model => null ()
end subroutine process_config_data_init
```

Since the captured model is a separate object allocated via a pointer, we need a finalizer.

⟨*Process config: process config data: TBP*⟩+≡
```
procedure :: final => process_config_data_final
```

⟨*Process config: procedures*⟩+≡
```
subroutine process_config_data_final (config)
  class(process_config_data_t), intent(inout) :: config
  if (associated (config%model)) then
     call config%model%final ()
     deallocate (config%model)
  end if
end subroutine process_config_data_final
```

Compute the MD5 sum of the configuration data. This encodes, in particular, the model and the expressions for cut, scales, weight, etc. It should not contain the IDs and number of components, etc., since the MD5 sum should be useful for integrating individual components.

This is done only once. If the MD5 sum is nonempty, the calculation is skipped.

⟨*Process config: process config data: TBP*⟩+≡
```
procedure :: compute_md5sum => process_config_data_compute_md5sum
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_config_data_compute_md5sum (config)
  class(process_config_data_t), intent(inout) :: config
  integer :: u
  if (config%md5sum == "") then
     u = free_unit ()
     open (u, status = "scratch", action = "readwrite")
     call config%write (u, counters = .false., os_data = .false., &
          rng_factory = .false., model = .true., expressions = .true.)
     rewind (u)
     config%md5sum = md5sum (u)
     close (u)
  end if
end subroutine process_config_data_compute_md5sum
```

⟨*Process config: process config data: TBP*⟩+≡

```
procedure :: get_md5sum => process_config_data_get_md5sum
```

⟨*Process config: procedures*⟩+≡

```
pure function process_config_data_get_md5sum (config) result (md5)
  character(32) :: md5
  class(process_config_data_t), intent(in) :: config
  md5 = config%md5sum
end function process_config_data_get_md5sum
```

30.6.2 Metadata

This information describes the process and its environment. It is fixed upon initialization.

The `id` string is the name of the process object, as given by the user. The matrix element generator will use this string for naming Fortran procedures and types, so it should qualify as a Fortran name.

The `num_id` is meaningful if nonzero. It is used for communication with external programs or file standards which do not support string IDs.

The `run_id` string distinguishes among several runs for the same process. It identifies process instances with respect to adapted integration grids and similar run-specific data. The run ID is kept when copying processes for creating instances, however, so it does not distinguish event samples.

The `var_list` is a snapshot of the variable list, taken at the point where the process was initialized.

The `lib` pointer accesses the process library where the process definition and the process driver are located.

The `lib_index` is the index of entry in the process library that corresponds to the current process.

The `component_id` array identifies the individual process components.

The `component_description` is an array of human-readable strings that characterize the process components, for instance `a, b => c, d`.

The `active` mask array marks those components which are active. The others are skipped.

⟨*Process config: public*⟩+≡

```
public :: process_metadata_t
```

⟨*Process config: types*⟩+≡

```
type :: process_metadata_t
   integer :: type = PRC_UNKNOWN
   type(string_t) :: id
   integer :: num_id = 0
   type(string_t) :: run_id
   type(var_list_t) :: var_list
   type(process_library_t), pointer :: lib => null ()
   integer :: lib_update_counter = 0
   integer :: lib_index = 0
   integer :: n_components = 0
   type(string_t), dimension(:), allocatable :: component_id
   type(string_t), dimension(:), allocatable :: component_description
   logical, dimension(:), allocatable :: active
 contains
 ⟨Process config: process metadata: TBP⟩
end type process_metadata_t
```

The local var list is a snapshot and needs a finalizer.

⟨*Process config: process metadata: TBP*⟩≡

```
procedure :: final => process_metadata_final
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_metadata_final (meta)
  class(process_metadata_t), intent(inout) :: meta
  call meta%var_list%final (follow_link=.true.)
end subroutine process_metadata_final
```

Output: ID and run ID. We write the variable list only upon request.

⟨*Process config: process metadata: TBP*⟩+≡

```
procedure :: write => process_metadata_write
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_metadata_write (meta, u, var_list, screen)
  class(process_metadata_t), intent(in) :: meta
  integer, intent(in) :: u
  logical, intent(in) :: var_list, screen
  integer :: i
  select case (meta%type)
  case (PRC_UNKNOWN)
     if (screen) then
        write (msg_buffer, "(A)") "Process [undefined]"
     else
        write (u, "(1x,A)") "Process [undefined]"
     end if
     return
  case (PRC_DECAY)
     if (screen) then
        write (msg_buffer, "(A,1x,A,A,A)") "Process [decay]:", &
             "'", char (meta%id), "'"
     else
        write (u, "(1x,A)", advance="no") "Process [decay]:"
     end if
  case (PRC_SCATTERING)
```

```
   if (screen) then
      write (msg_buffer, "(A,1x,A,A,A)") "Process [scattering]:", &
           "'", char (meta%id), "'"
   else
      write (u, "(1x,A)", advance="no") "Process [scattering]:"
   end if
case default
   call msg_bug ("process_write: undefined process type")
end select
if (screen)  then
   call msg_message ()
else
   write (u, "(1x,A,A,A)") "'", char (meta%id), "'"
end if
if (meta%num_id /= 0) then
   if (screen) then
      write (msg_buffer, "(2x,A,I0)") "ID (num)      = ", meta%num_id
      call msg_message ()
   else
      write (u, "(3x,A,I0)") "ID (num)      = ", meta%num_id
   end if
end if
if (screen) then
   if (meta%run_id /= "") then
      write (msg_buffer, "(2x,A,A,A)") "Run ID        = '", &
           char (meta%run_id), "'"
      call msg_message ()
   end if
else
   write (u, "(3x,A,A,A)") "Run ID        = '", char (meta%run_id), "'"
end if
if (associated (meta%lib)) then
   if (screen) then
      write (msg_buffer, "(2x,A,A,A)")  "Library name  = '", &
           char (meta%lib%get_name ()), "'"
      call msg_message ()
   else
      write (u, "(3x,A,A,A)")  "Library name  = '", &
           char (meta%lib%get_name ()), "'"
   end if
else
   if (screen) then
      write (msg_buffer, "(2x,A)")  "Library name  = [not associated]"
      call msg_message ()
   else
      write (u, "(3x,A)")  "Library name  = [not associated]"
   end if
end if
if (screen) then
   write (msg_buffer, "(2x,A,I0)")  "Process index = ", meta%lib_index
   call msg_message ()
else
   write (u, "(3x,A,I0)")  "Process index = ", meta%lib_index
end if
```

```
    if (allocated (meta%component_id)) then
       if (screen) then
          if (any (meta%active)) then
             write (msg_buffer, "(2x,A)")  "Process components:"
          else
             write (msg_buffer, "(2x,A)")  "Process components: [none]"
          end if
          call msg_message ()
       else
          write (u, "(3x,A)")  "Process components:"
       end if
       do i = 1, size (meta%component_id)
          if (.not. meta%active(i))  cycle
          if (screen) then
             write (msg_buffer, "(4x,I0,9A)")  i, ": '", &
                  char (meta%component_id (i)), "':   ", &
                  char (meta%component_description (i))
             call msg_message ()
          else
             write (u, "(5x,I0,9A)")  i, ": '", &
                  char (meta%component_id (i)), "':   ", &
                  char (meta%component_description (i))
          end if
       end do
    end if
    if (screen) then
       write (msg_buffer, "(A)")  repeat ("-", 72)
       call msg_message ()
    else
       call write_separator (u)
    end if
    if (screen)  return
    if (var_list) then
       write (u, "(1x,A)")  "Variable list:"
       call write_separator (u)
       call var_list_write (meta%var_list, u)
    else
       write (u, "(1x,A)")  "Variable list: [not shown]"
    end if
  end subroutine process_metadata_write
```

Short output: list components.

⟨*Process config: process metadata: TBP*⟩+≡

```
  procedure :: show => process_metadata_show
```

⟨*Process config: procedures*⟩+≡

```
  subroutine process_metadata_show (meta, u, model_name)
    class(process_metadata_t), intent(in) :: meta
    integer, intent(in) :: u
    type(string_t), intent(in) :: model_name
    integer :: i
    select case (meta%type)
    case (PRC_UNKNOWN)
       write (u, "(A)") "Process: [undefined]"
```

```
      return
   case default
      write (u, "(A)", advance="no") "Process:"
   end select
   write (u, "(1x,A)", advance="no") char (meta%id)
   select case (meta%num_id)
   case (0)
   case default
      write (u, "(1x,'(',I0,')')", advance="no") meta%num_id
   end select
   select case (char (model_name))
   case ("")
   case default
      write (u, "(1x,'[',A,']')", advance="no")  char (model_name)
   end select
   write (u, *)
   if (allocated (meta%component_id)) then
      do i = 1, size (meta%component_id)
         if (meta%active(i)) then
            write (u, "(2x,I0,':',1x,A)")  i, &
                 char (meta%component_description (i))
         end if
      end do
   end if
end subroutine process_metadata_show
```

Initialize.

⟨*Process config: process metadata: TBP*⟩+≡

```
procedure :: init => process_metadata_init
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_metadata_init (meta, id, run_id, lib)
   class(process_metadata_t), intent(out) :: meta
   type(string_t), intent(in) :: id
   type(string_t), intent(in) :: run_id
   type(process_library_t), intent(in), target :: lib
   select case (lib%get_n_in (id))
   case (1);  meta%type = PRC_DECAY
   case (2);  meta%type = PRC_SCATTERING
   case default
      call msg_bug ("Process '" // char (id) // "': impossible n_in")
   end select
   meta%id = id
   meta%run_id = run_id
   meta%lib => lib
   meta%lib_update_counter = lib%get_update_counter ()
   meta%lib_index = lib%get_entry_index (id)
   meta%num_id = lib%get_num_id (id)
   call lib%get_component_list (id, meta%component_id)
   meta%n_components = size (meta%component_id)
   call lib%get_component_description_list (id, meta%component_description)
   allocate (meta%active (meta%n_components), source = .true.)
end subroutine process_metadata_init
```

Mark a component as inactive.

⟨*Process config: process metadata: TBP*⟩+≡
```
procedure :: deactivate_component => process_metadata_deactivate_component
```

⟨*Process config: procedures*⟩+≡
```
subroutine process_metadata_deactivate_component (meta, i)
  class(process_metadata_t), intent(inout) :: meta
  integer, intent(in) :: i
  call msg_message ("Process component '" &
       // char (meta%component_id(i)) // "': matrix element vanishes")
  meta%active(i) = .false.
end subroutine process_metadata_deactivate_component
```

30.6.3 Beam configuration

The object `data` holds all details about the initial beam configuration. The allocatable array `sf` holds the structure-function configuration blocks. There are `n_strfun` entries in the structure-function chain (not counting the initial beam object). We maintain `n_channel` independent parameterizations of this chain. If this is greater than zero, we need a multi-channel sampling algorithm, where for each point one channel is selected to generate kinematics.

The number of parameters that are required for generating a structure-function chain is `n_sfpar`.

The flag `azimuthal_dependence` tells whether the process setup is symmetric about the beam axis in the c.m. system. This implies that there is no transversal beam polarization. The flag `lab_is_cm_frame` is obvious.

⟨*Process config: public*⟩+≡
```
public :: process_beam_config_t
```

⟨*Process config: types*⟩+≡
```
type :: process_beam_config_t
   type(beam_data_t) :: data
   integer :: n_strfun = 0
   integer :: n_channel = 1
   integer :: n_sfpar = 0
   type(sf_config_t), dimension(:), allocatable :: sf
   type(sf_channel_t), dimension(:), allocatable :: sf_channel
   logical :: azimuthal_dependence = .false.
   logical :: lab_is_cm_frame = .true.
   character(32) :: md5sum = ""
   logical :: sf_trace = .false.
   type(string_t) :: sf_trace_file
 contains
 ⟨Process config: process beam config: TBP⟩
end type process_beam_config_t
```

Here we write beam data only if they are actually used.

The `verbose` flag is passed to the beam-data writer.

⟨*Process config: process beam config: TBP*⟩≡
```
procedure :: write => process_beam_config_write
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_beam_config_write (object, unit, verbose)
  class(process_beam_config_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: u, i, c
  u = given_output_unit (unit)
  call object%data%write (u, verbose = verbose)
  if (object%data%initialized) then
     write (u, "(3x,A,L1)")  "Azimuthal dependence    = ", &
          object%azimuthal_dependence
     write (u, "(3x,A,L1)")  "Lab frame is c.m. frame = ", &
          object%lab_is_cm_frame
     if (object%md5sum /= "") then
        write (u, "(3x,A,A,A)")  "MD5 sum (beams/strf) = '", &
             object%md5sum, "'"
     end if
     if (allocated (object%sf)) then
        do i = 1, size (object%sf)
           call object%sf(i)%write (u)
        end do
        if (any_sf_channel_has_mapping (object%sf_channel)) then
           write (u, "(1x,A,L1)")  "Structure-function mappings per channel:"
           do c = 1, object%n_channel
              write (u, "(3x,I0,':')", advance="no")  c
              call object%sf_channel(c)%write (u)
           end do
        end if
     end if
  end if
end subroutine process_beam_config_write
```

The beam data have a finalizer. We assume that there is none for the structure-function data.

⟨*Process config: process beam config: TBP*⟩+≡

```
procedure :: final => process_beam_config_final
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_beam_config_final (object)
  class(process_beam_config_t), intent(inout) :: object
  call object%data%final ()
end subroutine process_beam_config_final
```

Initialize the beam setup with a given beam structure object.

⟨*Process config: process beam config: TBP*⟩+≡

```
procedure :: init_beam_structure => process_beam_config_init_beam_structure
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_beam_config_init_beam_structure &
     (beam_config, beam_structure, sqrts, model, decay_rest_frame)
  class(process_beam_config_t), intent(out) :: beam_config
  type(beam_structure_t), intent(in) :: beam_structure
  logical, intent(in), optional :: decay_rest_frame
  real(default), intent(in) :: sqrts
```

```
    class(model_data_t), intent(in), target :: model
    call beam_config%data%init_structure (beam_structure, &
         sqrts, model, decay_rest_frame)
    beam_config%lab_is_cm_frame = beam_config%data%cm_frame ()
  end subroutine process_beam_config_init_beam_structure
```

Initialize the beam setup for a scattering process with specified flavor combination, other properties taken from the beam structure object (if any).

⟨*Process config: process beam config: TBP*⟩+≡

```
  procedure :: init_scattering => process_beam_config_init_scattering
```

⟨*Process config: procedures*⟩+≡

```
  subroutine process_beam_config_init_scattering &
       (beam_config, flv_in, sqrts, beam_structure)
    class(process_beam_config_t), intent(out) :: beam_config
    type(flavor_t), dimension(2), intent(in) :: flv_in
    real(default), intent(in) :: sqrts
    type(beam_structure_t), intent(in), optional :: beam_structure
    if (present (beam_structure)) then
       if (beam_structure%polarized ()) then
          call beam_config%data%init_sqrts (sqrts, flv_in, &
               beam_structure%get_smatrix (), beam_structure%get_pol_f ())
       else
          call beam_config%data%init_sqrts (sqrts, flv_in)
       end if
    else
       call beam_config%data%init_sqrts (sqrts, flv_in)
    end if
  end subroutine process_beam_config_init_scattering
```

Initialize the beam setup for a decay process with specified flavor, other properties taken from the beam structure object (if present).

For a cascade decay, we set `rest_frame` to false, indicating a event-wise varying momentum. The beam data itself are initialized for the particle at rest.

⟨*Process config: process beam config: TBP*⟩+≡

```
  procedure :: init_decay => process_beam_config_init_decay
```

⟨*Process config: procedures*⟩+≡

```
  subroutine process_beam_config_init_decay &
       (beam_config, flv_in, rest_frame, beam_structure)
    class(process_beam_config_t), intent(out) :: beam_config
    type(flavor_t), dimension(1), intent(in) :: flv_in
    logical, intent(in), optional :: rest_frame
    type(beam_structure_t), intent(in), optional :: beam_structure
    if (present (beam_structure)) then
       if (beam_structure%polarized ()) then
          call beam_config%data%init_decay (flv_in, &
               beam_structure%get_smatrix (), beam_structure%get_pol_f (), &
               rest_frame = rest_frame)
       else
          call beam_config%data%init_decay (flv_in, rest_frame = rest_frame)
       end if
    else
```

```
      call beam_config%data%init_decay (flv_in, &
           rest_frame = rest_frame)
    end if
    beam_config%lab_is_cm_frame = beam_config%data%cm_frame ()
  end subroutine process_beam_config_init_decay
```

Print an informative message.

⟨*Process config: process beam config: TBP*⟩+≡

```
  procedure :: startup_message => process_beam_config_startup_message
```

⟨*Process config: procedures*⟩+≡

```
  subroutine process_beam_config_startup_message &
      (beam_config, unit, beam_structure)
    class(process_beam_config_t), intent(in) :: beam_config
    integer, intent(in), optional :: unit
    type(beam_structure_t), intent(in), optional :: beam_structure
    integer :: u
    u = free_unit ()
    open (u, status="scratch", action="readwrite")
    if (present (beam_structure)) then
       call beam_structure%write (u)
    end if
    call beam_config%data%write (u)
    rewind (u)
    do
       read (u, "(1x,A)", end=1)  msg_buffer
       call msg_message ()
    end do
1   continue
    close (u)
  end subroutine process_beam_config_startup_message
```

Allocate the structure-function array.

⟨*Process config: process beam config: TBP*⟩+≡

```
  procedure :: init_sf_chain => process_beam_config_init_sf_chain
```

⟨*Process config: procedures*⟩+≡

```
  subroutine process_beam_config_init_sf_chain &
      (beam_config, sf_config, sf_trace_file)
    class(process_beam_config_t), intent(inout) :: beam_config
    type(sf_config_t), dimension(:), intent(in) :: sf_config
    type(string_t), intent(in), optional :: sf_trace_file
    integer :: i
    beam_config%n_strfun = size (sf_config)
    allocate (beam_config%sf (beam_config%n_strfun))
    do i = 1, beam_config%n_strfun
       associate (sf => sf_config(i))
         call beam_config%sf(i)%init (sf%i, sf%data)
         if (.not. sf%data%is_generator ()) then
            beam_config%n_sfpar = beam_config%n_sfpar + sf%data%get_n_par ()
         end if
       end associate
    end do
    if (present (sf_trace_file)) then
```

```
      beam_config%sf_trace = .true.
      beam_config%sf_trace_file = sf_trace_file
    end if
  end subroutine process_beam_config_init_sf_chain
```

Allocate the structure-function mapping channel array, given the requested number of channels.

⟨*Process config: process beam config: TBP*⟩+≡

```
  procedure :: allocate_sf_channels => process_beam_config_allocate_sf_channels
```

⟨*Process config: procedures*⟩+≡

```
  subroutine process_beam_config_allocate_sf_channels (beam_config, n_channel)
    class(process_beam_config_t), intent(inout) :: beam_config
    integer, intent(in) :: n_channel
    beam_config%n_channel = n_channel
    call allocate_sf_channels (beam_config%sf_channel, &
         n_channel = n_channel, &
         n_strfun = beam_config%n_strfun)
  end subroutine process_beam_config_allocate_sf_channels
```

Set a structure-function mapping channel for an array of structure-function entries, for a single channel. (The default is no mapping.)

⟨*Process config: process beam config: TBP*⟩+≡

```
  procedure :: set_sf_channel => process_beam_config_set_sf_channel
```

⟨*Process config: procedures*⟩+≡

```
  subroutine process_beam_config_set_sf_channel (beam_config, c, sf_channel)
    class(process_beam_config_t), intent(inout) :: beam_config
    integer, intent(in) :: c
    type(sf_channel_t), intent(in) :: sf_channel
    beam_config%sf_channel(c) = sf_channel
  end subroutine process_beam_config_set_sf_channel
```

Print an informative startup message.

⟨*Process config: process beam config: TBP*⟩+≡

```
  procedure :: sf_startup_message => process_beam_config_sf_startup_message
```

⟨*Process config: procedures*⟩+≡

```
  subroutine process_beam_config_sf_startup_message &
       (beam_config, sf_string, unit)
    class(process_beam_config_t), intent(in) :: beam_config
    type(string_t), intent(in) :: sf_string
    integer, intent(in), optional :: unit
    if (beam_config%n_strfun > 0) then
       call msg_message ("Beam structure: " // char (sf_string), unit = unit)
       write (msg_buffer, "(A,3(1x,I0,1x,A))") &
            "Beam structure:", &
            beam_config%n_channel, "channels,", &
            beam_config%n_sfpar, "dimensions"
       call msg_message (unit = unit)
       if (beam_config%sf_trace) then
          call msg_message ("Beam structure: tracing &
               &values in '" // char (beam_config%sf_trace_file) // "'")
```

```
      end if
    end if
  end subroutine process_beam_config_sf_startup_message
```

Return the PDF set currently in use, if any. This should be unique, so we scan the structure functions until we get a nonzero number.

(This implies that if the PDF set is not unique (e.g., proton and photon structure used together), this doesn't work correctly.)

⟨*Process config: process beam config: TBP*⟩+≡

```
  procedure :: get_pdf_set => process_beam_config_get_pdf_set
```

⟨*Process config: procedures*⟩+≡

```
  function process_beam_config_get_pdf_set (beam_config) result (pdf_set)
    class(process_beam_config_t), intent(in) :: beam_config
    integer :: pdf_set
    integer :: i
    pdf_set = 0
    if (allocated (beam_config%sf)) then
       do i = 1, size (beam_config%sf)
          pdf_set = beam_config%sf(i)%get_pdf_set ()
          if (pdf_set /= 0)  return
       end do
    end if
  end function process_beam_config_get_pdf_set
```

Return the beam file.

⟨*Process config: process beam config: TBP*⟩+≡

```
  procedure :: get_beam_file => process_beam_config_get_beam_file
```

⟨*Process config: procedures*⟩+≡

```
  function process_beam_config_get_beam_file (beam_config) result (file)
    class(process_beam_config_t), intent(in) :: beam_config
    type(string_t) :: file
    integer :: i
    file = ""
    if (allocated (beam_config%sf)) then
       do i = 1, size (beam_config%sf)
          file = beam_config%sf(i)%get_beam_file ()
          if (file /= "")  return
       end do
    end if
  end function process_beam_config_get_beam_file
```

Compute the MD5 sum for the complete beam setup. We rely on the default output of `write` to contain all relevant data.

This is done only once, when the MD5 sum is still empty.

⟨*Process config: process beam config: TBP*⟩+≡

```
  procedure :: compute_md5sum => process_beam_config_compute_md5sum
```

⟨*Process config: procedures*⟩+≡

```
  subroutine process_beam_config_compute_md5sum (beam_config)
    class(process_beam_config_t), intent(inout) :: beam_config
    integer :: u
```

```
    if (beam_config%md5sum == "") then
       u = free_unit ()
       open (u, status = "scratch", action = "readwrite")
       call beam_config%write (u, verbose=.true.)
       rewind (u)
       beam_config%md5sum = md5sum (u)
       close (u)
    end if
  end subroutine process_beam_config_compute_md5sum
```

⟨*Process config: process beam config: TBP*⟩+≡

```
  procedure :: get_md5sum => process_beam_config_get_md5sum
```

⟨*Process config: procedures*⟩+≡

```
  pure function process_beam_config_get_md5sum (beam_config) result (md5)
    character(32) :: md5
    class(process_beam_config_t), intent(in) :: beam_config
    md5 = beam_config%md5sum
  end function process_beam_config_get_md5sum
```

30.6.4 Process components

A process component is an individual contribution to a process (scattering or decay) which needs not be physical. The sum over all components should be physical.

The `index` indentifies this component within its parent process.

The actual process component is stored in the `core` subobject. We use a polymorphic subobject instead of an extension of `process_component_t`, because the individual entries in the array of process components can have different types. In short, `process_component_t` is a wrapper for the actual process variants.

If the `active` flag is false, we should skip this component. This happens if the associated process has vanishing matrix element.

The index array `i_term` points to the individual terms generated by this component. The indices refer to the parent process.

The index `i_mci` is the index of the MC integrator and parameter set which are associated to this process component.

⟨*Process config: public*⟩+≡

```
  public :: process_component_t
```

⟨*Process config: types*⟩+≡

```
  type :: process_component_t
     type(process_component_def_t), pointer :: config => null ()
     integer :: index = 0
     logical :: active = .false.
     class(mci_t), allocatable :: mci_template
     integer, dimension(:), allocatable :: i_term
     integer :: i_mci = 0
     class(phs_config_t), allocatable :: phs_config
     character(32) :: md5sum_phs = ""
     integer :: component_type = COMP_DEFAULT
```

```
 contains
 ⟨Process config: process component: TBP⟩
end type process_component_t
```

Finalizer. The MCI template may (potentially) need a finalizer. The process configuration finalizer may include closing an open scratch file.

⟨*Process config: process component: TBP*⟩≡

```
procedure :: final => process_component_final
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_component_final (object)
  class(process_component_t), intent(inout) :: object
  if (allocated (object%mci_template)) then
     call object%mci_template%final ()
  end if
  if (allocated (object%phs_config)) then
     call object%phs_config%final ()
  end if
end subroutine process_component_final
```

The meaning of `verbose` depends on the process variant.

⟨*Process config: process component: TBP*⟩+≡

```
procedure :: write => process_component_write
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_component_write (object, unit)
  class(process_component_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  if (associated (object%config)) then
     write (u, "(1x,A,I0)")  "Component #", object%index
     call object%config%write (u)
     if (object%md5sum_phs /= "") then
        write (u, "(3x,A,A,A)")  "MD5 sum (phs)       = '", &
             object%md5sum_phs, "'"
     end if
  else
     write (u, "(1x,A)") "Process component: [not allocated]"
  end if
  if (.not. object%active) then
     write (u, "(1x,A)") "[Inactive]"
     return
  end if
  write (u, "(1x,A)") "Referenced data:"
  if (allocated (object%i_term)) then
     write (u, "(3x,A,999(1x,I0))") "Terms                    =", &
          object%i_term
  else
     write (u, "(3x,A)") "Terms                     = [undefined]"
  end if
  if (object%i_mci /= 0) then
     write (u, "(3x,A,I0)") "MC dataset               = ", object%i_mci
  else
```

```
      write (u, "(3x,A)") "MC dataset               = [undefined]"
    end if
    if (allocated (object%phs_config)) then
      call object%phs_config%write (u)
    end if
  end subroutine process_component_write
```

Initialize the component.

⟨*Process config: process component: TBP*⟩+≡

```
  procedure :: init => process_component_init
```

⟨*Process config: procedures*⟩+≡

```
  subroutine process_component_init (component, &
      i_component, meta, config, &
      active, data, &
      mci_template, phs_config_template)
    class(process_component_t), intent(out) :: component
    integer, intent(in) :: i_component
    type(process_metadata_t), intent(in), target :: meta
    type(process_config_data_t), intent(in) :: config
    logical, intent(in) :: active
    type(process_constants_t), intent(in) :: data
    class(mci_t), intent(in), allocatable :: mci_template
    class(phs_config_t), intent(in), allocatable :: phs_config_template

    component%index = i_component
    component%config => meta%lib%get_component_def_ptr (meta%id, i_component)

    component%active = active
    if (component%active) then
      if (allocated (mci_template)) &
        allocate (component%mci_template, source = mci_template)
      allocate (component%phs_config, source = phs_config_template)
      call component%phs_config%init (data, config%model)
      call component%phs_config%set_component_index (component%index)
    end if
  end subroutine process_component_init
```

⟨*Process config: process component: TBP*⟩+≡

```
  procedure :: is_active => process_component_is_active
```

⟨*Process config: procedures*⟩+≡

```
  elemental function process_component_is_active (component) result (active)
    logical :: active
    class(process_component_t), intent(in) :: component
    active = component%active
  end function process_component_is_active
```

⟨*Process config: process component: TBP*⟩+≡

```
  procedure :: has_mci_template => process_component_has_mci_template
```

⟨*Process config: procedures*⟩+≡

```
  pure function process_component_has_mci_template (component) &
      result (is_allocated)
```

```
    logical :: is_allocated
    class(process_component_t), intent(in) :: component
    is_allocated = allocated (component%mci_template)
  end function process_component_has_mci_template
```

⟨*Process config: process component: TBP*⟩+≡

```
  procedure :: extract_mci_template => process_component_extract_mci_template
```

⟨*Process config: procedures*⟩+≡

```
  function process_component_extract_mci_template (component) &
         result (mci_template)
    class(mci_t), allocatable :: mci_template
    class(process_component_t), intent(in) :: component
    if (allocated (component%mci_template)) &
       allocate (mci_template, source = component%mci_template)
  end function process_component_extract_mci_template
```

Finalize the phase-space configuration.

⟨*Process config: process component: TBP*⟩+≡

```
  procedure :: configure_phs => process_component_configure_phs
```

⟨*Process config: procedures*⟩+≡

```
  subroutine process_component_configure_phs &
       (component, sqrts, beam_config, rebuild, &
        ignore_mismatch)
    class(process_component_t), intent(inout) :: component
    real(default), intent(in) :: sqrts
    type(process_beam_config_t), intent(in) :: beam_config
    logical, intent(in), optional :: rebuild
    logical, intent(in), optional :: ignore_mismatch
    logical :: no_strfun
    integer :: nlo_type
    no_strfun = beam_config%n_strfun == 0
    nlo_type = component%config%get_nlo_type ()
    call component%phs_config%configure (sqrts, &
         azimuthal_dependence = beam_config%azimuthal_dependence, &
         sqrts_fixed = no_strfun, &
         cm_frame = beam_config%lab_is_cm_frame .and. no_strfun, &
         rebuild = rebuild, ignore_mismatch = ignore_mismatch, &
         nlo_type = nlo_type)
  end subroutine process_component_configure_phs
```

The process component possesses two MD5 sums: the checksum of the component definition, which should be available when the component is initialized, and the phase-space MD5 sum, which is available after configuration.

⟨*Process config: process component: TBP*⟩+≡

```
  procedure :: compute_md5sum => process_component_compute_md5sum
```

⟨*Process config: procedures*⟩+≡

```
  subroutine process_component_compute_md5sum (component)
    class(process_component_t), intent(inout) :: component
    component%md5sum_phs = component%phs_config%get_md5sum ()
  end subroutine process_component_compute_md5sum
```

Match phase-space channels with structure-function channels, where applicable.

This calls a method of the phs_config phase-space implementation.

⟨*Process config: process component: TBP*⟩+≡
```
procedure :: collect_channels => process_component_collect_channels
```

⟨*Process config: procedures*⟩+≡
```
subroutine process_component_collect_channels (component, coll)
  class(process_component_t), intent(inout) :: component
  type(phs_channel_collection_t), intent(inout) :: coll
  call component%phs_config%collect_channels (coll)
end subroutine process_component_collect_channels
```

⟨*Process config: process component: TBP*⟩+≡
```
procedure :: get_config => process_component_get_config
```

⟨*Process config: procedures*⟩+≡
```
function process_component_get_config (component) &
       result (config)
  type(process_component_def_t) :: config
  class(process_component_t), intent(in) :: component
  config = component%config
end function process_component_get_config
```

⟨*Process config: process component: TBP*⟩+≡
```
procedure :: get_md5sum => process_component_get_md5sum
```

⟨*Process config: procedures*⟩+≡
```
pure function process_component_get_md5sum (component) result (md5)
  type(string_t) :: md5
  class(process_component_t), intent(in) :: component
  md5 = component%config%get_md5sum () // component%md5sum_phs
end function process_component_get_md5sum
```

Return the number of phase-space parameters.

⟨*Process config: process component: TBP*⟩+≡
```
procedure :: get_n_phs_par => process_component_get_n_phs_par
```

⟨*Process config: procedures*⟩+≡
```
function process_component_get_n_phs_par (component) result (n_par)
  class(process_component_t), intent(in) :: component
  integer :: n_par
  n_par = component%phs_config%get_n_par ()
end function process_component_get_n_phs_par
```

⟨*Process config: process component: TBP*⟩+≡
```
procedure :: get_phs_config => process_component_get_phs_config
```

⟨*Process config: procedures*⟩+≡
```
subroutine process_component_get_phs_config (component, phs_config)
  class(process_component_t), intent(in), target :: component
  class(phs_config_t), intent(out), pointer :: phs_config
  phs_config => component%phs_config
end subroutine process_component_get_phs_config
```

⟨*Process config: process component: TBP*⟩+≡

```
procedure :: get_nlo_type => process_component_get_nlo_type
```

⟨*Process config: procedures*⟩+≡

```
elemental function process_component_get_nlo_type (component) result (nlo_type)
   integer :: nlo_type
   class(process_component_t), intent(in) :: component
   nlo_type = component%config%get_nlo_type ()
end function process_component_get_nlo_type
```

⟨*Process config: process component: TBP*⟩+≡

```
procedure :: needs_mci_entry => process_component_needs_mci_entry
```

⟨*Process config: procedures*⟩+≡

```
function process_component_needs_mci_entry (component, combined_integration) result (value)
  logical :: value
  class(process_component_t), intent(in) :: component
  logical, intent(in), optional :: combined_integration
  value = component%active
  if (present (combined_integration)) then
     if (combined_integration) &
          value = value .and. component%component_type <= COMP_MASTER
  end if
end function process_component_needs_mci_entry
```

⟨*Process config: process component: TBP*⟩+≡

```
procedure :: can_be_integrated => process_component_can_be_integrated
```

⟨*Process config: procedures*⟩+≡

```
elemental function process_component_can_be_integrated (component) result (active)
  logical :: active
  class(process_component_t), intent(in) :: component
  active = component%config%can_be_integrated ()
end function process_component_can_be_integrated
```

30.6.5 Process terms

For straightforward tree-level calculations, each process component corresponds to a unique elementary interaction. However, in the case of NLO calculations with subtraction terms, a process component may split into several separate contributions to the scattering, which are qualified by interactions with distinct kinematics and particle content. We represent their configuration as `process_term_t` objects, the actual instances will be introduced below as `term_instance_t`. In any case, the process term contains an elementary interaction with a definite quantum-number and momentum content.

The index `i_term_global` identifies the term relative to the process.

The index `i_component` identifies the process component which generates this term, relative to the parent process.

The index `i_term` identifies the term relative to the process component (not the process).

The `data` subobject holds all process constants.

The number of allowed flavor/helicity/color combinations is stored as `n_allowed`. This is the total number of independent entries in the density matrix. For each combination, the index of the flavor, helicity, and color state is stored in the arrays `flv`, `hel`, and `col`, respectively.

The flag `rearrange` is true if we need to rearrange the particles of the hard interaction, to obtain the effective parton state.

The interaction `int` holds the quantum state for the (resolved) hard interaction, the parent-child relations of the particles, and their momenta. The momenta are not filled yet; this is postponed to copies of `int` which go into the process instances.

If recombination is in effect, we should allocate `int_eff` to describe the rearranged partonic state.

This type is public only for use in a unit test.

⟨*Process config: public*⟩+≡
```
public :: process_term_t
```

⟨*Process config: types*⟩+≡
```
type :: process_term_t
   integer :: i_term_global = 0
   integer :: i_component = 0
   integer :: i_term = 0
   integer :: i_sub = 0
   integer :: i_core = 0
   integer :: n_allowed = 0
   type(process_constants_t) :: data
   real(default) :: alpha_s = 0
   integer, dimension(:), allocatable :: flv, hel, col
   type(interaction_t) :: int
   type(interaction_t), pointer :: int_eff => null ()
 contains
 ⟨Process config: process term: TBP⟩
end type process_term_t
```

For the output, we skip the process constants and the tables of allowed quantum numbers. Those can also be read off from the interaction object.

⟨*Process config: process term: TBP*⟩≡
```
procedure :: write => process_term_write
```

⟨*Process config: procedures*⟩+≡
```
subroutine process_term_write (term, unit)
  class(process_term_t), intent(in) :: term
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A,I0)")  "Term #", term%i_term_global
  write (u, "(3x,A,I0)")  "Process component index      = ", &
       term%i_component
  write (u, "(3x,A,I0)")  "Term index w.r.t. component  = ", &
       term%i_term
  call write_separator (u)
  write (u, "(1x,A)")  "Hard interaction:"
  call write_separator (u)
  call term%int%basic_write (u)
```

```
end subroutine process_term_write
```

Write an account of all quantum number states and their current status.

⟨*Process config: process term: TBP*⟩+≡

```
procedure :: write_state_summary => process_term_write_state_summary
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_term_write_state_summary (term, core, unit)
  class(process_term_t), intent(in) :: term
  class(prc_core_t), intent(in) :: core
  integer, intent(in), optional :: unit
  integer :: u, i, f, h, c
  type(state_iterator_t) :: it
  character :: sgn
  u = given_output_unit (unit)
  write (u, "(1x,A,I0)")  "Term #", term%i_term_global
  call it%init (term%int%get_state_matrix_ptr ())
  do while (it%is_valid ())
     i = it%get_me_index ()
     f = term%flv(i)
     h = term%hel(i)
     if (allocated (term%col)) then
        c = term%col(i)
     else
        c = 1
     end if
     if (core%is_allowed (term%i_term, f, h, c)) then
        sgn = "+"
     else
        sgn = " "
     end if
     write (u, "(1x,A1,1x,I0,2x)", advance="no")  sgn, i
     call quantum_numbers_write (it%get_quantum_numbers (), u)
     write (u, *)
     call it%advance ()
  end do
end subroutine process_term_write_state_summary
```

Finalizer: the `int` and potentially `int_eff` components have a finalizer that we must call.

⟨*Process config: process term: TBP*⟩+≡

```
procedure :: final => process_term_final
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_term_final (term)
  class(process_term_t), intent(inout) :: term
  call term%int%final ()
end subroutine process_term_final
```

Initialize the term. We copy the process constants from the `core` object and set up the `int` hard interaction accordingly.

The `alpha_s` value is useful for writing external event records. This is the constant value which may be overridden by a event-specific running value. If the model does not contain the strong coupling, the value is zero.

The `rearrange` part is commented out; this or something equivalent could become relevant for NLO algorithms.

⟨*Process config: process term: TBP*⟩+≡

```
procedure :: init => process_term_init
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_term_init &
     (term, i_term_global, i_component, i_term, core, model, &
      nlo_type, use_beam_pol, subtraction_method)
  class(process_term_t), intent(inout), target :: term
  integer, intent(in) :: i_term_global
  integer, intent(in) :: i_component
  integer, intent(in) :: i_term
  class(prc_core_t), intent(inout) :: core
  class(model_data_t), intent(in), target :: model
  integer, intent(in), optional :: nlo_type
  logical, intent(in), optional :: use_beam_pol
  type(string_t), intent(in), optional :: subtraction_method
  class(modelpar_data_t), pointer :: alpha_s_ptr
  logical :: use_internal_color
  term%i_term_global = i_term_global
  term%i_component = i_component
  term%i_term = i_term
  call core%get_constants (term%data, i_term)
  alpha_s_ptr => model%get_par_data_ptr (var_str ("alphas"))
  if (associated (alpha_s_ptr)) then
     term%alpha_s = alpha_s_ptr%get_real ()
  else
     term%alpha_s = -1
  end if
  use_internal_color = .false.
  if (present (subtraction_method)) &
       use_internal_color = (char (subtraction_method) == 'omega') &
       .or. (char (subtraction_method) == 'threshold')
  call term%setup_interaction (core, model, nlo_type = nlo_type, &
       pol_beams = use_beam_pol, use_internal_color = use_internal_color)
end subroutine process_term_init
```

We fetch the process constants which determine the quantum numbers and use those to create the interaction. The interaction contains incoming and outgoing particles, no virtuals. The incoming particles are parents of the outgoing ones.

Keeping previous `WHIZARD` conventions, we invert the color assignment (but not flavor or helicity) for the incoming particles. When the color-flow square matrix is evaluated, this inversion is done again, so in the color-flow sequence we get the color assignments of the matrix element.

⟨*Process config: process term: TBP*⟩+≡

```
procedure :: setup_interaction => process_term_setup_interaction
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_term_setup_interaction (term, core, model, &
   nlo_type, pol_beams, use_internal_color)
  class(process_term_t), intent(inout) :: term
  class(prc_core_t), intent(inout) :: core
  class(model_data_t), intent(in), target :: model
```

```
  logical, intent(in), optional :: pol_beams
  integer, intent(in), optional :: nlo_type
  logical, intent(in), optional :: use_internal_color
  integer :: n, n_tot
  type(flavor_t), dimension(:), allocatable :: flv
  type(color_t), dimension(:), allocatable :: col
  type(helicity_t), dimension(:), allocatable :: hel
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  logical :: is_pol, use_color
  integer :: nlo_t, n_sub
  is_pol = .false.; if (present (pol_beams)) is_pol = pol_beams
  nlo_t = BORN; if (present (nlo_type)) nlo_t = nlo_type
  n_tot = term%data%n_in + term%data%n_out
  call count_number_of_states ()
  term%n_allowed = n
  call compute_n_sub ()
  call fill_quantum_numbers ()
  call term%int%basic_init &
       (term%data%n_in, 0, term%data%n_out, set_relations = .true.)
  select type (core)
  type is (prc_openloops_t)
     call setup_states_openloops ()
  type is (prc_threshold_t)
     call setup_states_threshold ()
  class is (prc_user_defined_base_t)
     call setup_states_other_user_defined ()
  class default
     call setup_states_omega ()
  end select
  call term%int%freeze ()
contains
  subroutine count_number_of_states ()
    integer :: f, h, c
    n = 0
    select type (core)
    class is (prc_user_defined_base_t)
       do f = 1, term%data%n_flv
          do h = 1, term%data%n_hel
             do c = 1, term%data%n_col
                n = n + 1
             end do
          end do
       end do
    class default !!! Omega and all test cores
       do f = 1, term%data%n_flv
          do h = 1, term%data%n_hel
             do c = 1, term%data%n_col
                if (core%is_allowed (term%i_term, f, h, c))  n = n + 1
             end do
          end do
       end do
    end select
  end subroutine count_number_of_states
```

```
subroutine compute_n_sub ()
  if (nlo_t == NLO_VIRTUAL) then
     use_color = .false.; if (present (use_internal_color)) &
          use_color = use_internal_color
     n_sub = 1
     if (.not. use_color) n_sub = n_sub + n_tot * (n_tot - 1) / 2
  else
     n_sub = 0
  end if
end subroutine compute_n_sub

subroutine fill_quantum_numbers ()
  if (nlo_t == NLO_VIRTUAL) then
     allocate (term%flv ((n_sub + 1) * n), &
          term%col ((n_sub + 1) * n), term%hel ((n_sub + 1) * n))
  else
     allocate (term%flv (n), term%col (n), term%hel (n))
  end if
  allocate (flv (n_tot), col (n_tot), hel (n_tot))
  allocate (qn (n_tot))
end subroutine fill_quantum_numbers

subroutine setup_states_openloops ()
  integer :: s, f, c, h, i
  i = 0
  associate (data => term%data)
     do s = 0, n_sub
         do f = 1, data%n_flv
            do h = 1, data%n_hel
               do c = 1, data%n_col
                  i = i + 1
                  term%flv(i) = f
                  term%hel(i) = h
                  !!! Dummy-initialization of color
                  term%col(i) = c
                  call flv%init (data%flv_state (:,f), model)
                  call color_init_from_array (col, &
                       data%col_state(:,:,c), data%ghost_flag(:,c))
                  call col(:data%n_in)%invert ()
                  if (is_pol) then
                     call hel%init (data%hel_state (:,h))
                     call qn%init (flv, hel, col, s)
                  else
                     call qn%init (flv, col, s)
                  end if
                  call term%int%add_state (qn)
               end do
            end do
         end do
     end do
  end associate
end subroutine setup_states_openloops

subroutine setup_states_threshold ()
```

```
  integer :: s, f, c, h, i
  if (is_pol) &
       call msg_fatal ("Polarized beams only supported by OpenLoops")
  i = 0
  n_sub = 0; if (nlo_t == NLO_VIRTUAL) n_sub = 1
  associate (data => term%data)
     do s = 0, n_sub
        do f = 1, term%data%n_flv
           do h = 1, data%n_hel
              do c = 1, data%n_col
                 i = i + 1
                 term%flv(i) = f
                 !!! Dumy initialization of helicity
                 term%hel(i) = 1
                 !!! Dummy-initialization of color
                 term%col(i) = 1
                 call flv%init (term%data%flv_state (:,f), model)
                 call qn%init (flv, s)
                 call term%int%add_state (qn)
              end do
           end do
        end do
     end do
  end associate
end subroutine setup_states_threshold

subroutine setup_states_other_user_defined ()
  integer :: s, f, i
  integer :: n_sub
  if (is_pol) &
     call msg_fatal ("Polarized beams only supported by OpenLoops")
  i = 0
  n_sub = 0; if (nlo_t == NLO_VIRTUAL) n_sub = 1
  do s = 0, n_sub
     do f = 1, term%data%n_flv
        i = i + 1
        term%flv(i) = f
        term%hel(i) = 1
        term%col(i) = 1
        call flv%init (term%data%flv_state (:,f), model)
        call qn%init (flv, s)
        call term%int%add_state (qn)
     end do
  end do
end subroutine setup_states_other_user_defined

subroutine setup_states_omega ()
  integer :: f, h, c, i
  i = 0
  associate (data => term%data)
     do f = 1, data%n_flv
        do h = 1, data%n_hel
          do c = 1, data%n_col
             if (core%is_allowed (term%i_term, f, h, c)) then
```

```
                    i = i + 1
                    term%flv(i) = f
                    term%hel(i) = h
                    term%col(i) = c
                    call flv%init (data%flv_state(:,f), model)
                    call color_init_from_array (col, &
                         data%col_state(:,:,c), &
                         data%ghost_flag(:,c))
                    call col(:data%n_in)%invert ()
                    call hel%init (data%hel_state(:,h))
                    call qn%init (flv, col, hel)
                    call term%int%add_state (qn)
                 end if
              end do
           end do
        end do
     end associate
   end subroutine setup_states_omega

end subroutine process_term_setup_interaction
```

⟨*Process config: process term: TBP*⟩+≡

```
procedure :: get_process_constants => process_term_get_process_constants
```

⟨*Process config: procedures*⟩+≡

```
subroutine process_term_get_process_constants &
    (term, prc_constants)
  class(process_term_t), intent(inout) :: term
  type(process_constants_t), intent(out) :: prc_constants
  prc_constants = term%data
end subroutine process_term_get_process_constants
```

30.7 Process call statistics

Very simple object for statistics. Could be moved to a more basic chapter.

⟨`process_counter.f90`⟩≡

```
⟨File header⟩

module process_counter

⟨Use kinds⟩
  use io_units

⟨Standard module head⟩

⟨Process counter: public⟩

⟨Process counter: parameters⟩

⟨Process counter: types⟩
```

```
contains

⟨Process counter: procedures⟩

end module process_counter
```

This object can record process calls, categorized by evaluation status. It is a part of the `mci_entry` component below.

⟨*Process counter: public*⟩≡
```
public :: process_counter_t
```

⟨*Process counter: types*⟩≡
```
type :: process_counter_t
   integer :: total = 0
   integer :: failed_kinematics = 0
   integer :: failed_cuts = 0
   integer :: has_passed = 0
   integer :: evaluated = 0
   integer :: complete = 0
 contains
 ⟨Process counter: process counter: TBP⟩
end type process_counter_t
```

Here are the corresponding numeric codes:

⟨*Process counter: parameters*⟩≡
```
integer, parameter, public :: STAT_UNDEFINED = 0
integer, parameter, public :: STAT_INITIAL = 1
integer, parameter, public :: STAT_ACTIVATED = 2
integer, parameter, public :: STAT_BEAM_MOMENTA = 3
integer, parameter, public :: STAT_FAILED_KINEMATICS = 4
integer, parameter, public :: STAT_SEED_KINEMATICS = 5
integer, parameter, public :: STAT_HARD_KINEMATICS = 6
integer, parameter, public :: STAT_EFF_KINEMATICS = 7
integer, parameter, public :: STAT_FAILED_CUTS = 8
integer, parameter, public :: STAT_PASSED_CUTS = 9
integer, parameter, public :: STAT_EVALUATED_TRACE = 10
integer, parameter, public :: STAT_EVENT_COMPLETE = 11
```

Output.

⟨*Process counter: process counter: TBP*⟩≡
```
procedure :: write => process_counter_write
```

⟨*Process counter: procedures*⟩≡
```
subroutine process_counter_write (object, unit)
  class(process_counter_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  if (object%total > 0) then
     write (u, "(1x,A)")  "Call statistics (current run):"
     write (u, "(3x,A,I0)")  "total       = ", object%total
     write (u, "(3x,A,I0)")  "failed kin. = ", object%failed_kinematics
     write (u, "(3x,A,I0)")  "failed cuts = ", object%failed_cuts
     write (u, "(3x,A,I0)")  "passed cuts = ", object%has_passed
```

```
      write (u, "(3x,A,I0)")  "evaluated   = ", object%evaluated
    else
      write (u, "(1x,A)")  "Call statistics (current run): [no calls]"
    end if
  end subroutine process_counter_write
```

Reset. Just enforce default initialization.

⟨*Process counter: process counter: TBP*⟩+≡

```
  procedure :: reset => process_counter_reset
```

⟨*Process counter: procedures*⟩+≡

```
  subroutine process_counter_reset (counter)
    class(process_counter_t), intent(out) :: counter
  end subroutine process_counter_reset
```

We record an event according to the lowest status code greater or equal to the actual status. This is actually done by the process instance; the process object just copies the instance counter.

⟨*Process counter: process counter: TBP*⟩+≡

```
  procedure :: record => process_counter_record
```

⟨*Process counter: procedures*⟩+≡

```
  subroutine process_counter_record (counter, status)
    class(process_counter_t), intent(inout) :: counter
    integer, intent(in) :: status
    if (status <= STAT_FAILED_KINEMATICS) then
      counter%failed_kinematics = counter%failed_kinematics + 1
    else if (status <= STAT_FAILED_CUTS) then
      counter%failed_cuts = counter%failed_cuts + 1
    else if (status <= STAT_PASSED_CUTS) then
      counter%has_passed = counter%has_passed + 1
    else
      counter%evaluated = counter%evaluated + 1
    end if
    counter%total = counter%total + 1
  end subroutine process_counter_record
```

30.8 Multi-channel integration

⟨process_mci.f90⟩≡

```
⟨File header⟩

module process_mci

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use diagnostics
  use physics_defs
  use md5
  use cputime
```

```
  use rng_base
  use mci_base
  use variables
  use integration_results
  use process_libraries
  use phs_base
  use process_counter
  use process_config
```

⟨*Standard module head*⟩

⟨*Process mci: public*⟩

⟨*Process mci: parameters*⟩

⟨*Process mci: types*⟩

```
contains
```

⟨*Process mci: procedures*⟩

```
end module process_mci
```

30.8.1 Process MCI entry

The `process_mci_entry_t` block contains, for each process component that is integrated independently, the configuration data for its MC input parameters. Each input parameter set is handled by a `mci_t` integrator.

The MC input parameter set is broken down into the parameters required by the structure-function chain and the parameters required by the phase space of the elementary process.

The MD5 sum collects all information about the associated processes that may affect the integration. It does not contain the MCI object itself or integration results.

MC integration is organized in passes. Each pass may consist of several iterations, and for each iteration there is a number of calls. We store explicitly the values that apply to the current pass. Previous values are archived in the `results` object.

The `counter` receives the counter statistics from the associated process instance, for diagnostics.

The `results` object records results, broken down in passes and iterations.

⟨*Process mci: public*⟩≡

```
  public :: process_mci_entry_t
```

⟨*Process mci: types*⟩≡

```
  type :: process_mci_entry_t
     integer :: i_mci = 0
     integer, dimension(:), allocatable :: i_component
     integer :: process_type = PRC_UNKNOWN
     integer :: n_par = 0
     integer :: n_par_sf = 0
```

```
    integer :: n_par_phs = 0
    character(32) :: md5sum = ""
    integer :: pass = 0
    integer :: n_it = 0
    integer :: n_calls = 0
    logical :: activate_timer = .false.
    real(default) :: error_threshold = 0
    class(mci_t), allocatable :: mci
    type(process_counter_t) :: counter
    type(integration_results_t) :: results
    logical :: negative_weights
    logical :: combined_integration = .false.
    integer :: real_partition_type = REAL_FULL
    integer :: associated_real_component = 0
  contains
  ⟨Process mci: process mci entry: TBP⟩
end type process_mci_entry_t
```

Finalizer for the `mci` component.

⟨*Process mci: process mci entry: TBP*⟩≡

```
procedure :: final => process_mci_entry_final
```

⟨*Process mci: procedures*⟩≡

```
subroutine process_mci_entry_final (object)
  class(process_mci_entry_t), intent(inout) :: object
  if (allocated (object%mci))  call object%mci%final ()
end subroutine process_mci_entry_final
```

Output. Write pass/iteration information only if set (the pass index is nonzero). Write the MCI block only if it exists (for some self-tests it does not). Write results only if there are any.

⟨*Process mci: process mci entry: TBP*⟩+≡

```
procedure :: write => process_mci_entry_write
```

⟨*Process mci: procedures*⟩+≡

```
subroutine process_mci_entry_write (object, unit, pacify)
  class(process_mci_entry_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacify
  integer :: u
  u = given_output_unit (unit)
  write (u, "(3x,A,I0)")  "Associated components = ", object%i_component
  write (u, "(3x,A,I0)")  "MC input parameters   = ", object%n_par
  write (u, "(3x,A,I0)")  "MC parameters (SF)    = ", object%n_par_sf
  write (u, "(3x,A,I0)")  "MC parameters (PHS)   = ", object%n_par_phs
  if (object%pass > 0) then
     write (u, "(3x,A,I0)")  "Current pass          = ", object%pass
     write (u, "(3x,A,I0)")  "Number of iterations  = ", object%n_it
     write (u, "(3x,A,I0)")  "Number of calls       = ", object%n_calls
  end if
  if (object%md5sum /= "") then
     write (u, "(3x,A,A,A)") "MD5 sum (components)  = '", object%md5sum, "'"
  end if
  if (allocated (object%mci)) then
```

```
      call object%mci%write (u)
    end if
    call object%counter%write (u)
    if (object%results%exist ()) then
      call object%results%write (u, suppress = pacify)
      call object%results%write_chain_weights (u)
    end if
  end subroutine process_mci_entry_write
```

Initialize. From the existing configuration, we obtain the number of channels and the number of parameters, separately for the structure-function chain and for the associated process component. We assume that the phase-space object has already been configured.

The allocation of the MCI integrator with the appropriate concrete type is the duty of the process core.

We assume that there is only one component associated with a MCI entry. This restriction should be relaxed.

⟨*Process mci: process mci entry: TBP*⟩+≡

```
  procedure :: init => process_mci_entry_init
```

⟨*Process mci: procedures*⟩+≡

```
  subroutine process_mci_entry_init (mci_entry, &
      process_type, i_mci, i_component, component, &
      n_sfpar, rng_factory)
    class(process_mci_entry_t), intent(inout) :: mci_entry
    integer, intent(in) :: process_type
    integer, intent(in) :: i_mci
    integer, intent(in) :: i_component
    type(process_component_t), intent(in), target :: component
    integer, intent(in) :: n_sfpar
    class(rng_factory_t), intent(inout) :: rng_factory
    class(rng_t), allocatable :: rng
    associate (phs_config => component%phs_config)
      mci_entry%i_mci = i_mci
      call mci_entry%create_component_list (i_component, component%get_config ())
      mci_entry%n_par_sf = n_sfpar
      mci_entry%n_par_phs = phs_config%get_n_par ()
      mci_entry%n_par = mci_entry%n_par_sf + mci_entry%n_par_phs
      mci_entry%process_type = process_type
      if (component%has_mci_template ()) then
         allocate (mci_entry%mci, source=component%extract_mci_template ())
         call mci_entry%mci%record_index (mci_entry%i_mci)
         call mci_entry%mci%set_dimensions &
              (mci_entry%n_par, phs_config%get_n_channel ())
         call mci_entry%mci%declare_flat_dimensions &
              (phs_config%get_flat_dimensions ())
         if (phs_config%provides_equivalences) then
            call mci_entry%mci%declare_equivalences &
                 (phs_config%channel, mci_entry%n_par_sf)
         end if
         if (phs_config%provides_chains) then
            call mci_entry%mci%declare_chains (phs_config%chain)
         end if
```

```
        call rng_factory%make (rng)
        call mci_entry%mci%import_rng (rng)
      end if
      call mci_entry%results%init (process_type)
    end associate
  end subroutine process_mci_entry_init
```

⟨*Process mci: parameters*⟩≡

```
  integer, parameter, public :: REAL_FULL = 0
  integer, parameter, public :: REAL_SINGULAR = 1
  integer, parameter, public :: REAL_FINITE = 2
```

⟨*Process mci: process mci entry: TBP*⟩+≡

```
  procedure :: create_component_list => &
     process_mci_entry_create_component_list
```

⟨*Process mci: procedures*⟩+≡

```
  subroutine process_mci_entry_create_component_list (mci_entry, &
     i_component, component_config)
    class (process_mci_entry_t), intent(inout) :: mci_entry
    integer, intent(in) :: i_component
    type(process_component_def_t), intent(in) :: component_config
    integer, dimension(:), allocatable :: i_list
    integer :: n
    integer, save :: i_rfin_offset = 0
    call msg_debug (D_PROCESS_INTEGRATION, "process_mci_entry_create_component_list")
    if (mci_entry%combined_integration) then
       n = get_n_components (mci_entry%real_partition_type)
       allocate (i_list (n))
       call msg_debug (D_PROCESS_INTEGRATION, &
            "mci_entry%real_partition_type", mci_entry%real_partition_type)
       select case (mci_entry%real_partition_type)
       case (REAL_FULL)
          i_list = component_config%get_association_list ()
          allocate (mci_entry%i_component (size (i_list)))
          mci_entry%i_component = i_list
       case (REAL_SINGULAR)
          i_list = component_config%get_association_list (ASSOCIATED_REAL_FIN)
          allocate (mci_entry%i_component (size(i_list)))
          mci_entry%i_component = i_list
       case (REAL_FINITE)
          allocate (mci_entry%i_component (1))
          mci_entry%i_component(1) = &
               component_config%get_associated_real_fin () + i_rfin_offset
          i_rfin_offset = i_rfin_offset + 1
       end select
    else
       allocate (mci_entry%i_component (1))
       mci_entry%i_component(1) = i_component
    end if
  contains
    function get_n_components (damping_type) result (n_components)
      integer :: n_components
      integer, intent(in) :: damping_type
      select case (damping_type)
```

```
      case (REAL_FULL)
         n_components = size (component_config%get_association_list ())
      case (REAL_SINGULAR)
         n_components = size (component_config%get_association_list &
            (ASSOCIATED_REAL_FIN))
      end select
      call msg_debug (D_PROCESS_INTEGRATION, "n_components", n_components)
    end function get_n_components
  end subroutine process_mci_entry_create_component_list
```

⟨*Process mci: process mci entry: TBP*⟩+≡

```
  procedure :: deactivate_real_component => &
     process_mci_entry_deactivate_real_component
```

⟨*Process mci: procedures*⟩+≡

```
  subroutine process_mci_entry_deactivate_real_component (mci_entry, config)
    class(process_mci_entry_t), intent(inout) :: mci_entry
    type(process_component_def_t), intent(in) :: config
    integer, dimension(:), allocatable :: i_list
    integer :: n_component_old, i, j
    if (allocated (mci_entry%i_component)) then
       n_component_old = size (mci_entry%i_component)
       deallocate (mci_entry%i_component)
       allocate (mci_entry%i_component (n_component_old - 1))
       allocate (i_list (size (config%get_association_list ())))
       i_list = config%get_association_list ()
       j = 1
       do i = 1, n_component_old
          if (i_list(i) /= 2) then
             mci_entry%i_component(j) = i_list(i)
             j = j + 1
          end if
       end do
    else
       call msg_fatal ("Trying to reset deallocated component list")
    end if
  end subroutine process_mci_entry_deactivate_real_component
```

⟨*Process mci: process mci entry: TBP*⟩+≡

```
  procedure :: set_combined_integration => &
       process_mci_entry_set_combined_integration
```

⟨*Process mci: procedures*⟩+≡

```
  subroutine process_mci_entry_set_combined_integration (mci_entry, value)
    class(process_mci_entry_t), intent(inout) :: mci_entry
    logical, intent(in), optional :: value
    if (present (value)) &
       mci_entry%combined_integration = value
  end subroutine process_mci_entry_set_combined_integration
```

⟨*Process mci: process mci entry: TBP*⟩+≡

```
  procedure :: set_associated_real_component &
      => process_mci_entry_set_associated_real_component
```

⟨*Process mci: procedures*⟩+≡

```
subroutine process_mci_entry_set_associated_real_component (mci_entry, i)
   class(process_mci_entry_t), intent(inout) :: mci_entry
   integer, intent(in) :: i
   mci_entry%associated_real_component = i
end subroutine process_mci_entry_set_associated_real_component
```

Set some additional parameters.

⟨*Process mci: process mci entry: TBP*⟩+≡

```
procedure :: set_parameters => process_mci_entry_set_parameters
```

⟨*Process mci: procedures*⟩+≡

```
subroutine process_mci_entry_set_parameters (mci_entry, var_list)
  class(process_mci_entry_t), intent(inout) :: mci_entry
  type(var_list_t), intent(in) :: var_list
  real(default) :: error_threshold
  error_threshold = &
       var_list%get_rval (var_str ("error_threshold"))
  mci_entry%activate_timer = &
       var_list%get_lval (var_str ("?integration_timer"))
  call mci_entry%results%set_error_threshold (error_threshold)
end subroutine process_mci_entry_set_parameters
```

Compute a MD5 sum that summarizes all information that could influence integration results, for the associated process components. We take the process-configuration MD5 sum which represents parameters, cuts, etc., the MD5 sums for the process component definitions and their phase space objects (which should be configured), and the beam configuration MD5 sum. (The QCD setup is included in the process configuration data MD5 sum.)

Done only once, when the MD5 sum is still empty.

⟨*Process mci: process mci entry: TBP*⟩+≡

```
procedure :: compute_md5sum => process_mci_entry_compute_md5sum
```

⟨*Process mci: procedures*⟩+≡

```
subroutine process_mci_entry_compute_md5sum (mci_entry, &
     config, component, beam_config)
  class(process_mci_entry_t), intent(inout) :: mci_entry
  type(process_config_data_t), intent(in) :: config
  type(process_component_t), dimension(:), intent(in) :: component
  type(process_beam_config_t), intent(in) :: beam_config
  type(string_t) :: buffer
  integer :: i
  if (mci_entry%md5sum == "") then
     buffer = config%get_md5sum () // beam_config%get_md5sum ()
     do i = 1, size (component)
        if (component(i)%is_active ()) then
           buffer = buffer // component(i)%get_md5sum ()
        end if
     end do
     mci_entry%md5sum = md5sum (char (buffer))
  end if
  if (allocated (mci_entry%mci)) then
     call mci_entry%mci%set_md5sum (mci_entry%md5sum)
```

```
    end if
  end subroutine process_mci_entry_compute_md5sum
```

Test the MCI sampler by calling it a given number of time, discarding the results. The instance should be initialized.

The `mci_entry` is `intent(inout)` because the integrator contains the random-number state.

⟨*Process mci: process mci entry: TBP*⟩+≡

```
  procedure :: sampler_test => process_mci_entry_sampler_test
```

⟨*Process mci: procedures*⟩+≡

```
  subroutine process_mci_entry_sampler_test (mci_entry, mci_sampler, n_calls)
    class(process_mci_entry_t), intent(inout) :: mci_entry
    class(mci_sampler_t), intent(inout), target :: mci_sampler
    integer, intent(in) :: n_calls
    call mci_entry%mci%sampler_test (mci_sampler, n_calls)
  end subroutine process_mci_entry_sampler_test
```

Integrate.

The `integrate` method counts as an integration pass; the pass count is increased by one. We transfer the pass parameters (number of iterations and number of calls) to the actual integration routine.

The `mci_entry` is `intent(inout)` because the integrator contains the random-number state.

Note: The results are written to screen and to logfile. This behavior is hardcoded.

⟨*Process mci: process mci entry: TBP*⟩+≡

```
  procedure :: integrate => process_mci_entry_integrate
  procedure :: final_integration => process_mci_entry_final_integration
```

⟨*Process mci: procedures*⟩+≡

```
  subroutine process_mci_entry_integrate (mci_entry, mci_instance, &
         mci_sampler, n_it, n_calls, &
       adapt_grids, adapt_weights, final, pacify, &
       nlo_type)
    class(process_mci_entry_t), intent(inout) :: mci_entry
    class(mci_instance_t), intent(inout) :: mci_instance
    class(mci_sampler_t), intent(inout) :: mci_sampler
    integer, intent(in) :: n_it
    integer, intent(in) :: n_calls
    logical, intent(in), optional :: adapt_grids
    logical, intent(in), optional :: adapt_weights
    logical, intent(in), optional :: final, pacify
    integer, intent(in), optional :: nlo_type
    integer :: u_log
    u_log = logfile_unit ()
    mci_entry%pass = mci_entry%pass + 1
    mci_entry%n_it = n_it
    mci_entry%n_calls = n_calls
    if (mci_entry%pass == 1)  &
         call mci_entry%mci%startup_message (n_calls = n_calls)
    call mci_entry%mci%set_timer (active = mci_entry%activate_timer)
    call mci_entry%results%display_init &
```

```
        (mci_entry%process_type, screen = .true., unit = u_log)
    call mci_entry%results%new_pass ()
    if (present (nlo_type)) then
       select case (nlo_type)
       case (NLO_VIRTUAL, NLO_REAL, NLO_MISMATCH, NLO_DGLAP)
          mci_instance%negative_weights = .true.
       end select
    end if
    call mci_entry%mci%add_pass (adapt_grids, adapt_weights, final)
    call mci_entry%mci%start_timer ()
    call mci_entry%mci%integrate (mci_instance, mci_sampler, n_it, &
         n_calls, mci_entry%results, pacify = pacify)
    call mci_entry%mci%stop_timer ()
    if (signal_is_pending ())  return
  end subroutine process_mci_entry_integrate

  subroutine process_mci_entry_final_integration (mci_entry)
    class(process_mci_entry_t), intent(inout) :: mci_entry
    call mci_entry%results%display_final ()
    call mci_entry%time_message ()
  end subroutine process_mci_entry_final_integration
```

If appropriate, issue an informative message about the expected time for an event sample.

⟨*Process mci: process mci entry: TBP*⟩+≡

```
  procedure :: get_time => process_mci_entry_get_time
  procedure :: time_message => process_mci_entry_time_message
```

⟨*Process mci: procedures*⟩+≡

```
  subroutine process_mci_entry_get_time (mci_entry, time, sample)
    class(process_mci_entry_t), intent(in) :: mci_entry
    type(time_t), intent(out) :: time
    integer, intent(in) :: sample
    real(default) :: time_last_pass, efficiency, calls
    time_last_pass = mci_entry%mci%get_time ()
    calls = mci_entry%results%get_n_calls ()
    efficiency = mci_entry%mci%get_efficiency ()
    if (time_last_pass > 0 .and. calls > 0 .and. efficiency > 0) then
       time = nint (time_last_pass / calls / efficiency * sample)
    end if
  end subroutine process_mci_entry_get_time

  subroutine process_mci_entry_time_message (mci_entry)
    class(process_mci_entry_t), intent(in) :: mci_entry
    type(time_t) :: time
    integer :: sample
    sample = 10000
    call mci_entry%get_time (time, sample)
    if (time%is_known ()) then
       call msg_message ("Time estimate for generating 10000 events: " &
            // char (time%to_string_dhms ()))
    end if
  end subroutine process_mci_entry_time_message
```

Prepare event generation. (For the test integrator, this does nothing. It is relevant for the VAMP integrator.)

⟨*Process mci: process mci entry: TBP*⟩+≡

```
procedure :: prepare_simulation => process_mci_entry_prepare_simulation
```

⟨*Process mci: procedures*⟩+≡

```
subroutine process_mci_entry_prepare_simulation (mci_entry)
  class(process_mci_entry_t), intent(inout) :: mci_entry
  call mci_entry%mci%prepare_simulation ()
end subroutine process_mci_entry_prepare_simulation
```

Generate an event. The instance should be initialized, otherwise event generation is directed by the `mci` integrator subobject. The integrator instance is contained in a `mci_work` subobject of the process instance, which simultaneously serves as the sampler object. (We avoid the anti-aliasing rules if we assume that the sampling itself does not involve the integrator instance contained in the process instance.)

Regarding weighted events, we only take events which are valid, which means that they have valid kinematics and have passed cuts. Therefore, we have a rejection loop. For unweighted events, the unweighting routine should already take care of this.

⟨*Process mci: process mci entry: TBP*⟩+≡

```
procedure :: generate_weighted_event => &
     process_mci_entry_generate_weighted_event
procedure :: generate_unweighted_event => &
     process_mci_entry_generate_unweighted_event
```

⟨*Process mci: procedures*⟩+≡

```
subroutine process_mci_entry_generate_weighted_event (mci_entry, &
    mci_instance, mci_sampler, keep_failed)
  class(process_mci_entry_t), intent(inout) :: mci_entry
  class(mci_instance_t), intent(inout) :: mci_instance
  class(mci_sampler_t), intent(inout) :: mci_sampler
  logical, intent(in) :: keep_failed
  logical :: generate_new
  generate_new = .true.
  REJECTION: do while (generate_new)
     call mci_entry%mci%generate_weighted_event (mci_instance, mci_sampler)
     if (signal_is_pending ())  return
     if (.not. mci_sampler%is_valid()) then
        if (keep_failed) then
           generate_new = .false.
        else
           generate_new = .true.
        end if
     else
        generate_new = .false.
     end if
  end do REJECTION
end subroutine process_mci_entry_generate_weighted_event

subroutine process_mci_entry_generate_unweighted_event (mci_entry, mci_instance, mci_sampler)
  class(process_mci_entry_t), intent(inout) :: mci_entry
```

```
    class(mci_instance_t), intent(inout) :: mci_instance
    class(mci_sampler_t), intent(inout) :: mci_sampler
    call mci_entry%mci%generate_unweighted_event (mci_instance, mci_sampler)
  end subroutine process_mci_entry_generate_unweighted_event
```

Extract results.

⟨*Process mci: process mci entry: TBP*⟩+≡

```
  procedure :: has_integral => process_mci_entry_has_integral
  procedure :: get_integral => process_mci_entry_get_integral
  procedure :: get_error => process_mci_entry_get_error
  procedure :: get_accuracy => process_mci_entry_get_accuracy
  procedure :: get_chi2 => process_mci_entry_get_chi2
  procedure :: get_efficiency => process_mci_entry_get_efficiency
```

⟨*Process mci: procedures*⟩+≡

```
  function process_mci_entry_has_integral (mci_entry) result (flag)
    class(process_mci_entry_t), intent(in) :: mci_entry
    logical :: flag
    flag = mci_entry%results%exist ()
  end function process_mci_entry_has_integral

  function process_mci_entry_get_integral (mci_entry) result (integral)
    class(process_mci_entry_t), intent(in) :: mci_entry
    real(default) :: integral
    integral = mci_entry%results%get_integral ()
  end function process_mci_entry_get_integral

  function process_mci_entry_get_error (mci_entry) result (error)
    class(process_mci_entry_t), intent(in) :: mci_entry
    real(default) :: error
    error = mci_entry%results%get_error ()
  end function process_mci_entry_get_error

  function process_mci_entry_get_accuracy (mci_entry) result (accuracy)
    class(process_mci_entry_t), intent(in) :: mci_entry
    real(default) :: accuracy
    accuracy = mci_entry%results%get_accuracy ()
  end function process_mci_entry_get_accuracy

  function process_mci_entry_get_chi2 (mci_entry) result (chi2)
    class(process_mci_entry_t), intent(in) :: mci_entry
    real(default) :: chi2
    chi2 = mci_entry%results%get_chi2 ()
  end function process_mci_entry_get_chi2

  function process_mci_entry_get_efficiency (mci_entry) result (efficiency)
    class(process_mci_entry_t), intent(in) :: mci_entry
    real(default) :: efficiency
    efficiency = mci_entry%results%get_efficiency ()
  end function process_mci_entry_get_efficiency
```

Return the MCI checksum. This may be the one used for configuration, but may also incorporate results, if they change the state of the integrator (adaptation).

⟨*Process mci: process mci entry: TBP*⟩+≡

```
procedure :: get_md5sum => process_mci_entry_get_md5sum
```

⟨*Process mci: procedures*⟩+≡

```
pure function process_mci_entry_get_md5sum (entry) result (md5sum)
  class(process_mci_entry_t), intent(in) :: entry
  character(32) :: md5sum
  md5sum = entry%mci%get_md5sum ()
end function process_mci_entry_get_md5sum
```

30.8.2 MC parameter set and MCI instance

For each process component that is associated with a multi-channel integration (MCI) object, the `mci_work_t` object contains the currently active parameter set. It also holds the implementation of the `mci_instance_t` that the integrator needs for doing its work.

⟨*Process mci: public*⟩+≡

```
public :: mci_work_t
```

⟨*Process mci: types*⟩+≡

```
type :: mci_work_t
   type(process_mci_entry_t), pointer :: config => null ()
   real(default), dimension(:), allocatable :: x
   class(mci_instance_t), pointer :: mci => null ()
   type(process_counter_t) :: counter
   logical :: keep_failed_events = .false.
 contains
 ⟨Process mci: mci work: TBP⟩
end type mci_work_t
```

First write configuration data, then the current values.

⟨*Process mci: mci work: TBP*⟩≡

```
procedure :: write => mci_work_write
```

⟨*Process mci: procedures*⟩+≡

```
subroutine mci_work_write (mci_work, unit, testflag)
  class(mci_work_t), intent(in) :: mci_work
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1x,A,I0,A)")  "Active MCI instance #", &
       mci_work%config%i_mci, " ="
  write (u, "(2x)", advance="no")
  do i = 1, mci_work%config%n_par
     write (u, "(1x,F7.5)", advance="no")  mci_work%x(i)
     if (i == mci_work%config%n_par_sf) &
          write (u, "(1x,'|')", advance="no")
  end do
  write (u, *)
  if (associated (mci_work%mci)) then
     call mci_work%mci%write (u, pacify = testflag)
     call mci_work%counter%write (u)
  end if
```

```
end subroutine mci_work_write
```

The `mci` component may require finalization.

⟨*Process mci: mci work: TBP*⟩+≡

```
procedure :: final => mci_work_final
```

⟨*Process mci: procedures*⟩+≡

```
subroutine mci_work_final (mci_work)
  class(mci_work_t), intent(inout) :: mci_work
  if (associated (mci_work%mci)) then
     call mci_work%mci%final ()
     deallocate (mci_work%mci)
  end if
end subroutine mci_work_final
```

Initialize with the maximum length that we will need. Contents are not initialized.

The integrator inside the `mci_entry` object is responsible for allocating and initializing its own instance, which is referred to by a pointer in the `mci_work` object.

⟨*Process mci: mci work: TBP*⟩+≡

```
procedure :: init => mci_work_init
```

⟨*Process mci: procedures*⟩+≡

```
subroutine mci_work_init (mci_work, mci_entry)
  class(mci_work_t), intent(out) :: mci_work
  type(process_mci_entry_t), intent(in), target :: mci_entry
  mci_work%config => mci_entry
  allocate (mci_work%x (mci_entry%n_par))
  if (allocated (mci_entry%mci)) then
     call mci_entry%mci%allocate_instance (mci_work%mci)
     call mci_work%mci%init (mci_entry%mci)
  end if
end subroutine mci_work_init
```

Set parameters explicitly, either all at once, or separately for the structure-function and process parts.

⟨*Process mci: mci work: TBP*⟩+≡

```
procedure :: set => mci_work_set
procedure :: set_x_strfun => mci_work_set_x_strfun
procedure :: set_x_process => mci_work_set_x_process
```

⟨*Process mci: procedures*⟩+≡

```
subroutine mci_work_set (mci_work, x)
  class(mci_work_t), intent(inout) :: mci_work
  real(default), dimension(:), intent(in) :: x
  mci_work%x = x
end subroutine mci_work_set

subroutine mci_work_set_x_strfun (mci_work, x)
  class(mci_work_t), intent(inout) :: mci_work
  real(default), dimension(:), intent(in) :: x
  mci_work%x(1 : mci_work%config%n_par_sf) = x
```

```
end subroutine mci_work_set_x_strfun

subroutine mci_work_set_x_process (mci_work, x)
  class(mci_work_t), intent(inout) :: mci_work
  real(default), dimension(:), intent(in) :: x
  mci_work%x(mci_work%config%n_par_sf + 1 : mci_work%config%n_par) = x
end subroutine mci_work_set_x_process
```

Return the array of active components, i.e., those that correspond to the currently selected MC parameter set.

⟨*Process mci: mci work: TBP*⟩+≡

```
procedure :: get_active_components => mci_work_get_active_components
```

⟨*Process mci: procedures*⟩+≡

```
function mci_work_get_active_components (mci_work) result (i_component)
  class(mci_work_t), intent(in) :: mci_work
  integer, dimension(:), allocatable :: i_component
  allocate (i_component (size (mci_work%config%i_component)))
  i_component = mci_work%config%i_component
end function mci_work_get_active_components
```

Return the active parameters as a simple array with correct length. Do this separately for the structure-function parameters and the process parameters.

⟨*Process mci: mci work: TBP*⟩+≡

```
procedure :: get_x_strfun => mci_work_get_x_strfun
procedure :: get_x_process => mci_work_get_x_process
```

⟨*Process mci: procedures*⟩+≡

```
function mci_work_get_x_strfun (mci_work) result (x)
  class(mci_work_t), intent(in) :: mci_work
  real(default), dimension(mci_work%config%n_par_sf) :: x
  x = mci_work%x(1 : mci_work%config%n_par_sf)
end function mci_work_get_x_strfun

function mci_work_get_x_process (mci_work) result (x)
  class(mci_work_t), intent(in) :: mci_work
  real(default), dimension(mci_work%config%n_par_phs) :: x
  x = mci_work%x(mci_work%config%n_par_sf + 1 : mci_work%config%n_par)
end function mci_work_get_x_process
```

Initialize and finalize event generation for the specified MCI entry. This also resets the counter.

⟨*Process mci: mci work: TBP*⟩+≡

```
procedure :: init_simulation => mci_work_init_simulation
procedure :: final_simulation => mci_work_final_simulation
```

⟨*Process mci: procedures*⟩+≡

```
subroutine mci_work_init_simulation (mci_work, safety_factor, keep_failed_events)
  class(mci_work_t), intent(inout) :: mci_work
  real(default), intent(in), optional :: safety_factor
  logical, intent(in), optional :: keep_failed_events
  call mci_work%mci%init_simulation (safety_factor)
  call mci_work%counter%reset ()
```

```
    if (present (keep_failed_events)) &
       mci_work%keep_failed_events = keep_failed_events
  end subroutine mci_work_init_simulation

  subroutine mci_work_final_simulation (mci_work)
    class(mci_work_t), intent(inout) :: mci_work
    call mci_work%mci%final_simulation ()
  end subroutine mci_work_final_simulation
```

Counter.

⟨*Process mci: mci work: TBP*⟩+≡

```
  procedure :: reset_counter => mci_work_reset_counter
  procedure :: record_call => mci_work_record_call
  procedure :: get_counter => mci_work_get_counter
```

⟨*Process mci: procedures*⟩+≡

```
  subroutine mci_work_reset_counter (mci_work)
    class(mci_work_t), intent(inout) :: mci_work
    call mci_work%counter%reset ()
  end subroutine mci_work_reset_counter

  subroutine mci_work_record_call (mci_work, status)
    class(mci_work_t), intent(inout) :: mci_work
    integer, intent(in) :: status
    call mci_work%counter%record (status)
  end subroutine mci_work_record_call

  pure function mci_work_get_counter (mci_work) result (counter)
    class(mci_work_t), intent(in) :: mci_work
    type(process_counter_t) :: counter
    counter = mci_work%counter
  end function mci_work_get_counter
```

30.9 Process component manager

⟨pcm.f90⟩≡

```
  ⟨File header⟩

  module pcm

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use constants, only: zero, two
    use diagnostics
    use lorentz
    use io_units, only: free_unit
    use process_constants, only: process_constants_t
    use physics_defs
    use model_data, only: model_data_t
    use quantum_numbers, only: quantum_numbers_t
    use flavors, only: flavor_t
    use nlo_data, only: nlo_settings_t
```

```
  use phs_fks, only: isr_kinematics_t, real_kinematics_t
  use phs_fks, only: phs_identifier_t
  use nlo_color_data, only: nlo_color_data_t
  use fks_regions, only: region_data_t
  use phs_fks, only: phs_fks_generator_t
  use phs_fks, only: get_boost_for_threshold_projection
  use phs_fks, only: threshold_projection_born
  use phs_fks, only: dalitz_plot_t
  use real_subtraction, only: real_subtraction_t, soft_mismatch_t
  use real_subtraction, only: FIXED_ORDER_EVENTS, POWHEG, this_purpose
  use real_subtraction, only: real_partition_t, powheg_damping_simple_t
  use real_subtraction, only: real_partition_fixed_order_t
  use virtual, only: virtual_t
  use dglap_remnant, only: dglap_remnant_t
  use ttv_formfactors, only: m1s_to_mpole

  use pcm_base
```

⟨*Standard module head*⟩

⟨*Pcm: public*⟩

⟨*Pcm: types*⟩

```
contains
```

⟨*Pcm: procedures*⟩

```
end module pcm
```

30.9.1 Default process component manager

This is the configuration object which has the duty of allocating the corresponding instance. The default version is trivial.

⟨*Pcm: public*⟩≡

```
  public :: pcm_default_t
```

⟨*Pcm: types*⟩≡

```
  type, extends (pcm_t) :: pcm_default_t
   contains
   ⟨Pcm: pcm default: TBP⟩
  end type pcm_default_t
```

⟨*Pcm: pcm default: TBP*⟩≡

```
  procedure :: allocate_instance => pcm_default_allocate_instance
```

⟨*Pcm: procedures*⟩≡

```
  subroutine pcm_default_allocate_instance (pcm, instance)
    class(pcm_default_t), intent(in) :: pcm
    class(pcm_instance_t), intent(inout), allocatable :: instance
    allocate (pcm_instance_default_t :: instance)
  end subroutine pcm_default_allocate_instance
```

⟨*Pcm: pcm default: TBP*⟩+≡
```
procedure :: final => pcm_default_final
```
⟨*Pcm: procedures*⟩+≡
```
subroutine pcm_default_final (pcm)
  class(pcm_default_t), intent(inout) :: pcm
end subroutine pcm_default_final
```

⟨*Pcm: pcm default: TBP*⟩+≡
```
procedure :: is_nlo => pcm_default_is_nlo
```
⟨*Pcm: procedures*⟩+≡
```
function pcm_default_is_nlo (pcm) result (is_nlo)
  logical :: is_nlo
  class(pcm_default_t), intent(in) :: pcm
  is_nlo = .false.
end function pcm_default_is_nlo
```

⟨*Pcm: types*⟩+≡
```
type, extends (pcm_instance_t) :: pcm_instance_default_t
contains
```
⟨*Pcm: pcm instance default: TBP*⟩
```
end type pcm_instance_default_t
```

⟨*Pcm: pcm instance default: TBP*⟩≡
```
procedure :: final => pcm_instance_default_final
```
⟨*Pcm: procedures*⟩+≡
```
subroutine pcm_instance_default_final (pcm_instance)
  class(pcm_instance_default_t), intent(inout) :: pcm_instance
end subroutine pcm_instance_default_final
```

⟨*Pcm: pcm instance default: TBP*⟩+≡
```
procedure :: is_valid => pcm_instance_default_is_valid
```
⟨*Pcm: procedures*⟩+≡
```
function pcm_instance_default_is_valid (pcm_instance) result (valid)
  logical :: valid
  class(pcm_instance_default_t), intent(in) :: pcm_instance
  valid = .true.
end function pcm_instance_default_is_valid
```

30.9.2 NLO process component manager

The NLO-aware version of the process-component manager.

This is the configuration object, which has the duty of allocating the corresponding instance. This is the nontrivial NLO version.

⟨*Pcm: public*⟩+≡
```
public :: pcm_nlo_t
```

⟨*Pcm: types*⟩+≡

```
type, extends (pcm_t) :: pcm_nlo_t
  type(nlo_settings_t) :: settings
  type(region_data_t) :: region_data
  type(nlo_color_data_t) :: color_data
  class(real_partition_t), allocatable :: real_partition
  type(dalitz_plot_t) :: dalitz_plot
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn_real, qn_born
contains
⟨Pcm: pcm nlo: TBP⟩
end type pcm_nlo_t
```

We need one subtraction index for the Born matrix element. Further, if we do not use the internal color correlation, we need $n(n-1)/2$ indices for each entry of the BLHA output array.

⟨*Pcm: pcm nlo: TBP*⟩≡

```
procedure :: get_n_sub => pcm_nlo_get_n_sub
```

⟨*Pcm: procedures*⟩+≡

```
function pcm_nlo_get_n_sub (pcm_nlo) result (n_sub)
  integer :: n_sub
  class(pcm_nlo_t), intent(in) :: pcm_nlo
  integer :: n_tot
  n_sub = 1
  if (.not. pcm_nlo%settings%use_internal_color_correlations) then
     n_tot = pcm_nlo%region_data%n_legs_born
     n_sub = n_sub + n_tot * (n_tot - 1) / 2
  else
     !!! TODO (cw-2016-12-05): Implementation
  end if
end function pcm_nlo_get_n_sub
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
procedure :: get_n_flv_born => pcm_nlo_get_n_flv_born
```

⟨*Pcm: procedures*⟩+≡

```
function pcm_nlo_get_n_flv_born (pcm_nlo) result (n_flv)
  integer :: n_flv
  class(pcm_nlo_t), intent(in) :: pcm_nlo
  n_flv = pcm_nlo%region_data%n_flv_born
end function pcm_nlo_get_n_flv_born
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
procedure :: get_n_flv_real => pcm_nlo_get_n_flv_real
```

⟨*Pcm: procedures*⟩+≡

```
function pcm_nlo_get_n_flv_real (pcm_nlo) result (n_flv)
  integer :: n_flv
  class(pcm_nlo_t), intent(in) :: pcm_nlo
  n_flv = pcm_nlo%region_data%n_flv_real
end function pcm_nlo_get_n_flv_real
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
procedure :: get_n_alr => pcm_nlo_get_n_alr
```

⟨*Pcm: procedures*⟩+≡

```
function pcm_nlo_get_n_alr (pcm) result (n_alr)
  integer :: n_alr
  class(pcm_nlo_t), intent(in) :: pcm
  n_alr = pcm%region_data%n_regions
end function pcm_nlo_get_n_alr
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
procedure :: get_flv_states => pcm_nlo_get_flv_states
```

⟨*Pcm: procedures*⟩+≡

```
function pcm_nlo_get_flv_states (pcm, born) result (flv)
  integer, dimension(:,:), allocatable :: flv
  class(pcm_nlo_t), intent(in) :: pcm
  logical, intent(in) :: born
  if (born) then
     !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
     allocate (flv (pcm%region_data%n_legs_born, pcm%region_data%n_flv_born))
     flv = pcm%region_data%get_flv_states_born ()
  else
     !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
     allocate (flv (pcm%region_data%n_legs_real, pcm%region_data%n_flv_real))
     flv = pcm%region_data%get_flv_states_real ()
  end if
end function pcm_nlo_get_flv_states
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
procedure :: get_qn => pcm_nlo_get_qn
```

⟨*Pcm: procedures*⟩+≡

```
function pcm_nlo_get_qn (pcm, born) result (qn)
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn
  class(pcm_nlo_t), intent(in) :: pcm
  logical, intent(in) :: born
  if (born) then
     !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
     allocate (qn (size (pcm%qn_born, dim = 1), size (pcm%qn_born, dim = 2)))
     qn = pcm%qn_born
  else
     !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
     allocate (qn (size (pcm%qn_real, dim = 1), size (pcm%qn_real, dim = 2)))
     qn = pcm%qn_real
  end if
end function pcm_nlo_get_qn
```

Check if there are massive emitters. Since the mass-structure of all underlying Born configurations have to be the same (**This does not have to be the case when different components are generated at LO**) , we just use the first one to determine this.

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
procedure :: has_massive_emitter => pcm_nlo_has_massive_emitter
```

⟨*Pcm: procedures*⟩+≡

```
function pcm_nlo_has_massive_emitter (pcm) result (val)
  logical :: val
  class(pcm_nlo_t), intent(in) :: pcm
  integer :: i
  val = .false.
  associate (reg_data => pcm%region_data)
     do i = reg_data%n_in + 1, reg_data%n_legs_born
        if (any (i == reg_data%emitters)) &
           val = val .or. reg_data%flv_born(1)%massive(i)
     end do
  end associate
end function pcm_nlo_has_massive_emitter
```

Returns an array which specifies if the particle at position `i` is massive.

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
procedure :: get_mass_info => pcm_nlo_get_mass_info
```

⟨*Pcm: procedures*⟩+≡

```
function pcm_nlo_get_mass_info (pcm, i_flv) result (massive)
  class(pcm_nlo_t), intent(in) :: pcm
  integer, intent(in) :: i_flv
  logical, dimension(:), allocatable :: massive
  allocate (massive (size (pcm%region_data%flv_born(i_flv)%massive)))
  massive = pcm%region_data%flv_born(i_flv)%massive
end function pcm_nlo_get_mass_info
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
procedure :: allocate_instance => pcm_nlo_allocate_instance
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_nlo_allocate_instance (pcm, instance)
  class(pcm_nlo_t), intent(in) :: pcm
  class(pcm_instance_t), intent(inout), allocatable :: instance
  allocate (pcm_instance_nlo_t :: instance)
end subroutine pcm_nlo_allocate_instance
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
procedure :: init_color_data => pcm_nlo_init_color_data
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_nlo_init_color_data (pcm, prc_constants, &
   flavor_indices, color_indices)
  class(pcm_nlo_t), intent(inout) :: pcm
  type(process_constants_t), intent(in), dimension(2) :: prc_constants
  integer, intent(in), dimension(:) :: flavor_indices, color_indices
  call pcm%color_data%prepare (pcm%region_data, flavor_indices, color_indices)
  call pcm%color_data%init_color_matrix (pcm%region_data, prc_constants, &
     pcm%settings%factorization_mode)
end subroutine pcm_nlo_init_color_data
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
procedure :: init_qn => pcm_nlo_init_qn
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_nlo_init_qn (pcm, model)
  class(pcm_nlo_t), intent(inout) :: pcm
  class(model_data_t), intent(in) :: model
  integer, dimension(:,:), allocatable :: flv_states
  type(flavor_t), dimension(:), allocatable :: flv
  integer :: i
  type(quantum_numbers_t), dimension(:), allocatable :: qn
  !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
  allocate (flv_states (pcm%region_data%n_legs_born, pcm%region_data%n_flv_born))
  flv_states = pcm%get_flv_states (.true.)
  allocate (pcm%qn_born (size (flv_states, dim = 1), size (flv_states, dim = 2)))
  allocate (flv (size (flv_states, dim = 1)))
  allocate (qn (size (flv_states, dim = 1)))
  do i = 1, pcm%get_n_flv_born ()
     call flv%init (flv_states (:,i), model)
     call qn%init (flv)
     pcm%qn_born(:,i) = qn
  end do
  deallocate (flv); deallocate (qn)
  deallocate (flv_states)
  !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
  allocate (flv_states (pcm%region_data%n_legs_real, pcm%region_data%n_flv_real))
  flv_states = pcm%get_flv_states (.false.)
  allocate (pcm%qn_real (size (flv_states, dim = 1), size (flv_states, dim = 2)))
  allocate (flv (size (flv_states, dim = 1)))
  allocate (qn (size (flv_states, dim = 1)))
  do i = 1, pcm%get_n_flv_real ()
     call flv%init (flv_states (:,i), model)
     call qn%init (flv)
     pcm%qn_real(:,i) = qn
  end do
end subroutine pcm_nlo_init_qn
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
procedure :: allocate_ps_matching => pcm_nlo_allocate_ps_matching
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_nlo_allocate_ps_matching (pcm)
  class(pcm_nlo_t), intent(inout) :: pcm
  if (.not. allocated (pcm%real_partition)) then
     allocate (powheg_damping_simple_t :: pcm%real_partition)
  end if
end subroutine pcm_nlo_allocate_ps_matching
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
procedure :: activate_dalitz_plot => pcm_nlo_activate_dalitz_plot
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_nlo_activate_dalitz_plot (pcm, filename)
  class(pcm_nlo_t), intent(inout) :: pcm
  type(string_t), intent(in) :: filename
  call pcm%dalitz_plot%init (free_unit (), filename, .false.)
  call pcm%dalitz_plot%write_header ()
```

```
    end subroutine pcm_nlo_activate_dalitz_plot
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
    procedure :: setup_real_partition => pcm_nlo_setup_real_partition
```

⟨*Pcm: procedures*⟩+≡

```
    subroutine pcm_nlo_setup_real_partition (pcm, scale)
      class(pcm_nlo_t), intent(inout) :: pcm
      real(default), intent(in) :: scale
      if (.not. allocated (pcm%real_partition)) then
         allocate (real_partition_fixed_order_t :: pcm%real_partition)
         select type (partition => pcm%real_partition)
         type is (real_partition_fixed_order_t)
            call pcm%region_data%get_all_ftuples (partition%fks_pairs)
            partition%scale = scale
         end select
      end if
    end subroutine pcm_nlo_setup_real_partition
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
    procedure :: register_dalitz_plot => pcm_nlo_register_dalitz_plot
```

⟨*Pcm: procedures*⟩+≡

```
    subroutine pcm_nlo_register_dalitz_plot (pcm, emitter, p)
      class(pcm_nlo_t), intent(inout) :: pcm
      integer, intent(in) :: emitter
      type(vector4_t), intent(in), dimension(:) :: p
      real(default) :: k0_n, k0_np1
      k0_n = p(emitter)%p(0)
      k0_np1 = p(size(p))%p(0)
      call pcm%dalitz_plot%register (k0_n, k0_np1)
    end subroutine pcm_nlo_register_dalitz_plot
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
    procedure :: setup_phs_generator => pcm_nlo_setup_phs_generator
```

⟨*Pcm: procedures*⟩+≡

```
    subroutine pcm_nlo_setup_phs_generator (pcm, pcm_instance, generator, &
       sqrts, mode, singular_jacobian)
      class(pcm_nlo_t), intent(in) :: pcm
      type(phs_fks_generator_t), intent(inout) :: generator
      type(pcm_instance_nlo_t), intent(in), target :: pcm_instance
      real(default), intent(in) :: sqrts
      integer, intent(in), optional:: mode
      logical, intent(in), optional :: singular_jacobian
      logical :: yorn
      yorn = .false.; if (present (singular_jacobian)) yorn = singular_jacobian
      call generator%connect_kinematics (pcm_instance%isr_kinematics, &
         pcm_instance%real_kinematics, pcm%has_massive_emitter ())
      generator%n_in = pcm%region_data%n_in
      call generator%set_sqrts_hat (sqrts)
      call generator%set_emitters (pcm%region_data%emitters)
      call generator%setup_masses (pcm%region_data%n_legs_born)
      generator%is_massive = pcm%get_mass_info (1)
```

```
    generator%singular_jacobian = yorn
    if (present (mode)) generator%mode = mode
  end subroutine pcm_nlo_setup_phs_generator
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
  procedure :: final => pcm_nlo_final
```

⟨*Pcm: procedures*⟩+≡

```
  subroutine pcm_nlo_final (pcm)
    class(pcm_nlo_t), intent(inout) :: pcm
    if (allocated (pcm%real_partition)) deallocate (pcm%real_partition)
    call pcm%dalitz_plot%final ()
  end subroutine pcm_nlo_final
```

⟨*Pcm: pcm nlo: TBP*⟩+≡

```
  procedure :: is_nlo => pcm_nlo_is_nlo
```

⟨*Pcm: procedures*⟩+≡

```
  function pcm_nlo_is_nlo (pcm) result (is_nlo)
    logical :: is_nlo
    class(pcm_nlo_t), intent(in) :: pcm
    is_nlo = .true.
  end function pcm_nlo_is_nlo
```

As a first implementation, it acts as a wrapper for the NLO controller object and the squared matrix-element collector.

⟨*Pcm: public*⟩+≡

```
  public :: pcm_instance_nlo_t
```

⟨*Pcm: types*⟩+≡

```
  type, extends (pcm_instance_t) :: pcm_instance_nlo_t
     integer :: active_real_component = 1
     logical :: use_internal_color_correlation = .true.
     type(real_kinematics_t), pointer :: real_kinematics => null ()
     type(isr_kinematics_t), pointer :: isr_kinematics => null ()
     type(real_subtraction_t) :: real_sub
     type(virtual_t) :: virtual
     type(soft_mismatch_t) :: soft_mismatch
     type(dglap_remnant_t) :: dglap_remnant
     logical, dimension(:), allocatable :: bad_blha_point
  contains
  ⟨Pcm: pcm instance: TBP⟩
  end type pcm_instance_nlo_t
```

⟨*Pcm: pcm instance: TBP*⟩≡

```
  procedure :: set_radiation_event => pcm_instance_nlo_set_radiation_event
  procedure :: set_subtraction_event => pcm_instance_nlo_set_subtraction_event
```

⟨*Pcm: procedures*⟩+≡

```
  subroutine pcm_instance_nlo_set_radiation_event (pcm_instance)
    class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
    pcm_instance%real_sub%radiation_event = .true.
    pcm_instance%real_sub%subtraction_event = .false.
```

```
end subroutine pcm_instance_nlo_set_radiation_event

subroutine pcm_instance_nlo_set_subtraction_event (pcm_instance)
  class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
  pcm_instance%real_sub%radiation_event = .false.
  pcm_instance%real_sub%subtraction_event = .true.
end subroutine pcm_instance_nlo_set_subtraction_event
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: disable_subtraction => pcm_instance_nlo_disable_subtraction
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_instance_nlo_disable_subtraction (pcm_instance)
  class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
  pcm_instance%real_sub%subtraction_deactivated = .true.
end subroutine pcm_instance_nlo_disable_subtraction
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: init_config => pcm_instance_nlo_init_config
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_instance_nlo_init_config (pcm_instance, active_components, &
   nlo_types, sqrts, i_real_fin)
  class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
  logical, intent(in), dimension(:) :: active_components
  integer, intent(in), dimension(:) :: nlo_types
  real(default), intent(in) :: sqrts
  integer, intent(in) :: i_real_fin
  integer :: i_component
  select type (pcm => pcm_instance%config)
  type is (pcm_nlo_t)
     do i_component = 1, size (active_components)
        if (active_components(i_component)) then
           select case (nlo_types(i_component))
           case (NLO_REAL)
              pcm_instance%active_real_component = i_component
              if (i_component /= i_real_fin) then
                 call pcm_instance%setup_real_component &
                      (pcm%settings%fks_template%subtraction_disabled, sqrts)
              end if
           case (NLO_VIRTUAL)
              call pcm_instance%init_virtual ()
           case (NLO_DGLAP)
              call pcm_instance%init_dglap_remnant ()
           end select
        end if
     end do
     allocate (pcm_instance%bad_blha_point (size (active_components)))
     pcm_instance%bad_blha_point = .false.
  end select
end subroutine pcm_instance_nlo_init_config
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: setup_real_component => pcm_instance_nlo_setup_real_component
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_instance_nlo_setup_real_component (pcm_instance, &
   subtraction_disabled, sqrts)
  class(pcm_instance_nlo_t), intent(inout), target :: pcm_instance
  logical, intent(in) :: subtraction_disabled
  real(default), intent(in) :: sqrts
  call pcm_instance%init_real_and_isr_kinematics (sqrts)
  call pcm_instance%init_real_subtraction ()
  if (subtraction_disabled) call pcm_instance%disable_subtraction ()
  select type (config => pcm_instance%config)
  type is (pcm_nlo_t)
     associate (region_data => config%region_data)
        call pcm_instance%soft_mismatch%init (region_data, &
             pcm_instance%real_kinematics, config%settings%factorization_mode)
     end associate
  end select
end subroutine pcm_instance_nlo_setup_real_component
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: init_real_and_isr_kinematics => &
     pcm_instance_nlo_init_real_and_isr_kinematics
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_instance_nlo_init_real_and_isr_kinematics (pcm_instance, sqrts)
  class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
  real(default) :: sqrts
  integer :: n_contr
  allocate (pcm_instance%real_kinematics)
  allocate (pcm_instance%isr_kinematics)
  select type (config => pcm_instance%config)
  type is (pcm_nlo_t)
     associate (region_data => config%region_data)
        if (allocated (region_data%alr_contributors)) then
           n_contr = size (region_data%alr_contributors)
        else if (config%settings%factorization_mode == FACTORIZATION_THRESHOLD) then
           n_contr = 2
        else
           n_contr = 1
        end if
        call pcm_instance%real_kinematics%init &
             (region_data%n_legs_real, region_data%n_phs, &
             region_data%n_regions, n_contr)
        if (config%settings%factorization_mode == FACTORIZATION_THRESHOLD) &
           call pcm_instance%real_kinematics%init_onshell &
                (region_data%n_legs_real, region_data%n_phs)
        pcm_instance%isr_kinematics%n_in = region_data%n_in
     end associate
  end select
  pcm_instance%isr_kinematics%beam_energy = sqrts / two
end subroutine pcm_instance_nlo_init_real_and_isr_kinematics
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: set_real_and_isr_kinematics => &
    pcm_instance_nlo_set_real_and_isr_kinematics
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_instance_nlo_set_real_and_isr_kinematics (pcm_instance, phs_identifiers, sqrts)
  class(pcm_instance_nlo_t), intent(inout), target :: pcm_instance
  type(phs_identifier_t), intent(in), dimension(:) :: phs_identifiers
  real(default), intent(in) :: sqrts
  call pcm_instance%real_sub%set_real_kinematics &
       (pcm_instance%real_kinematics)
  call pcm_instance%real_sub%set_isr_kinematics &
       (pcm_instance%isr_kinematics)
end subroutine pcm_instance_nlo_set_real_and_isr_kinematics
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: init_real_subtraction => pcm_instance_nlo_init_real_subtraction
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_instance_nlo_init_real_subtraction (pcm_instance)
  class(pcm_instance_nlo_t), intent(inout), target :: pcm_instance
  select type (config => pcm_instance%config)
  type is (pcm_nlo_t)
     associate (region_data => config%region_data)
        call pcm_instance%real_sub%init (region_data)
        call pcm_instance%real_sub%set_resonance_mappings &
             (config%settings%use_resonance_mappings)
           if (allocated (config%settings%selected_alr)) then
              associate (selected_alr => config%settings%selected_alr)
              if (any (selected_alr < 0)) then
                 call msg_fatal ("Fixed alpha region must be non-negative!")
              else if (any (selected_alr > region_data%n_regions)) then
                 call msg_fatal ("Fixed alpha region is larger than the total"&
                      &" number of singular regions!")
              else
                 allocate (pcm_instance%real_sub%selected_alr (size (selected_alr)))
                 pcm_instance%real_sub%selected_alr = selected_alr
              end if
        end associate
           end if
        pcm_instance%real_sub%sub_soft%factorization_mode &
             = config%settings%factorization_mode
     end associate
  end select
end subroutine pcm_instance_nlo_init_real_subtraction
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: set_momenta_and_scales_virtual => &
   pcm_instance_nlo_set_momenta_and_scales_virtual
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_instance_nlo_set_momenta_and_scales_virtual (pcm_instance, p, &
   ren_scale, fac_scale)
  class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
  type(vector4_t), intent(in), dimension(:) :: p
  real(default), intent(in) :: ren_scale, fac_scale
  real(default) :: sqrts, mtop
  type(lorentz_transformation_t) :: L_to_cms
```

```
    select type (config => pcm_instance%config)
    type is (pcm_nlo_t)
       associate (virtual => pcm_instance%virtual)
          call virtual%set_ren_scale (p, ren_scale)
          call virtual%set_fac_scale (p, fac_scale)
          call virtual%set_ellis_sexton_scale ()
       end associate
    end select
  end subroutine pcm_instance_nlo_set_momenta_and_scales_virtual
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
  procedure :: set_fac_scale => pcm_instance_nlo_set_fac_scale
```

⟨*Pcm: procedures*⟩+≡

```
  subroutine pcm_instance_nlo_set_fac_scale (pcm_instance, fac_scale)
    class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
    real(default), intent(in) :: fac_scale
    pcm_instance%isr_kinematics%fac_scale = fac_scale
  end subroutine pcm_instance_nlo_set_fac_scale
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
  procedure :: set_momenta => pcm_instance_nlo_set_momenta
```

⟨*Pcm: procedures*⟩+≡

```
  subroutine pcm_instance_nlo_set_momenta (pcm_instance, p_born, p_real, i_phs, cms)
    class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
    type(vector4_t), dimension(:), intent(in) :: p_born, p_real
    integer, intent(in) :: i_phs
    logical, intent(in), optional :: cms
    logical :: yorn
    yorn = .false.; if (present (cms)) yorn = cms
    associate (kinematics => pcm_instance%real_kinematics)
       if (yorn) then
          if (.not. kinematics%p_born_cms%initialized) &
             call kinematics%p_born_cms%init (size (p_born), 1)
          if (.not. kinematics%p_real_cms%initialized) &
             call kinematics%p_real_cms%init (size (p_real), 1)
          kinematics%p_born_cms%phs_point(1)%p = p_born
          kinematics%p_real_cms%phs_point(i_phs)%p = p_real
       else
          if (.not. kinematics%p_born_lab%initialized) &
             call kinematics%p_born_lab%init (size (p_born), 1)
          if (.not. kinematics%p_real_lab%initialized) &
             call kinematics%p_real_lab%init (size (p_real), 1)
          kinematics%p_born_lab%phs_point(1)%p = p_born
          kinematics%p_real_lab%phs_point(i_phs)%p = p_real
       end if
    end associate
  end subroutine pcm_instance_nlo_set_momenta
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
  procedure :: get_momenta => pcm_instance_nlo_get_momenta
```

⟨*Pcm: procedures*⟩+≡

```
function pcm_instance_nlo_get_momenta (pcm_instance, i_phs, born_phsp, cms) result (p)
  type(vector4_t), dimension(:), allocatable :: p
  class(pcm_instance_nlo_t), intent(in) :: pcm_instance
  integer, intent(in) :: i_phs
  logical, intent(in) :: born_phsp
  logical, intent(in), optional :: cms
  logical :: yorn
  yorn = .false.; if (present (cms)) yorn = cms
  select type (config => pcm_instance%config)
  type is (pcm_nlo_t)
     if (born_phsp) then
        if (yorn) then
           allocate (p (1 : config%region_data%n_legs_born), &
              source = pcm_instance%real_kinematics%p_born_cms%phs_point(1)%p)
        else
           allocate (p (1 : config%region_data%n_legs_born), &
              source = pcm_instance%real_kinematics%p_born_lab%phs_point(1)%p)
        end if
     else
        if (yorn) then
           allocate (p (1 : config%region_data%n_legs_real), &
              source = pcm_instance%real_kinematics%p_real_cms%phs_point(i_phs)%p)
        else
           allocate (p ( 1 : config%region_data%n_legs_real), &
                 source = pcm_instance%real_kinematics%p_real_lab%phs_point(i_phs)%p)
        end if
     end if
  end select
end function pcm_instance_nlo_get_momenta
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: get_xi_max => pcm_instance_nlo_get_xi_max
```

⟨*Pcm: procedures*⟩+≡

```
function pcm_instance_nlo_get_xi_max (pcm_instance, alr) result (xi_max)
  real(default) :: xi_max
  class(pcm_instance_nlo_t), intent(in) :: pcm_instance
  integer, intent(in) :: alr
  integer :: i_phs
  i_phs = pcm_instance%real_kinematics%alr_to_i_phs (alr)
  xi_max = pcm_instance%real_kinematics%xi_max (i_phs)
end function pcm_instance_nlo_get_xi_max
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: get_n_born => pcm_instance_nlo_get_n_born
```

⟨*Pcm: procedures*⟩+≡

```
function pcm_instance_nlo_get_n_born (pcm_instance) result (n_born)
  integer :: n_born
  class(pcm_instance_nlo_t), intent(in) :: pcm_instance
  select type (config => pcm_instance%config)
  type is (pcm_nlo_t)
     n_born = config%region_data%n_legs_born
```

```
    end select
  end function pcm_instance_nlo_get_n_born
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
  procedure :: get_n_real => pcm_instance_nlo_get_n_real
```

⟨*Pcm: procedures*⟩+≡

```
  function pcm_instance_nlo_get_n_real (pcm_instance) result (n_real)
    integer :: n_real
    class(pcm_instance_nlo_t), intent(in) :: pcm_instance
    select type (config => pcm_instance%config)
    type is (pcm_nlo_t)
       n_real = config%region_data%n_legs_real
    end select
  end function pcm_instance_nlo_get_n_real
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
  procedure :: get_n_regions => pcm_instance_nlo_get_n_regions
```

⟨*Pcm: procedures*⟩+≡

```
  function pcm_instance_nlo_get_n_regions (pcm_instance) result (n_regions)
    integer :: n_regions
    class(pcm_instance_nlo_t), intent(in) :: pcm_instance
    select type (config => pcm_instance%config)
    type is (pcm_nlo_t)
       n_regions = config%region_data%n_regions
    end select
  end function pcm_instance_nlo_get_n_regions
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
  procedure :: set_x_rad => pcm_instance_nlo_set_x_rad
```

⟨*Pcm: procedures*⟩+≡

```
  subroutine pcm_instance_nlo_set_x_rad (pcm_instance, x_tot)
    class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
    real(default), intent(in), dimension(:) :: x_tot
    integer :: n_par
    n_par = size (x_tot)
    if (n_par < 3) then
       pcm_instance%real_kinematics%x_rad = zero
    else
       pcm_instance%real_kinematics%x_rad = x_tot (n_par - 2 : n_par)
    end if
  end subroutine pcm_instance_nlo_set_x_rad
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
  procedure :: init_virtual => pcm_instance_nlo_init_virtual
```

⟨*Pcm: procedures*⟩+≡

```
  subroutine pcm_instance_nlo_init_virtual (pcm_instance)
    class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
    select type (config => pcm_instance%config)
    type is (pcm_nlo_t)
       associate (region_data => config%region_data)
```

```
            call pcm_instance%virtual%init (region_data%get_flv_states_born (), &
                 region_data%n_in, config%settings%use_internal_color_correlations, &
                 config%settings%virtual_selection, config%settings%virtual_resonance_aware_collinea
            pcm_instance%virtual%factorization_mode = config%settings%factorization_mode
         end associate
      end select
   end subroutine pcm_instance_nlo_init_virtual
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
   procedure :: disable_virtual_subtraction => pcm_instance_nlo_disable_virtual_subtraction
```

⟨*Pcm: procedures*⟩+≡

```
   subroutine pcm_instance_nlo_disable_virtual_subtraction (pcm_instance)
      class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
   end subroutine pcm_instance_nlo_disable_virtual_subtraction
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
   procedure :: compute_sqme_virt => pcm_instance_nlo_compute_sqme_virt
```

⟨*Pcm: procedures*⟩+≡

```
   subroutine pcm_instance_nlo_compute_sqme_virt (pcm_instance, p, &
          alpha_s, me, separate_alrs, sqme_virt)
      class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
      type(vector4_t), intent(in), dimension(:) :: p
      real(default), intent(in) :: alpha_s
      complex(default), intent(in), dimension(:) :: me
      logical, intent(in) :: separate_alrs
      real(default), dimension(:), allocatable, intent(inout) :: sqme_virt
      type(vector4_t), dimension(:), allocatable :: pp
      associate (virtual => pcm_instance%virtual)
         allocate (pp (size (p)))
         if (virtual%factorization_mode == FACTORIZATION_THRESHOLD) then
            pp = pcm_instance%real_kinematics%p_born_onshell%get_momenta (1)
         else
            pp = p
         end if
         select type (config => pcm_instance%config)
         type is (pcm_nlo_t)
            if (separate_alrs) then
               allocate (sqme_virt (config%get_n_flv_born ()))
            else
               allocate (sqme_virt (1))
            end if
            sqme_virt = zero
            call virtual%evaluate (config%region_data, &
                 alpha_s, pp, real(me), separate_alrs, sqme_virt)
         end select
      end associate
   end subroutine pcm_instance_nlo_compute_sqme_virt
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
   procedure :: compute_sqme_mismatch => pcm_instance_nlo_compute_sqme_mismatch
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_instance_nlo_compute_sqme_mismatch (pcm_instance, &
         alpha_s, separate_alrs, sqme_mism)
  class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
  real(default), intent(in) :: alpha_s
  logical, intent(in) :: separate_alrs
  real(default), dimension(:), allocatable, intent(inout) :: sqme_mism
  select type (config => pcm_instance%config)
  type is (pcm_nlo_t)
     if (separate_alrs) then
        allocate (sqme_mism (config%get_n_flv_born ()))
     else
        allocate (sqme_mism (1))
     end if
     sqme_mism = zero
     sqme_mism = pcm_instance%soft_mismatch%evaluate (alpha_s)
  end select
end subroutine pcm_instance_nlo_compute_sqme_mismatch
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: compute_sqme_dglap_remnant => pcm_instance_nlo_compute_sqme_dglap_remnant
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_instance_nlo_compute_sqme_dglap_remnant (pcm_instance, &
          alpha_s, sqme_born, separate_alrs, sqme_dglap)
  class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
  real(default), intent(in) :: alpha_s
  real(default), dimension(:), intent(in) :: sqme_born
  logical, intent(in) :: separate_alrs
  real(default), dimension(:), allocatable, intent(inout) :: sqme_dglap
  select type (config => pcm_instance%config)
  type is (pcm_nlo_t)
     if (separate_alrs) then
        allocate (sqme_dglap (config%get_n_flv_born ()))
     else
        allocate (sqme_dglap (1))
     end if
  end select
  sqme_dglap = pcm_instance%dglap_remnant%evaluate (alpha_s, sqme_born)
end subroutine pcm_instance_nlo_compute_sqme_dglap_remnant
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: set_fixed_order_event_mode => pcm_instance_nlo_set_fixed_order_event_mode
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_instance_nlo_set_fixed_order_event_mode (pcm_instance)
  class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
  pcm_instance%real_sub%purpose = FIXED_ORDER_EVENTS
end subroutine pcm_instance_nlo_set_fixed_order_event_mode
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: set_powheg_mode => pcm_instance_nlo_set_powheg_mode
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_instance_nlo_set_powheg_mode (pcm_instance)
  class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
  pcm_instance%real_sub%purpose = POWHEG
end subroutine pcm_instance_nlo_set_powheg_mode
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: init_dglap_remnant => pcm_instance_nlo_init_dglap_remnant
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_instance_nlo_init_dglap_remnant (pcm_instance)
  class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
  select type (config => pcm_instance%config)
  type is (pcm_nlo_t)
     call pcm_instance%dglap_remnant%init (pcm_instance%isr_kinematics, &
        config%region_data%get_flv_states_born (), config%get_n_alr ())
  end select
end subroutine pcm_instance_nlo_init_dglap_remnant
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: is_fixed_order_nlo_events &
     => pcm_instance_nlo_is_fixed_order_nlo_events
```

⟨*Pcm: procedures*⟩+≡

```
function pcm_instance_nlo_is_fixed_order_nlo_events (pcm_instance) result (is_nlo)
  logical :: is_nlo
  class(pcm_instance_nlo_t), intent(in) :: pcm_instance
  is_nlo = pcm_instance%real_sub%purpose == FIXED_ORDER_EVENTS
end function pcm_instance_nlo_is_fixed_order_nlo_events
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: final => pcm_instance_nlo_final
```

⟨*Pcm: procedures*⟩+≡

```
subroutine pcm_instance_nlo_final (pcm_instance)
  class(pcm_instance_nlo_t), intent(inout) :: pcm_instance
  call pcm_instance%real_sub%final ()
  call pcm_instance%virtual%final ()
  call pcm_instance%soft_mismatch%final ()
  call pcm_instance%dglap_remnant%final ()
  if (associated (pcm_instance%real_kinematics)) then
     call pcm_instance%real_kinematics%final ()
     nullify (pcm_instance%real_kinematics)
  end if
  if (associated (pcm_instance%isr_kinematics)) then
     nullify (pcm_instance%isr_kinematics)
  end if
end subroutine pcm_instance_nlo_final
```

⟨*Pcm: pcm instance: TBP*⟩+≡

```
procedure :: is_valid => pcm_instance_nlo_is_valid
```

⟨*Pcm: procedures*⟩+≡

```
function pcm_instance_nlo_is_valid (pcm_instance) result (valid)
  logical :: valid
  class(pcm_instance_nlo_t), intent(in) :: pcm_instance
  valid = .not. any (pcm_instance%bad_blha_point)
end function pcm_instance_nlo_is_valid
```

30.10 Kinematics instance

In this data type we combine all objects (instances) necessary for generating (or recovering) a kinematical configuration. The components work together as an implementation of multi-channel phase space.

`sf_chain` is an instance of the structure-function chain. It is used both for generating kinematics and, after the proper scale has been determined, evaluating the structure function entries.

`phs` is an instance of the phase space for the elementary process.

The array `f` contains the products of the Jacobians that originate from parameter mappings in the structure-function chain or in the phase space. We allocate this explicitly if either `sf_chain` or `phs` are explicitly allocated, otherwise we can take over a pointer.

All components are implemented as pointers to (anonymous) targets. For each component, there is a flag that tells whether this component is to be regarded as a proper component ('owned' by the object) or as a pointer.

⟨`kinematics.f90`⟩≡

⟨*File header*⟩

```
module kinematics
```

⟨*Use kinds*⟩

```
  use format_utils, only: write_separator
  use diagnostics
  use constants, only: two
  use io_units
  use lorentz
  use physics_defs
  use sf_base
  use phs_base
  use interactions
  use mci_base
  use phs_fks
  use fks_regions
  use process_config
  use process_mci
  use pcm, only: pcm_instance_nlo_t
  use ttv_formfactors, only: m1s_to_mpole
```

⟨*Standard module head*⟩

⟨*Kinematics: public*⟩

```
⟨Kinematics: types⟩

contains

⟨Kinematics: procedures⟩

end module kinematics
```

⟨*Kinematics: public*⟩≡

```
public :: kinematics_t
```

⟨*Kinematics: types*⟩≡

```
type :: kinematics_t
   integer :: n_in = 0
   integer :: n_channel = 0
   integer :: selected_channel = 0
   type(sf_chain_instance_t), pointer :: sf_chain => null ()
   class(phs_t), pointer :: phs => null ()
   real(default), dimension(:), pointer :: f => null ()
   real(default) :: phs_factor
   logical :: sf_chain_allocated = .false.
   logical :: phs_allocated = .false.
   logical :: f_allocated = .false.
   integer :: emitter = -1
   integer :: i_phs = 0
   integer :: i_con = 0
   logical :: only_cm_frame = .false.
   logical :: new_seed = .true.
   logical :: threshold = .false.
 contains
 ⟨Kinematics: kinematics: TBP⟩
end type kinematics_t
```

Output. Show only those components which are marked as owned.

⟨*Kinematics: kinematics: TBP*⟩≡

```
procedure :: write => kinematics_write
```

⟨*Kinematics: procedures*⟩≡

```
subroutine kinematics_write (object, unit)
  class(kinematics_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, c
  u = given_output_unit (unit)
  if (object%f_allocated) then
     write (u, "(1x,A)")  "Flux * PHS volume:"
     write (u, "(2x,ES19.12)")  object%phs_factor
     write (u, "(1x,A)")  "Jacobian factors per channel:"
     do c = 1, size (object%f)
        write (u, "(3x,I0,':',1x,ES14.7)", advance="no")  c, object%f(c)
        if (c == object%selected_channel) then
           write (u, "(1x,A)")  "[selected]"
        else
           write (u, *)
        end if
     end do
```

```
    end if
    if (object%sf_chain_allocated) then
       call write_separator (u)
       call object%sf_chain%write (u)
    end if
    if (object%phs_allocated) then
       call write_separator (u)
       call object%phs%write (u)
    end if
  end subroutine kinematics_write
```

Finalizer. Delete only those components which are marked as owned.

⟨*Kinematics: kinematics: TBP*⟩+≡

```
  procedure :: final => kinematics_final
```

⟨*Kinematics: procedures*⟩+≡

```
  subroutine kinematics_final (object)
    class(kinematics_t), intent(inout) :: object
    if (object%sf_chain_allocated) then
       call object%sf_chain%final ()
       deallocate (object%sf_chain)
       object%sf_chain_allocated = .false.
    end if
    if (object%phs_allocated) then
       call object%phs%final ()
       deallocate (object%phs)
       object%phs_allocated = .false.
    end if
    if (object%f_allocated) then
       deallocate (object%f)
       object%f_allocated = .false.
    end if
  end subroutine kinematics_final
```

Set the flags indicating whether the phase space shall be set up for the calculation of the real contribution. For this case, also set the emitter.

⟨*Kinematics: kinematics: TBP*⟩+≡

```
  procedure :: set_nlo_info => kinematics_set_nlo_info
```

⟨*Kinematics: procedures*⟩+≡

```
  subroutine kinematics_set_nlo_info (k, nlo_type)
    class(kinematics_t), intent(inout) :: k
    integer, intent(in) :: nlo_type
    if (nlo_type == NLO_VIRTUAL) k%only_cm_frame = .true.
  end subroutine kinematics_set_nlo_info
```

Allocate the structure-function chain instance, initialize it as a copy of the `sf_chain` template, and prepare it for evaluation.

The `sf_chain` remains a target because the (usually constant) beam momenta are taken from there.

⟨*Kinematics: kinematics: TBP*⟩+≡

```
  procedure :: init_sf_chain => kinematics_init_sf_chain
```

⟨*Kinematics: procedures*⟩+≡

```
subroutine kinematics_init_sf_chain (k, sf_chain, config)
  class(kinematics_t), intent(inout) :: k
  type(sf_chain_t), intent(in), target :: sf_chain
  type(process_beam_config_t), intent(in) :: config
  integer :: n_strfun, n_channel
  integer :: c
  k%n_in = config%data%get_n_in ()
  n_strfun = config%n_strfun
  n_channel = config%n_channel
  allocate (k%sf_chain)
  k%sf_chain_allocated = .true.
  call k%sf_chain%init (sf_chain, n_channel)
  if (n_strfun /= 0) then
     do c = 1, n_channel
        call k%sf_chain%set_channel (c, config%sf_channel(c))
     end do
  end if
  call k%sf_chain%link_interactions ()
  call k%sf_chain%exchange_mask ()
  call k%sf_chain%init_evaluators ()
end subroutine kinematics_init_sf_chain
```

Allocate and initialize the phase-space part and the array of Jacobian factors.

⟨*Kinematics: kinematics: TBP*⟩+≡

```
procedure :: init_phs => kinematics_init_phs
```

⟨*Kinematics: procedures*⟩+≡

```
subroutine kinematics_init_phs (k, config)
  class(kinematics_t), intent(inout) :: k
  class(phs_config_t), intent(in), target :: config
  k%n_channel = config%get_n_channel ()
  call config%allocate_instance (k%phs)
  call k%phs%init (config)
  k%phs_allocated = .true.
  allocate (k%f (k%n_channel))
  k%f = 0
  k%f_allocated = .true.
end subroutine kinematics_init_phs
```

⟨*Kinematics: kinematics: TBP*⟩+≡

```
procedure :: evaluate_radiation_kinematics => kinematics_evaluate_radiation_kinematics
```

⟨*Kinematics: procedures*⟩+≡

```
subroutine kinematics_evaluate_radiation_kinematics (k, r_in, reg_data, nlo_type)
  class(kinematics_t), intent(inout) :: k
  real(default), intent(in), dimension(:) :: r_in
  type(region_data_t), intent(in) :: reg_data
  integer, intent(in) :: nlo_type
  logical :: use_contributors
  use_contributors = allocated (reg_data%alr_contributors)
  select type (phs => k%phs)
  type is (phs_fks_t)
     if (use_contributors) then
```

```
         call phs%compute_xi_ref_momenta (reg_data%alr_contributors)
      else if (k%threshold) then
         if (.not. is_subtraction_component (k%emitter, nlo_type)) &
              call phs%compute_xi_ref_momenta_threshold ()
      else
         call phs%compute_xi_ref_momenta ()
      end if
      call phs%generate_radiation_variables &
           (r_in(phs%n_r_born + 1 : phs%n_r_born + 3), k%threshold)
      call phs%compute_cms_energy ()
    end select
  end subroutine kinematics_evaluate_radiation_kinematics
```

Generate kinematics, given a phase-space channel and a MC parameter set. The main result is the momentum array `p`, but we also fill the momentum entries in the structure-function chain and the Jacobian-factor array `f`. Regarding phase space, We fill only the parameter arrays for the selected channel.

⟨*Kinematics: kinematics: TBP*⟩+≡

```
  procedure :: compute_selected_channel => kinematics_compute_selected_channel
```

⟨*Kinematics: procedures*⟩+≡

```
  subroutine kinematics_compute_selected_channel &
       (k, mci_work, phs_channel, p, success)
    class(kinematics_t), intent(inout) :: k
    type(mci_work_t), intent(in) :: mci_work
    integer, intent(in) :: phs_channel
    type(vector4_t), dimension(:), intent(out) :: p
    logical, intent(out) :: success
    integer :: sf_channel
    k%selected_channel = phs_channel
    sf_channel = k%phs%config%get_sf_channel (phs_channel)
    call k%sf_chain%compute_kinematics (sf_channel, mci_work%get_x_strfun ())
    call k%sf_chain%get_out_momenta (p(1:k%n_in))
    call k%phs%set_incoming_momenta (p(1:k%n_in))
    call k%phs%compute_flux ()
    call k%phs%select_channel (phs_channel)
    call k%phs%evaluate_selected_channel (phs_channel, &
         mci_work%get_x_process ())

    select type (phs => k%phs)
    type is (phs_fks_t)
      if (phs%q_defined) then
         call phs%get_born_momenta (p)
         k%phs_factor = phs%get_overall_factor ()
         success = .true.
      else
         k%phs_factor = 0
         success = .false.
      end if
    class default
      if (phs%q_defined) then
         call k%phs%get_outgoing_momenta (p(k%n_in + 1 :))
         k%phs_factor = k%phs%get_overall_factor ()
         success = .true.
```

```
        if (k%only_cm_frame) then
           if (.not. k%lab_is_cm_frame()) &
              call k%boost_to_cm_frame (p)
        end if
     else
        k%phs_factor = 0
        success = .false.
     end if
   end select
end subroutine kinematics_compute_selected_channel
```

Complete kinematics by filling the non-selected phase-space parameter arrays.

⟨*Kinematics: kinematics: TBP*⟩+≡

```
procedure :: compute_other_channels => kinematics_compute_other_channels
```

⟨*Kinematics: procedures*⟩+≡

```
subroutine kinematics_compute_other_channels (k, mci_work, phs_channel)
  class(kinematics_t), intent(inout) :: k
  type(mci_work_t), intent(in) :: mci_work
  integer, intent(in) :: phs_channel
  integer :: c, c_sf
  call k%phs%evaluate_other_channels (phs_channel)
  do c = 1, k%n_channel
     c_sf = k%phs%config%get_sf_channel (c)
     k%f(c) = k%sf_chain%get_f (c_sf) * k%phs%get_f (c)
  end do
end subroutine kinematics_compute_other_channels
```

Just fetch the outgoing momenta of the `sf_chain` subobject, which become the incoming (seed) momenta of the hard interaction.

This is a stripped down-version of the above which we use when recovering kinematics. Momenta are known, but no MC parameters yet.

(We do not use the `get_out_momenta` method of the chain, since this relies on the structure-function interactions, which are not necessary filled here. We do rely on the momenta of the last evaluator in the chain, however.)

⟨*Kinematics: kinematics: TBP*⟩+≡

```
procedure :: get_incoming_momenta => kinematics_get_incoming_momenta
```

⟨*Kinematics: procedures*⟩+≡

```
subroutine kinematics_get_incoming_momenta (k, p)
  class(kinematics_t), intent(in) :: k
  type(vector4_t), dimension(:), intent(out) :: p
  type(interaction_t), pointer :: int
  integer :: i
  int => k%sf_chain%get_out_int_ptr ()
  do i = 1, k%n_in
     p(i) = int%get_momentum (k%sf_chain%get_out_i (i))
  end do
end subroutine kinematics_get_incoming_momenta
```

This inverts the remainder of the above `compute` method. We know the momenta and recover the rest, as far as needed. If we select a channel, we can

complete the inversion and reconstruct the MC parameter set.

⟨*Kinematics: kinematics: TBP*⟩+≡

```
procedure :: recover_mcpar => kinematics_recover_mcpar
```

⟨*Kinematics: procedures*⟩+≡

```
subroutine kinematics_recover_mcpar (k, mci_work, phs_channel, p)
  class(kinematics_t), intent(inout) :: k
  type(mci_work_t), intent(inout) :: mci_work
  integer, intent(in) :: phs_channel
  type(vector4_t), dimension(:), intent(in) :: p
  integer :: c, c_sf
  real(default), dimension(:), allocatable :: x_sf, x_phs
  c = phs_channel
  c_sf = k%phs%config%get_sf_channel (c)
  k%selected_channel = c
  call k%sf_chain%recover_kinematics (c_sf)
  call k%phs%set_incoming_momenta (p(1:k%n_in))
  call k%phs%compute_flux ()
  call k%phs%set_outgoing_momenta (p(k%n_in+1:))
  call k%phs%inverse ()
  do c = 1, k%n_channel
     c_sf = k%phs%config%get_sf_channel (c)
     k%f(c) = k%sf_chain%get_f (c_sf) * k%phs%get_f (c)
  end do
  k%phs_factor = k%phs%get_overall_factor ()
  c = phs_channel
  c_sf = k%phs%config%get_sf_channel (c)
  allocate (x_sf (k%sf_chain%config%get_n_bound ()))
  allocate (x_phs (k%phs%config%get_n_par ()))
  call k%phs%select_channel (c)
  call k%sf_chain%get_mcpar (c_sf, x_sf)
  call k%phs%get_mcpar (c, x_phs)
  call mci_work%set_x_strfun (x_sf)
  call mci_work%set_x_process (x_phs)
end subroutine kinematics_recover_mcpar
```

Retrieve the MC input parameter array for a specific channel. We assume that the kinematics is complete, so this is known for all channels.

⟨*Kinematics: kinematics: TBP*⟩+≡

```
procedure :: get_mcpar => kinematics_get_mcpar
```

⟨*Kinematics: procedures*⟩+≡

```
subroutine kinematics_get_mcpar (k, phs_channel, r)
  class(kinematics_t), intent(in) :: k
  integer, intent(in) :: phs_channel
  real(default), dimension(:), intent(out) :: r
  integer :: sf_channel, n_par_sf, n_par_phs
  sf_channel = k%phs%config%get_sf_channel (phs_channel)
  n_par_phs = k%phs%config%get_n_par ()
  n_par_sf = k%sf_chain%config%get_n_bound ()
  if (n_par_sf > 0) then
     call k%sf_chain%get_mcpar (sf_channel, r(1:n_par_sf))
  end if
  if (n_par_phs > 0) then
```

```
      call k%phs%get_mcpar (phs_channel, r(n_par_sf+1:))
    end if
  end subroutine kinematics_get_mcpar
```

Evaluate the structure function chain, assuming that kinematics is known.

The status must be precisely SF_DONE_KINEMATICS. We thus avoid evaluating the chain twice via different pointers to the same target.

⟨*Kinematics: kinematics: TBP*⟩+≡

```
  procedure :: evaluate_sf_chain => kinematics_evaluate_sf_chain
```

⟨*Kinematics: procedures*⟩+≡

```
  subroutine kinematics_evaluate_sf_chain (k, fac_scale)
    class(kinematics_t), intent(inout) :: k
    real(default), intent(in) :: fac_scale
    select case (k%sf_chain%get_status ())
    case (SF_DONE_KINEMATICS)
      call k%sf_chain%evaluate (fac_scale)
    end select
  end subroutine kinematics_evaluate_sf_chain
```

Recover beam momenta, i.e., return the beam momenta stored in the current sf_chain to their source. This is a side effect.

⟨*Kinematics: kinematics: TBP*⟩+≡

```
  procedure :: return_beam_momenta => kinematics_return_beam_momenta
```

⟨*Kinematics: procedures*⟩+≡

```
  subroutine kinematics_return_beam_momenta (k)
    class(kinematics_t), intent(in) :: k
    call k%sf_chain%return_beam_momenta ()
  end subroutine kinematics_return_beam_momenta
```

Check wether the phase space is configured in the center-of-mass frame. Relevant for using the proper momenta input for BLHA matrix elements.

⟨*Kinematics: kinematics: TBP*⟩+≡

```
  procedure :: lab_is_cm_frame => kinematics_lab_is_cm_frame
```

⟨*Kinematics: procedures*⟩+≡

```
  function kinematics_lab_is_cm_frame (k) result (cm_frame)
     logical :: cm_frame
     class(kinematics_t), intent(in) :: k
     cm_frame = k%phs%config%cm_frame
  end function kinematics_lab_is_cm_frame
```

Boost to center-of-mass frame

⟨*Kinematics: kinematics: TBP*⟩+≡

```
  procedure :: boost_to_cm_frame => kinematics_boost_to_cm_frame
```

⟨*Kinematics: procedures*⟩+≡

```
  subroutine kinematics_boost_to_cm_frame (k, p)
     class(kinematics_t), intent(in) :: k
     type(vector4_t), intent(inout), dimension(:) :: p
     p = inverse (k%phs%lt_cm_to_lab) * p
  end subroutine kinematics_boost_to_cm_frame
```

⟨*Kinematics: kinematics: TBP*⟩+≡

```
procedure :: modify_momenta_for_subtraction => kinematics_modify_momenta_for_subtraction
```

⟨*Kinematics: procedures*⟩+≡

```
subroutine kinematics_modify_momenta_for_subtraction (k, p_in, p_out)
  class(kinematics_t), intent(inout) :: k
  type(vector4_t), intent(in), dimension(:) :: p_in
  type(vector4_t), intent(out), dimension(:), allocatable :: p_out
  allocate (p_out (size (p_in)))
  if (k%threshold) then
     select type (phs => k%phs)
     type is (phs_fks_t)
        p_out = phs%get_onshell_projected_momenta ()
     end select
  else
     p_out = p_in
  end if
end subroutine kinematics_modify_momenta_for_subtraction
```

⟨*Kinematics: kinematics: TBP*⟩+≡

```
procedure :: threshold_projection => kinematics_threshold_projection
```

⟨*Kinematics: procedures*⟩+≡

```
subroutine kinematics_threshold_projection (k, pcm_instance, nlo_type)
  class(kinematics_t), intent(inout) :: k
  type(pcm_instance_nlo_t), intent(inout) :: pcm_instance
  integer, intent(in) :: nlo_type
  real(default) :: sqrts, mtop
  type(lorentz_transformation_t) :: L_to_cms
  type(vector4_t), dimension(:), allocatable :: p_tot
  integer :: n_tot
  n_tot = k%phs%get_n_tot ()
  select type (phs => k%phs)
  type is (phs_fks_t)
     if (nlo_type == NLO_REAL) then !!! Pure Real
          n_tot = n_tot - 1
          allocate (p_tot (n_tot))
          p_tot(1 : k%n_in) = phs%p_born
          p_tot(k%n_in + 1 : n_tot) = phs%q_born
     else !!! Mismatch
        allocate (p_tot (n_tot))
        p_tot(1 : k%n_in) = k%phs%p
        p_tot(k%n_in + 1 : n_tot) = phs%q
     end if
     sqrts = two * energy (phs%p_born(1))
  class default !!! Virtual, Born must be checked before
     allocate (p_tot (n_tot))
     p_tot(1 : k%n_in) = k%phs%p
     p_tot(k%n_in + 1 : n_tot) = phs%q
     sqrts = two * energy (phs%p(1))
  end select
  mtop = m1s_to_mpole (sqrts)
  L_to_cms = get_boost_for_threshold_projection (p_tot, sqrts, mtop)
  call pcm_instance%real_kinematics%p_born_cms%set_momenta (1, p_tot)
  associate (p_onshell => pcm_instance%real_kinematics%p_born_onshell%phs_point(1)%p)
```

```
      call threshold_projection_born (mtop, L_to_cms, p_tot, p_onshell)
      if (debug2_active (D_THRESHOLD)) then
         print *, 'On-shell projected Born: '
         call vector4_write_set (p_onshell)
      end if
    end associate
  end subroutine kinematics_threshold_projection
```

⟨*Kinematics: kinematics: TBP*⟩+≡

```
  procedure :: evaluate_radiation => kinematics_evaluate_radiation
```

⟨*Kinematics: procedures*⟩+≡

```
  subroutine kinematics_evaluate_radiation (k, p_in, p_out, success)
    class(kinematics_t), intent(inout) :: k
    type(vector4_t), intent(in), dimension(:) :: p_in
    type(vector4_t), intent(out), dimension(:), allocatable :: p_out
    logical, intent(out) :: success
    type(phs_point_t) :: p
    type(vector4_t), dimension(:), allocatable :: p_born
    real(default) :: xi_max_offshell, xi_offshell, y_offshell, jac_rand_dummy, phi
    select type (phs => k%phs)
    type is (phs_fks_t)
       allocate (p_born (size (p_in)))
       if (k%threshold) then
          p_born = phs%get_onshell_projected_momenta ()
       else
          p_born = p_in
       end if
       call phs%compute_xi_max (p_born, k%threshold)
       if (k%emitter >= 0) then
          p = size (p_born) + 1
          allocate (p_out (size (p_born) + 1))
          if (k%emitter <= k%n_in) then
             call phs%generate_isr (k%i_phs, p_born, p)
          else
             if (k%threshold) then
                jac_rand_dummy = 1._default
                call compute_y_from_emitter (phs%generator%real_kinematics%x_rad (I_Y), &
                     phs%generator%real_kinematics%p_born_cms%get_momenta(1), &
                     k%emitter, .false., phs%generator%y_max, jac_rand_dummy, &
                     y_offshell)
                call phs%compute_xi_max (k%emitter, k%i_phs, y_offshell, &
                     phs%generator%real_kinematics%p_born_cms%get_momenta(1), &
                     xi_max_offshell)
                xi_offshell = xi_max_offshell * phs%generator%real_kinematics%xi_tilde
                phi = phs%generator%real_kinematics%phi
                call phs%generate_fsr (k%emitter, k%i_phs, p, &
                     xi_y_phi = [xi_offshell, y_offshell, phi], no_jacobians = .true.)
                call phs%generator%real_kinematics%p_real_cms%set_momenta (k%i_phs, p)
                call phs%generate_fsr_threshold (k%emitter, k%i_phs, p)
                if (debug2_active (D_SUBTRACTION)) &
                     call generate_fsr_threshold_for_other_emitters (k%emitter, k%i_phs)
             else if (k%i_con > 0) then
                call phs%generate_fsr (k%emitter, k%i_phs, p, k%i_con)
```

```
            else
               call phs%generate_fsr (k%emitter, k%i_phs, p)
            end if
         end if
         success = check_scalar_products (p%p)
         if (debug_active (D_SUBTRACTION)) then
            call msg_debug (D_SUBTRACTION, "Real phase-space: ")
            call p%write ()
         end if
         p_out = p%p
      else
         allocate (p_out (size (p_in))); p_out = p_in
         success = .true.
      end if
   end select
contains
   subroutine generate_fsr_threshold_for_other_emitters (emitter, i_phs)
     integer, intent(in) :: emitter, i_phs
     integer :: ii_phs, this_emitter
     select type (phs => k%phs)
     type is (phs_fks_t)
        do ii_phs = 1, size (phs%phs_identifiers)
           this_emitter = phs%phs_identifiers(ii_phs)%emitter
           if (ii_phs /= i_phs .and. this_emitter /= emitter) &
                call phs%generate_fsr_threshold (this_emitter, i_phs)
        end do
     end select
   end subroutine
end subroutine kinematics_evaluate_radiation
```

30.11 Instances

⟨instances.f90⟩≡

```
⟨File header⟩

module instances

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_utils, only: write_separator
  use constants
  use diagnostics
  use os_interface
  use numeric_utils
  use lorentz
  use mci_base
  use particles
  use interactions
  use quantum_numbers
  use model_data
  use helicities
```

```
  use flavors
  use beam_structures
  use variables
  use sf_base
  use physics_defs
  use process_constants
  use process_libraries
  use state_matrices
  use integration_results
  use phs_base
  use prc_core, only: prc_core_t, prc_core_state_t

  !!! We should depend less on these modules (move it to pcm_nlo_t e.g.)
  use phs_wood, only: phs_wood_t, phs_wood_config_t
  use phs_fks
  use blha_olp_interfaces, only: prc_blha_t
  use blha_config, only: BLHA_AMP_CC
  use prc_user_defined, only: prc_user_defined_base_t, user_defined_state_t
  use prc_threshold, only: prc_threshold_t
  use blha_olp_interfaces, only: blha_result_array_size
  use prc_openloops, only: prc_openloops_t, openloops_state_t
  use prc_recola, only: prc_recola_t

  use ttv_formfactors, only: m1s_to_mpole
  !!! local modules
  use parton_states
  use process_counter
  use pcm_base
  use pcm
  use core_manager
  use process_config
  use process_mci
  use process
  use kinematics
```

⟨*Standard module head*⟩

⟨*Instances: public*⟩

⟨*Instances: types*⟩

⟨*Instances: interfaces*⟩

```
contains
```

⟨*Instances: procedures*⟩

```
end module instances
```

30.11.1 Term instance

A `term_instance_t` object contains all data that describe a term. Each process component consists of one or more distinct terms which may differ in kinematics,

but whose squared transition matrices have to be added pointwise.

The `active` flag is set when this term is connected to an active process component. Inactive terms are skipped for kinematics and evaluation.

The `k_term` object is the instance of the kinematics setup (structure-function chain, phase space, etc.) that applies specifically to this term. In ordinary cases, it consists of straight pointers to the seed kinematics.

The `amp` array stores the amplitude values when we get them from evaluating the associated matrix-element code.

The `int_hard` interaction describes the elementary hard process. It receives the momenta and the amplitude entries for each sampling point.

The `isolated` object holds the effective parton state for the elementary interaction. The amplitude entries are computed from `int_hard`.

The `connected` evaluator set convolutes this scattering matrix with the beam (and possibly structure-function) density matrix.

The `checked` flag is set once we have applied cuts on this term. The result of this is stored in the `passed` flag. Once the term has passed cuts, we calculate the various scale and weight expressions.

⟨*Instances: types*⟩≡

```
type :: term_instance_t
   type(process_term_t), pointer :: config => null ()
   logical :: active = .false.
   type(kinematics_t) :: k_term
   complex(default), dimension(:), allocatable :: amp
   type(interaction_t) :: int_hard
   type(isolated_state_t) :: isolated
   type(connected_state_t) :: connected
   class(prc_core_state_t), allocatable :: core_state
   logical :: checked = .false.
   logical :: passed = .false.
   real(default) :: scale = 0
   real(default) :: fac_scale = 0
   real(default) :: ren_scale = 0
   real(default), allocatable :: alpha_qcd_forced
   real(default) :: weight = 1
   type(vector4_t), dimension(:), allocatable :: p_seed
   type(vector4_t), dimension(:), allocatable :: p_hard
   type(pcm_instance_nlo_t), pointer :: pcm_instance => null ()
   integer :: nlo_type = BORN
   integer, dimension(:), allocatable :: same_kinematics
 contains
 ⟨Instances: term instance: TBP⟩
end type term_instance_t
```

⟨*Instances: term instance: TBP*⟩≡

```
procedure :: write => term_instance_write
```

⟨*Instances: procedures*⟩≡

```
subroutine term_instance_write (term, unit, show_eff_state, testflag)
  class(term_instance_t), intent(in) :: term
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: show_eff_state
  logical, intent(in), optional :: testflag
```

```
    integer :: u
    logical :: state
    u = given_output_unit (unit)
    state = .true.;  if (present (show_eff_state))  state = show_eff_state
    if (term%active) then
       if (associated (term%config)) then
          write (u, "(1x,A,I0,A,I0,A)")  "Term #", term%config%i_term, &
               " (component #", term%config%i_component, ")"
       else
          write (u, "(1x,A)")  "Term [undefined]"
       end if
    else
       write (u, "(1x,A,I0,A)")  "Term #", term%config%i_term, &
            " [inactive]"
    end if
    if (term%checked) then
       write (u, "(3x,A,L1)")      "passed cuts           = ", term%passed
    end if
    if (term%passed) then
       write (u, "(3x,A,ES19.12)")  "overall scale         = ", term%scale
       write (u, "(3x,A,ES19.12)")  "factorization scale   = ", term%fac_scale
       write (u, "(3x,A,ES19.12)")  "renormalization scale = ", term%ren_scale
       if (allocated (term%alpha_qcd_forced)) then
          write (u, "(3x,A,ES19.12)")  "alpha(QCD) forced     = ", &
               term%alpha_qcd_forced
       end if
       write (u, "(3x,A,ES19.12)")  "reweighting factor    = ", term%weight
    end if
    call term%k_term%write (u)
    call write_separator (u)
    write (u, "(1x,A)")  "Amplitude (transition matrix of the &
         &hard interaction):"
    call write_separator (u)
    call term%int_hard%basic_write (u, testflag = testflag)
    if (state .and. term%isolated%has_trace) then
       call write_separator (u)
       write (u, "(1x,A)")  "Evaluators for the hard interaction:"
       call term%isolated%write (u, testflag = testflag)
    end if
    if (state .and. term%connected%has_trace) then
       call write_separator (u)
       write (u, "(1x,A)")  "Evaluators for the connected process:"
       call term%connected%write (u, testflag = testflag)
    end if
  end subroutine term_instance_write
```

The interactions and evaluators must be finalized.

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: final => term_instance_final
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_final (term)
    class(term_instance_t), intent(inout) :: term
    if (allocated (term%amp)) deallocate (term%amp)
```

```
    if (allocated (term%core_state)) deallocate (term%core_state)
    if (allocated (term%alpha_qcd_forced)) &
       deallocate (term%alpha_qcd_forced)
    if (allocated (term%p_seed)) deallocate(term%p_seed)
    if (allocated (term%p_hard)) deallocate (term%p_hard)
    call term%k_term%final ()
    call term%connected%final ()
    call term%isolated%final ()
    call term%int_hard%final ()
    term%pcm_instance => null ()
  end subroutine term_instance_final
```

For initialization, we make use of defined assignment for the `interaction_t` type. This creates a deep copy.

The hard interaction (incoming momenta) is linked to the structure function instance. In the isolated state, we either set pointers to both, or we create modified copies (`rearrange`) as effective structure-function chain and interaction, respectively.

Finally, we set up the `subevt` component that will be used for evaluating observables, collecting particles from the trace evaluator in the effective connected state. Their quantum numbers must be determined by following back source links and set explicitly, since they are already eliminated in that trace.

The `rearrange` parts are still commented out; they could become relevant for a NLO algorithm.

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: init => term_instance_init
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_init (term, process, i_term, real_finite)
    class(term_instance_t), intent(inout), target :: term
    type(process_t), intent(in), target:: process
    integer, intent(in) :: i_term
    logical, intent(in), optional :: real_finite
    class(prc_core_t), pointer :: core => null ()
    type(process_beam_config_t) :: beam_config
    type(interaction_t), pointer :: sf_chain_int
    type(interaction_t), pointer :: src_int
    type(quantum_numbers_mask_t), dimension(:), allocatable :: mask_in
    type(state_matrix_t), pointer :: state_matrix
    type(flavor_t), dimension(:), allocatable :: flv_int, flv_src, f_in, f_out
    integer :: n_in, n_vir, n_out, n_tot, n_sub
    !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
    integer :: i, j, k
    type(quantum_numbers_t), dimension(:), allocatable :: qn
    logical :: me_already_squared, keep_fs_flavors
    logical :: decrease_n_tot

    me_already_squared = .false.
    keep_fs_flavors = .false.

    term%config => process%get_term_ptr (i_term)
    term%int_hard = term%config%int
    core => process%get_core_term (i_term)
```

```
call core%allocate_workspace (term%core_state)
select type (core)
class is (prc_user_defined_base_t)
   call reduce_interaction (term%int_hard, &
        core%includes_polarization (), .true., .false.)
   me_already_squared = .true.
   allocate (term%amp (term%int_hard%get_n_matrix_elements ()))
class default
   allocate (term%amp (term%config%n_allowed))
end select
if (allocated (term%core_state)) then
   select type (core_state => term%core_state)
   type is (openloops_state_t)
      call core_state%init_threshold (process%get_model_ptr ())
   end select
end if
term%amp = cmplx (0, 0, default)
decrease_n_tot = term%nlo_type == NLO_REAL .and. &
     term%config%i_term_global /= term%config%i_sub
if (present (real_finite)) then
   if (real_finite) decrease_n_tot = .false.
end if
if (decrease_n_tot) then
   allocate (term%p_seed (term%int_hard%get_n_tot () - 1))
else
   allocate (term%p_seed (term%int_hard%get_n_tot ()))
end if
allocate (term%p_hard (term%int_hard%get_n_tot ()))
sf_chain_int => term%k_term%sf_chain%get_out_int_ptr ()
n_in = term%int_hard%get_n_in ()
do j = 1, n_in
   i = term%k_term%sf_chain%get_out_i (j)
   call term%int_hard%set_source_link (j, sf_chain_int, i)
end do
call term%isolated%init (term%k_term%sf_chain, term%int_hard)
allocate (mask_in (n_in))
mask_in = term%k_term%sf_chain%get_out_mask ()
select type (phs => term%k_term%phs)
  type is (phs_wood_t)
     if (me_already_squared) then
        call term%isolated%setup_identity_trace (core, mask_in, .true., .false.)
     else
        call term%isolated%setup_square_trace (core, mask_in, term%config%col, .false.)
     end if
  type is (phs_fks_t)
     select case (phs%mode)
     case (PHS_MODE_ADDITIONAL_PARTICLE)
        if (me_already_squared) then
           call term%isolated%setup_identity_trace (core, mask_in, .true., .false.)
        else
           keep_fs_flavors = term%config%data%n_flv > 1
           call term%isolated%setup_square_trace (core, mask_in, term%config%col, &
                keep_fs_flavors)
        end if
```

```
     case (PHS_MODE_COLLINEAR_REMNANT)
        if (me_already_squared) then
           call term%isolated%setup_identity_trace (core, mask_in)
        else
           call term%isolated%setup_square_trace (core, mask_in, term%config%col, .false.)
        end if
     end select
  class default
     call term%isolated%setup_square_trace (core, mask_in, term%config%col, .false.)
end select
if (term%nlo_type == NLO_VIRTUAL) then
   select type (pcm => process%get_pcm_ptr ())
   type is (pcm_nlo_t)
      n_sub = pcm%get_n_sub ()
   end select
else
   !!! No integration of real subtraction in interactions yet
   n_sub = 0
end if
keep_fs_flavors = keep_fs_flavors .or. me_already_squared
call term%connected%setup_connected_trace (term%isolated, &
     undo_helicities = undo_helicities (core, me_already_squared), &
     n_sub = n_sub, keep_fs_flavors = keep_fs_flavors)

associate (int_eff => term%isolated%int_eff)
  state_matrix => int_eff%get_state_matrix_ptr ()
  n_tot = int_eff%get_n_tot  ()
  !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
  allocate (flv_int (n_tot), qn (n_tot))
  qn = state_matrix%get_quantum_number (1)
  do k = 1, n_tot
     flv_int(k) = quantum_numbers_get_flavor (qn (k))
  end do
  deallocate (qn)
  !!! flv_int = quantum_numbers_get_flavor &
  !!!      (state_matrix%get_quantum_number (1))
  allocate (f_in (n_in))
  f_in = flv_int(1:n_in)
  deallocate (flv_int)
end associate
n_in = term%connected%trace%get_n_in ()
n_vir = term%connected%trace%get_n_vir ()
n_out = term%connected%trace%get_n_out ()
allocate (f_out (n_out))
do j = 1, n_out
   call term%connected%trace%find_source &
        (n_in + n_vir + j, src_int, i)
   if (associated (src_int)) then
      state_matrix => src_int%get_state_matrix_ptr ()
      !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
      n_tot = src_int%get_n_tot ()
      allocate (flv_src (n_tot), qn (n_tot))
      qn = state_matrix%get_quantum_number (1)
      do k = 1, n_tot
```

```
          flv_src(k) = quantum_numbers_get_flavor (qn (k))
       end do
       deallocate (qn)
       !!! flv_src = quantum_numbers_get_flavor &
       !!!      (state_matrix%get_quantum_number (1))
       f_out(j) = flv_src(i)
       deallocate (flv_src)
    end if
  end do

  beam_config = process%get_beam_config ()

  call term%connected%setup_subevt (term%isolated%sf_chain_eff, &
       beam_config%data%flv, f_in, f_out)
  call term%connected%setup_var_list &
       (process%get_var_list_ptr (), beam_config%data)

contains

 function undo_helicities (core, me_squared) result (val)
   logical :: val
   class(prc_core_t), intent(in) :: core
   logical, intent(in) :: me_squared
   select type (core)
   class is (prc_user_defined_base_t)
      val = me_squared .and. .not. core%includes_polarization ()
   class default
      val = .false.
   end select
 end function undo_helicities

 subroutine reduce_interaction (int, polarized_beams, keep_fs_flavors, &
    keep_colors)
   type(interaction_t), intent(inout) :: int
   logical, intent(in) :: polarized_beams
   logical, intent(in) :: keep_fs_flavors, keep_colors
   type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
   logical, dimension(:), allocatable :: mask_f, mask_c, mask_h
   integer :: n_tot, n_in
   n_in = int%get_n_in (); n_tot = int%get_n_tot ()
   allocate (qn_mask (n_tot))
   allocate (mask_f (n_tot), mask_c (n_tot), mask_h (n_tot))
   mask_c = .not. keep_colors
   mask_f (1 : n_in) = .false.
   if (keep_fs_flavors) then
      mask_f (n_in + 1 : ) = .false.
   else
      mask_f (n_in + 1 : ) = .true.
   end if
   if (polarized_beams) then
      mask_h (1 : n_in) = .false.
   else
      mask_h (1 : n_in) = .true.
   end if
```

```
      mask_h (n_in + 1 : ) = .true.
      call qn_mask%init (mask_f, mask_c, mask_h)
      call int%reduce_state_matrix (qn_mask)
    end subroutine

  end subroutine term_instance_init
```

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: init_from_process => term_instance_init_from_process
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_init_from_process (term_instance, &
         process, i, pcm_instance, sf_chain)
    class(term_instance_t), intent(inout), target :: term_instance
    type(process_t), intent(in), target :: process
    integer, intent(in) :: i
    class(pcm_instance_t), intent(in), target :: pcm_instance
    type(sf_chain_t), intent(in), target :: sf_chain
    type(process_term_t) :: term
    integer :: i_component
    term = process%get_term_ptr (i)
    i_component = term%i_component
    if (i_component /= 0) then
       select type (pcm_instance)
       type is (pcm_instance_nlo_t)
          term_instance%pcm_instance => pcm_instance
       end select
       term_instance%nlo_type = process%get_nlo_type_component (i_component)
       call term_instance%setup_kinematics (sf_chain, &
            process%get_beam_config (), process%get_phs_config (i_component))
       call term_instance%init (process, i, &
            real_finite = process%component_is_real_finite (i_component))
       select type (phs => term_instance%k_term%phs)
       type is (phs_fks_t)
          call term_instance%set_emitter (process%get_pcm_ptr ())
          call term_instance%setup_fks_kinematics (process%get_var_list_ptr ())
       end select
       call term_instance%set_threshold (process%get_pcm_ptr ())
       call term_instance%setup_expressions (process%get_meta (), process%get_config ())
    end if
  end subroutine term_instance_init_from_process
```

Initialize the seed-kinematics configuration. All subobjects are allocated explicitly.

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: setup_kinematics => term_instance_setup_kinematics
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_setup_kinematics (term, sf_chain, &
     beam_config, phs_config)
    class(term_instance_t), intent(inout) :: term
    type(sf_chain_t), intent(in), target :: sf_chain
    type(process_beam_config_t), intent(in) :: beam_config
    class(phs_config_t), intent(in), target :: phs_config
```

```
      call term%k_term%init_sf_chain (sf_chain, beam_config)
      call term%k_term%init_phs (phs_config)
      call term%k_term%set_nlo_info (term%nlo_type)
      select type (phs => term%k_term%phs)
      type is (phs_fks_t)
         call phs%allocate_momenta (phs_config, &
              .not. (term%nlo_type == NLO_REAL))
         select type (config => term%pcm_instance%config)
         type is (pcm_nlo_t)
            call config%region_data%init_phs_identifiers (phs%phs_identifiers)
            call config%region_data%set_alr_to_i_phs (phs%phs_identifiers, &
                 term%pcm_instance%real_kinematics%alr_to_i_phs)
         end select
      end select
    end subroutine term_instance_setup_kinematics
```

⟨*Instances: term instance: TBP*⟩+≡

```
    procedure :: setup_fks_kinematics => term_instance_setup_fks_kinematics
```

⟨*Instances: procedures*⟩+≡

```
    subroutine term_instance_setup_fks_kinematics (term, var_list)
      class(term_instance_t), intent(inout), target :: term
      type(var_list_t), intent(in) :: var_list
      integer :: mode
      logical :: singular_jacobian
      if (.not. (term%nlo_type == NLO_REAL .or. term%nlo_type == NLO_DGLAP .or. &
         term%nlo_type == NLO_MISMATCH)) return
      singular_jacobian = var_list%get_lval (var_str ("?powheg_use_singular_jacobian"))
      if (term%nlo_type == NLO_REAL) then
         mode = check_generator_mode (GEN_REAL_PHASE_SPACE)
      else if (term%nlo_type == NLO_MISMATCH) then
         mode = check_generator_mode (GEN_SOFT_MISMATCH)
      else
         mode = PHS_MODE_UNDEFINED
      end if
      select type (phs => term%k_term%phs)
      type is (phs_fks_t)
         select type (config => term%pcm_instance%config)
         type is (pcm_nlo_t)
            call config%setup_phs_generator (term%pcm_instance, &
                 phs%generator, phs%config%sqrts, mode, singular_jacobian)
         end select
      class default
         call msg_fatal ("Phase space should be an FKS phase space!")
      end select
    contains
      function check_generator_mode (gen_mode_default) result (gen_mode)
         integer :: gen_mode
         integer, intent(in) :: gen_mode_default
         select type (config => term%pcm_instance%config)
         type is (pcm_nlo_t)
            associate (settings => config%settings)
               if (settings%test_coll_limit .and. settings%test_anti_coll_limit) &
                  call msg_fatal ("You cannot check the collinear and anti-collinear limit "&
```

```
            &"at the same time!")
         if (settings%test_soft_limit .and. .not. settings%test_coll_limit) then
            gen_mode = GEN_SOFT_LIMIT_TEST
         else if (.not. settings%test_soft_limit .and. settings%test_coll_limit) then
            gen_mode = GEN_COLL_LIMIT_TEST
         else if (.not. settings%test_soft_limit .and. settings%test_anti_coll_limit) then
            gen_mode = GEN_ANTI_COLL_LIMIT_TEST
         else if (settings%test_soft_limit .and. settings%test_coll_limit) then
            gen_mode = GEN_SOFT_COLL_LIMIT_TEST
         else if (settings%test_soft_limit .and. settings%test_anti_coll_limit) then
            gen_mode = GEN_SOFT_ANTI_COLL_LIMIT_TEST
         else
            gen_mode = gen_mode_default
         end if
      end associate
   end select
  end function check_generator_mode
end subroutine term_instance_setup_fks_kinematics
```

Setup seed kinematics, starting from the MC parameter set given as argument. As a result, the `k_seed` kinematics object is evaluated (except for the structure-function matrix-element evaluation, which we postpone until we know the factorization scale), and we have a valid `p_seed` momentum array.

⟨*Instances: term instance: TBP*⟩+≡

```
procedure :: compute_seed_kinematics => term_instance_compute_seed_kinematics
```

⟨*Instances: procedures*⟩+≡

```
subroutine term_instance_compute_seed_kinematics &
     (term, mci_work, phs_channel, success)
  class(term_instance_t), intent(inout), target :: term
  type(mci_work_t), intent(in) :: mci_work
  integer, intent(in) :: phs_channel
  logical, intent(out) :: success
  call term%k_term%compute_selected_channel &
       (mci_work, phs_channel, term%p_seed, success)
end subroutine term_instance_compute_seed_kinematics
```

⟨*Instances: term instance: TBP*⟩+≡

```
procedure :: evaluate_radiation_kinematics => term_instance_evaluate_radiation_kinematics
```

⟨*Instances: procedures*⟩+≡

```
subroutine term_instance_evaluate_radiation_kinematics (term, mci_work)
  class(term_instance_t), intent(inout) :: term
  type(mci_work_t), intent(in) :: mci_work
  select type (phs => term%k_term%phs)
  type is (phs_fks_t)
     if (phs%mode == PHS_MODE_ADDITIONAL_PARTICLE) then
        select type (config => term%pcm_instance%config)
        type is (pcm_nlo_t)
           call term%k_term%evaluate_radiation_kinematics &
                  (mci_work%get_x_process (), config%region_data, term%nlo_type)
        end select
     end if
  end select
```

```
    end subroutine term_instance_evaluate_radiation_kinematics
```

⟨*Instances: term instance: TBP*⟩+≡

```
    procedure :: evaluate_projections => term_instance_evaluate_projections
```

⟨*Instances: procedures*⟩+≡

```
    subroutine term_instance_evaluate_projections (term)
      class(term_instance_t), intent(inout) :: term
      real(default) :: sqrts, mtop
      type(lorentz_transformation_t) :: L_to_cms
      if (term%k_term%threshold .and. term%nlo_type > BORN) then
         if (debug2_active (D_THRESHOLD)) &
              print *, 'Evaluate on-shell projection: ', char (component_status (term%nlo_type))
         call term%k_term%threshold_projection (term%pcm_instance, term%nlo_type)
      end if
    end subroutine term_instance_evaluate_projections
```

⟨*Instances: term instance: TBP*⟩+≡

```
    procedure :: redo_sf_chain => term_instance_redo_sf_chain
```

⟨*Instances: procedures*⟩+≡

```
    subroutine term_instance_redo_sf_chain (term, mci_work, phs_channel)
      class(term_instance_t), intent(inout) :: term
      type(mci_work_t), intent(in) :: mci_work
      integer, intent(in) :: phs_channel
      integer :: sf_channel
      associate (k => term%k_term)
         sf_channel = k%phs%config%get_sf_channel (phs_channel)
         call k%sf_chain%compute_kinematics (sf_channel, mci_work%get_x_strfun ())
      end associate
    end subroutine term_instance_redo_sf_chain
```

Inverse: recover missing parts of the kinematics, given a complete set of seed momenta. Select a channel and reconstruct the MC parameter set.

⟨*Instances: term instance: TBP*⟩+≡

```
    procedure :: recover_mcpar => term_instance_recover_mcpar
```

⟨*Instances: procedures*⟩+≡

```
    subroutine term_instance_recover_mcpar (term, mci_work, phs_channel)
      class(term_instance_t), intent(inout), target :: term
      type(mci_work_t), intent(inout) :: mci_work
      integer, intent(in) :: phs_channel
      call term%k_term%recover_mcpar (mci_work, phs_channel, term%p_seed)
    end subroutine term_instance_recover_mcpar
```

Compute the momenta in the hard interactions, one for each term that constitutes this process component. In simple cases this amounts to just copying momenta. In more advanced cases, we may generate distinct sets of momenta from the seed kinematics.

The interactions in the term instances are accessed individually. We may choose to calculate all terms at once together with the seed kinematics, use `component%core_state` for storage, and just fill the interactions here.

⟨*Instances: term instance: TBP*⟩+≡

```
procedure :: compute_hard_kinematics => &
     term_instance_compute_hard_kinematics
```

⟨*Instances: procedures*⟩+≡

```
subroutine term_instance_compute_hard_kinematics (term, skip_term, success)
  class(term_instance_t), intent(inout) :: term
  integer, intent(in), optional :: skip_term
  type(vector4_t), dimension(:), allocatable :: p
  logical, intent(out) :: success
  if (allocated (term%core_state)) &
     call term%core_state%reset_new_kinematics ()
  if (present (skip_term)) then
     if (term%config%i_term_global == skip_term) return
  end if

  if (term%nlo_type == NLO_REAL .and. term%k_term%emitter >= 0) then
     call term%k_term%evaluate_radiation (term%p_seed, p, success)
     select type (config => term%pcm_instance%config)
     type is (pcm_nlo_t)
        if (config%dalitz_plot%active) &
              call config%register_dalitz_plot (term%k_term%emitter, p)
     end select
  else if (is_subtraction_component (term%k_term%emitter, term%nlo_type)) then
     call term%k_term%modify_momenta_for_subtraction (term%p_seed, p)
     success = .true.
  else
     allocate (p (size (term%p_seed))); p = term%p_seed
     success = .true.
  end if
  call term%int_hard%set_momenta (p)
end subroutine term_instance_compute_hard_kinematics
```

Here, we invert this. We fetch the incoming momenta which reside in the appropriate `sf_chain` object, stored within the `k_seed` subobject. On the other hand, we have the outgoing momenta of the effective interaction. We rely on the process core to compute the remaining seed momenta and to fill the momenta within the hard interaction. (The latter is trivial if hard and effective interaction coincide.)

After this is done, the incoming momenta in the trace evaluator that corresponds to the hard (effective) interaction, are still left undefined. We remedy this by calling `receive_kinematics` once.

⟨*Instances: term instance: TBP*⟩+≡

```
procedure :: recover_seed_kinematics => &
     term_instance_recover_seed_kinematics
```

⟨*Instances: procedures*⟩+≡

```
subroutine term_instance_recover_seed_kinematics (term)
  class(term_instance_t), intent(inout) :: term
  integer :: n_in
  n_in = term%k_term%n_in
  call term%k_term%get_incoming_momenta (term%p_seed(1:n_in))
  associate (int_eff => term%isolated%int_eff)
     call int_eff%set_momenta (term%p_seed(1:n_in), outgoing = .false.)
     term%p_seed(n_in + 1 : ) = int_eff%get_momenta (outgoing = .true.)
```

```
    end associate
    call term%isolated%receive_kinematics ()
  end subroutine term_instance_recover_seed_kinematics
```

Compute the integration parameters for all channels except the selected one.

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: compute_other_channels => &
      term_instance_compute_other_channels
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_compute_other_channels &
      (term, mci_work, phs_channel)
    class(term_instance_t), intent(inout), target :: term
    type(mci_work_t), intent(in) :: mci_work
    integer, intent(in) :: phs_channel
    call term%k_term%compute_other_channels (mci_work, phs_channel)
  end subroutine term_instance_compute_other_channels
```

Recover beam momenta, i.e., return the beam momenta as currently stored in the kinematics subobject to their source. This is a side effect.

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: return_beam_momenta => term_instance_return_beam_momenta
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_return_beam_momenta (term)
    class(term_instance_t), intent(in) :: term
    call term%k_term%return_beam_momenta ()
  end subroutine term_instance_return_beam_momenta
```

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: apply_real_partition => term_instance_apply_real_partition
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_apply_real_partition (term, process)
    class(term_instance_t), intent(inout) :: term
    type(process_t), intent(in) :: process
    real(default) :: f, sqme
    integer :: i_component, nlegs
    integer :: i_amp, n_amps
    logical :: is_subtraction
    i_component = term%config%i_component
    if (process%component_is_selected (i_component) .and. &
          process%get_component_nlo_type (i_component) == NLO_REAL) then
       is_subtraction = process%get_component_type (i_component) == COMP_REAL_SING &
            .and. term%k_term%emitter < 0
       if (is_subtraction) return
       select type (pcm => process%get_pcm_ptr ())
       type is (pcm_nlo_t)
          f = pcm%real_partition%get_f (term%p_hard)
       end select
       n_amps = term%connected%trace%get_n_matrix_elements ()
       do i_amp = 1, n_amps
          sqme = real (term%connected%trace%get_matrix_element (i_amp))
          call msg_debug2 (D_PROCESS_INTEGRATION, "term_instance_apply_real_partition")
```

```
            select type (pcm => term%pcm_instance%config)
            type is (pcm_nlo_t)
               select case (process%get_component_type (i_component))
               case (COMP_REAL_FIN, COMP_REAL_SING)
                  select case (process%get_component_type (i_component))
                  case (COMP_REAL_FIN)
                     call msg_debug2 (D_PROCESS_INTEGRATION, "Real finite")
                     sqme = sqme * (one - f)
                  case (COMP_REAL_SING)
                     call msg_debug2 (D_PROCESS_INTEGRATION, "Real singular")
                     sqme = sqme * f
                  end select
               end select
            end select
            call msg_debug2 (D_PROCESS_INTEGRATION, "apply_damping: sqme", sqme)
            call term%connected%trace%set_matrix_element (i_amp, cmplx (sqme, zero, default))
         end do
      end if
   end subroutine term_instance_apply_real_partition
```

⟨*Instances: term instance: TBP*⟩+≡

```
   procedure :: get_lorentz_transformation => term_instance_get_lorentz_transformation
```

⟨*Instances: procedures*⟩+≡

```
   function term_instance_get_lorentz_transformation (term) result (lt)
      type(lorentz_transformation_t) :: lt
      class(term_instance_t), intent(in) :: term
      lt = term%k_term%phs%get_lorentz_transformation ()
   end function term_instance_get_lorentz_transformation
```

⟨*Instances: term instance: TBP*⟩+≡

```
   procedure :: set_emitter => term_instance_set_emitter
```

⟨*Instances: procedures*⟩+≡

```
   subroutine term_instance_set_emitter (term, pcm)
      class(term_instance_t), intent(inout) :: term
      class(pcm_t), intent(in) :: pcm
      integer :: i_phs
      logical :: set_emitter
      select type (pcm)
      type is (pcm_nlo_t)
         !!! Without resonances, i_alr = i_phs
         i_phs = term%config%i_term
         term%k_term%i_phs = term%config%i_term
         select type (phs => term%k_term%phs)
         type is (phs_fks_t)
            set_emitter = i_phs <= pcm%region_data%n_phs .and. term%nlo_type == NLO_REAL
            if (set_emitter) then
               term%k_term%emitter = phs%phs_identifiers(i_phs)%emitter
               select type (pcm => term%pcm_instance%config)
               type is (pcm_nlo_t)
                  if (allocated (pcm%region_data%i_phs_to_i_con)) &
                     term%k_term%i_con = pcm%region_data%i_phs_to_i_con (i_phs)
               end select
```

```
        end if
      end select
    end select
  end subroutine term_instance_set_emitter
```

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: set_threshold => term_instance_set_threshold
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_set_threshold (term, pcm)
    class(term_instance_t), intent(inout) :: term
    class(pcm_t), intent(in) :: pcm
    select type (pcm)
    type is (pcm_nlo_t)
      term%k_term%threshold = pcm%settings%factorization_mode == FACTORIZATION_THRESHOLD
    class default
      term%k_term%threshold = .false.
    end select
  end subroutine term_instance_set_threshold
```

For initializing the expressions, we need the local variable list and the parse trees.

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: setup_expressions => term_instance_setup_expressions
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_setup_expressions (term, meta, config)
    class(term_instance_t), intent(inout), target :: term
    type(process_metadata_t), intent(in), target :: meta
    type(process_config_data_t), intent(in) :: config
    if (allocated (config%ef_cuts)) &
         call term%connected%setup_cuts (config%ef_cuts)
    if (allocated (config%ef_scale)) &
         call term%connected%setup_scale (config%ef_scale)
    if (allocated (config%ef_fac_scale)) &
         call term%connected%setup_fac_scale (config%ef_fac_scale)
    if (allocated (config%ef_ren_scale)) &
         call term%connected%setup_ren_scale (config%ef_ren_scale)
    if (allocated (config%ef_weight)) &
         call term%connected%setup_weight (config%ef_weight)
  end subroutine term_instance_setup_expressions
```

Prepare the extra evaluators that we need for processing events.

The quantum numbers mask of the incoming particle

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: setup_event_data => term_instance_setup_event_data
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_setup_event_data (term, core, model)
    class(term_instance_t), intent(inout), target :: term
    class(prc_core_t), intent(in) :: core
    class(model_data_t), intent(in), target :: model
    integer :: n_in
```

```
      type(quantum_numbers_mask_t), dimension(:), allocatable :: mask_in
      n_in = term%int_hard%get_n_in ()
      allocate (mask_in (n_in))
      mask_in = term%k_term%sf_chain%get_out_mask ()
      call setup_isolated (term%isolated, core, model, mask_in, term%config%col)
      call setup_connected (term%connected, term%isolated, term%nlo_type)
   contains
     subroutine setup_isolated (isolated, core, model, mask, color)
       type(isolated_state_t), intent(inout), target :: isolated
       class(prc_core_t), intent(in) :: core
       class(model_data_t), intent(in), target :: model
       type(quantum_numbers_mask_t), intent(in), dimension(:) :: mask
       integer, intent(in), dimension(:) :: color
       call isolated%setup_square_matrix (core, model, mask, color)
       call isolated%setup_square_flows (core, model, mask)
     end subroutine setup_isolated

     subroutine setup_connected (connected, isolated, nlo_type)
       type(connected_state_t), intent(inout), target :: connected
       type(isolated_state_t), intent(in), target :: isolated
       integer :: nlo_type
       type(quantum_numbers_mask_t), dimension(:), allocatable :: mask
       call connected%setup_connected_matrix (isolated)
       if (term%nlo_type == NLO_VIRTUAL) then
          !!! We don't care about the subtraction matrix elements in
          !!! connected%matrix, because all entries there are supposed
          !!! to be squared. To be able to match with flavor quantum numbers,
          !!! we remove the subtraction quantum entries from the state matrix.
          allocate (mask (connected%matrix%get_n_tot()))
          call mask%set_sub (1)
          call connected%matrix%reduce_state_matrix (mask)
       end if
       call connected%setup_connected_flows (isolated)
       call connected%setup_state_flv (isolated%get_n_out ())
     end subroutine setup_connected
   end subroutine term_instance_setup_event_data
```

⟨*Instances: term instance: TBP*⟩+≡

```
    procedure :: evaluate_color_correlations => &
       term_instance_evaluate_color_correlations
```

⟨*Instances: procedures*⟩+≡

```
    subroutine term_instance_evaluate_color_correlations (term, p, core, bad_point)
      class(term_instance_t), intent(inout) :: term
      type(vector4_t), intent(in), dimension(:) :: p
      class(prc_core_t), intent(inout) :: core
      logical, intent(out), optional :: bad_point
      integer :: i_flv_born
      logical :: bp
      real(default), dimension(:,:), allocatable :: sqme_cc
      real(default) :: sqme_born
      associate (pcm_instance => term%pcm_instance)
         select type (config => pcm_instance%config)
         type is (pcm_nlo_t)
```

```
            allocate (sqme_cc (config%region_data%n_legs_born, &
                 config%region_data%n_legs_born))
            call msg_debug2 (D_SUBTRACTION, &
                 "term_instance_evaluate_color_correlations: " // &
                 "use_internal_color_correlations:", &
                 config%settings%use_internal_color_correlations)
            call msg_debug2 (D_SUBTRACTION, "fac_scale", term%fac_scale)

            do i_flv_born = 1, config%region_data%n_flv_born
               if (config%settings%use_internal_color_correlations) then
                  select case (term%nlo_type)
                  case (NLO_REAL)
                     sqme_born = pcm_instance%real_sub%sqme_born (i_flv_born)
                     sqme_cc = sqme_born * config%color_data%beta_ij (:, :, i_flv_born)
                  case (NLO_MISMATCH)
                     sqme_born = pcm_instance%soft_mismatch%sqme_born (i_flv_born)
                     sqme_cc = sqme_born * config%color_data%beta_ij (:, :, i_flv_born)
                  end select
                  bp = .false.
               else
                  select type (core)
                  class is (prc_user_defined_base_t)
                     call core%update_alpha_s (term%core_state, term%fac_scale)
                     call core%compute_sqme_cc (i_flv_born, p, &
                          term%fac_scale, sqme_cc, bp, sqme_born)
                  end select
               end if

               select case (term%nlo_type)
               case (NLO_REAL)
                  pcm_instance%real_sub%sqme_born_cc (:, :, i_flv_born) = sqme_cc
               case (NLO_MISMATCH)
                  pcm_instance%soft_mismatch%sqme_born_cc (:, :, i_flv_born) = sqme_cc
               case (NLO_VIRTUAL)
                  if (config%settings%use_internal_color_correlations) then
                     pcm_instance%virtual%sqme_cc (:, :, i_flv_born) = &
                          config%color_data%beta_ij (:, :, i_flv_born)
                  else
                     pcm_instance%virtual%sqme_cc (:, :, i_flv_born) = sqme_cc
                  end if
               end select
            end do
         end select
      end associate
      if (present (bad_point)) bad_point = bp
   end subroutine term_instance_evaluate_color_correlations
```

The information about spin correlations is not stored in the `nlo_settings` because it is only available after the `fks_regions` have been created.

⟨*Instances: term instance: TBP*⟩+≡

```
   procedure :: evaluate_spin_correlations => term_instance_evaluate_spin_correlations
```

⟨*Instances: procedures*⟩+≡

```
   subroutine term_instance_evaluate_spin_correlations (term, p, core, bad_point)
```

```
    class(term_instance_t), intent(inout) :: term
    type(vector4_t), intent(in), dimension(:) :: p
    class(prc_core_t), intent(inout) :: core
    logical, intent(out), optional :: bad_point
    integer :: n_flv_born, i_flv_born, i_emitter, emitter
    real(default), dimension(0:3, 0:3) :: sqme_sc
    logical :: this_bp, bp
    bp = .false.
    associate (pcm_instance => term%pcm_instance)
       if (pcm_instance%real_sub%requires_spin_correlations () .and. &
            term%nlo_type == NLO_REAL) then
          n_flv_born = term%config%data%get_n_flv ()
          select type (core)
          type is (prc_openloops_t)
             call core%update_alpha_s (term%core_state, term%fac_scale)
             do i_flv_born = 1, n_flv_born
                select type (config => pcm_instance%config)
                type is (pcm_nlo_t)
                   do i_emitter = 1, config%region_data%n_emitters
                      emitter = config%region_data%emitters(i_emitter)
                      call core%compute_sqme_sc (i_flv_born, emitter, &
                           p, term%ren_scale, sqme_sc, this_bp)
                      bp = bp .or. this_bp
                      pcm_instance%real_sub%sqme_born_sc(:,:,emitter,i_flv_born) = sqme_sc
                   end do
                end select
             end do
          class default
             call msg_fatal ("Spin correlations so far only supported by OpenLoops.")
          end select
       end if
    end associate
    if (present (bad_point)) bad_point = bp
  end subroutine term_instance_evaluate_spin_correlations
```

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: apply_fks => term_instance_apply_fks
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_apply_fks (term, core, alpha_s_sub)
    class(term_instance_t), intent(inout) :: term
    class(prc_core_t), intent(inout) :: core
    real(default), intent(in) :: alpha_s_sub
    real(default), dimension(:), allocatable :: sqme
    integer :: i_phs, emitter
    logical :: is_subtraction
    associate (pcm_instance => term%pcm_instance)
       select type (config => pcm_instance%config)
       type is (pcm_nlo_t)
          if (term%connected%has_matrix) then
             allocate (sqme (config%get_n_alr ()))
          else
             allocate (sqme (1))
          end if
```

```
            sqme = zero
            select type (phs => term%k_term%phs)
            type is (phs_fks_t)
               call pcm_instance%set_real_and_isr_kinematics &
                    (phs%phs_identifiers, term%k_term%phs%get_sqrts ())
               if (term%k_term%emitter < 0) then
                  call pcm_instance%set_subtraction_event ()
                  do i_phs = 1, config%region_data%n_phs
                     emitter = phs%phs_identifiers(i_phs)%emitter
                     call pcm_instance%real_sub%compute (emitter, &
                          i_phs, alpha_s_sub, term%connected%has_matrix, sqme)
                  end do
               else
                  call pcm_instance%set_radiation_event ()
                  emitter = term%k_term%emitter; i_phs = term%k_term%i_phs
                  pcm_instance%real_sub%sqme_real_non_sub (:, i_phs) = &
                       real (term%connected%trace%get_matrix_element ())
                  call pcm_instance%real_sub%compute (emitter, i_phs, alpha_s_sub, &
                       term%connected%has_matrix, sqme)
               end if
            end select
         end select
      end associate
      if (term%connected%has_trace) &
           call term%connected%trace%set_only_matrix_element &
                (1, cmplx (sum(sqme), 0, default))
      select type (config => term%pcm_instance%config)
      type is (pcm_nlo_t)
         is_subtraction = term%k_term%emitter < 0
         if (term%connected%has_matrix) &
             call refill_evaluator (cmplx (sqme, 0, default), &
                  config%get_qn (is_subtraction), &
                  config%region_data%get_flavor_indices (is_subtraction), &
                  term%connected%matrix)
         if (term%connected%has_flows) &
              call refill_evaluator (cmplx (sqme, 0, default), &
                   config%get_qn (is_subtraction), &
                   config%region_data%get_flavor_indices (is_subtraction), &
                   term%connected%flows)
      end select
   end subroutine term_instance_apply_fks
```

⟨*Instances: term instance: TBP*⟩+≡

```
   procedure :: evaluate_sqme_virt => term_instance_evaluate_sqme_virt
```

⟨*Instances: procedures*⟩+≡

```
   subroutine term_instance_evaluate_sqme_virt (term, alpha_s)
      class(term_instance_t), intent(inout) :: term
      real(default), intent(in) :: alpha_s
      type(vector4_t), dimension(:), allocatable :: p_born
      real(default), dimension(:), allocatable :: sqme_virt
      if (term%nlo_type /= NLO_VIRTUAL) call msg_fatal &
         ("Trying to evaluate virtual matrix element with unsuited term_instance.")
      if (debug2_active (D_VIRTUAL)) then
```

```
      call msg_debug2 (D_VIRTUAL, "Evaluating virtual-subtracted matrix elements")
      print *, 'alpha_s: ', alpha_s
      print *, 'ren_scale: ', term%ren_scale
      print *, 'fac_scale: ', term%fac_scale
   end if

   select type (config => term%pcm_instance%config)
   type is (pcm_nlo_t)
      allocate (p_born (config%region_data%n_legs_born))
      if (config%settings%factorization_mode == FACTORIZATION_THRESHOLD) then
         p_born = term%pcm_instance%real_kinematics%p_born_onshell%get_momenta(1)
      else
         p_born = term%int_hard%get_momenta ()
      end if
   end select

   call term%pcm_instance%set_momenta_and_scales_virtual &
        (p_born, term%ren_scale, term%fac_scale)
   call term%pcm_instance%compute_sqme_virt (term%p_hard, alpha_s, &
        term%connected%trace%get_matrix_element (), &
        term%connected%has_matrix, sqme_virt)
   call term%connected%trace%set_only_matrix_element &
        (1, cmplx (sum(sqme_virt) * term%weight, 0, default))
   if (term%connected%has_matrix) then
      select type (config => term%pcm_instance%config)
      type is (pcm_nlo_t)
         call refill_evaluator (cmplx (sqme_virt * term%weight, 0, default), &
              config%get_qn (.true.), config%region_data%get_flavor_indices (.true.), &
              term%connected%matrix)
      end select
   end if
end subroutine term_instance_evaluate_sqme_virt
```

⟨*Instances: term instance: TBP*⟩+≡

```
procedure :: evaluate_sqme_mismatch => term_instance_evaluate_sqme_mismatch
```

⟨*Instances: procedures*⟩+≡

```
subroutine term_instance_evaluate_sqme_mismatch (term, alpha_s)
  class(term_instance_t), intent(inout) :: term
  real(default), intent(in) :: alpha_s
  real(default), dimension(:), allocatable :: sqme_mism
  if (term%nlo_type /= NLO_MISMATCH) call msg_fatal &
     ("Trying to evaluate soft mismatch with unsuited term_instance.")
  call term%pcm_instance%compute_sqme_mismatch &
       (alpha_s, term%connected%has_matrix, sqme_mism)
  call term%connected%trace%set_matrix_element &
     (cmplx (sqme_mism * term%weight, zero, default))
  if (term%connected%has_matrix) then
     select type (config => term%pcm_instance%config)
     type is (pcm_nlo_t)
        call refill_evaluator (cmplx (sqme_mism * term%weight, 0, default), &
             config%get_qn (.true.), config%region_data%get_flavor_indices (.true.), &
             term%connected%matrix)
     end select
```

```
    end if
  end subroutine term_instance_evaluate_sqme_mismatch
```

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: evaluate_sqme_dglap => term_instance_evaluate_sqme_dglap
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_evaluate_sqme_dglap (term, alpha_s)
    class(term_instance_t), intent(inout) :: term
    real(default), intent(in) :: alpha_s
    real(default), dimension(:), allocatable :: sqme_dglap
    if (term%nlo_type /= NLO_DGLAP) call msg_fatal &
       ("Trying to evaluate DGLAP remnant with unsuited term_instance.")
    call term%pcm_instance%compute_sqme_dglap_remnant (alpha_s, &
       real (term%connected%trace%get_matrix_element ()), &
       term%connected%has_matrix, sqme_dglap)
    call term%connected%trace%set_matrix_element &
         (cmplx (sqme_dglap * term%weight, 0, default))
    if (term%connected%has_matrix) then
       select type (config => term%pcm_instance%config)
       type is (pcm_nlo_t)
          call refill_evaluator (cmplx (sqme_dglap * term%weight, 0, default), &
               config%get_qn (.true.), config%region_data%get_flavor_indices (.true.), &
               term%connected%matrix)
       end select
    end if
  end subroutine term_instance_evaluate_sqme_dglap
```

Reset the term instance: clear the parton-state expressions and deactivate.

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: reset => term_instance_reset
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_reset (term)
    class(term_instance_t), intent(inout) :: term
    call term%connected%reset_expressions ()
    if (allocated (term%alpha_qcd_forced))  deallocate (term%alpha_qcd_forced)
    term%active = .false.
  end subroutine term_instance_reset
```

Force an α_s value that should be used in the matrix-element calculation.

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: set_alpha_qcd_forced => term_instance_set_alpha_qcd_forced
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_set_alpha_qcd_forced (term, alpha_qcd)
    class(term_instance_t), intent(inout) :: term
    real(default), intent(in) :: alpha_qcd
    if (allocated (term%alpha_qcd_forced)) then
       term%alpha_qcd_forced = alpha_qcd
    else
       allocate (term%alpha_qcd_forced, source = alpha_qcd)
    end if
  end subroutine term_instance_set_alpha_qcd_forced
```

Complete the kinematics computation for the effective parton states.

We assume that the `compute_hard_kinematics` method of the process component instance has already been called, so the `int_hard` contains the correct hard kinematics. The duty of this procedure is first to compute the effective kinematics and store this in the `int_eff` effective interaction inside the `isolated` parton state. The effective kinematics may differ from the kinematics in the hard interaction. It may involve parton recombination or parton splitting. The `rearrange_partons` method is responsible for this part.

We may also call a method to compute the effective structure-function chain at this point. This is not implemented yet.

In the simple case that no rearrangement is necessary, as indicated by the `rearrange` flag, the effective interaction is a pointer to the hard interaction, and we can skip the rearrangement method. Similarly for the effective structure-function chain. (If we have an algorithm that uses rarrangement, it should evaluate `k_term` explicitly.)

The final step of kinematics setup is to transfer the effective kinematics to the evaluators and to the `subevt`.

⟨*Instances: term instance: TBP*⟩+≡
```
procedure :: compute_eff_kinematics => &
     term_instance_compute_eff_kinematics
```

⟨*Instances: procedures*⟩+≡
```
subroutine term_instance_compute_eff_kinematics (term)
  class(term_instance_t), intent(inout) :: term
  term%checked = .false.
  term%passed = .false.
  call term%isolated%receive_kinematics ()
  call term%connected%receive_kinematics ()
end subroutine term_instance_compute_eff_kinematics
```

Inverse. Reconstruct the connected state from the momenta in the trace evaluator (which we assume to be set), then reconstruct the isolated state as far as possible. The second part finalizes the momentum configuration, using the incoming seed momenta

⟨*Instances: term instance: TBP*⟩+≡
```
procedure :: recover_hard_kinematics => &
     term_instance_recover_hard_kinematics
```

⟨*Instances: procedures*⟩+≡
```
subroutine term_instance_recover_hard_kinematics (term)
  class(term_instance_t), intent(inout) :: term
  term%checked = .false.
  term%passed = .false.
  call term%connected%send_kinematics ()
  call term%isolated%send_kinematics ()
end subroutine term_instance_recover_hard_kinematics
```

Check the term whether it passes cuts and, if successful, evaluate scales and weights. The factorization scale is also given to the term kinematics, enabling structure-function evaluation.

⟨*Instances: term instance: TBP*⟩+≡
```
procedure :: evaluate_expressions => &
```

```
term_instance_evaluate_expressions
```

⟨*Instances: procedures*⟩+≡

```
subroutine term_instance_evaluate_expressions (term, scale_forced)
  class(term_instance_t), intent(inout) :: term
  real(default), intent(in), allocatable, optional :: scale_forced
  call term%connected%evaluate_expressions (term%passed, &
       term%scale, term%fac_scale, term%ren_scale, term%weight, &
       scale_forced, force_evaluation = .true.)
  term%checked = .true.
end subroutine term_instance_evaluate_expressions
```

Evaluate the trace: first evaluate the hard interaction, then the trace evaluator. We use the `evaluate_interaction` method of the process component which generated this term. The `subevt` and cut expressions are not yet filled.

The `component` argument is intent(inout) because the `compute_amplitude` method may modify the `core_state` workspace object.

⟨*Instances: term instance: TBP*⟩+≡

```
procedure :: evaluate_interaction => term_instance_evaluate_interaction
```

⟨*Instances: procedures*⟩+≡

```
subroutine term_instance_evaluate_interaction (term, core)
  class(term_instance_t), intent(inout) :: term
  class(prc_core_t), intent(in), pointer :: core

  term%p_hard = term%int_hard%get_momenta ()
  select type (core)
  class is (prc_user_defined_base_t)
     call term%evaluate_interaction_userdef (core)
  class default
     call term%evaluate_interaction_default (core)
  end select
  call term%int_hard%set_matrix_element (term%amp)
end subroutine term_instance_evaluate_interaction
```

⟨*Instances: term instance: TBP*⟩+≡

```
procedure :: evaluate_interaction_default &
   => term_instance_evaluate_interaction_default
```

⟨*Instances: procedures*⟩+≡

```
subroutine term_instance_evaluate_interaction_default (term, core)
  class(term_instance_t), intent(inout) :: term
  class(prc_core_t), intent(in) :: core
  integer :: i
  do i = 1, term%config%n_allowed
     term%amp(i) = core%compute_amplitude (term%config%i_term, term%p_hard, &
          term%config%flv(i), term%config%hel(i), term%config%col(i), &
          term%fac_scale, term%ren_scale, term%alpha_qcd_forced, &
          term%core_state)
  end do
  if (associated (term%pcm_instance)) &
       call term%pcm_instance%set_fac_scale (term%fac_scale)
end subroutine term_instance_evaluate_interaction_default
```

⟨*Instances: term instance: TBP*⟩+≡

```
procedure :: evaluate_interaction_userdef &
   => term_instance_evaluate_interaction_userdef
```

⟨*Instances: procedures*⟩+≡

```
subroutine term_instance_evaluate_interaction_userdef (term, core)
  class(term_instance_t), intent(inout) :: term
  class(prc_core_t), intent(inout) :: core

  select type (core_state => term%core_state)
  type is (openloops_state_t)
     select type (core)
     type is (prc_openloops_t)
        call core%compute_alpha_s (core_state, term%ren_scale)
        if (allocated (core_state%threshold_data)) &
             call evaluate_threshold_parameters (core_state, core, term%k_term%phs%get_sqrts ())
     end select
  class is (user_defined_state_t)
     select type (core)
     class is (prc_user_defined_base_t)
        call core%compute_alpha_s (core_state, term%ren_scale)
     end select
  end select
  call evaluate_threshold_interaction ()
  if (term%nlo_type == NLO_VIRTUAL) then
     call term%evaluate_interaction_userdef_loop (core)
  else
     call term%evaluate_interaction_userdef_tree (core)
  end if

contains
  subroutine evaluate_threshold_parameters (core_state, core, sqrts)
     type(openloops_state_t), intent(inout) :: core_state
     type(prc_openloops_t), intent(inout) :: core
     real(default), intent(in) :: sqrts
     real(default) :: mtop, wtop
     mtop = m1s_to_mpole (sqrts)
     wtop = core_state%threshold_data%compute_top_width &
            (mtop, core_state%alpha_qcd)
     call core%set_mass_and_width (6, mtop, wtop)
  end subroutine

  subroutine evaluate_threshold_interaction ()
     integer :: leg
     select type (core)
     type is (prc_threshold_t)
        if (term%nlo_type > BORN) then
           associate (pcm => term%pcm_instance)
              if (term%k_term%emitter >= 0) then
                 call core%set_offshell_momenta &
                      (pcm%real_kinematics%p_real_cms%get_momenta(term%config%i_term))
                 leg = thr_leg (term%k_term%emitter)
                 call core%set_leg (leg)
                 call core%set_onshell_momenta &
                      (pcm%real_kinematics%p_real_onshell(leg)%get_momenta(term%config%i_term))
```

```
            else
               call core%set_leg (0)
               call core%set_offshell_momenta &
                    (pcm%real_kinematics%p_born_cms%get_momenta(1))
            end if
         end associate
      else
         call core%set_leg (-1)
         call core%set_offshell_momenta (term%p_hard)
      end if
   end select
  end subroutine evaluate_threshold_interaction
end subroutine term_instance_evaluate_interaction_userdef
```

⟨*Instances: term instance: TBP*⟩+≡

```
procedure :: evaluate_interaction_userdef_tree &
   => term_instance_evaluate_interaction_userdef_tree
```

⟨*Instances: procedures*⟩+≡

```
subroutine term_instance_evaluate_interaction_userdef_tree (term, core)
  class(term_instance_t), intent(inout) :: term
  class(prc_core_t), intent(in) :: core
  real(default) :: sqme
  integer :: n_amps, i, i_born
  logical :: bad_point
  n_amps = term%int_hard%get_n_matrix_elements ()
  select type (core)
  class is (prc_user_defined_base_t)
     call core%update_alpha_s (term%core_state, term%fac_scale)
     do i = 1, n_amps
        call core%compute_sqme (i, term%p_hard, term%ren_scale, &
             sqme, bad_point)
        i_born = core%get_helicity_list_base (i)
        term%amp(i_born) = cmplx (sqme, 0, default)
     end do
  end select
end subroutine term_instance_evaluate_interaction_userdef_tree
```

⟨*Instances: term instance: TBP*⟩+≡

```
procedure :: evaluate_interaction_userdef_loop &
   => term_instance_evaluate_interaction_userdef_loop
```

⟨*Instances: procedures*⟩+≡

```
subroutine term_instance_evaluate_interaction_userdef_loop (term, core)
  class(term_instance_t), intent(inout) :: term
  class(prc_core_t), intent(in) :: core
  integer :: n_virtuals, n_hel, n_sub, n_flv
  integer :: i, i_born, i_virt, i_cc, i_sub
  logical :: bad_point
  real(default), dimension(4) :: sqme_virt
  real(default), dimension(:), allocatable :: sqme_cc
  allocate (sqme_cc (blha_result_array_size &
       (term%int_hard%get_n_tot (), BLHA_AMP_CC)))
```

```
      call term%compute_virt_me_array_sizes (core, n_hel, n_sub, n_flv, n_virtuals)
      do i = 1, n_virtuals
         select type (core)
         class is (prc_blha_t)
            i_virt = core%get_helicity_list (i)
         class default
            i_virt = 1
         end select
         i_born = i_virt + n_hel
         select type (core)
         class is (prc_user_defined_base_t)
            call core%compute_sqme_virt (i, term%p_hard, &
                 term%ren_scale, sqme_virt, bad_point)
            associate (bad_blha => term%pcm_instance%bad_blha_point (term%config%i_component))
               bad_blha = bad_blha .or. bad_point
            end associate
         end select
         term%amp(i_virt) = cmplx (sqme_virt(3), 0, default)
         select type (config => term%pcm_instance%config)
         type is (pcm_nlo_t)
            if (config%settings%use_internal_color_correlations) then
               term%amp(i_born) = cmplx (sqme_virt(4), 0, default)
            else
               select type (core)
               class is (prc_blha_t)
                  call core%compute_sqme_cc_raw (i, term%p_hard, term%ren_scale, &
                       sqme_cc, bad_point)
                  term%amp (i_born) = cmplx (sqme_virt(4), 0, default)
                  do i_sub = 1, n_sub - 1
                     i_cc = core%get_helicity_list (i + n_hel - 1)
                     i_cc = i_cc + (i_sub + 1) * n_hel
                     term%amp(i_cc) = cmplx (sqme_cc(i_sub), 0, default)
                  end do
               end select
            end if
         end select
      end do
   end subroutine term_instance_evaluate_interaction_userdef_loop
```

⟨*Instances: term instance: TBP*⟩+≡

```
   procedure :: compute_virt_me_array_sizes => &
       term_instance_compute_virt_me_array_sizes
```

⟨*Instances: procedures*⟩+≡

```
   subroutine term_instance_compute_virt_me_array_sizes &
          (term_instance, core, n_hel, n_sub, n_flv, n_virtuals)
      class(term_instance_t), intent(in) :: term_instance
      class(prc_core_t), intent(in) :: core
      integer, intent(out) :: n_hel, n_sub, n_flv, n_virtuals
      integer :: n_in
      logical :: includes_polarization
      n_sub = term_instance%int_hard%get_n_sub ()
      n_virtuals = term_instance%int_hard%get_n_matrix_elements ()
      n_flv = term_instance%config%data%n_flv
```

```
    n_in = term_instance%config%data%n_in
    select type (core)
    class is (prc_user_defined_base_t)
      includes_polarization = core%includes_polarization ()
    class default
      includes_polarization = .false.
    end select
    if (includes_polarization) then
       n_hel = term_instance%int_hard%get_n_in_helicities ()
    else
       n_hel = 1
    end if
    if (n_sub > 0) then
       n_virtuals = n_virtuals - n_hel * n_sub * n_flv
    end if
  end subroutine term_instance_compute_virt_me_array_sizes
```

Evaluate the trace. First evaluate the structure-function chain (i.e., the density matrix of the incoming partons). Do this twice, in case the sf-chain instances within `k_term` and `isolated` differ. Next, evaluate the hard interaction, then compute the convolution with the initial state.

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: evaluate_trace => term_instance_evaluate_trace
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_evaluate_trace (term)
    class(term_instance_t), intent(inout) :: term
    call term%k_term%evaluate_sf_chain (term%fac_scale)
    call term%isolated%evaluate_sf_chain (term%fac_scale)
    call term%isolated%evaluate_trace ()
    call term%connected%evaluate_trace ()
  end subroutine term_instance_evaluate_trace
```

Evaluate the extra data that we need for processing the object as a physical event.

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: evaluate_event_data => term_instance_evaluate_event_data
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_evaluate_event_data (term)
    class(term_instance_t), intent(inout) :: term
    logical :: only_momenta
    only_momenta = term%nlo_type > BORN
    call term%isolated%evaluate_event_data (only_momenta)
    call term%connected%evaluate_event_data (only_momenta)
  end subroutine term_instance_evaluate_event_data
```

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: set_fac_scale => term_instance_set_fac_scale
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_set_fac_scale (term, fac_scale)
    class(term_instance_t), intent(inout) :: term
```

```
    real(default), intent(in) :: fac_scale
    term%fac_scale = fac_scale
  end subroutine term_instance_set_fac_scale
```

Return data that might be useful for external processing. The factorization scale:

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: get_fac_scale => term_instance_get_fac_scale
```

⟨*Instances: procedures*⟩+≡

```
  function term_instance_get_fac_scale (term) result (fac_scale)
    class(term_instance_t), intent(in) :: term
    real(default) :: fac_scale
    fac_scale = term%fac_scale
  end function term_instance_get_fac_scale
```

We take the strong coupling from the process core. The value is calculated when a new event is requested, so we should call it only after the event has been evaluated. If it is not available there (a negative number is returned), we take the value stored in the term configuration, which should be determined by the model. If the model does not provide a value, the result is zero.

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: get_alpha_s => term_instance_get_alpha_s
```

⟨*Instances: procedures*⟩+≡

```
  function term_instance_get_alpha_s (term, core) result (alpha_s)
    class(term_instance_t), intent(in) :: term
    class(prc_core_t), intent(in) :: core
    real(default) :: alpha_s
    alpha_s = core%get_alpha_s (term%core_state)
    if (alpha_s < zero)  alpha_s = term%config%alpha_s
  end function term_instance_get_alpha_s
```

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: reset_phs_identifiers => term_instance_reset_phs_identifiers
```

⟨*Instances: procedures*⟩+≡

```
  subroutine term_instance_reset_phs_identifiers (term)
    class(term_instance_t), intent(inout) :: term
    select type (phs => term%k_term%phs)
    type is (phs_fks_t)
       phs%phs_identifiers%evaluated = .false.
    end select
  end subroutine term_instance_reset_phs_identifiers
```

The second helicity for `helicities` comes with a minus sign because OpenLoops inverts the helicity index of antiparticles.

⟨*Instances: term instance: TBP*⟩+≡

```
  procedure :: get_helicities_for_openloops => term_instance_get_helicities_for_openloops
```

⟨*Instances: procedures*⟩+≡

```
subroutine term_instance_get_helicities_for_openloops (term, helicities)
  class(term_instance_t), intent(in) :: term
  integer, dimension(:,:), allocatable, intent(out) :: helicities
  type(helicity_t), dimension(:), allocatable :: hel
  type(quantum_numbers_t), dimension(:,:), allocatable :: qn
  type(quantum_numbers_mask_t) :: qn_mask
  integer :: h, i, j, n_in
  call qn_mask%set_sub (1)
  call term%isolated%trace%get_quantum_numbers_mask (qn_mask, qn)
  n_in = term%int_hard%get_n_in ()
  allocate (helicities (size (qn, dim=1), n_in))
  allocate (hel (size (qn, dim=1)))
  do i = 1, size (qn, dim=1)
     do j = 1, n_in
        hel(j) = qn(i, j)%get_helicity ()
        call hel(j)%diagonalize ()
        call hel(j)%get_indices (h, h)
        helicities (i, j) = h
     end do
  end do
end subroutine term_instance_get_helicities_for_openloops
```

⟨*Instances: term instance: TBP*⟩+≡

```
procedure :: get_boost_to_lab => term_instance_get_boost_to_lab
```

⟨*Instances: procedures*⟩+≡

```
function term_instance_get_boost_to_lab (term) result (lt)
  type(lorentz_transformation_t) :: lt
  class(term_instance_t), intent(in) :: term
  lt = term%k_term%phs%get_lorentz_transformation ()
end function term_instance_get_boost_to_lab
```

⟨*Instances: term instance: TBP*⟩+≡

```
procedure :: get_boost_to_cms => term_instance_get_boost_to_cms
```

⟨*Instances: procedures*⟩+≡

```
function term_instance_get_boost_to_cms (term) result (lt)
  type(lorentz_transformation_t) :: lt
  class(term_instance_t), intent(in) :: term
  lt = inverse (term%k_term%phs%get_lorentz_transformation ())
end function term_instance_get_boost_to_cms
```

⟨*Instances: term instance: TBP*⟩+≡

```
procedure :: get_i_term_global => term_instance_get_i_term_global
```

⟨*Instances: procedures*⟩+≡

```
elemental function term_instance_get_i_term_global (term) result (i_term)
  integer :: i_term
  class(term_instance_t), intent(in) :: term
  i_term = term%config%i_term_global
end function term_instance_get_i_term_global
```

30.11.2 The process instance

A process instance contains all process data that depend on the sampling point and thus change often. In essence, it is an event record at the elementary (parton) level. We do not call it such, to avoid confusion with the actual event records. If decays are involved, the latter are compositions of several elementary processes (i.e., their instances).

We implement the process instance as an extension of the `mci_sampler_t` that we need for computing integrals and generate events.

The base type contains: the `integrand`, the `selected_channel`, the two-dimensional array `x` of parameters, and the one-dimensional array `f` of Jacobians. These subobjects are public and used for communicating with the multi-channel integrator.

The `process` pointer accesses the process of which this record is an instance. It is required whenever the calculation needs invariant configuration data, therefore the process should stay in memory for the whole lifetime of its instances.

The `evaluation_status` code is used to check the current status. In particular, failure at various stages is recorded there.

The `count` object records process evaluations, broken down according to status.

The `sqme` value is the single real number that results from evaluating and tracing the kinematics and matrix elements. This is the number that is handed over to an integration routine.

The `weight` value is the event weight. It is defined when an event has been generated from the process instance, either weighted or unweighted. The value is the `sqme` value times Jacobian weights from the integration, or unity, respectively.

The `i_mci` index chooses a subset of components that are associated with a common parameter set and integrator, i.e., that are added coherently.

The `sf_chain` subobject is a realization of the beam and structure-function configuration in the `process` object. It is not used for calculation directly but serves as the template for the sf-chain instances that are contained in the `component` objects.

The `component` subobjects determine the state of each component.

The `term` subobjects are workspace for evaluating kinematics, matrix elements, cuts etc.

The `mci_work` subobject contains the array of real input parameters (random numbers) that generates the kinematical point. It also contains the workspace for the MC integrators. The active entry of the `mci_work` array is selected by the `i_mci` index above.

⟨*Instances: public*⟩≡
```
public :: process_instance_t
```

⟨*Instances: types*⟩+≡
```
type, extends (mci_sampler_t) :: process_instance_t
   type(process_t), pointer :: process => null ()
   integer :: evaluation_status = STAT_UNDEFINED
   real(default) :: sqme = 0
   real(default) :: weight = 0
   real(default) :: excess = 0
   integer :: i_mci = 0
```

```
    integer :: selected_channel = 0
    type(sf_chain_t) :: sf_chain
    type(term_instance_t), dimension(:), allocatable :: term
    type(mci_work_t), dimension(:), allocatable :: mci_work
    class(pcm_instance_t), allocatable :: pcm
  contains
  ⟨Instances: process instance: TBP⟩
end type process_instance_t
```

The output routine contains a header with the most relevant information about the process, copied from `process_metadata_write`. We mark the active components by an asterisk.

The next section is the MC parameter input. The following sections are written only if the evaluation status is beyond setting the parameters, or if the `verbose` option is set.

⟨*Instances: process instance: TBP*⟩≡

```
procedure :: write_header => process_instance_write_header
procedure :: write => process_instance_write
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_write_header (object, unit, testflag)
  class(process_instance_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u
  u = given_output_unit (unit)
  call write_separator (u, 2)
  if (associated (object%process)) then
     call object%process%write_meta (u, testflag)
  else
     write (u, "(1x,A)") "Process instance [undefined process]"
     return
  end if
  write (u, "(3x,A)", advance = "no")  "status = "
  select case (object%evaluation_status)
  case (STAT_INITIAL);            write (u, "(A)")  "initialized"
  case (STAT_ACTIVATED);          write (u, "(A)")  "activated"
  case (STAT_BEAM_MOMENTA);       write (u, "(A)")  "beam momenta set"
  case (STAT_FAILED_KINEMATICS);  write (u, "(A)")  "failed kinematics"
  case (STAT_SEED_KINEMATICS);    write (u, "(A)")  "seed kinematics"
  case (STAT_HARD_KINEMATICS);    write (u, "(A)")  "hard kinematics"
  case (STAT_EFF_KINEMATICS);     write (u, "(A)")  "effective kinematics"
  case (STAT_FAILED_CUTS);        write (u, "(A)")  "failed cuts"
  case (STAT_PASSED_CUTS);        write (u, "(A)")  "passed cuts"
  case (STAT_EVALUATED_TRACE);    write (u, "(A)")  "evaluated trace"
     call write_separator (u)
     write (u, "(3x,A,ES19.12)")  "sqme   = ", object%sqme
  case (STAT_EVENT_COMPLETE);   write (u, "(A)")  "event complete"
     call write_separator (u)
     write (u, "(3x,A,ES19.12)")  "sqme   = ", object%sqme
     write (u, "(3x,A,ES19.12)")  "weight = ", object%weight
     if (.not. vanishes (object%excess)) &
          write (u, "(3x,A,ES19.12)")  "excess = ", object%excess
  case default;                 write (u, "(A)")  "undefined"
```

```
  end select
  if (object%i_mci /= 0) then
     call write_separator (u)
     call object%mci_work(object%i_mci)%write (u, testflag)
  end if
  call write_separator (u, 2)
end subroutine process_instance_write_header

subroutine process_instance_write (object, unit, testflag)
  class(process_instance_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag
  integer :: u, i
  u = given_output_unit (unit)
  call object%write_header (u)
  if (object%evaluation_status >= STAT_BEAM_MOMENTA) then
     call object%sf_chain%write (u)
     call write_separator (u, 2)
     if (object%evaluation_status >= STAT_SEED_KINEMATICS) then
        if (object%evaluation_status >= STAT_HARD_KINEMATICS) then
           call write_separator (u, 2)
           write (u, "(1x,A)") "Active terms:"
           if (any (object%term%active)) then
              do i = 1, size (object%term)
                 if (object%term(i)%active) then
                    call write_separator (u)
                    call object%term(i)%write (u, &
                         show_eff_state = &
                         object%evaluation_status >= STAT_EFF_KINEMATICS, &
                         testflag = testflag)
                 end if
              end do
           end if
        end if
        call write_separator (u, 2)
     end if
  end if
end subroutine process_instance_write
```

Initialization connects the instance with a process. All initial information is transferred from the process object. The process object contains templates for the interaction subobjects (beam and term), but no evaluators. The initialization routine creates evaluators for the matrix element trace, other evaluators are left untouched.

Before we start generating, we double-check if the process library has been updated after the process was initialized (`check_library_sanity`). This may happen if between integration and event generation the library has been recompiled, so all links become broken.

The `instance` object must have the `target` attribute (also in any caller) since the initialization routine assigns various pointers to subobject of `instance`.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: init => process_instance_init
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_init (instance, process)
  class(process_instance_t), intent(out), target :: instance
  type(process_t), intent(inout), target :: process
  integer :: i
  class(pcm_t), pointer :: pcm
  type(process_term_t) :: term
  integer :: i_born, i_real, i_real_fin

  instance%process => process

  call instance%process%check_library_sanity ()

  call instance%setup_sf_chain (process%get_beam_config_ptr ())

  allocate (instance%mci_work (process%get_n_mci ()))
  do i = 1, size (instance%mci_work)
     call instance%process%init_mci_work (instance%mci_work(i), i)
  end do

  call instance%process%reset_selected_cores ()

  pcm => instance%process%get_pcm_ptr ()
  call pcm%allocate_instance (instance%pcm)
  call instance%pcm%link_config (pcm)

  select type (pcm)
  type is (pcm_nlo_t)
     !!! The process is kept when the integration is finalized, but not the
     !!! process_instance. Thus, we check whether pcm has been initialized
     !!! but set up the pcm_instance each time.
     i_real_fin = process%get_associated_real_fin (1)
     if (.not. pcm%initialized) then
        i_born = process%get_i_core_nlo_type (BORN)
        i_real = process%get_i_core_nlo_type (NLO_REAL, include_sub = .false.)
        term = process%get_term_ptr (process%get_i_term (i_real))
        if (pcm%settings%use_internal_color_correlations) &
             call pcm%init_color_data &
                  ([process%get_constants (i_born), &
                  process%get_constants (i_real)], &
                  term%flv, term%col)
        call pcm%init_qn (process%get_model_ptr ())
        if (i_real_fin > 0) call pcm%allocate_ps_matching ()
        associate (var_list => process%get_var_list_ptr ())
           if (var_list%get_sval (var_str ("$dalitz_plot")) /= var_str ('')) &
                call pcm%activate_dalitz_plot (var_list%get_sval (var_str ("$dalitz_plot")))
        end associate
     end if
     pcm%initialized = .true.
     select type (pcm_instance => instance%pcm)
     type is (pcm_instance_nlo_t)
        call pcm_instance%init_config (process%component_is_active (), &
             process%get_nlo_type_component (), process%get_sqrts (), i_real_fin)
     end select
```

```
    end select

    allocate (instance%term (process%get_n_terms ()))
    do i = 1, process%get_n_terms ()
       call instance%term(i)%init_from_process (process, i, instance%pcm, &
            instance%sf_chain)
    end do
    call instance%find_same_kinematics ()

    instance%evaluation_status = STAT_INITIAL
  end subroutine process_instance_init
```

Finalize all subobjects that may contain allocated pointers.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: final => process_instance_final
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_final (instance)
    class(process_instance_t), intent(inout) :: instance
    integer :: i
    instance%process => null ()
    if (allocated (instance%mci_work)) then
       do i = 1, size (instance%mci_work)
          call instance%mci_work(i)%final ()
       end do
       deallocate (instance%mci_work)
    end if
    call instance%sf_chain%final ()
    if (allocated (instance%term)) then
       do i = 1, size (instance%term)
          call instance%term(i)%final ()
       end do
       deallocate (instance%term)
    end if
    call instance%pcm%final ()
    instance%evaluation_status = STAT_UNDEFINED
  end subroutine process_instance_final
```

Revert the process instance to initial state. We do not deallocate anything, just reset the state index and deactivate all components and terms.

We do not reset the choice of the MCI set `i_mci` unless this is required explicitly.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: reset => process_instance_reset
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_reset (instance, reset_mci)
    class(process_instance_t), intent(inout) :: instance
    logical, intent(in), optional :: reset_mci
    integer :: i
    call instance%process%reset_selected_cores ()
    do i = 1, size (instance%term)
       call instance%term(i)%reset ()
    end do
```

```
    instance%term%checked = .false.
    instance%term%passed = .false.
    instance%term%k_term%new_seed = .true.
    if (present (reset_mci)) then
       if (reset_mci)  instance%i_mci = 0
    end if
    instance%selected_channel = 0
    instance%evaluation_status = STAT_INITIAL
  end subroutine process_instance_reset
```

Integration and event generation

The sampler test should just evaluate the squared matrix element `n_calls` times, discarding the results, and return. This can be done before integration, e.g., for timing estimates.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: sampler_test => process_instance_sampler_test
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_sampler_test (instance, i_mci, n_calls)
    class(process_instance_t), intent(inout), target :: instance
    integer, intent(in) :: i_mci
    integer, intent(in) :: n_calls
    integer :: i_mci_work
    i_mci_work = instance%process%get_i_mci_work (i_mci)
    call instance%choose_mci (i_mci_work)
    call instance%reset_counter ()
    call instance%process%sampler_test (instance, n_calls, i_mci_work)
    call instance%process%set_counter_mci_entry (i_mci_work, instance%get_counter ())
  end subroutine process_instance_sampler_test
```

Generate a weighted event. We select one of the available MCI integrators by its index `i_mci` and thus generate an event for the associated (group of) process component(s). The arguments exactly correspond to the initializer and finalizer above.

The resulting event is stored in the `process_instance` object, which also holds the workspace of the integrator.

Note: The `process` object contains the random-number state, which changes for each event. Otherwise, all volatile data are inside the `instance` object.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: generate_weighted_event => process_instance_generate_weighted_event
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_generate_weighted_event (instance, i_mci)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_mci
    integer :: i_mci_work
    i_mci_work = instance%process%get_i_mci_work (i_mci)
    call instance%choose_mci (i_mci_work)
    associate (mci_work => instance%mci_work(i_mci_work))
       call instance%process%generate_weighted_event &
          (i_mci_work, mci_work, instance, &
```

```
            instance%keep_failed_events ())
      end associate
    end subroutine process_instance_generate_weighted_event
```

⟨*Instances: process instance: TBP*⟩+≡

```
    procedure :: generate_unweighted_event => process_instance_generate_unweighted_event
```

⟨*Instances: procedures*⟩+≡

```
    subroutine process_instance_generate_unweighted_event (instance, i_mci)
      class(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: i_mci
      integer :: i_mci_work
      i_mci_work = instance%process%get_i_mci_work (i_mci)
      call instance%choose_mci (i_mci_work)
      associate (mci_work => instance%mci_work(i_mci_work))
         call instance%process%generate_unweighted_event &
            (i_mci_work, mci_work, instance)
      end associate
    end subroutine process_instance_generate_unweighted_event
```

This replaces the event generation methods for the situation that the process instance object has been filled by other means (i.e., reading and/or recalculating its contents). We just have to fill in missing MCI data, especially the event weight.

⟨*Instances: process instance: TBP*⟩+≡

```
    procedure :: recover_event => process_instance_recover_event
```

⟨*Instances: procedures*⟩+≡

```
    subroutine process_instance_recover_event (instance)
      class(process_instance_t), intent(inout) :: instance
      integer :: i_mci
      i_mci = instance%i_mci
      call instance%process%set_i_mci_work (i_mci)
      associate (mci_instance => instance%mci_work(i_mci)%mci)
        call mci_instance%fetch (instance, instance%selected_channel)
      end associate
    end subroutine process_instance_recover_event
```

Activate the components and terms that correspond to a currently selected MCI parameter set.

⟨*Instances: process instance: TBP*⟩+≡

```
    procedure :: activate => process_instance_activate
```

⟨*Instances: procedures*⟩+≡

```
    subroutine process_instance_activate (instance)
      class(process_instance_t), intent(inout) :: instance
      integer :: i, j
      integer, dimension(:), allocatable :: i_term
      associate (mci_work => instance%mci_work(instance%i_mci))
         call instance%process%select_components (mci_work%get_active_components ())
      end associate
      associate (process => instance%process)
         do i = 1, instance%process%get_n_components ()
```

```
            if (instance%process%component_is_selected (i)) then
               allocate (i_term (size (process%get_component_i_terms (i))))
               i_term = process%get_component_i_terms (i)
               do j = 1, size (i_term)
                  instance%term(i_term(j))%active = .true.
               end do
            end if
            if (allocated (i_term)) deallocate (i_term)
         end do
      end associate
      instance%evaluation_status = STAT_ACTIVATED
   end subroutine process_instance_activate
```

⟨*Instances: process instance: TBP*⟩+≡

```
   procedure :: find_same_kinematics => process_instance_find_same_kinematics
```

⟨*Instances: procedures*⟩+≡

```
   subroutine process_instance_find_same_kinematics (instance)
      class(process_instance_t), intent(inout) :: instance
      integer :: i_term1, i_term2, k, n_same
      do i_term1 = 1, size (instance%term)
         if (.not. allocated (instance%term(i_term1)%same_kinematics)) then
            n_same = 1 !!! Index group includes the index of its term_instance
            do i_term2 = 1, size (instance%term)
               if (i_term1 == i_term2) cycle
               if (compare_md5s (i_term1, i_term2)) n_same = n_same + 1
            end do
            allocate (instance%term(i_term1)%same_kinematics (n_same))
            associate (same_kinematics1 => instance%term(i_term1)%same_kinematics)
               same_kinematics1 = 0
               k = 1
               do i_term2 = 1, size (instance%term)
                  if (compare_md5s (i_term1, i_term2)) then
                     same_kinematics1(k) = i_term2
                     k = k + 1
                  end if
               end do
               do k = 1, size (same_kinematics1)
                  if (same_kinematics1(k) == i_term1) cycle
                  i_term2 = same_kinematics1(k)
                  allocate (instance%term(i_term2)%same_kinematics (n_same))
                  instance%term(i_term2)%same_kinematics = same_kinematics1
               end do
            end associate
         end if
      end do
   contains
      function compare_md5s (i, j) result (same)
         logical :: same
         integer, intent(in) :: i, j
         character(32) :: md5sum_1, md5sum_2
         integer :: mode_1, mode_2
         mode_1 = 0; mode_2 = 0
         select type (phs => instance%term(i)%k_term%phs%config)
```

```
      type is (phs_fks_config_t)
         md5sum_1 = phs%md5sum_born_config
         mode_1 = phs%mode
      class default
         md5sum_1 = phs%md5sum_phs_config
      end select
      select type (phs => instance%term(j)%k_term%phs%config)
      type is (phs_fks_config_t)
         md5sum_2 = phs%md5sum_born_config
         mode_2 = phs%mode
      class default
         md5sum_2 = phs%md5sum_phs_config
      end select
      same = (md5sum_1 == md5sum_2) .and. (mode_1 == mode_2)
    end function compare_md5s
  end subroutine process_instance_find_same_kinematics
```

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: transfer_same_kinematics => process_instance_transfer_same_kinematics
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_transfer_same_kinematics (instance, i_term)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_term
    integer :: i, i_term_same
    associate (same_kinematics => instance%term(i_term)%same_kinematics)
       do i = 1, size (same_kinematics)
          i_term_same = same_kinematics(i)
          if (i_term_same /= i_term) then
             instance%term(i_term_same)%p_seed = instance%term(i_term)%p_seed
             associate (phs => instance%term(i_term_same)%k_term%phs)
                call phs%set_lorentz_transformation &
                   (instance%term(i_term)%k_term%phs%get_lorentz_transformation ())
                select type (phs)
                type is (phs_fks_t)
                   call phs%set_momenta (instance%term(i_term_same)%p_seed)
                   call phs%set_reference_frames ()
                   call phs%set_isr_kinematics ()
                end select
             end associate
          end if
          instance%term(i_term_same)%k_term%new_seed = .false.
       end do
    end associate
  end subroutine process_instance_transfer_same_kinematics
```

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: redo_sf_chains => process_instance_redo_sf_chains
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_redo_sf_chains (instance, i_term, phs_channel)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_term
    integer, intent(in) :: phs_channel
```

```
    integer :: i, i_term_same
    associate (same_kinematics => instance%term(i_term)%same_kinematics)
       do i = 1, size (same_kinematics)
          i_term_same = same_kinematics(i)
          if (i_term_same /= i_term) &
             call instance%term(i_term_same)%redo_sf_chain &
                  (instance%mci_work(instance%i_mci), phs_channel)
       end do
    end associate
  end subroutine process_instance_redo_sf_chains
```

Integrate the process, using a previously initialized process instance. We select one of the available MCI integrators by its index `i_mci` and thus integrate over (structure functions and) phase space for the associated (group of) process component(s).

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: integrate => process_instance_integrate
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_integrate (instance, i_mci, n_it, n_calls, &
       adapt_grids, adapt_weights, final, pacify)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_mci
    integer, intent(in) :: n_it
    integer, intent(in) :: n_calls
    logical, intent(in), optional :: adapt_grids
    logical, intent(in), optional :: adapt_weights
    logical, intent(in), optional :: final, pacify
    integer :: nlo_type, i_mci_work
    nlo_type = instance%process%get_component_nlo_type (i_mci)
    i_mci_work = instance%process%get_i_mci_work (i_mci)
    call instance%choose_mci (i_mci_work)
    call instance%reset_counter ()
    associate (mci_work => instance%mci_work(i_mci_work), &
               process => instance%process)
       call process%integrate (i_mci_work, mci_work, &
          instance, n_it, n_calls, adapt_grids, adapt_weights, &
          final, pacify, nlo_type = nlo_type)
       call process%set_counter_mci_entry (i_mci_work, instance%get_counter ())
    end associate
  end subroutine process_instance_integrate
```

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: transfer_helicities => process_instance_transfer_helicities
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_transfer_helicities (instance, i_component, core)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_component
    !!! intent(inout) is annoying. Transfer list should be somewhere else
    class(prc_blha_t), intent(inout) :: core
    integer, dimension(:,:), allocatable :: helicities
    integer, dimension(:), allocatable :: i_term
    allocate (i_term (size (instance%process%get_component_i_terms (i_component))))
```

```
    i_term = instance%process%get_component_i_terms (i_component)
    call instance%term(i_term(1))%get_helicities_for_openloops (helicities)
    call core%set_helicity_list (helicities)
  end subroutine process_instance_transfer_helicities
```

Subroutine of the initialization above: initialize the beam and structure-function chain template. We establish pointers to the configuration data, so `beam_config` must have a `target` attribute.

The resulting chain is not used directly for calculation. It will acquire instances which are stored in the process-component instance objects.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: setup_sf_chain => process_instance_setup_sf_chain
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_setup_sf_chain (instance, config)
    class(process_instance_t), intent(inout) :: instance
    type(process_beam_config_t), intent(in), target :: config
    integer :: n_strfun
    n_strfun = config%n_strfun
    if (n_strfun /= 0) then
       call instance%sf_chain%init (config%data, config%sf)
    else
       call instance%sf_chain%init (config%data)
    end if
    if (config%sf_trace) then
       call instance%sf_chain%setup_tracing (config%sf_trace_file)
    end if
  end subroutine process_instance_setup_sf_chain
```

This initialization routine should be called only for process instances which we intend as a source for physical events. It initializes the evaluators in the parton states of the terms. They describe the (semi-)exclusive transition matrix and the distribution of color flow for the partonic process, convoluted with the beam and structure-function chain.

If the model is not provided explicitly, we may use the model instance that belongs to the process. However, an explicit model allows us to override particle settings.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: setup_event_data => process_instance_setup_event_data
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_setup_event_data (instance, model, i_core)
    class(process_instance_t), intent(inout), target :: instance
    class(model_data_t), intent(in), optional, target :: model
    integer, intent(in), optional :: i_core
    class(model_data_t), pointer :: current_model
    integer :: i
    class(prc_core_t), pointer :: core => null ()
    if (present (model)) then
       current_model => model
    else
       current_model => instance%process%get_model_ptr ()
    end if
```

```
    do i = 1, size (instance%term)
       associate (term => instance%term(i))
         if (associated (term%config)) then
            core => instance%process%get_core_term (i)
            call term%setup_event_data (core, current_model)
         end if
       end associate
    end do
    core => null ()
  end subroutine process_instance_setup_event_data
```

Choose a MC parameter set and the corresponding integrator. The choice persists beyond calls of the `reset` method above. This method is automatically called here.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: choose_mci => process_instance_choose_mci
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_choose_mci (instance, i_mci)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_mci
    instance%i_mci = i_mci
    call instance%reset ()
  end subroutine process_instance_choose_mci
```

Explicitly set a MC parameter set. Works only if we are in initial state. We assume that the length of the parameter set is correct.

After setting the parameters, activate the components and terms that correspond to the chosen MC parameter set.

The `warmup_flag` is used when a dummy phase-space point is computed for the warmup of e.g. OpenLoops helicities. The setting of the the `evaluation_status` has to be avoided then.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: set_mcpar => process_instance_set_mcpar
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_set_mcpar (instance, x, warmup_flag)
    class(process_instance_t), intent(inout) :: instance
    real(default), dimension(:), intent(in) :: x
    logical, intent(in), optional :: warmup_flag
    logical :: activate
    activate = .true.; if (present (warmup_flag)) activate = .not. warmup_flag
    if (instance%evaluation_status == STAT_INITIAL) then
       associate (mci_work => instance%mci_work(instance%i_mci))
          call mci_work%set (x)
       end associate
       if (activate) call instance%activate ()
    end if
  end subroutine process_instance_set_mcpar
```

Receive the beam momentum/momenta from a source interaction. This applies to a cascade decay process instance, where the 'beam' momentum varies event by event.

The master beam momentum array is contained in the main structure function chain subobject `sf_chain`. The sf-chain instance that reside in the components will take their beam momenta from there.

The procedure transforms the instance status into `STAT_BEAM_MOMENTA`. For process instance with fixed beam, this intermediate status is skipped.

⟨*Instances: process instance: TBP*⟩+≡
```
procedure :: receive_beam_momenta => process_instance_receive_beam_momenta
```
⟨*Instances: procedures*⟩+≡
```
subroutine process_instance_receive_beam_momenta (instance)
  class(process_instance_t), intent(inout) :: instance
  if (instance%evaluation_status >= STAT_INITIAL) then
     call instance%sf_chain%receive_beam_momenta ()
     instance%evaluation_status = STAT_BEAM_MOMENTA
  end if
end subroutine process_instance_receive_beam_momenta
```

Set the beam momentum/momenta explicitly. Otherwise, analogous to the previous procedure.

⟨*Instances: process instance: TBP*⟩+≡
```
procedure :: set_beam_momenta => process_instance_set_beam_momenta
```
⟨*Instances: procedures*⟩+≡
```
subroutine process_instance_set_beam_momenta (instance, p)
  class(process_instance_t), intent(inout) :: instance
  type(vector4_t), dimension(:), intent(in) :: p
  if (instance%evaluation_status >= STAT_INITIAL) then
     call instance%sf_chain%set_beam_momenta (p)
     instance%evaluation_status = STAT_BEAM_MOMENTA
  end if
end subroutine process_instance_set_beam_momenta
```

Recover the initial beam momenta (those in the `sf_chain` component), given a valid (recovered) `sf_chain_instance` in one of the active components. We need to do this only if the lab frame is not the c.m. frame, otherwise those beams would be fixed anyway.

⟨*Instances: process instance: TBP*⟩+≡
```
procedure :: recover_beam_momenta => process_instance_recover_beam_momenta
```
⟨*Instances: procedures*⟩+≡
```
subroutine process_instance_recover_beam_momenta (instance, i_term)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: i_term
  if (.not. instance%process%lab_is_cm_frame ()) then
     if (instance%evaluation_status >= STAT_EFF_KINEMATICS) then
        call instance%term(i_term)%return_beam_momenta ()
     end if
  end if
end subroutine process_instance_recover_beam_momenta
```

Explicitly choose MC integration channel. We assume here that the channel count is identical for all active components.

⟨*Instances: process instance: TBP*⟩+≡
```
procedure :: select_channel => process_instance_select_channel
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_select_channel (instance, channel)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: channel
  instance%selected_channel = channel
end subroutine process_instance_select_channel
```

First step of process evaluation: set up seed kinematics. That is, for each active process component, compute a momentum array from the MC input parameters.

If `skip_term` is set, we skip the component that accesses this term. We can assume that the associated data have already been recovered, and we are just computing the rest.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: compute_seed_kinematics => &
     process_instance_compute_seed_kinematics
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_compute_seed_kinematics (instance, skip_term)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in), optional :: skip_term
  integer :: channel, skip_component, i, j
  logical :: success
  integer, dimension(:), allocatable :: i_term
  channel = instance%selected_channel
  if (channel == 0) then
     call msg_bug ("Compute seed kinematics: undefined integration channel")
  end if
  if (present (skip_term)) then
     skip_component = instance%term(skip_term)%config%i_component
  else
     skip_component = 0
  end if
  if (instance%evaluation_status >= STAT_ACTIVATED) then
     success = .true.
     do i = 1, instance%process%get_n_components ()
        if (i == skip_component)  cycle
        if (instance%process%component_is_selected (i)) then
           allocate (i_term (size (instance%process%get_component_i_terms (i))))
           i_term = instance%process%get_component_i_terms (i)
           do j = 1, size (i_term)
              if (instance%term(i_term(j))%k_term%new_seed) then
                 call instance%term(i_term(j))%compute_seed_kinematics &
                    (instance%mci_work(instance%i_mci), channel, success)
                 call instance%transfer_same_kinematics (i_term(j))
                 call instance%redo_sf_chains (i_term(j), channel)
              end if
              if (.not. success)  exit
              call instance%term(i_term(j))%evaluate_projections ()
              call instance%term(i_term(j))%evaluate_radiation_kinematics &
                     (instance%mci_work(instance%i_mci))
           end do
        end if
        if (allocated (i_term)) deallocate (i_term)
     end do
```

```
      if (success) then
         instance%evaluation_status = STAT_SEED_KINEMATICS
      else
         instance%evaluation_status = STAT_FAILED_KINEMATICS
      end if
   end if
   associate (mci_work => instance%mci_work(instance%i_mci))
      select type (pcm => instance%pcm)
      class is (pcm_instance_nlo_t)
         call pcm%set_x_rad (mci_work%get_x_process ())
      end select
   end associate
end subroutine process_instance_compute_seed_kinematics
```

Inverse: recover missing parts of the kinematics from the momentum configuration, which we know for a single term and component. Given a channel, reconstruct the MC parameter set.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: recover_mcpar => process_instance_recover_mcpar
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_recover_mcpar (instance, i_term)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: i_term
  integer :: channel
  if (instance%evaluation_status >= STAT_EFF_KINEMATICS) then
     channel = instance%selected_channel
     if (channel == 0) then
        call msg_bug ("Recover MC parameters: undefined integration channel")
     end if
     call instance%term(i_term)%recover_mcpar &
        (instance%mci_work(instance%i_mci), channel)
  end if
end subroutine process_instance_recover_mcpar
```

Second step of process evaluation: compute all momenta, for all active components, from the seed kinematics.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: compute_hard_kinematics => &
     process_instance_compute_hard_kinematics
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_compute_hard_kinematics (instance, skip_term)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in), optional :: skip_term
  integer :: i
  logical :: success
  success = .true.
  if (instance%evaluation_status >= STAT_SEED_KINEMATICS) then
     do i = 1, size (instance%term)
        if (instance%term(i)%active) &
           call instance%term(i)%compute_hard_kinematics (skip_term, success)
        if (.not. success) exit
     end do
```

```
      if (success) then
         instance%evaluation_status = STAT_HARD_KINEMATICS
      else
         instance%evaluation_status = STAT_FAILED_KINEMATICS
      end if
   end if
end subroutine process_instance_compute_hard_kinematics
```

Inverse: recover seed kinematics. We know the beam momentum configuration and the outgoing momenta of the effective interaction, for one specific term.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: recover_seed_kinematics => &
     process_instance_recover_seed_kinematics
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_recover_seed_kinematics (instance, i_term)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: i_term
  if (instance%evaluation_status >= STAT_EFF_KINEMATICS) &
     call instance%term(i_term)%recover_seed_kinematics ()
end subroutine process_instance_recover_seed_kinematics
```

Third step of process evaluation: compute the effective momentum configurations, for all active terms, from the hard kinematics.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: compute_eff_kinematics => &
     process_instance_compute_eff_kinematics
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_compute_eff_kinematics (instance, skip_term)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in), optional :: skip_term
  integer :: i
  if (instance%evaluation_status >= STAT_HARD_KINEMATICS) then
     do i = 1, size (instance%term)
        if (present (skip_term)) then
           if (i == skip_term)  cycle
        end if
        if (instance%term(i)%active) then
           call instance%term(i)%compute_eff_kinematics ()
        end if
     end do
     instance%evaluation_status = STAT_EFF_KINEMATICS
  end if
end subroutine process_instance_compute_eff_kinematics
```

Inverse: recover the hard kinematics from effective kinematics for one term, then compute effective kinematics for the other terms.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: recover_hard_kinematics => &
     process_instance_recover_hard_kinematics
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_recover_hard_kinematics (instance, i_term)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: i_term
  integer :: i
  if (instance%evaluation_status >= STAT_EFF_KINEMATICS) then
     call instance%term(i_term)%recover_hard_kinematics ()
     do i = 1, size (instance%term)
        if (i /= i_term) then
           if (instance%term(i)%active) then
              call instance%term(i)%compute_eff_kinematics ()
           end if
        end if
     end do
     instance%evaluation_status = STAT_EFF_KINEMATICS
  end if
end subroutine process_instance_recover_hard_kinematics
```

Fourth step of process evaluation: check cuts for all terms. Where sucessful, compute any scales and weights. Otherwise, deactive the term. If any of the terms has passed, set the state to `STAT_PASSED_CUTS`.

The argument `scale_forced`, if present, will override the scale calculation in the term expressions.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: evaluate_expressions => &
     process_instance_evaluate_expressions
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_evaluate_expressions (instance, scale_forced)
  class(process_instance_t), intent(inout) :: instance
  real(default), intent(in), allocatable, optional :: scale_forced
  integer :: i
  logical :: passed_real
  if (instance%evaluation_status >= STAT_EFF_KINEMATICS) then
     do i = 1, size (instance%term)
        if (instance%term(i)%active) then
           call instance%term(i)%evaluate_expressions (scale_forced)
        end if
     end do
     call evaluate_real_scales_and_cuts ()
     if (.not. passed_real) then
        instance%evaluation_status = STAT_FAILED_CUTS
     else
        if (any (instance%term%passed)) then
           instance%evaluation_status = STAT_PASSED_CUTS
        else
           instance%evaluation_status = STAT_FAILED_CUTS
        end if
    end if
  end if
contains
  subroutine evaluate_real_scales_and_cuts ()
    integer :: i
    passed_real = .true.
```

```
      select type (config => instance%pcm%config)
      type is (pcm_nlo_t)
         do i = 1, size (instance%term)
            if (instance%term(i)%active .and. instance%term(i)%nlo_type == NLO_REAL) then
               if (config%settings%cut_all_sqmes) &
                    passed_real = passed_real .and. instance%term(i)%passed
               if (config%settings%use_born_scale) &
                    call replace_scales (instance%term(i))
            end if
         end do
      end select
    end subroutine evaluate_real_scales_and_cuts

    subroutine replace_scales (this_term)
      type(term_instance_t), intent(inout) :: this_term
      integer :: i_sub
      i_sub = this_term%config%i_sub
      if (this_term%config%i_term_global /= i_sub) then
         this_term%ren_scale = instance%term(i_sub)%ren_scale
         this_term%fac_scale = instance%term(i_sub)%fac_scale
      end if
    end subroutine replace_scales
  end subroutine process_instance_evaluate_expressions
```

Fifth step of process evaluation: fill the parameters for the non-selected ,channels, that have not been used for seeding. We should do this after evaluating cuts, since we may save some expensive calculations if the phase space point fails the cuts.

If `skip_term` is set, we skip the component that accesses this term. We can assume that the associated data have already been recovered, and we are just computing the rest.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: compute_other_channels => &
       process_instance_compute_other_channels
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_compute_other_channels (instance, skip_term)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in), optional :: skip_term
    integer :: channel, skip_component, i, j
    integer, dimension(:), allocatable :: i_term
    channel = instance%selected_channel
    if (channel == 0) then
       call msg_bug ("Compute other channels: undefined integration channel")
    end if
    if (present (skip_term)) then
       skip_component = instance%term(skip_term)%config%i_component
    else
       skip_component = 0
    end if
    if (instance%evaluation_status >= STAT_PASSED_CUTS) then
       do i = 1, instance%process%get_n_components ()
          if (i == skip_component)  cycle
```

```
        if (instance%process%component_is_selected (i)) then
           allocate (i_term (size (instance%process%get_component_i_terms (i))))
           i_term = instance%process%get_component_i_terms (i)
           do j = 1, size (i_term)
              call instance%term(i_term(j))%compute_other_channels &
                 (instance%mci_work(instance%i_mci), channel)
           end do
        end if
        if (allocated (i_term)) deallocate (i_term)
     end do
  end if
end subroutine process_instance_compute_other_channels
```

Sixth step of process evaluation: evaluate the matrix elements, and compute the trace (summed over quantum numbers) for all terms. Finally, sum up the terms, iterating over all active process components.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: evaluate_trace => process_instance_evaluate_trace
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_evaluate_trace (instance)
  class(process_instance_t), intent(inout) :: instance
  class(prc_core_t), pointer :: core => null ()
  integer :: i, i_real_fin
  real(default) :: alpha_s
  class(prc_core_t), pointer :: core_sub => null ()
  integer :: em
  instance%sqme = zero
  call instance%reset_matrix_elements ()
  if (instance%evaluation_status >= STAT_PASSED_CUTS) then
     do i = 1, size (instance%term)
        associate (term => instance%term(i))
          if (term%active .and. term%passed) then
             core => instance%process%get_core_term (i)
             if (instance%pcm%config%is_nlo ()) &
                  core_sub => instance%process%get_subtraction_core ()
             call term%evaluate_interaction (core)
             call term%evaluate_trace ()
             i_real_fin = instance%process%get_associated_real_fin (1)
             if (instance%process%uses_real_partition ()) &
                  call term%apply_real_partition (instance%process)

             if (term%config%i_component /= i_real_fin) then
                if ((term%nlo_type == NLO_REAL .and. term%k_term%emitter < 0) &
                      .or. term%nlo_type == NLO_MISMATCH) call set_born_sqmes (term)
                if (term%nlo_type > BORN) then
                   if (.not. (term%nlo_type == NLO_REAL .and. term%k_term%emitter >= 0)) then
                      call term%evaluate_color_correlations (term%p_seed, core_sub)
                      call term%evaluate_spin_correlations (term%p_seed, core_sub)
                   end if
                end if
                alpha_s = core%get_alpha_s (term%core_state)
                select case (term%nlo_type)
                case (NLO_REAL)
```

```
                  call term%apply_fks (core, alpha_s)
               case (NLO_VIRTUAL)
                  call term%evaluate_sqme_virt (alpha_s)
               case (NLO_MISMATCH)
                  call term%evaluate_sqme_mismatch (alpha_s)
               case (NLO_DGLAP)
                  call term%evaluate_sqme_dglap (alpha_s)
               end select
            end if
         end if
         core_sub => null ()
         instance%sqme = instance%sqme + real (sum (&
            term%connected%trace%get_matrix_element () * &
            term%weight))

      end associate
   end do

   core => null ()
   if (instance%pcm%is_valid ()) then
      instance%evaluation_status = STAT_EVALUATED_TRACE
   else
      instance%evaluation_status = STAT_FAILED_KINEMATICS
   end if

  else
     !!! Failed kinematics or failed cuts: set sqme to zero
     instance%sqme = zero
  end if
contains

  subroutine set_born_sqmes (term)
    type(term_instance_t), intent(inout) :: term
    integer :: i_flv
    real(default) :: sqme
    select type (config => instance%pcm%config)
    type is (pcm_nlo_t)
       do i_flv = 1, term%connected%trace%get_n_matrix_elements ()
          sqme = real (term%connected%trace%get_matrix_element (i_flv))
          select case (term%nlo_type)
          case (NLO_REAL)
             term%pcm_instance%real_sub%sqme_born (i_flv) = sqme
          case (NLO_MISMATCH)
             term%pcm_instance%soft_mismatch%sqme_born (i_flv) = sqme
          end select
       end do
    end select
  end subroutine set_born_sqmes
end subroutine process_instance_evaluate_trace
```

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: apply_real_partition => process_instance_apply_real_partition
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_apply_real_partition (instance, i_component)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: i_component
  integer :: i_term
  integer, dimension(:), allocatable :: i_terms
  associate (process => instance%process)
     if (process%component_is_selected (i_component) .and. &
            process%get_component_nlo_type (i_component) == NLO_REAL) then
        allocate (i_terms (size (process%get_component_i_terms (i_component))))
        i_terms = process%get_component_i_terms (i_component)
        do i_term = 1, size (i_terms)
           call instance%term(i_terms(i_term))%apply_real_partition (process)
        end do
     end if
     if (allocated (i_terms)) deallocate (i_terms)
  end associate
end subroutine process_instance_apply_real_partition
```

Final step of process evaluation: evaluate the matrix elements, and compute the trace (summed over quantum numbers) for all terms. Finally, sum up the terms, iterating over all active process components.

If `weight` is provided, we already know the kinematical event weight (the MCI weight which depends on the kinematics sampling algorithm, but not on the matrix element), so we do not need to take it from the MCI record.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: evaluate_event_data => process_instance_evaluate_event_data
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_evaluate_event_data (instance, weight)
  class(process_instance_t), intent(inout) :: instance
  real(default), intent(in), optional :: weight
  integer :: i
  if (instance%evaluation_status >= STAT_EVALUATED_TRACE) then
     do i = 1, size (instance%term)
        associate (term => instance%term(i))
          if (term%active .and. term%passed) then
             call term%evaluate_event_data ()
          end if
        end associate
     end do
     if (present (weight)) then
        instance%weight = weight
     else
        instance%weight = &
             instance%mci_work(instance%i_mci)%mci%get_event_weight ()
        instance%excess = &
             instance%mci_work(instance%i_mci)%mci%get_event_excess ()
     end if
     instance%evaluation_status = STAT_EVENT_COMPLETE
  else
     !!! failed kinematics etc.: set weight to zero
     instance%weight = zero
     !!! Maybe we want to keep the event nevertheless
     if (instance%keep_failed_events ()) then
```

```
         !!! Force factorization scale, otherwise writing to event output fails
         do i = 1, size (instance%term)
            instance%term(i)%fac_scale = zero
         end do
         instance%evaluation_status = STAT_EVENT_COMPLETE
      end if
   end if
end subroutine process_instance_evaluate_event_data
```

Computes the real-emission matrix element for externally supplied momenta. Also, e.g. for Powheg, there is the possibility to supply an external α_s

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: compute_sqme_rad => process_instance_compute_sqme_rad
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_compute_sqme_rad &
     (instance, i_term, i_phs, is_subtraction, alpha_s_external)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: i_term, i_phs
  logical, intent(in) :: is_subtraction
  real(default), intent(in), optional :: alpha_s_external
  class(prc_core_t), pointer :: core
  integer :: i_real, i_real_fin

  select type (pcm => instance%pcm)
  type is (pcm_instance_nlo_t)
     associate (term => instance%term(i_term))
       core => instance%process%get_core_term (i_term)
       if (is_subtraction) then
          call pcm%set_subtraction_event ()
       else
          call pcm%set_radiation_event ()
       end if
       call term%int_hard%set_momenta (pcm%get_momenta &
          (i_phs = i_phs, born_phsp = is_subtraction))
       if (allocated (term%core_state)) &
          call term%core_state%reset_new_kinematics ()
       if (present (alpha_s_external)) &
          call term%set_alpha_qcd_forced (alpha_s_external)
       call term%compute_eff_kinematics ()
       call term%evaluate_expressions ()
       call term%evaluate_interaction (core)
       call term%evaluate_trace ()
       pcm%real_sub%sqme_born (1) = &
            real (term%connected%trace%get_matrix_element (1))
       if (term%nlo_type == NLO_REAL .and. term%k_term%emitter < 0) then
          call term%evaluate_color_correlations (term%p_seed, core)
          call term%evaluate_spin_correlations (term%p_seed, core)
       end if
       i_real_fin = instance%process%get_associated_real_fin (1)
       if (term%config%i_component /= i_real_fin) &
            call term%apply_fks (core, core%get_alpha_s (term%core_state))
       if (instance%process%uses_real_partition ()) then
          i_real = instance%get_associated_real ()
```

```
            call instance%apply_real_partition (i_real)
          end if
        end associate
      end select
      core => null ()
    end subroutine process_instance_compute_sqme_rad
```

For unweighted event generation, we should reset the reported event weight to unity (signed) or zero. The latter case is appropriate for an event which failed for whatever reason.

⟨*Instances: process instance: TBP*⟩+≡

```
    procedure :: normalize_weight => process_instance_normalize_weight
```

⟨*Instances: procedures*⟩+≡

```
    subroutine process_instance_normalize_weight (instance)
      class(process_instance_t), intent(inout) :: instance
      if (.not. vanishes (instance%weight)) then
         instance%weight = sign (1._default, instance%weight)
      end if
    end subroutine process_instance_normalize_weight
```

This is a convenience routine that performs the computations of the steps 1 to 5 in a single step. The arguments are the input for `set_mcpar`. After this, the evaluation status should be either `STAT_FAILED_KINEMATICS`, `STAT_FAILED_CUTS` or `STAT_EVALUATED_TRACE`.

Before calling this, we should call `choose_mci`.

⟨*Instances: process instance: TBP*⟩+≡

```
    procedure :: evaluate_sqme => process_instance_evaluate_sqme
```

⟨*Instances: procedures*⟩+≡

```
    subroutine process_instance_evaluate_sqme (instance, channel, x)
      class(process_instance_t), intent(inout) :: instance
      integer, intent(in) :: channel
      real(default), dimension(:), intent(in) :: x
      call instance%reset ()
      call instance%set_mcpar (x)
      call instance%select_channel (channel)
      call instance%compute_seed_kinematics ()
      call instance%compute_hard_kinematics ()
      call instance%compute_eff_kinematics ()
      call instance%evaluate_expressions ()
      call instance%compute_other_channels ()
      call instance%evaluate_trace ()
    end subroutine process_instance_evaluate_sqme
```

This is the inverse. Assuming that the final trace evaluator contains a valid momentum configuration, recover kinematics and recalculate the matrix elements and their trace.

To be precise, we first recover kinematics for the given term and associated component, then recalculate from that all other terms and active components. The `channel` is not really required to obtain the matrix element, but it allows

us to reconstruct the exact MC parameter set that corresponds to the given phase space point.

Before calling this, we should call `choose_mci`.

⟨*Instances: process instance: TBP*⟩+≡
```
procedure :: recover => process_instance_recover
```
⟨*Instances: procedures*⟩+≡
```
subroutine process_instance_recover &
    (instance, channel, i_term, update_sqme, scale_forced)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: channel
  integer, intent(in) :: i_term
  logical, intent(in) :: update_sqme
  real(default), intent(in), allocatable, optional :: scale_forced
  call instance%activate ()
  instance%evaluation_status = STAT_EFF_KINEMATICS
  call instance%recover_hard_kinematics (i_term)
  call instance%recover_seed_kinematics (i_term)
  call instance%select_channel (channel)
  call instance%recover_mcpar (i_term)
  call instance%recover_beam_momenta (i_term)
  call instance%compute_seed_kinematics (i_term)
  call instance%compute_hard_kinematics (i_term)
  call instance%compute_eff_kinematics (i_term)
  call instance%compute_other_channels (i_term)
  call instance%evaluate_expressions (scale_forced)
  if (update_sqme)  call instance%evaluate_trace ()
end subroutine process_instance_recover
```

The `evaluate` method is required by the `sampler_t` base type of which the process instance is an extension.

The requirement is that after the process instance is evaluated, the integrand, the selected channel, the x array, and the f Jacobian array are exposed by the `sampler_t` object.

⟨*Instances: process instance: TBP*⟩+≡
```
procedure :: evaluate => process_instance_evaluate
```
⟨*Instances: procedures*⟩+≡
```
subroutine process_instance_evaluate (sampler, c, x_in, val, x, f)
  class(process_instance_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  call sampler%evaluate_sqme (c, x_in)
  if (sampler%is_valid ())  call sampler%fetch (val, x, f)
  call sampler%record_call ()
end subroutine process_instance_evaluate
```

The phase-space point is valid if the event has valid kinematics and has passed the cuts.

⟨*Instances: process instance: TBP*⟩+≡
```
procedure :: is_valid => process_instance_is_valid
```

⟨*Instances: procedures*⟩+≡

```
function process_instance_is_valid (sampler) result (valid)
  class(process_instance_t), intent(in) :: sampler
  logical :: valid
  valid = sampler%evaluation_status >= STAT_PASSED_CUTS
end function process_instance_is_valid
```

The `rebuild` method should rebuild the kinematics section out of the `x_in` parameter set. The integrand value `val` should not be computed, but is provided as input.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: rebuild => process_instance_rebuild
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_rebuild (sampler, c, x_in, val, x, f)
  class(process_instance_t), intent(inout) :: sampler
  integer, intent(in) :: c
  real(default), dimension(:), intent(in) :: x_in
  real(default), intent(in) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  call msg_bug ("process_instance_rebuild not implemented yet")
  x = 0
  f = 0
end subroutine process_instance_rebuild
```

This is another method required by the `sampler_t` base type: fetch the data that are relevant for the MCI record.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: fetch => process_instance_fetch
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_fetch (sampler, val, x, f)
  class(process_instance_t), intent(in) :: sampler
  real(default), intent(out) :: val
  real(default), dimension(:,:), intent(out) :: x
  real(default), dimension(:), intent(out) :: f
  integer, dimension(:), allocatable :: i_terms
  integer :: i, i_term_base, cc
  integer :: n_channel

  val = 0
  associate (process => sampler%process)
     FIND_COMPONENT: do i = 1, process%get_n_components ()
       if (sampler%process%component_is_selected (i)) then
          allocate (i_terms (size (process%get_component_i_terms (i))))
          i_terms = process%get_component_i_terms (i)
          i_term_base = i_terms(1)
          associate (k => sampler%term(i_term_base)%k_term)
            n_channel = k%n_channel
            do cc = 1, n_channel
               call k%get_mcpar (cc, x(:,cc))
            end do
            f = k%f
```

```
            val = sampler%sqme * k%phs_factor
          end associate
          if (allocated (i_terms)) deallocate (i_terms)
          exit FIND_COMPONENT
        end if
      end do FIND_COMPONENT
    end associate
  end subroutine process_instance_fetch
```

Initialize and finalize event generation for the specified MCI entry.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: init_simulation => process_instance_init_simulation
  procedure :: final_simulation => process_instance_final_simulation
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_init_simulation (instance, i_mci, &
     safety_factor, keep_failed_events)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_mci
    real(default), intent(in), optional :: safety_factor
    logical, intent(in), optional :: keep_failed_events
    call instance%mci_work(i_mci)%init_simulation (safety_factor, keep_failed_events)
  end subroutine process_instance_init_simulation

  subroutine process_instance_final_simulation (instance, i_mci)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_mci
    call instance%mci_work(i_mci)%final_simulation ()
  end subroutine process_instance_final_simulation
```

Accessing the process instance

Once the seed kinematics is complete, we can retrieve the MC input parameters for all channels, not just the seed channel.

Note: We choose the first active component. This makes sense only if the seed kinematics is identical for all active components.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: get_mcpar => process_instance_get_mcpar
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_get_mcpar (instance, channel, x)
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: channel
    real(default), dimension(:), intent(out) :: x
    integer :: i
    if (instance%evaluation_status >= STAT_SEED_KINEMATICS) then
       do i = 1, size (instance%term)
          if (instance%term(i)%active) then
             call instance%term(i)%k_term%get_mcpar (channel, x)
             return
          end if
       end do
       call msg_bug ("Process instance: get_mcpar: no active channels")
```

```
    else
       call msg_bug ("Process instance: get_mcpar: no seed kinematics")
    end if
  end subroutine process_instance_get_mcpar
```

Return true if the sqme value is known. This also implies that the event is kinematically valid and has passed all cuts.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: has_evaluated_trace => process_instance_has_evaluated_trace
```

⟨*Instances: procedures*⟩+≡

```
  function process_instance_has_evaluated_trace (instance) result (flag)
    class(process_instance_t), intent(in) :: instance
    logical :: flag
    flag = instance%evaluation_status >= STAT_EVALUATED_TRACE
  end function process_instance_has_evaluated_trace
```

Return true if the event is complete. In particular, the event must be kinematically valid, passed all cuts, and the event data have been computed.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: is_complete_event => process_instance_is_complete_event
```

⟨*Instances: procedures*⟩+≡

```
  function process_instance_is_complete_event (instance) result (flag)
    class(process_instance_t), intent(in) :: instance
    logical :: flag
    flag = instance%evaluation_status >= STAT_EVENT_COMPLETE
  end function process_instance_is_complete_event
```

Select the term for the process instance that will provide the basic event record (used in evt_trivial_make_particle_set). It might be necessary to write out additional events corresponding to other terms (done in evt_nlo).

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: select_i_term => process_instance_select_i_term
```

⟨*Instances: procedures*⟩+≡

```
  function process_instance_select_i_term (instance) result (i_term)
    integer :: i_term
    class(process_instance_t), intent(in) :: instance
    integer :: i_mci
    i_mci = instance%i_mci
    i_term = instance%process%select_i_term (i_mci)
  end function process_instance_select_i_term
```

Return pointer to the master beam interaction.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: get_beam_int_ptr => process_instance_get_beam_int_ptr
```

⟨*Instances: procedures*⟩+≡

```
  function process_instance_get_beam_int_ptr (instance) result (ptr)
    class(process_instance_t), intent(in), target :: instance
    type(interaction_t), pointer :: ptr
    ptr => instance%sf_chain%get_beam_int_ptr ()
```

```
end function process_instance_get_beam_int_ptr
```

Return pointers to the matrix and flows interactions, given a term index.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: get_trace_int_ptr => process_instance_get_trace_int_ptr
procedure :: get_matrix_int_ptr => process_instance_get_matrix_int_ptr
procedure :: get_flows_int_ptr => process_instance_get_flows_int_ptr
```

⟨*Instances: procedures*⟩+≡

```
function process_instance_get_trace_int_ptr (instance, i_term) result (ptr)
  class(process_instance_t), intent(in), target :: instance
  integer, intent(in) :: i_term
  type(interaction_t), pointer :: ptr
  ptr => instance%term(i_term)%connected%get_trace_int_ptr ()
end function process_instance_get_trace_int_ptr

function process_instance_get_matrix_int_ptr (instance, i_term) result (ptr)
  class(process_instance_t), intent(in), target :: instance
  integer, intent(in) :: i_term
  type(interaction_t), pointer :: ptr
  ptr => instance%term(i_term)%connected%get_matrix_int_ptr ()
end function process_instance_get_matrix_int_ptr

function process_instance_get_flows_int_ptr (instance, i_term) result (ptr)
  class(process_instance_t), intent(in), target :: instance
  integer, intent(in) :: i_term
  type(interaction_t), pointer :: ptr
  ptr => instance%term(i_term)%connected%get_flows_int_ptr ()
end function process_instance_get_flows_int_ptr
```

Return the complete account of flavor combinations in the underlying interaction object, including beams, radiation, and hard interaction.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: get_state_flv => process_instance_get_state_flv
```

⟨*Instances: procedures*⟩+≡

```
function process_instance_get_state_flv (instance, i_term) result (state_flv)
  class(process_instance_t), intent(in) :: instance
  integer, intent(in) :: i_term
  type(state_flv_content_t) :: state_flv
  state_flv = instance%term(i_term)%connected%get_state_flv ()
end function process_instance_get_state_flv
```

Return pointers to the parton states of a selected term.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: get_isolated_state_ptr => &
     process_instance_get_isolated_state_ptr
procedure :: get_connected_state_ptr => &
     process_instance_get_connected_state_ptr
```

⟨*Instances: procedures*⟩+≡

```
function process_instance_get_isolated_state_ptr (instance, i_term) &
     result (ptr)
```

```
    class(process_instance_t), intent(in), target :: instance
    integer, intent(in) :: i_term
    type(isolated_state_t), pointer :: ptr
    ptr => instance%term(i_term)%isolated
  end function process_instance_get_isolated_state_ptr

  function process_instance_get_connected_state_ptr (instance, i_term) &
       result (ptr)
    class(process_instance_t), intent(in), target :: instance
    integer, intent(in) :: i_term
    type(connected_state_t), pointer :: ptr
    ptr => instance%term(i_term)%connected
  end function process_instance_get_connected_state_ptr
```

Return the indices of the beam particles and incoming partons within the currently active state matrix, respectively.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: get_beam_index => process_instance_get_beam_index
  procedure :: get_in_index => process_instance_get_in_index
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_get_beam_index (instance, i_term, i_beam)
    class(process_instance_t), intent(in) :: instance
    integer, intent(in) :: i_term
    integer, dimension(:), intent(out) :: i_beam
    call instance%term(i_term)%connected%get_beam_index (i_beam)
  end subroutine process_instance_get_beam_index

  subroutine process_instance_get_in_index (instance, i_term, i_in)
    class(process_instance_t), intent(in) :: instance
    integer, intent(in) :: i_term
    integer, dimension(:), intent(out) :: i_in
    call instance%term(i_term)%connected%get_in_index (i_in)
  end subroutine process_instance_get_in_index
```

Return squared matrix element and event weight, and event weight excess where applicable.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: get_sqme => process_instance_get_sqme
  procedure :: get_weight => process_instance_get_weight
  procedure :: get_excess => process_instance_get_excess
```

⟨*Instances: procedures*⟩+≡

```
  function process_instance_get_sqme (instance, i_term) result (sqme)
    real(default) :: sqme
    class(process_instance_t), intent(in) :: instance
    integer, intent(in), optional :: i_term
    if (instance%evaluation_status >= STAT_EVALUATED_TRACE) then
       if (present (i_term)) then
          sqme = instance%term(i_term)%connected%trace%get_matrix_element (1)
       else
          sqme = instance%sqme
       end if
    else
```

```
      sqme = 0
    end if
  end function process_instance_get_sqme

  function process_instance_get_weight (instance) result (weight)
    real(default) :: weight
    class(process_instance_t), intent(in) :: instance
    if (instance%evaluation_status >= STAT_EVENT_COMPLETE) then
      weight = instance%weight
    else
      weight = 0
    end if
  end function process_instance_get_weight

  function process_instance_get_excess (instance) result (excess)
    real(default) :: excess
    class(process_instance_t), intent(in) :: instance
    if (instance%evaluation_status >= STAT_EVENT_COMPLETE) then
      excess = instance%excess
    else
      excess = 0
    end if
  end function process_instance_get_excess
```

Return the currently selected MCI channel.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: get_channel => process_instance_get_channel
```

⟨*Instances: procedures*⟩+≡

```
  function process_instance_get_channel (instance) result (channel)
    integer :: channel
    class(process_instance_t), intent(in) :: instance
    channel = instance%selected_channel
  end function process_instance_get_channel
```

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: set_fac_scale => process_instance_set_fac_scale
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_set_fac_scale (instance, fac_scale)
    class(process_instance_t), intent(inout) :: instance
    real(default), intent(in) :: fac_scale
    integer :: i_term
    i_term = 1
    call instance%term(i_term)%set_fac_scale (fac_scale)
  end subroutine process_instance_set_fac_scale
```

Return factorization scale and strong coupling. We have to select a term instance.

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: get_fac_scale => process_instance_get_fac_scale
  procedure :: get_alpha_s => process_instance_get_alpha_s
```

⟨*Instances: procedures*⟩+≡

```
function process_instance_get_fac_scale (instance, i_term) result (fac_scale)
  class(process_instance_t), intent(in) :: instance
  integer, intent(in) :: i_term
  real(default) :: fac_scale
  fac_scale = instance%term(i_term)%get_fac_scale ()
end function process_instance_get_fac_scale

function process_instance_get_alpha_s (instance, i_term) result (alpha_s)
  real(default) :: alpha_s
  class(process_instance_t), intent(in) :: instance
  integer, intent(in) :: i_term
  class(prc_core_t), pointer :: core => null ()
  core => instance%process%get_core_term (i_term)
  alpha_s = instance%term(i_term)%get_alpha_s (core)
  core => null ()
end function process_instance_get_alpha_s
```

Counter.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: reset_counter => process_instance_reset_counter
procedure :: record_call => process_instance_record_call
procedure :: get_counter => process_instance_get_counter
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_reset_counter (process_instance)
  class(process_instance_t), intent(inout) :: process_instance
  call process_instance%mci_work(process_instance%i_mci)%reset_counter ()
end subroutine process_instance_reset_counter

subroutine process_instance_record_call (process_instance)
  class(process_instance_t), intent(inout) :: process_instance
  call process_instance%mci_work(process_instance%i_mci)%record_call &
       (process_instance%evaluation_status)
end subroutine process_instance_record_call

pure function process_instance_get_counter (process_instance) result (counter)
  class(process_instance_t), intent(in) :: process_instance
  type(process_counter_t) :: counter
  counter = process_instance%mci_work(process_instance%i_mci)%get_counter ()
end function process_instance_get_counter
```

Sum up the total number of calls for all MCI records.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: get_actual_calls_total => process_instance_get_actual_calls_total
```

⟨*Instances: procedures*⟩+≡

```
pure function process_instance_get_actual_calls_total (process_instance) &
     result (n)
  class(process_instance_t), intent(in) :: process_instance
  integer :: n
  integer :: i
  type(process_counter_t) :: counter
  n = 0
```

```
    do i = 1, size (process_instance%mci_work)
       counter = process_instance%mci_work(i)%get_counter ()
       n = n + counter%total
    end do
  end function process_instance_get_actual_calls_total
```

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: reset_matrix_elements => process_instance_reset_matrix_elements
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_reset_matrix_elements (instance)
    class(process_instance_t), intent(inout) :: instance
    integer :: i_term
    do i_term = 1, size (instance%term)
       call instance%term(i_term)%connected%trace%set_matrix_element (cmplx (0, 0, default))
       call instance%term(i_term)%connected%matrix%set_matrix_element (cmplx (0, 0, default))
    end do
  end subroutine process_instance_reset_matrix_elements
```

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: get_phase_space_point &
     => process_instance_get_phase_space_point
```

⟨*Instances: procedures*⟩+≡

```
  subroutine process_instance_get_phase_space_point (instance, &
         i_component, i_core, p)
    type(vector4_t), dimension(:), allocatable, intent(out) :: p
    class(process_instance_t), intent(inout) :: instance
    integer, intent(in) :: i_component, i_core
    real(default), dimension(:), allocatable :: x
    logical :: success
    integer :: i_term
    instance%i_mci = i_component
    i_term = instance%process%get_i_term (i_core)
    associate (term => instance%term(i_term))
       allocate (x (instance%mci_work(i_component)%config%n_par))
       x = 0.5_default
       call instance%set_mcpar (x, .true.)
       call instance%select_channel (1)
       call term%compute_seed_kinematics &
            (instance%mci_work(i_component), 1, success)
       allocate (p (size (term%p_hard)))
       p = term%p_seed
    end associate
  end subroutine process_instance_get_phase_space_point
```

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: get_first_active_i_term => process_instance_get_first_active_i_term
```

⟨*Instances: procedures*⟩+≡

```
  function process_instance_get_first_active_i_term (instance) result (i_term)
    integer :: i_term
    class(process_instance_t), intent(in) :: instance
    integer :: i
```

```
    i_term = 0
    do i = 1, size (instance%term)
       if (instance%term(i)%active) then
          i_term = i
          exit
       end if
    end do
  end function process_instance_get_first_active_i_term
```

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: get_associated_real => process_instance_get_associated_real
```

⟨*Instances: procedures*⟩+≡

```
  function process_instance_get_associated_real (instance, avoid_finite) result (i_real)
    integer :: i_real
    class(process_instance_t), intent(in) :: instance
    logical, intent(in), optional :: avoid_finite
    logical :: avoid
    avoid = .false.; if (present (avoid_finite)) avoid = avoid_finite
    select type (pcm => instance%pcm)
    type is (pcm_instance_nlo_t)
       if (instance%process%get_component_type (pcm%active_real_component) &
          == COMP_REAL_FIN .and. .not. avoid) then
          i_real = instance%process%get_associated_real_fin (pcm%active_real_component)
       else
          i_real = pcm%active_real_component
       end if
    end select
  end function process_instance_get_associated_real
```

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: get_connected_states => process_instance_get_connected_states
```

⟨*Instances: procedures*⟩+≡

```
  function process_instance_get_connected_states (instance, i_component) result (connected)
    type(connected_state_t), dimension(:), allocatable :: connected
    class(process_instance_t), intent(in) :: instance
    integer, intent(in) :: i_component
    connected = instance%process%get_connected_states (i_component, &
        instance%term(:)%connected)
  end function process_instance_get_connected_states
```

Get the hadronic center-of-mass energy

⟨*Instances: process instance: TBP*⟩+≡

```
  procedure :: get_sqrts => process_instance_get_sqrts
```

⟨*Instances: procedures*⟩+≡

```
  function process_instance_get_sqrts (instance) result (sqrts)
    class(process_instance_t), intent(in) :: instance
    real(default) :: sqrts
    sqrts = instance%process%get_sqrts ()
  end function process_instance_get_sqrts
```

Get the polarizations

⟨*Instances: process instance: TBP*⟩+≡
```
procedure :: get_polarization => process_instance_get_polarization
```

⟨*Instances: procedures*⟩+≡
```
function process_instance_get_polarization (instance) result (pol)
  class(process_instance_t), intent(in) :: instance
  real(default), dimension(2) :: pol
  pol = instance%process%get_polarization ()
end function process_instance_get_polarization
```

Get the beam spectrum

⟨*Instances: process instance: TBP*⟩+≡
```
procedure :: get_beam_file => process_instance_get_beam_file
```

⟨*Instances: procedures*⟩+≡
```
function process_instance_get_beam_file (instance) result (file)
  class(process_instance_t), intent(in) :: instance
  type(string_t) :: file
  file = instance%process%get_beam_file ()
end function process_instance_get_beam_file
```

Get the process name

⟨*Instances: process instance: TBP*⟩+≡
```
procedure :: get_process_name => process_instance_get_process_name
```

⟨*Instances: procedures*⟩+≡
```
function process_instance_get_process_name (instance) result (name)
  class(process_instance_t), intent(in) :: instance
  type(string_t) :: name
  name = instance%process%get_id ()
end function process_instance_get_process_name
```

Particle sets

Here we provide two procedures that convert the process instance from/to a particle set. The conversion applies to the trace evaluator which has no quantum-number information, thus it involves only the momenta and the parent-child relations. We keep virtual particles.

Nevertheless, it is possible to reconstruct the complete structure from a particle set. The reconstruction implies a re-evaluation of the structure function and matrix-element codes.

The `i_term` index is needed for both input and output, to select among different active trace evaluators.

In both cases, the `instance` object must be properly initialized.

NB: The `recover_beams` option should be used only when the particle set originates from an external event file, and the user has asked for it. It should be switched off when reading from raw event file.

⟨*Instances: process instance: TBP*⟩+≡
```
procedure :: get_trace => process_instance_get_trace
procedure :: set_trace => process_instance_set_trace
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_get_trace (instance, pset, i_term)
  class(process_instance_t), intent(in), target :: instance
  type(particle_set_t), intent(out) :: pset
  integer, intent(in) :: i_term
  type(interaction_t), pointer :: int
  logical :: ok
  int => instance%get_trace_int_ptr (i_term)
  call pset%init (ok, int, int, FM_IGNORE_HELICITY, &
       [0._default, 0._default], .false., .true.)
end subroutine process_instance_get_trace

subroutine process_instance_set_trace &
     (instance, pset, i_term, recover_beams, check_match)
  class(process_instance_t), intent(inout), target :: instance
  type(particle_set_t), intent(in) :: pset
  integer, intent(in) :: i_term
  logical, intent(in), optional :: recover_beams, check_match
  type(interaction_t), pointer :: int
  integer :: n_in
  int => instance%get_trace_int_ptr (i_term)
  n_in = instance%process%get_n_in ()
  call pset%fill_interaction (int, n_in, &
       recover_beams = recover_beams, &
       check_match = check_match, &
       state_flv = instance%get_state_flv (i_term))
end subroutine process_instance_set_trace
```

This procedure allows us to override any QCD setting of the WHIZARD process and directly set the coupling value that comes together with a particle set.

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: set_alpha_qcd_forced => process_instance_set_alpha_qcd_forced
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_set_alpha_qcd_forced (instance, i_term, alpha_qcd)
  class(process_instance_t), intent(inout) :: instance
  integer, intent(in) :: i_term
  real(default), intent(in) :: alpha_qcd
  call instance%term(i_term)%set_alpha_qcd_forced (alpha_qcd)
end subroutine process_instance_set_alpha_qcd_forced
```

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: has_nlo_component => process_instance_has_nlo_component
```

⟨*Instances: procedures*⟩+≡

```
function process_instance_has_nlo_component (instance) result (nlo)
  class(process_instance_t), intent(in) :: instance
  logical :: nlo
  nlo = instance%process%is_nlo_calculation ()
end function process_instance_has_nlo_component
```

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: keep_failed_events => process_instance_keep_failed_events
```

⟨*Instances: procedures*⟩+≡

```
function process_instance_keep_failed_events (instance) result (keep)
  logical :: keep
  class(process_instance_t), intent(in) :: instance
  keep = instance%mci_work(instance%i_mci)%keep_failed_events
end function process_instance_keep_failed_events
```

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: get_term_indices => process_instance_get_term_indices
```

⟨*Instances: procedures*⟩+≡

```
function process_instance_get_term_indices (instance, nlo_type) result (i_term)
  integer, dimension(:), allocatable :: i_term
  class(process_instance_t), intent(in) :: instance
  integer :: nlo_type
  allocate (i_term (count (instance%term%nlo_type == nlo_type)))
  i_term = pack (instance%term%get_i_term_global (), instance%term%nlo_type == nlo_type)
end function process_instance_get_term_indices
```

⟨*Instances: process instance: TBP*⟩+≡

```
procedure :: setup_blha_helicities => process_instance_setup_blha_helicities
```

⟨*Instances: procedures*⟩+≡

```
subroutine process_instance_setup_blha_helicities (instance)
  class(process_instance_t), intent(inout), target :: instance
  integer :: i, i_component, n_sub, n_flv
  type(vector4_t), dimension(:), allocatable :: p
  type(core_manager_t), pointer :: cm => null ()

  cm => instance%process%get_core_manager_ptr ()
  do i = 1, cm%n_cores
     select type (core => cm%cores(i)%core)
     class is (prc_blha_t)
        if (cm%nlo_type(i) /= NLO_SUBTRACTION .and. &
              core%includes_polarization ()) then
           i_component = cm%i_core_to_first_i_component (i)
           if (instance%process%component_can_be_integrated (i_component)) then
              call instance%transfer_helicities (i_component, core)
              call instance%get_phase_space_point (i_component, i, p)
              call core%warmup_helicities (p)
           end if
        else
           select type (pcm => instance%process%get_pcm_ptr ())
           type is (pcm_nlo_t)
              n_flv = pcm%get_n_flv_born ()
              n_sub = pcm%get_n_sub ()
           class default
              n_flv = core%data%n_flv
              n_sub = 0
           end select
           call core%set_helicity_list_trivial (n_flv, n_sub)
        end if
     end select
  end do
```

```
    end subroutine process_instance_setup_blha_helicities
```

⟨*Instances: process instance: TBP*⟩+≡

```
    procedure :: get_boost_to_lab => process_instance_get_boost_to_lab
```

⟨*Instances: procedures*⟩+≡

```
    function process_instance_get_boost_to_lab (instance, i_term) result (lt)
      type(lorentz_transformation_t) :: lt
      class(process_instance_t), intent(in) :: instance
      integer, intent(in) :: i_term
      lt = instance%term(i_term)%get_boost_to_lab ()
    end function process_instance_get_boost_to_lab
```

⟨*Instances: process instance: TBP*⟩+≡

```
    procedure :: get_boost_to_cms => process_instance_get_boost_to_cms
```

⟨*Instances: procedures*⟩+≡

```
    function process_instance_get_boost_to_cms (instance, i_term) result (lt)
      type(lorentz_transformation_t) :: lt
      class(process_instance_t), intent(in) :: instance
      integer, intent(in) :: i_term
      lt = instance%term(i_term)%get_boost_to_cms ()
    end function process_instance_get_boost_to_cms
```

⟨*Process: process: TBP*⟩+≡

```
    !procedure :: setup_recola => process_setup_recola
```

⟨*Process: procedures*⟩+≡

```
    !subroutine process_setup_recola (process)
    !  class(process_t), intent(inout) :: process
    !  integer :: i, i_recola
    !  i_recola = 0
    !  do i = 1, process%cm%n_cores
    !     select type (core => process%cm%cores(i)%core)
    !     type is (prc_recola_t)
    !        if (process%cm%nlo_type(i) == NLO_VIRTUAL) then
    !           call core%set_nlo ()
    !        end if
    !        call core%register_processes (i_recola)
    !        call core%replace_helicity_and_color_arrays ()
    !     end select
    !  end do
    !end subroutine process_setup_recola
```

⟨*Instances: process instance: TBP*⟩+≡

```
    procedure :: is_cm_frame => process_instance_is_cm_frame
```

⟨*Instances: procedures*⟩+≡

```
    function process_instance_is_cm_frame (instance, i_term) result (cm_frame)
      logical :: cm_frame
      class(process_instance_t), intent(in) :: instance
      integer, intent(in) :: i_term
      cm_frame = instance%term(i_term)%k_term%phs%is_cm_frame ()
    end function process_instance_is_cm_frame
```

The `pacify` subroutine has the purpose of setting numbers to zero which are (by comparing with a `tolerance` parameter) considered equivalent with zero. We do this in some unit tests. Here, we a apply this to the phase space subobject of the process instance.

⟨*Instances: public*⟩+≡
```
public :: pacify
```

⟨*Instances: interfaces*⟩≡
```
interface pacify
   module procedure pacify_process_instance
end interface pacify
```

⟨*Instances: procedures*⟩+≡
```
subroutine pacify_process_instance (instance)
  type(process_instance_t), intent(inout) :: instance
  integer :: i
  do i = 1, size (instance%term)
     call pacify (instance%term(i)%k_term%phs)
  end do
end subroutine pacify_process_instance
```

30.12 Unit tests

Test module, followed by the corresponding implementation module.

⟨`processes_ut.f90`⟩≡
```
⟨File header⟩

module processes_ut
  use unit_tests
  use processes_uti

⟨Standard module head⟩

⟨Processes: public test⟩

⟨Processes: public test auxiliary⟩

contains

⟨Processes: test driver⟩

end module processes_ut
```

⟨`processes_uti.f90`⟩≡
```
⟨File header⟩

module processes_uti

⟨Use kinds⟩
⟨Use strings⟩
  use format_utils, only: write_separator
  use constants, only: TWOPI4
```

```
  use physics_defs, only: CONV
  use os_interface
  use sm_qcd
  use lorentz
  use pdg_arrays
  use model_data
  use var_base, only: vars_t
  use model_testbed, only: prepare_model
  use particle_specifiers, only: new_prt_spec
  use flavors
  use interactions, only: reset_interaction_counter
  use particles
  use rng_base
  use mci_base
  use mci_midpoint
  use sf_mappings
  use sf_base
  use phs_base
  use phs_single
  use phs_forests, only: syntax_phs_forest_init, syntax_phs_forest_final
  use phs_wood, only: phs_wood_config_t
  use resonances, only: resonance_history_t
  use process_constants
  use prc_core_def, only: prc_core_def_t
  use prc_core
  use prc_test, only: prc_test_create_library
  use prc_template_me, only: template_me_def_t
  use process_libraries
  use prc_test_core

  use process_counter
  use process_config, only: process_term_t
  use process, only: process_t
  use instances, only: process_instance_t

  use rng_base_ut, only: rng_test_factory_t
  use sf_base_ut, only: sf_test_data_t
  use mci_base_ut, only: mci_test_t
  use phs_base_ut, only: phs_test_config_t
```

⟨*Standard module head*⟩

⟨*Processes: public test auxiliary*⟩

⟨*Processes: test declarations*⟩

```
contains
```

⟨*Processes: tests*⟩

⟨*Processes: test auxiliary*⟩

```
end module processes_uti
```

API: driver for the unit tests below.

⟨*Processes: public test*⟩≡
```
public :: processes_test
```

⟨*Processes: test driver*⟩≡
```
subroutine processes_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨Processes: execute tests⟩
end subroutine processes_test
```

Write an empty process object

The most trivial test is to write an uninitialized process object.

⟨*Processes: execute tests*⟩≡
```
call test (processes_1, "processes_1", &
     "write an empty process object", &
     u, results)
```

⟨*Processes: test declarations*⟩≡
```
public :: processes_1
```

⟨*Processes: tests*⟩≡
```
subroutine processes_1 (u)
  integer, intent(in) :: u
  type(process_t) :: process

  write (u, "(A)")  "* Test output: processes_1"
  write (u, "(A)")  "*   Purpose: display an empty process object"
  write (u, "(A)")

  call process%write (.false., u)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: processes_1"

end subroutine processes_1
```

Initialize a process object

Initialize a process and display it.

⟨*Processes: execute tests*⟩+≡
```
call test (processes_2, "processes_2", &
     "initialize a simple process object", &
     u, results)
```

⟨*Processes: test declarations*⟩+≡
```
public :: processes_2
```

⟨*Processes: tests*⟩+≡

```
subroutine processes_2 (u)
  integer, intent(in) :: u
  type(process_library_t), target :: lib
  type(string_t) :: libname
  type(string_t) :: procname
  type(string_t) :: run_id
  type(os_data_t) :: os_data
  type(qcd_t) :: qcd
  class(rng_factory_t), allocatable :: rng_factory
  class(model_data_t), pointer :: model
  type(process_t), allocatable :: process
  class(mci_t), allocatable :: mci_template
  class(phs_config_t), allocatable :: phs_config_template

  write (u, "(A)")  "* Test output: processes_2"
  write (u, "(A)")  "*   Purpose: initialize a simple process object"
  write (u, "(A)")

  write (u, "(A)")  "* Build and load a test library with one process"
  write (u, "(A)")

  libname = "processes2"
  procname = libname
  run_id = "run2"
  call os_data_init (os_data)
  allocate (rng_test_factory_t :: rng_factory)
  call prc_test_create_library (libname, lib)

  write (u, "(A)")  "* Initialize a process object"
  write (u, "(A)")

  allocate (model)
  call model%init_test ()

  allocate (process)
  call process%init (procname, run_id, &
       lib, os_data, qcd, rng_factory, model)
  call process%setup_test_cores ()

  allocate (phs_test_config_t :: phs_config_template)
  call process%init_component &
     (1, .true., mci_template, phs_config_template)

  call process%setup_mci ()

  call process%write (.false., u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call process%final ()
  deallocate (process)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_2"

  end subroutine processes_2
```

Compute a trivial matrix element

Initialize a process, retrieve some information and compute a matrix element.

We use the same trivial process as for the previous test. All momentum and state dependence is trivial, so we just test basic functionality.

⟨*Processes: execute tests*⟩+≡

```
  call test (processes_3, "processes_3", &
       "retrieve a trivial matrix element", &
       u, results)
```

⟨*Processes: test declarations*⟩+≡

```
  public :: processes_3
```

⟨*Processes: tests*⟩+≡

```
  subroutine processes_3 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(qcd_t) :: qcd
    class(rng_factory_t), allocatable :: rng_factory
    class(model_data_t), pointer :: model
    type(process_t), allocatable :: process
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    type(process_constants_t) :: data
    type(vector4_t), dimension(:), allocatable :: p

    write (u, "(A)")  "* Test output: processes_3"
    write (u, "(A)")  "*   Purpose: create a process &
         &and compute a matrix element"
    write (u, "(A)")

    write (u, "(A)")  "* Build and load a test library with one process"
    write (u, "(A)")

    libname = "processes3"
    procname = libname
    run_id = "run3"
    call os_data_init (os_data)
    allocate (rng_test_factory_t :: rng_factory)
    call prc_test_create_library (libname, lib)

    allocate (model)
    call model%init_test ()
```

```
allocate (process)
call process%init (procname, run_id, &
     lib, os_data, qcd, rng_factory, model)
call process%setup_test_cores ()

allocate (mci_test_t :: mci_template)
select type (mci_template)
type is (mci_test_t)
   call mci_template%set_dimensions (2, 2)
   call mci_template%set_divisions (100)
end select
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
   (1, .true., mci_template, phs_config_template)

write (u, "(A)")  "* Return the number of process components"
write (u, "(A)")

write (u, "(A,I0)")  "n_components = ", process%get_n_components ()

write (u, "(A)")
write (u, "(A)")  "* Return the number of flavor states"
write (u, "(A)")

data = process%get_constants (1)

write (u, "(A,I0)")  "n_flv(1) = ", data%n_flv

write (u, "(A)")
write (u, "(A)")  "* Return the first flavor state"
write (u, "(A)")

write (u, "(A,4(1x,I0))")  "flv_state(1) =", data%flv_state (:,1)

write (u, "(A)")
write (u, "(A)")  "* Set up kinematics &
     &[arbitrary, the matrix element is constant]"

allocate (p (4))

write (u, "(A)")
write (u, "(A)")  "* Retrieve the matrix element"
write (u, "(A)")


write (u, "(A,F5.3,' + ',F5.3,' I')")  "me (1, p, 1, 1, 1) = ", &
     process%compute_amplitude (1, 1, 1, p, 1, 1, 1)


write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call process%final ()
deallocate (process)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_3"

  end subroutine processes_3
```

Generate a process instance

Initialize a process and process instance, choose a sampling point and fill the process instance.

We use the same trivial process as for the previous test. All momentum and state dependence is trivial, so we just test basic functionality.

⟨*Processes: execute tests*⟩+≡

```
  call test (processes_4, "processes_4", &
       "create and fill a process instance (partonic event)", &
       u, results)
```

⟨*Processes: test declarations*⟩+≡

```
  public :: processes_4
```

⟨*Processes: tests*⟩+≡

```
  subroutine processes_4 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(qcd_t) :: qcd
    class(rng_factory_t), allocatable :: rng_factory
    class(model_data_t), pointer :: model
    type(process_t), allocatable, target :: process
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    real(default) :: sqrts
    type(process_instance_t), allocatable, target :: process_instance
    type(particle_set_t) :: pset

    write (u, "(A)")  "* Test output: processes_4"
    write (u, "(A)")  "*   Purpose: create a process &
         &and fill a process instance"
    write (u, "(A)")

    write (u, "(A)")  "* Build and initialize a test process"
    write (u, "(A)")

    libname = "processes4"
    procname = libname
    run_id = "run4"
    call os_data_init (os_data)
    allocate (rng_test_factory_t :: rng_factory)
    call prc_test_create_library (libname, lib)
```

```
call reset_interaction_counter ()

allocate (model)
call model%init_test ()

allocate (process)
call process%init (procname, run_id, &
     lib, os_data, qcd, rng_factory, model)

call process%setup_test_cores ()
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
   (1, .true., mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

sqrts = 1000
call process%setup_beams_sqrts (sqrts, i_core = 1)
call process%configure_phs ()
call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_terms ()
call process%write (.false., u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Inject a set of random numbers"
write (u, "(A)")

call process_instance%choose_mci (1)
call process_instance%set_mcpar ([0._default, 0._default])
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Set up hard kinematics"
write (u, "(A)")

call process_instance%select_channel (1)
call process_instance%compute_seed_kinematics ()
call process_instance%compute_hard_kinematics ()
call process_instance%compute_eff_kinematics ()
call process_instance%evaluate_expressions ()
call process_instance%compute_other_channels ()
```

```
write (u, "(A)")  "* Evaluate matrix element and square"
write (u, "(A)")

call process_instance%evaluate_trace ()
call process_instance%write (u)

call process_instance%get_trace (pset, 1)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

call write_separator (u)
call pset%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Recover process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)
call process_instance%choose_mci (1)
call process_instance%set_trace (pset, 1, check_match = .false.)

call process_instance%activate ()
process_instance%evaluation_status = STAT_EFF_KINEMATICS
call process_instance%recover_hard_kinematics (i_term = 1)
call process_instance%recover_seed_kinematics (i_term = 1)
call process_instance%select_channel (1)
call process_instance%recover_mcpar (i_term = 1)

call process_instance%compute_seed_kinematics (skip_term = 1)
call process_instance%compute_hard_kinematics (skip_term = 1)
call process_instance%compute_eff_kinematics (skip_term = 1)

call process_instance%evaluate_expressions ()
call process_instance%compute_other_channels (skip_term = 1)
call process_instance%evaluate_trace ()
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call pset%final ()
call process_instance%final ()
deallocate (process_instance)

call process%final ()
deallocate (process)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_4"

  end subroutine processes_4
```

Structure function configuration

Configure structure functions (multi-channel) in a process object.

⟨*Processes: execute tests*⟩+≡

```
  call test (processes_7, "processes_7", &
       "process configuration with structure functions", &
       u, results)
```

⟨*Processes: test declarations*⟩+≡

```
  public :: processes_7
```

⟨*Processes: tests*⟩+≡

```
  subroutine processes_7 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(qcd_t) :: qcd
    class(rng_factory_t), allocatable :: rng_factory
    class(model_data_t), pointer :: model
    type(process_t), allocatable, target :: process
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    real(default) :: sqrts
    type(pdg_array_t) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    type(sf_config_t), dimension(:), allocatable :: sf_config
    type(sf_channel_t), dimension(2) :: sf_channel

    write (u, "(A)")  "* Test output: processes_7"
    write (u, "(A)")  "*   Purpose: initialize a process with &
         &structure functions"
    write (u, "(A)")

    write (u, "(A)")  "* Build and initialize a process object"
    write (u, "(A)")

    libname = "processes7"
    procname = libname
    run_id = "run7"
    call os_data_init (os_data)
    allocate (rng_test_factory_t :: rng_factory)
    call prc_test_create_library (libname, lib)

    allocate (model)
    call model%init_test ()
```

```
  allocate (process)
  call process%init (procname, run_id, &
       lib, os_data, qcd, rng_factory, model)

  call process%setup_test_cores ()
  allocate (phs_test_config_t :: phs_config_template)
  call process%init_component &
     (1, .true., mci_template, phs_config_template)

  write (u, "(A)")  "* Set beam, structure functions, and mappings"
  write (u, "(A)")

  sqrts = 1000
  call process%setup_beams_sqrts (sqrts, i_core = 1)
  call process%configure_phs ()

  pdg_in = 25
  allocate (sf_test_data_t :: data)
  select type (data)
  type is (sf_test_data_t)
     model => process%get_model_ptr ()
     call data%init (model, pdg_in)
  end select

  allocate (sf_config (2))
  call sf_config(1)%init ([1], data)
  call sf_config(2)%init ([2], data)
  call process%init_sf_chain (sf_config)
  deallocate (sf_config)

  call process%test_allocate_sf_channels (3)

  call sf_channel(1)%init (2)
  call sf_channel(1)%activate_mapping ([1,2])
  call process%set_sf_channel (2, sf_channel(1))

  call sf_channel(2)%init (2)
  call sf_channel(2)%set_s_mapping ([1,2])
  call process%set_sf_channel (3, sf_channel(2))

  call process%setup_mci ()

  call process%write (.false., u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call process%final ()
  deallocate (process)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: processes_7"

end subroutine processes_7
```

Evaluating a process with structure function

Configure structure functions (single-channel) in a process object, create an instance, compute kinematics and evaluate.

Note the order of operations when setting up structure functions and phase space. The beams are first, they determine the `sqrts` value. We can also set up the chain of structure functions. We then configure the phase space. From this, we can obtain information about special configurations (resonances, etc.), which we need for allocating the possible structure-function channels (parameterizations and mappings). Finally, we match phase-space channels onto structure-function channels.

In the current example, this matching is trivial; we only have one structure-function channel.

⟨*Processes: execute tests*⟩+≡
```
call test (processes_8, "processes_8", &
     "process evaluation with structure functions", &
     u, results)
```

⟨*Processes: test declarations*⟩+≡
```
public :: processes_8
```

⟨*Processes: tests*⟩+≡
```
subroutine processes_8 (u)
  integer, intent(in) :: u
  type(process_library_t), target :: lib
  type(string_t) :: libname
  type(string_t) :: procname
  type(string_t) :: run_id
  type(os_data_t) :: os_data
  type(qcd_t) :: qcd
  class(rng_factory_t), allocatable :: rng_factory
  class(model_data_t), pointer :: model
  type(process_t), allocatable, target :: process
  class(mci_t), allocatable :: mci_template
  class(phs_config_t), allocatable :: phs_config_template
  real(default) :: sqrts
  type(process_instance_t), allocatable, target :: process_instance
  type(pdg_array_t) :: pdg_in
  class(sf_data_t), allocatable, target :: data
  type(sf_config_t), dimension(:), allocatable :: sf_config
  type(sf_channel_t) :: sf_channel
  type(particle_set_t) :: pset

  write (u, "(A)")  "* Test output: processes_8"
  write (u, "(A)")  "*   Purpose: evaluate a process with &
       &structure functions"
  write (u, "(A)")

  write (u, "(A)")  "* Build and initialize a process object"
  write (u, "(A)")

  libname = "processes8"
```

```
procname = libname
run_id = "run8"
call os_data_init (os_data)
allocate (rng_test_factory_t :: rng_factory)
call prc_test_create_library (libname, lib)

call reset_interaction_counter ()

allocate (model)
call model%init_test ()

allocate (process)
call process%init (procname, run_id, &
     lib, os_data, qcd, rng_factory, model)

call process%setup_test_cores ()
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
   (1, .true., mci_template, phs_config_template)

write (u, "(A)")  "* Set beam, structure functions, and mappings"
write (u, "(A)")

sqrts = 1000
call process%setup_beams_sqrts (sqrts, i_core = 1)

pdg_in = 25
allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
   model => process%get_model_ptr ()
   call data%init (model, pdg_in)
end select

allocate (sf_config (2))
call sf_config(1)%init ([1], data)
call sf_config(2)%init ([2], data)
call process%init_sf_chain (sf_config)
deallocate (sf_config)

call process%configure_phs ()

call process%test_allocate_sf_channels (1)

call sf_channel%init (2)
call sf_channel%activate_mapping ([1,2])
call process%set_sf_channel (1, sf_channel)

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_mci ()
call process%setup_terms ()
```

```
call process%write (.false., u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)

write (u, "(A)")  "* Set up kinematics and evaluate"
write (u, "(A)")

call process_instance%choose_mci (1)
call process_instance%evaluate_sqme (1, &
     [0.8_default, 0.8_default, 0.1_default, 0.2_default])
call process_instance%write (u)

call process_instance%get_trace (pset, 1)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

call write_separator (u)
call pset%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Recover process instance"
write (u, "(A)")

call reset_interaction_counter (2)

allocate (process_instance)
call process_instance%init (process)

call process_instance%choose_mci (1)
call process_instance%set_trace (pset, 1, check_match = .false.)
call process_instance%recover &
     (channel = 1, i_term = 1, update_sqme = .true.)
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call pset%final ()

call process_instance%final ()
deallocate (process_instance)

call process%final ()
deallocate (process)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_8"

  end subroutine processes_8
```

Multi-channel phase space and structure function

This is an extension of the previous example. This time, we have two distinct structure-function channels which are matched to the two distinct phase-space channels.

⟨*Processes: execute tests*⟩+≡

```
  call test (processes_9, "processes_9", &
       "multichannel kinematics and structure functions", &
       u, results)
```

⟨*Processes: test declarations*⟩+≡

```
  public :: processes_9
```

⟨*Processes: tests*⟩+≡

```
  subroutine processes_9 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(qcd_t) :: qcd
    class(rng_factory_t), allocatable :: rng_factory
    class(model_data_t), pointer :: model
    type(process_t), allocatable, target :: process
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    real(default) :: sqrts
    type(process_instance_t), allocatable, target :: process_instance
    type(pdg_array_t) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    type(sf_config_t), dimension(:), allocatable :: sf_config
    type(sf_channel_t) :: sf_channel
    real(default), dimension(4) :: x_saved
    type(particle_set_t) :: pset

    write (u, "(A)")  "* Test output: processes_9"
    write (u, "(A)")  "*   Purpose: evaluate a process with &
         &structure functions"
    write (u, "(A)")  "*            in a multi-channel configuration"
    write (u, "(A)")

    write (u, "(A)")  "* Build and initialize a process object"
    write (u, "(A)")

    libname = "processes9"
    procname = libname
```

```
run_id = "run9"
call os_data_init (os_data)
allocate (rng_test_factory_t :: rng_factory)
call prc_test_create_library (libname, lib)

call reset_interaction_counter ()

allocate (model)
call model%init_test ()

allocate (process)
call process%init (procname, run_id, &
     lib, os_data, qcd, rng_factory, model)

call process%setup_test_cores ()
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
   (1, .true., mci_template, phs_config_template)

write (u, "(A)")  "* Set beam, structure functions, and mappings"
write (u, "(A)")

sqrts = 1000
call process%setup_beams_sqrts (sqrts, i_core = 1)

pdg_in = 25
allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
   model => process%get_model_ptr ()
   call data%init (model, pdg_in)
end select

allocate (sf_config (2))
call sf_config(1)%init ([1], data)
call sf_config(2)%init ([2], data)
call process%init_sf_chain (sf_config)
deallocate (sf_config)

call process%configure_phs ()

call process%test_allocate_sf_channels (2)

call sf_channel%init (2)
call process%set_sf_channel (1, sf_channel)

call sf_channel%init (2)
call sf_channel%activate_mapping ([1,2])
call process%set_sf_channel (2, sf_channel)

call process%test_set_component_sf_channel ([1, 2])

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")
```

```
call process%setup_mci ()
call process%setup_terms ()

call process%write (.false., u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)

write (u, "(A)")  "* Set up kinematics in channel 1 and evaluate"
write (u, "(A)")

call process_instance%choose_mci (1)
call process_instance%evaluate_sqme (1, &
     [0.8_default, 0.8_default, 0.1_default, 0.2_default])
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Extract MC input parameters"
write (u, "(A)")

write (u, "(A)")  "Channel 1:"
call process_instance%get_mcpar (1, x_saved)
write (u, "(2x,9(1x,F7.5))")  x_saved

write (u, "(A)")  "Channel 2:"
call process_instance%get_mcpar (2, x_saved)
write (u, "(2x,9(1x,F7.5))")  x_saved

write (u, "(A)")
write (u, "(A)")  "* Set up kinematics in channel 2 and evaluate"
write (u, "(A)")

call process_instance%evaluate_sqme (2, x_saved)
call process_instance%write (u)

call process_instance%get_trace (pset, 1)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Recover process instance for channel 2"
write (u, "(A)")

call reset_interaction_counter (2)

allocate (process_instance)
call process_instance%init (process)

call process_instance%choose_mci (1)
```

```
    call process_instance%set_trace (pset, 1, check_match = .false.)
    call process_instance%recover &
         (channel = 2, i_term = 1, update_sqme = .true.)
    call process_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call pset%final ()

    call process_instance%final ()
    deallocate (process_instance)

    call process%final ()
    deallocate (process)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_9"

  end subroutine processes_9
```

Event generation

Activate the MC integrator for the process object and use it to generate a single event. Note that the test integrator does not require integration in preparation for generating events.

⟨*Processes: execute tests*⟩+≡

```
  call test (processes_10, "processes_10", &
       "event generation", &
       u, results)
```

⟨*Processes: test declarations*⟩+≡

```
  public :: processes_10
```

⟨*Processes: tests*⟩+≡

```
  subroutine processes_10 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(qcd_t) :: qcd
    class(rng_factory_t), allocatable :: rng_factory
    class(model_data_t), pointer :: model
    type(process_t), allocatable, target :: process
    class(mci_t), allocatable :: mci_template
    class(mci_t), pointer :: mci
    class(phs_config_t), allocatable :: phs_config_template
    real(default) :: sqrts
    type(process_instance_t), allocatable, target :: process_instance

    write (u, "(A)")  "* Test output: processes_10"
```

```
write (u, "(A)")  "*   Purpose: generate events for a process without &
     &structure functions"
write (u, "(A)")  "*            in a multi-channel configuration"
write (u, "(A)")

write (u, "(A)")  "* Build and initialize a process object"
write (u, "(A)")

libname = "processes10"
procname = libname
run_id = "run10"
call os_data_init (os_data)
allocate (rng_test_factory_t :: rng_factory)
call prc_test_create_library (libname, lib)

call reset_interaction_counter ()

allocate (model)
call model%init_test ()

allocate (process)
call process%init (procname, run_id, &
     lib, os_data, qcd, rng_factory, model)

call process%setup_test_cores ()
allocate (mci_test_t :: mci_template)
select type (mci_template)
type is (mci_test_t);  call mci_template%set_divisions (100)
end select
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
   (1, .true., mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

sqrts = 1000
call process%setup_beams_sqrts (sqrts, i_core = 1)
call process%configure_phs ()

call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_terms ()
call process%write (.false., u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)
```

```
    write (u, "(A)")  "* Generate weighted event"
    write (u, "(A)")

    call process%test_get_mci_ptr (mci)
    select type (mci)
    type is (mci_test_t)
       ! This ensures that the next 'random' numbers are 0.3, 0.5, 0.7
       call mci%rng%init (3)
       ! Include the constant PHS factor in the stored maximum of the integrand
       call mci%set_max_factor (conv * twopi4 &
            / (2 * sqrt (lambda (sqrts **2, 125._default**2, 125._default**2))))
    end select

    call process_instance%generate_weighted_event (1)
    call process_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Generate unweighted event"
    write (u, "(A)")

    call process_instance%generate_unweighted_event (1)
    call process%test_get_mci_ptr (mci)
    select type (mci)
    type is (mci_test_t)
       write (u, "(A,I0)")  " Success in try ", mci%tries
       write (u, "(A)")
    end select

    call process_instance%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call process_instance%final ()
    deallocate (process_instance)

    call process%final ()
    deallocate (process)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_10"

  end subroutine processes_10
```

Integration

Activate the MC integrator for the process object and use it to integrate over phase space.

⟨*Processes: execute tests*⟩+≡

```
  call test (processes_11, "processes_11", &
       "integration", &
```

```
        u, results)
```

⟨*Processes: test declarations*⟩+≡

```
    public :: processes_11
```

⟨*Processes: tests*⟩+≡

```
    subroutine processes_11 (u)
      integer, intent(in) :: u
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      class(model_data_t), pointer :: model
      type(process_t), allocatable, target :: process
      class(mci_t), allocatable :: mci_template
      class(phs_config_t), allocatable :: phs_config_template
      real(default) :: sqrts
      type(process_instance_t), allocatable, target :: process_instance

      write (u, "(A)")  "* Test output: processes_11"
      write (u, "(A)")  "*   Purpose: integrate a process without &
           &structure functions"
      write (u, "(A)")  "*            in a multi-channel configuration"
      write (u, "(A)")

      write (u, "(A)")  "* Build and initialize a process object"
      write (u, "(A)")

      libname = "processes11"
      procname = libname
      run_id = "run11"
      call os_data_init (os_data)
      allocate (rng_test_factory_t :: rng_factory)
      call prc_test_create_library (libname, lib)

      call reset_interaction_counter ()

      allocate (model)
      call model%init_test ()

      allocate (process)
      call process%init (procname, run_id, &
           lib, os_data, qcd, rng_factory, model)

      call process%setup_test_cores ()
      allocate (mci_test_t :: mci_template)
      select type (mci_template)
      type is (mci_test_t)
         call mci_template%set_divisions (100)
      end select
      allocate (phs_test_config_t :: phs_config_template)
      call process%init_component &
```

```
      (1, .true., mci_template, phs_config_template)

   write (u, "(A)")  "* Prepare a trivial beam setup"
   write (u, "(A)")

   sqrts = 1000
   call process%setup_beams_sqrts (sqrts, i_core = 1)
   call process%configure_phs ()

   call process%setup_mci ()

   write (u, "(A)")  "* Complete process initialization"
   write (u, "(A)")

   call process%setup_terms ()
   call process%write (.false., u)

   write (u, "(A)")
   write (u, "(A)")  "* Create a process instance"
   write (u, "(A)")

   allocate (process_instance)
   call process_instance%init (process)

   write (u, "(A)")  "* Integrate with default test parameters"
   write (u, "(A)")

   call process_instance%integrate (1, n_it=1, n_calls=10000)
   call process%final_integration (1)

   call process%write (.false., u)

   write (u, "(A)")
   write (u, "(A,ES13.7)")  " Integral divided by phs factor = ", &
        process%get_integral (1) &
        / process_instance%term(1)%k_term%phs_factor

   write (u, "(A)")
   write (u, "(A)")  "* Cleanup"

   call process_instance%final ()
   deallocate (process_instance)

   call process%final ()
   deallocate (process)

   write (u, "(A)")
   write (u, "(A)")  "* Test output end: processes_11"

end subroutine processes_11
```

Complete events

For the purpose of simplifying further tests, we implement a convenience routine that initializes a process and prepares a single event. This is a wrapup of the test processes_10.

The procedure is re-exported by the processes_ut module.

⟨*Processes: public test auxiliary*⟩≡

```
public :: prepare_test_process
```

⟨*Processes: test auxiliary*⟩≡

```
subroutine prepare_test_process (process, process_instance, model)
  type(process_t), intent(out), target :: process
  type(process_instance_t), intent(out), target :: process_instance
  class(model_data_t), intent(in), target :: model
  type(process_library_t), target :: lib
  type(string_t) :: libname
  type(string_t) :: procname
  type(string_t) :: run_id
  type(os_data_t) :: os_data
  class(model_data_t), pointer :: process_model
  type(qcd_t) :: qcd
  class(rng_factory_t), allocatable :: rng_factory
  class(mci_t), allocatable :: mci_template
  class(mci_t), pointer :: mci
  class(phs_config_t), allocatable :: phs_config_template
  real(default) :: sqrts
  libname = "processes_test"
  procname = libname
  run_id = "run_test"
  call os_data_init (os_data)
  allocate (rng_test_factory_t :: rng_factory)
  call prc_test_create_library (libname, lib)
  call reset_interaction_counter ()
  allocate (process_model)
  call process_model%init (model%get_name (), &
       model%get_n_real (), &
       model%get_n_complex (), &
       model%get_n_field (), &
       model%get_n_vtx ())
  call process_model%copy_from (model)
  call process%init (procname, run_id, &
       lib, os_data, qcd, rng_factory, process_model)
  call process%setup_test_cores ()
  allocate (mci_test_t :: mci_template)
  select type (mci_template)
  type is (mci_test_t);  call mci_template%set_divisions (100)
  end select
  allocate (phs_test_config_t :: phs_config_template)
  call process%init_component &
     (1, .true., mci_template, phs_config_template)
  sqrts = 1000
  call process%setup_beams_sqrts (sqrts, i_core = 1)
  call process%configure_phs ()
  call process%setup_mci ()
  call process%setup_terms ()
```

```
    call process_instance%init (process)
    call process%test_get_mci_ptr (mci)
    select type (mci)
    type is (mci_test_t)
       ! This ensures that the next 'random' numbers are 0.3, 0.5, 0.7
       call mci%rng%init (3)
       ! Include the constant PHS factor in the stored maximum of the integrand
       call mci%set_max_factor (conv * twopi4 &
            / (2 * sqrt (lambda (sqrts **2, 125._default**2, 125._default**2))))
    end select
    call process%nullify_library_pointer ()  ! avoid dangling pointer
  end subroutine prepare_test_process
```

Here we do the cleanup of the process and process instance emitted by the previous routine.

⟨*Processes: public test auxiliary*⟩+≡

```
  public :: cleanup_test_process
```

⟨*Processes: test auxiliary*⟩+≡

```
  subroutine cleanup_test_process (process, process_instance)
    type(process_t), intent(inout) :: process
    type(process_instance_t), intent(inout) :: process_instance
    call process_instance%final ()
    call process%final ()
  end subroutine cleanup_test_process
```

This is the actual test. Prepare the test process and event, fill all evaluators, and display the results. Use a particle set as temporary storage, read kinematics and recalculate the event.

⟨*Processes: execute tests*⟩+≡

```
  call test (processes_12, "processes_12", &
       "event post-processing", &
       u, results)
```

⟨*Processes: test declarations*⟩+≡

```
  public :: processes_12
```

⟨*Processes: tests*⟩+≡

```
  subroutine processes_12 (u)
    integer, intent(in) :: u
    type(process_t), allocatable, target :: process
    type(process_instance_t), allocatable, target :: process_instance
    type(particle_set_t) :: pset
    type(model_data_t), target :: model

    write (u, "(A)")  "* Test output: processes_12"
    write (u, "(A)")  "*   Purpose: generate a complete partonic event"
    write (u, "(A)")

    call model%init_test ()

    write (u, "(A)")  "* Build and initialize process and process instance &
         &and generate event"
    write (u, "(A)")
```

```
  allocate (process)
  allocate (process_instance)
  call prepare_test_process (process, process_instance, model)
  call process_instance%setup_event_data (i_core = 1)

  call process%prepare_simulation (1)
  call process_instance%init_simulation (1)
  call process_instance%generate_weighted_event (1)
  call process_instance%evaluate_event_data ()

  call process_instance%write (u)

  call process_instance%get_trace (pset, 1)

  call process_instance%final_simulation (1)
  call process_instance%final ()
  deallocate (process_instance)

  write (u, "(A)")
  write (u, "(A)")  "* Recover kinematics and recalculate"
  write (u, "(A)")

  call reset_interaction_counter (2)

  allocate (process_instance)
  call process_instance%init (process)
  call process_instance%setup_event_data ()

  call process_instance%choose_mci (1)
  call process_instance%set_trace (pset, 1, check_match = .false.)
  call process_instance%recover &
       (channel = 1, i_term = 1, update_sqme = .true.)

  call process_instance%recover_event ()
  call process_instance%evaluate_event_data ()

  call process_instance%write (u)


  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call cleanup_test_process (process, process_instance)
  deallocate (process_instance)
  deallocate (process)

  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: processes_12"

end subroutine processes_12
```

Colored interaction

This test specifically checks the transformation of process data (flavor, helicity, and color) into an interaction in a process term.

We use the `test_t` process core (which has no nontrivial particles), but call only the `is_allowed` method, which always returns true.

⟨*Processes: execute tests*⟩+≡

```
    call test (processes_13, "processes_13", &
         "colored interaction", &
         u, results)
```

⟨*Processes: test declarations*⟩+≡

```
    public :: processes_13
```

⟨*Processes: tests*⟩+≡

```
    subroutine processes_13 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(model_data_t), target :: model
      type(process_term_t) :: term
      class(prc_core_t), allocatable :: core

      write (u, "(A)")  "* Test output: processes_13"
      write (u, "(A)")  "*   Purpose: initialized a colored interaction"
      write (u, "(A)")

      write (u, "(A)")  "* Set up a process constants block"
      write (u, "(A)")

      call os_data_init (os_data)
      call model%init_sm_test ()
!     call model_list%read_model (var_str ("QCD"), var_str ("QCD.mdl"), &
!          os_data, model)
      allocate (test_t :: core)

      associate (data => term%data)
        data%n_in = 2
        data%n_out = 3
        data%n_flv = 2
        data%n_hel = 2
        data%n_col = 2
        data%n_cin = 2

        allocate (data%flv_state (5, 2))
        data%flv_state (:,1) = [ 1, 21, 1, 21, 21]
        data%flv_state (:,2) = [ 2, 21, 2, 21, 21]

        allocate (data%hel_state (5, 2))
        data%hel_state (:,1) = [1, 1, 1, 1, 0]
        data%hel_state (:,2) = [1,-1, 1,-1, 0]

        allocate (data%col_state (2, 5, 2))
        data%col_state (:,:,1) = &
             reshape ([[1, 0], [2,-1], [3, 0], [2,-3], [0,0]], [2,5])
        data%col_state (:,:,2) = &
```

```
        reshape ([[1, 0], [2,-3], [3, 0], [2,-1], [0,0]], [2,5])

      allocate (data%ghost_flag (5, 2))
      data%ghost_flag(1:4,:) = .false.
      data%ghost_flag(5,:) = .true.

    end associate

    write (u, "(A)")  "* Set up the interaction"
    write (u, "(A)")

    call reset_interaction_counter ()
    call term%setup_interaction (core, model)
    call term%int%basic_write (u)

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_13"
  end subroutine processes_13
```

MD5 sums

Configure a process with structure functions (multi-channel) and compute MD5 sums

⟨*Processes: execute tests*⟩+≡

```
  call test (processes_14, "processes_14", &
       "process configuration and MD5 sum", &
       u, results)
```

⟨*Processes: test declarations*⟩+≡

```
  public :: processes_14
```

⟨*Processes: tests*⟩+≡

```
  subroutine processes_14 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(qcd_t) :: qcd
    class(rng_factory_t), allocatable :: rng_factory
    class(model_data_t), pointer :: model
    type(process_t), allocatable, target :: process
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    real(default) :: sqrts
    type(pdg_array_t) :: pdg_in
    class(sf_data_t), allocatable, target :: data
    type(sf_config_t), dimension(:), allocatable :: sf_config
    type(sf_channel_t), dimension(3) :: sf_channel
```

```
write (u, "(A)")  "* Test output: processes_14"
write (u, "(A)")  "*   Purpose: initialize a process with &
     &structure functions"
write (u, "(A)")  "*            and compute MD5 sum"
write (u, "(A)")

write (u, "(A)")  "* Build and initialize a process object"
write (u, "(A)")

libname = "processes7"
procname = libname
run_id = "run7"
call os_data_init (os_data)
allocate (rng_test_factory_t :: rng_factory)
call prc_test_create_library (libname, lib)
call lib%compute_md5sum ()

allocate (model)
call model%init_test ()

allocate (process)
call process%init (procname, run_id, &
     lib, os_data, qcd, rng_factory, model)

call process%setup_test_cores ()
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
   (1, .true., mci_template, phs_config_template)

write (u, "(A)")  "* Set beam, structure functions, and mappings"
write (u, "(A)")

sqrts = 1000
call process%setup_beams_sqrts (sqrts, i_core = 1)
call process%configure_phs ()

pdg_in = 25
allocate (sf_test_data_t :: data)
select type (data)
type is (sf_test_data_t)
   model => process%get_model_ptr ()
   call data%init (model, pdg_in)
end select

call process%test_allocate_sf_channels (3)

allocate (sf_config (2))
call sf_config(1)%init ([1], data)
call sf_config(2)%init ([2], data)
call process%init_sf_chain (sf_config)
deallocate (sf_config)

call sf_channel(1)%init (2)
call process%set_sf_channel (1, sf_channel(1))
```

```
    call sf_channel(2)%init (2)
    call sf_channel(2)%activate_mapping ([1,2])
    call process%set_sf_channel (2, sf_channel(2))

    call sf_channel(3)%init (2)
    call sf_channel(3)%set_s_mapping ([1,2])
    call process%set_sf_channel (3, sf_channel(3))

    call process%setup_mci ()

    call process%compute_md5sum ()

    call process%write (.false., u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call process%final ()
    deallocate (process)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_14"

  end subroutine processes_14
```

Decay Process Evaluation

Initialize an evaluate a decay process.

⟨*Processes: execute tests*⟩+≡

```
  call test (processes_15, "processes_15", &
       "decay process", &
       u, results)
```

⟨*Processes: test declarations*⟩+≡

```
  public :: processes_15
```

⟨*Processes: tests*⟩+≡

```
  subroutine processes_15 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(qcd_t) :: qcd
    class(rng_factory_t), allocatable :: rng_factory
    class(model_data_t), pointer :: model
    type(process_t), allocatable, target :: process
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    type(process_instance_t), allocatable, target :: process_instance
    type(particle_set_t) :: pset
```

```
write (u, "(A)")  "* Test output: processes_15"
write (u, "(A)")  "*   Purpose: initialize a decay process object"
write (u, "(A)")

write (u, "(A)")  "* Build and load a test library with one process"
write (u, "(A)")

libname = "processes15"
procname = libname
run_id = "run15"
call os_data_init (os_data)
allocate (rng_test_factory_t :: rng_factory)
call prc_test_create_library (libname, lib, scattering = .false., &
     decay = .true.)

allocate (model)
call model%init_test ()
call model%set_par (var_str ("ff"), 0.4_default)
call model%set_par (var_str ("mf"), &
     model%get_real (var_str ("ff")) * model%get_real (var_str ("ms")))

write (u, "(A)")  "* Initialize a process object"
write (u, "(A)")

allocate (process)
call process%init (procname, run_id, &
     lib, os_data, qcd, rng_factory, model)

call process%setup_test_cores ()
allocate (phs_single_config_t :: phs_config_template)
call process%init_component &
   (1, .true., mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

call process%setup_beams_decay (i_core = 1)
call process%configure_phs ()
call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_terms ()
call process%write (.false., u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

call reset_interaction_counter (3)

allocate (process_instance)
call process_instance%init (process)
```

```
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Inject a set of random numbers"
write (u, "(A)")

call process_instance%choose_mci (1)
call process_instance%set_mcpar ([0._default, 0._default])
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Set up hard kinematics"
write (u, "(A)")

call process_instance%select_channel (1)
call process_instance%compute_seed_kinematics ()
call process_instance%compute_hard_kinematics ()

write (u, "(A)")  "* Evaluate matrix element and square"
write (u, "(A)")

call process_instance%compute_eff_kinematics ()
call process_instance%evaluate_expressions ()
call process_instance%compute_other_channels ()
call process_instance%evaluate_trace ()
call process_instance%write (u)

call process_instance%get_trace (pset, 1)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

call write_separator (u)
call pset%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Recover process instance"
write (u, "(A)")

call reset_interaction_counter (3)

allocate (process_instance)
call process_instance%init (process)
call process_instance%choose_mci (1)
call process_instance%set_trace (pset, 1, check_match = .false.)
call process_instance%recover (1, 1, .true.)
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"
```

```
    call pset%final ()
    call process_instance%final ()
    deallocate (process_instance)

    call process%final ()
    deallocate (process)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_15"

  end subroutine processes_15
```

Integration: decay

Activate the MC integrator for the decay object and use it to integrate over phase space.

⟨*Processes: execute tests*⟩+≡

```
  call test (processes_16, "processes_16", &
       "decay integration", &
       u, results)
```

⟨*Processes: test declarations*⟩+≡

```
  public :: processes_16
```

⟨*Processes: tests*⟩+≡

```
  subroutine processes_16 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(qcd_t) :: qcd
    class(rng_factory_t), allocatable :: rng_factory
    class(model_data_t), pointer :: model
    type(process_t), allocatable, target :: process
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    type(process_instance_t), allocatable, target :: process_instance

    write (u, "(A)")  "* Test output: processes_16"
    write (u, "(A)")  "*   Purpose: integrate a process without &
         &structure functions"
    write (u, "(A)")  "*            in a multi-channel configuration"
    write (u, "(A)")

    write (u, "(A)")  "* Build and initialize a process object"
    write (u, "(A)")

    libname = "processes16"
    procname = libname
    run_id = "run16"
```

```
call os_data_init (os_data)
allocate (rng_test_factory_t :: rng_factory)
call prc_test_create_library (libname, lib, scattering = .false., &
     decay = .true.)

call reset_interaction_counter ()

allocate (model)
call model%init_test ()
call model%set_par (var_str ("ff"), 0.4_default)
call model%set_par (var_str ("mf"), &
     model%get_real (var_str ("ff")) * model%get_real (var_str ("ms")))

allocate (process)
call process%init (procname, run_id, &
     lib, os_data, qcd, rng_factory, model)

call process%setup_test_cores ()
allocate (mci_midpoint_t :: mci_template)
allocate (phs_single_config_t :: phs_config_template)
call process%init_component &
   (1, .true., mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

call process%setup_beams_decay (i_core = 1)
call process%configure_phs ()

call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_terms ()
call process%write (.false., u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)

write (u, "(A)")  "* Integrate with default test parameters"
write (u, "(A)")

call process_instance%integrate (1, n_it=1, n_calls=10000)
call process%final_integration (1)

call process%write (.false., u)

write (u, "(A)")
write (u, "(A,ES13.7)")  " Integral divided by phs factor = ", &
```

```
        process%get_integral (1) &
        / process_instance%term(1)%k_term%phs_factor

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call process_instance%final ()
    deallocate (process_instance)

    call process%final ()
    deallocate (process)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_16"

  end subroutine processes_16
```

Decay Process Evaluation

Initialize an evaluate a decay process for a moving particle.

⟨*Processes: execute tests*⟩+≡

```
  call test (processes_17, "processes_17", &
      "decay of moving particle", &
      u, results)
```

⟨*Processes: test declarations*⟩+≡

```
  public :: processes_17
```

⟨*Processes: tests*⟩+≡

```
  subroutine processes_17 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(qcd_t) :: qcd
    class(rng_factory_t), allocatable :: rng_factory
    class(model_data_t), pointer :: model
    type(process_t), allocatable, target :: process
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    type(process_instance_t), allocatable, target :: process_instance
    type(particle_set_t) :: pset
    type(flavor_t) :: flv_beam
    real(default) :: m, p, E

    write (u, "(A)")  "* Test output: processes_17"
    write (u, "(A)")  "*   Purpose: initialize a decay process object"
    write (u, "(A)")

    write (u, "(A)")  "* Build and load a test library with one process"
    write (u, "(A)")
```

```
libname = "processes17"
procname = libname
run_id = "run17"
call os_data_init (os_data)
allocate (rng_test_factory_t :: rng_factory)

call prc_test_create_library (libname, lib, scattering = .false., &
     decay = .true.)

write (u, "(A)")  "* Initialize a process object"
write (u, "(A)")

allocate (model)
call model%init_test ()
call model%set_par (var_str ("ff"), 0.4_default)
call model%set_par (var_str ("mf"), &
     model%get_real (var_str ("ff")) * model%get_real (var_str ("ms")))

allocate (process)
call process%init (procname, run_id, &
     lib, os_data, qcd, rng_factory, model)

call process%setup_test_cores ()
allocate (phs_single_config_t :: phs_config_template)
call process%init_component &
   (1, .true., mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

call process%setup_beams_decay (rest_frame = .false., i_core = 1)
call process%configure_phs ()
call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization"
write (u, "(A)")

call process%setup_terms ()
call process%write (.false., u)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

call reset_interaction_counter (3)

allocate (process_instance)
call process_instance%init (process)
call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Set parent momentum and random numbers"
write (u, "(A)")
```

```
call process_instance%choose_mci (1)
call process_instance%set_mcpar ([0._default, 0._default])

model => process%get_model_ptr ()
call flv_beam%init (25, model)
m = flv_beam%get_mass ()
p = 3 * m / 4
E = sqrt (m**2 + p**2)
call process_instance%set_beam_momenta ([vector4_moving (E, p, 3)])

call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Set up hard kinematics"
write (u, "(A)")

call process_instance%select_channel (1)
call process_instance%compute_seed_kinematics ()
call process_instance%compute_hard_kinematics ()

write (u, "(A)")  "* Evaluate matrix element and square"
write (u, "(A)")

call process_instance%compute_eff_kinematics ()
call process_instance%evaluate_expressions ()
call process_instance%compute_other_channels ()
call process_instance%evaluate_trace ()
call process_instance%write (u)

call process_instance%get_trace (pset, 1)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Particle content:"
write (u, "(A)")

call write_separator (u)
call pset%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Recover process instance"
write (u, "(A)")

call reset_interaction_counter (3)

allocate (process_instance)
call process_instance%init (process)

call process_instance%choose_mci (1)
call process_instance%set_trace (pset, 1, check_match = .false.)
call process_instance%recover (1, 1, .true.)
call process_instance%write (u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call pset%final ()
    call process_instance%final ()
    deallocate (process_instance)

    call process%final ()
    deallocate (process)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_17"

  end subroutine processes_17
```

Resonances in Phase Space

This test demonstrates the extraction of the resonance-history set from the generated phase space. We need a nontrivial process, but no matrix element. This is provided by the `prc_template` method, using the `SM` model. We also need the `phs_wood` method, otherwise we would not have resonances in the phase space configuration.

⟨*Processes: execute tests*⟩+≡

```
  call test (processes_18, "processes_18", &
       "extract resonance history set", &
       u, results)
```

⟨*Processes: test declarations*⟩+≡

```
  public :: processes_18
```

⟨*Processes: tests*⟩+≡

```
  subroutine processes_18 (u)
    integer, intent(in) :: u
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: model_name
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(string_t), dimension(:), allocatable :: prt_in, prt_out
    class(prc_core_def_t), allocatable :: def
    type(process_def_entry_t), pointer :: entry
    type(qcd_t) :: qcd
    class(rng_factory_t), allocatable :: rng_factory
    class(model_data_t), pointer :: model
    class(vars_t), pointer :: vars
    type(process_t), allocatable, target :: process
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    class(phs_config_t), pointer :: phs_config
    real(default) :: sqrts
    type(resonance_history_t), dimension(:), allocatable :: res_hist
```

```
integer :: i

write (u, "(A)")  "* Test output: processes_18"
write (u, "(A)")  "*   Purpose: extra resonance histories"
write (u, "(A)")

write (u, "(A)")  "* Build and load a test library with one process"
write (u, "(A)")

libname = "processes_18_lib"
procname = "processes_18_p"
run_id = "run18"
call os_data_init (os_data)

allocate (rng_test_factory_t :: rng_factory)

call syntax_phs_forest_init ()

write (u, "(A)")  "* Initialize a process library with one process"
write (u, "(A)")

model_name = "SM"
call prepare_model (model, model_name, vars)

call lib%init (libname)

allocate (prt_in (2), prt_out (3))
prt_in = [var_str ("e+"), var_str ("e-")]
prt_out = [var_str ("d"), var_str ("ubar"), var_str ("W+")]

allocate (template_me_def_t :: def)
select type (def)
type is (template_me_def_t)
   call def%init (model, prt_in, prt_out, unity = .false.)
end select
allocate (entry)
call entry%init (procname, &
     model_name = model_name, &
     n_in = 2, n_components = 1)
call entry%import_component (1, n_out = size (prt_out), &
     prt_in  = new_prt_spec (prt_in), &
     prt_out = new_prt_spec (prt_out), &
     method  = var_str ("template"), &
     variant = def)
call entry%write (u)

call lib%append (entry)

call lib%configure (os_data)
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)
```

```
  write (u, "(A)")
  write (u, "(A)")  "* Initialize a process object"
  write (u, "(A)")

  allocate (process)
  call process%init (procname, run_id, &
       lib, os_data, qcd, rng_factory, model)

  call process%setup_test_cores ()
  allocate (phs_wood_config_t :: phs_config_template)
  call process%init_component &
     (1, .true., mci_template, phs_config_template)

  write (u, "(A)")  "* Configure phase space"

  sqrts = 1000
  call process%setup_beams_sqrts (sqrts, i_core = 1)
  call process%configure_phs ()

  write (u, "(A)")
  write (u, "(A)")  "* Extract resonance history set"

  phs_config => process%get_phs_config (1)
  select type (phs_config)
  type is (phs_wood_config_t)
     call phs_config%extract_resonance_histories (res_hist)
  end select

  do i = 1, size (res_hist)
     write (u, *)
     call res_hist(i)%write (u)
  end do

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call process%final ()
  deallocate (process)

  call syntax_phs_forest_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: processes_18"

end subroutine processes_18
```

30.13 Process Stacks

For storing and handling multiple processes, we define process stacks. These are ordinary stacks where new process entries are pushed onto the top. We allow for multiple entries with identical process ID, but distinct run ID.

The implementation is essentially identical to the `prclib_stacks` module above. Unfortunately, Fortran supports no generic programming, so we do not make use of this fact.

When searching for a specific process ID, we will get (a pointer to) the topmost process entry with that ID on the stack, which was entered last. Usually, this is the best version of the process (in terms of integral, etc.) Thus the stack terminology makes sense.

⟨`process_stacks.f90`⟩≡

```
⟨File header⟩

module process_stacks

⟨Use strings⟩
  use io_units
  use format_utils, only: write_separator
  use diagnostics
  use os_interface
  use sm_qcd
  use model_data
  use rng_base
  use variables
  use observables
  use process_libraries
  use process

⟨Standard module head⟩

⟨Process stacks: public⟩

⟨Process stacks: types⟩

contains

⟨Process stacks: procedures⟩

end module process_stacks
```

30.13.1 The process entry type

A process entry is a process object, augmented by a pointer to the next entry. We do not need specific methods, all relevant methods are inherited.

On higher level, processes should be prepared as process entry objects.

⟨*Process stacks: public*⟩≡

```
public :: process_entry_t
```

⟨*Process stacks: types*⟩≡

```
type, extends (process_t) :: process_entry_t
   type(process_entry_t), pointer :: next => null ()
end type process_entry_t
```

30.13.2 The process stack type

For easy conversion and lookup it is useful to store the filling number in the object. The content is stored as a linked list.

The `var_list` component stores process-specific results, so they can be retrieved as (pseudo) variables.

The process stack can be linked to another one. This allows us to work with stacks of local scope.

⟨*Process stacks: public*⟩+≡
```
public :: process_stack_t
```

⟨*Process stacks: types*⟩+≡
```
type :: process_stack_t
   integer :: n = 0
   type(process_entry_t), pointer :: first => null ()
   type(var_list_t), pointer :: var_list => null ()
   type(process_stack_t), pointer :: next => null ()
 contains
 ⟨Process stacks: process stack: TBP⟩
end type process_stack_t
```

Finalize partly: deallocate the process stack and variable list entries, but keep the variable list as an empty object. This way, the variable list links are kept.

⟨*Process stacks: process stack: TBP*⟩≡
```
procedure :: clear => process_stack_clear
```

⟨*Process stacks: procedures*⟩≡
```
subroutine process_stack_clear (stack)
  class(process_stack_t), intent(inout) :: stack
  type(process_entry_t), pointer :: process
  if (associated (stack%var_list)) then
     call stack%var_list%final ()
  end if
  do while (associated (stack%first))
     process => stack%first
     stack%first => process%next
     call process%final ()
     deallocate (process)
  end do
  stack%n = 0
end subroutine process_stack_clear
```

Finalizer. Clear and deallocate the variable list.

⟨*Process stacks: process stack: TBP*⟩+≡
```
procedure :: final => process_stack_final
```

⟨*Process stacks: procedures*⟩+≡
```
subroutine process_stack_final (object)
  class(process_stack_t), intent(inout) :: object
  call object%clear ()
  if (associated (object%var_list)) then
     deallocate (object%var_list)
  end if
end subroutine process_stack_final
```

Output. The processes on the stack will be ordered LIFO, i.e., backwards.

⟨*Process stacks: process stack: TBP*⟩+≡

```
procedure :: write => process_stack_write
```

⟨*Process stacks: procedures*⟩+≡

```
recursive subroutine process_stack_write (object, unit, pacify)
  class(process_stack_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: pacify
  type(process_entry_t), pointer :: process
  integer :: u
  u = given_output_unit (unit)
  call write_separator (u, 2)
  select case (object%n)
  case (0)
     write (u, "(1x,A)")  "Process stack: [empty]"
     call write_separator (u, 2)
  case default
     write (u, "(1x,A)")  "Process stack:"
     process => object%first
     do while (associated (process))
        call process%write (.false., u, pacify = pacify)
        process => process%next
     end do
  end select
  if (associated (object%next)) then
     write (u, "(1x,A)")  "[Processes from context environment:]"
     call object%next%write (u, pacify)
  end if
end subroutine process_stack_write
```

The variable list is printed by a separate routine, since it should be linked to the global variable list, anyway.

⟨*Process stacks: process stack: TBP*⟩+≡

```
procedure :: write_var_list => process_stack_write_var_list
```

⟨*Process stacks: procedures*⟩+≡

```
subroutine process_stack_write_var_list (object, unit)
  class(process_stack_t), intent(in) :: object
  integer, intent(in), optional :: unit
  if (associated (object%var_list)) then
     call var_list_write (object%var_list, unit)
  end if
end subroutine process_stack_write_var_list
```

Short output.

⟨*Process stacks: process stack: TBP*⟩+≡

```
procedure :: show => process_stack_show
```

⟨*Process stacks: procedures*⟩+≡

```
recursive subroutine process_stack_show (object, unit)
  class(process_stack_t), intent(in) :: object
  integer, intent(in), optional :: unit
  type(process_entry_t), pointer :: process
```

```
    integer :: u
    u = given_output_unit (unit)
    select case (object%n)
    case (0)
    case default
       process => object%first
       do while (associated (process))
          call process%show (u, verbose=.false.)
          process => process%next
       end do
    end select
    if (associated (object%next))  call object%next%show ()
  end subroutine process_stack_show
```

30.13.3 Link

Link the current process stack to a global one.

⟨*Process stacks: process stack: TBP*⟩+≡

```
  procedure :: link => process_stack_link
```

⟨*Process stacks: procedures*⟩+≡

```
  subroutine process_stack_link (local_stack, global_stack)
    class(process_stack_t), intent(inout) :: local_stack
    type(process_stack_t), intent(in), target :: global_stack
    local_stack%next => global_stack
  end subroutine process_stack_link
```

Initialize the process variable list and link the main variable list to it.

⟨*Process stacks: process stack: TBP*⟩+≡

```
  procedure :: init_var_list => process_stack_init_var_list
```

⟨*Process stacks: procedures*⟩+≡

```
  subroutine process_stack_init_var_list (stack, var_list)
    class(process_stack_t), intent(inout) :: stack
    type(var_list_t), intent(inout), optional :: var_list
    allocate (stack%var_list)
    if (present (var_list))  call var_list%link (stack%var_list)
  end subroutine process_stack_init_var_list
```

Link the process variable list to a global variable list.

⟨*Process stacks: process stack: TBP*⟩+≡

```
  procedure :: link_var_list => process_stack_link_var_list
```

⟨*Process stacks: procedures*⟩+≡

```
  subroutine process_stack_link_var_list (stack, var_list)
    class(process_stack_t), intent(inout) :: stack
    type(var_list_t), intent(in), target :: var_list
    call stack%var_list%link (var_list)
  end subroutine process_stack_link_var_list
```

30.13.4 Push

We take a process pointer and push it onto the stack. The previous pointer is nullified. Subsequently, the process is 'owned' by the stack and will be finalized when the stack is deleted.

⟨*Process stacks: process stack: TBP*⟩+≡

```
procedure :: push => process_stack_push
```

⟨*Process stacks: procedures*⟩+≡

```
subroutine process_stack_push (stack, process)
  class(process_stack_t), intent(inout) :: stack
  type(process_entry_t), intent(inout), pointer :: process
  process%next => stack%first
  stack%first => process
  process => null ()
  stack%n = stack%n + 1
end subroutine process_stack_push
```

Initialize process variables for a given process ID, without setting values.

⟨*Process stacks: process stack: TBP*⟩+≡

```
procedure :: init_result_vars => process_stack_init_result_vars
```

⟨*Process stacks: procedures*⟩+≡

```
subroutine process_stack_init_result_vars (stack, id)
  class(process_stack_t), intent(inout) :: stack
  type(string_t), intent(in) :: id
  call var_list_init_num_id (stack%var_list, id)
  call var_list_init_process_results (stack%var_list, id)
end subroutine process_stack_init_result_vars
```

Fill process variables with values. This is executed after the integration pass.

Note: We set only integral and error. With multiple MCI records possible, the results for `n_calls`, `chi2` etc. are not necessarily unique. (We might set the efficiency, though.)

⟨*Process stacks: process stack: TBP*⟩+≡

```
procedure :: fill_result_vars => process_stack_fill_result_vars
```

⟨*Process stacks: procedures*⟩+≡

```
subroutine process_stack_fill_result_vars (stack, id)
  class(process_stack_t), intent(inout) :: stack
  type(string_t), intent(in) :: id
  type(process_t), pointer :: process
  process => stack%get_process_ptr (id)
  if (associated (process)) then
     call var_list_init_num_id (stack%var_list, id, process%get_num_id ())
     if (process%has_integral ()) then
        call var_list_init_process_results (stack%var_list, id, &
             integral = process%get_integral (), &
             error = process%get_error ())
     end if
  else
     call msg_bug ("process_stack_fill_result_vars: unknown process ID")
  end if
end subroutine process_stack_fill_result_vars
```

30.13.5 Data Access

Tell if a process exists.

⟨*Process stacks: process stack: TBP*⟩+≡
```
procedure :: exists => process_stack_exists
```

⟨*Process stacks: procedures*⟩+≡
```
function process_stack_exists (stack, id) result (flag)
  class(process_stack_t), intent(in) :: stack
  type(string_t), intent(in) :: id
  logical :: flag
  type(process_t), pointer :: process
  process => stack%get_process_ptr (id)
  flag = associated (process)
end function process_stack_exists
```

Return a pointer to a process with specific ID. Look also at a linked stack, if necessary.

⟨*Process stacks: process stack: TBP*⟩+≡
```
procedure :: get_process_ptr => process_stack_get_process_ptr
```

⟨*Process stacks: procedures*⟩+≡
```
recursive function process_stack_get_process_ptr (stack, id) result (ptr)
  class(process_stack_t), intent(in) :: stack
  type(string_t), intent(in) :: id
  type(process_t), pointer :: ptr
  type(process_entry_t), pointer :: entry
  ptr => null ()
  entry => stack%first
  do while (associated (entry))
     if (entry%get_id () == id) then
        ptr => entry%process_t
        return
     end if
     entry => entry%next
  end do
  if (associated (stack%next))  ptr => stack%next%get_process_ptr (id)
end function process_stack_get_process_ptr
```

30.13.6 Unit tests

Test module, followed by the corresponding implementation module.

⟨`process_stacks_ut.f90`⟩≡
```
⟨File header⟩

module process_stacks_ut
  use unit_tests
  use process_stacks_uti

⟨Standard module head⟩

⟨Process stacks: public test⟩
```

```
  contains

  ⟨Process stacks: test driver⟩

  end module process_stacks_ut
```

⟨process_stacks_uti.f90⟩≡

```
  ⟨File header⟩

  module process_stacks_uti

  ⟨Use strings⟩
    use os_interface
    use sm_qcd
    use model_data
    use process_libraries
    use rng_base
    use prc_test, only: prc_test_create_library
    use process, only: process_t
    use instances, only: process_instance_t
    use processes_ut, only: prepare_test_process

    use process_stacks

    use rng_base_ut, only: rng_test_factory_t

  ⟨Standard module head⟩

  ⟨Process stacks: test declarations⟩

  contains

  ⟨Process stacks: tests⟩

  end module process_stacks_uti
```

API: driver for the unit tests below.

⟨*Process stacks: public test*⟩≡

```
    public :: process_stacks_test
```

⟨*Process stacks: test driver*⟩≡

```
    subroutine process_stacks_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Process stacks: execute tests⟩
    end subroutine process_stacks_test
```

Write an empty process stack

The most trivial test is to write an uninitialized process stack.

⟨*Process stacks: execute tests*⟩≡

```
    call test (process_stacks_1, "process_stacks_1", &
         "write an empty process stack", &
```

```
        u, results)
```

⟨*Process stacks: test declarations*⟩≡

```
    public :: process_stacks_1
```

⟨*Process stacks: tests*⟩≡

```
    subroutine process_stacks_1 (u)
      integer, intent(in) :: u
      type(process_stack_t) :: stack

      write (u, "(A)")  "* Test output: process_stacks_1"
      write (u, "(A)")  "*   Purpose: display an empty process stack"
      write (u, "(A)")

      call stack%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: process_stacks_1"

    end subroutine process_stacks_1
```

Fill a process stack

Fill a process stack with two (identical) processes.

⟨*Process stacks: execute tests*⟩+≡

```
    call test (process_stacks_2, "process_stacks_2", &
         "fill a process stack", &
         u, results)
```

⟨*Process stacks: test declarations*⟩+≡

```
    public :: process_stacks_2
```

⟨*Process stacks: tests*⟩+≡

```
    subroutine process_stacks_2 (u)
      integer, intent(in) :: u
      type(process_stack_t) :: stack
      type(process_library_t), target :: lib
      type(string_t) :: libname
      type(string_t) :: procname
      type(string_t) :: run_id
      type(os_data_t) :: os_data
      type(qcd_t) :: qcd
      class(rng_factory_t), allocatable :: rng_factory
      class(model_data_t), pointer :: model
      type(process_entry_t), pointer :: process => null ()

      write (u, "(A)")  "* Test output: process_stacks_2"
      write (u, "(A)")  "*   Purpose: fill a process stack"
      write (u, "(A)")

      write (u, "(A)")  "* Build, initialize and store two test processes"
      write (u, "(A)")

      libname = "process_stacks2"
      procname = libname
```

```
    call os_data_init (os_data)
    allocate (rng_test_factory_t :: rng_factory)
    call prc_test_create_library (libname, lib)

    allocate (model)
    call model%init_test ()

    allocate (process)
    run_id = "run1"
    call process%init (procname, run_id, &
         lib, os_data, qcd, rng_factory, model)
    call stack%push (process)

    allocate (model)
    call model%init_test ()

    allocate (process)
    run_id = "run2"
    call process%init (procname, run_id, &
         lib, os_data, qcd, rng_factory, model)
    call stack%push (process)

    call stack%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call stack%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: process_stacks_2"

  end subroutine process_stacks_2
```

Fill a process stack

Fill a process stack with two (identical) processes.

⟨*Process stacks: execute tests*⟩+≡

```
  call test (process_stacks_3, "process_stacks_3", &
       "process variables", &
       u, results)
```

⟨*Process stacks: test declarations*⟩+≡

```
  public :: process_stacks_3
```

⟨*Process stacks: tests*⟩+≡

```
  subroutine process_stacks_3 (u)
    integer, intent(in) :: u
    type(process_stack_t) :: stack
    type(model_data_t), target :: model
    type(string_t) :: procname
    type(process_entry_t), pointer :: process => null ()
    type(process_instance_t), target :: process_instance
```

```
    write (u, "(A)")  "* Test output: process_stacks_3"
    write (u, "(A)")  "*   Purpose: setup process variables"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize process variables"
    write (u, "(A)")

    procname = "processes_test"
    call model%init_test ()

    write (u, "(A)")  "* Initialize process variables"
    write (u, "(A)")

    call stack%init_var_list ()
    call stack%init_result_vars (procname)
    call stack%write_var_list (u)

    write (u, "(A)")
    write (u, "(A)")  "* Build and integrate a test process"
    write (u, "(A)")

    allocate (process)
    call prepare_test_process (process%process_t, process_instance, model)
    call process_instance%integrate (1, 1, 1000)
    call process_instance%final ()
    call process%final_integration (1)
    call stack%push (process)

    write (u, "(A)")  "* Fill process variables"
    write (u, "(A)")

    call stack%fill_result_vars (procname)
    call stack%write_var_list (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call stack%final ()

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: process_stacks_3"

  end subroutine process_stacks_3
```

Linked a process stack

Fill two process stack, linked to each other.

⟨*Process stacks: execute tests*⟩+≡

```
  call test (process_stacks_4, "process_stacks_4", &
       "linked stacks", &
       u, results)
```

⟨*Process stacks: test declarations*⟩+≡

```
public :: process_stacks_4
```

⟨*Process stacks: tests*⟩+≡

```
subroutine process_stacks_4 (u)
  integer, intent(in) :: u
  type(process_library_t), target :: lib
  type(process_stack_t), target :: stack1, stack2
  class(model_data_t), pointer :: model
  type(string_t) :: libname
  type(string_t) :: procname
  type(string_t) :: run_id
  type(os_data_t) :: os_data
  type(qcd_t) :: qcd
  class(rng_factory_t), allocatable :: rng_factory
  type(process_entry_t), pointer :: process => null ()

  write (u, "(A)")  "* Test output: process_stacks_4"
  write (u, "(A)")  "*   Purpose: link process stacks"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize process variables"
  write (u, "(A)")

  libname = "process_stacks_4_lib"
  procname = "process_stacks_4a"

  call os_data_init (os_data)
  allocate (rng_test_factory_t :: rng_factory)

  write (u, "(A)")  "* Initialize first process"
  write (u, "(A)")

  call prc_test_create_library (procname, lib)

  allocate (model)
  call model%init_test ()

  allocate (process)
  run_id = "run1"
  call process%init (procname, run_id, &
       lib, os_data, qcd, rng_factory, model)
  call stack1%push (process)

  write (u, "(A)")  "* Initialize second process"
  write (u, "(A)")

  call stack2%link (stack1)

  procname = "process_stacks_4b"
  call prc_test_create_library (procname, lib)

  allocate (model)
  call model%init_test ()
```

```
  allocate (process)
  run_id = "run2"
  call process%init (procname, run_id, &
       lib, os_data, qcd, rng_factory, model)
  call stack2%push (process)

  write (u, "(A)")  "* Show linked stacks"
  write (u, "(A)")

  call stack2%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call stack2%final ()
  call stack1%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: process_stacks_4"

end subroutine process_stacks_4
```

Chapter 31

Matching

⟨matching_base.f90⟩≡
  ⟨*File header*⟩

```
module matching_base
```

  ⟨*Use strings*⟩

```
  use diagnostics
  use sm_qcd
  use model_data
  use particles
  use variables
  use shower_base
  use instances, only: process_instance_t
  use rng_base
```

  ⟨*Standard module head*⟩

  ⟨*Matching base: public*⟩

  ⟨*Matching base: parameters*⟩

  ⟨*Matching base: types*⟩

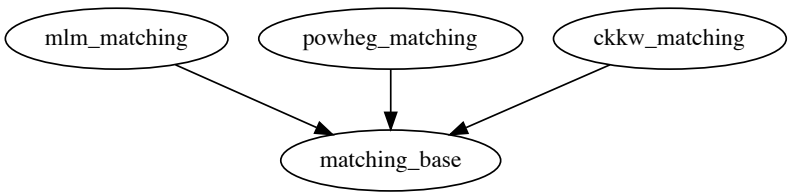

Figure 31.1: Module dependencies in `src/matching`.

⟨*Matching base: interfaces*⟩

```
contains
```

⟨*Matching base: procedures*⟩

```
end module matching_base
```

31.1 Abstract Matching Type

A matching will need access to the `shower` as well as matrix elements that we currently get over `process_instace`. The `model` is intended for the backup `model_hadrons`.

⟨*Matching base: public*⟩≡

```
public :: matching_t
```

⟨*Matching base: types*⟩≡

```
type, abstract :: matching_t
  logical :: is_hadron_collision = .false.
  type(qcd_t), pointer :: qcd => null ()
  class(shower_base_t), pointer :: shower => null ()
  type(process_instance_t), pointer :: process_instance => null ()
  class(model_data_t), pointer :: model => null ()
  class(rng_t), allocatable :: rng
  type(string_t) :: process_name
 contains
 ⟨Matching base: matching: TBP⟩
end type matching_t
```

⟨*Matching base: matching: TBP*⟩≡

```
procedure (matching_init), deferred :: init
```

⟨*Matching base: interfaces*⟩≡

```
abstract interface
   subroutine matching_init (matching, var_list, process_name)
     import
     class(matching_t), intent(out) :: matching
     type(var_list_t), intent(in) :: var_list
     type(string_t), intent(in) :: process_name
   end subroutine matching_init
end interface
```

If we use a polymorphic settings type, this boilerplate wouldn't be necessary but then we introduce `select type` statements all over the place.

⟨*default matching init*⟩≡

```
type(var_list_t), intent(in) :: var_list
type(string_t), intent(in) :: process_name
call msg_debug (D_MATCHING, "matching_init")
call matching%settings%init (var_list)
matching%process_name = process_name
```

⟨*Matching base: matching: TBP*⟩+≡

```
procedure (matching_write), deferred :: write
```

⟨*Matching base: interfaces*⟩+≡

```
abstract interface
   subroutine matching_write (matching, unit)
     import
     class(matching_t), intent(in) :: matching
     integer, intent(in), optional :: unit
   end subroutine matching_write
end interface
```

⟨*Matching base: matching: TBP*⟩+≡

```
procedure :: import_rng => matching_import_rng
```

⟨*Matching base: procedures*⟩≡

```
pure subroutine matching_import_rng (matching, rng)
  class(matching_t), intent(inout) :: matching
  class(rng_t), allocatable, intent(inout) :: rng
  call move_alloc (from = rng, to = matching%rng)
end subroutine matching_import_rng
```

⟨*Matching base: matching: TBP*⟩+≡

```
procedure :: connect => matching_connect
procedure :: base_connect => matching_connect
```

⟨*Matching base: procedures*⟩+≡

```
subroutine matching_connect (matching, process_instance, model, shower)
  class(matching_t), intent(inout) :: matching
  type(process_instance_t), intent(in), target :: process_instance
  class(model_data_t), intent(in), target :: model
  class(shower_base_t), intent(in), target :: shower
  call msg_debug (D_MATCHING, "matching_connect")
  matching%process_instance => process_instance
  matching%model => model
  matching%shower => shower
end subroutine matching_connect
```

⟨*Matching base: matching: TBP*⟩+≡

```
procedure (matching_before_shower), deferred :: before_shower
```

⟨*Matching base: interfaces*⟩+≡

```
abstract interface
   subroutine matching_before_shower (matching, particle_set, vetoed)
     import
     class(matching_t), intent(inout) :: matching
     type(particle_set_t), intent(inout) :: particle_set
     logical, intent(out) :: vetoed
   end subroutine matching_before_shower
end interface
```

⟨*Matching base: matching: TBP*⟩+≡

```
procedure (matching_after_shower), deferred :: after_shower
```

⟨*Matching base: interfaces*⟩+≡
```
abstract interface
   subroutine matching_after_shower (matching, particle_set, vetoed)
     import
     class(matching_t), intent(inout) :: matching
     type(particle_set_t), intent(inout) :: particle_set
     logical, intent(out) :: vetoed
   end subroutine matching_after_shower
end interface
```

Per default, do nothing here.

⟨*Matching base: matching: TBP*⟩+≡
```
procedure :: prepare_for_events => matching_prepare_for_events
```

⟨*Matching base: procedures*⟩+≡
```
subroutine matching_prepare_for_events (matching)
  class(matching_t), intent(inout), target :: matching
end subroutine matching_prepare_for_events
```

⟨*Matching base: matching: TBP*⟩+≡
```
procedure :: first_event => matching_first_event
```

⟨*Matching base: procedures*⟩+≡
```
subroutine matching_first_event (matching)
  class(matching_t), intent(inout), target :: matching
end subroutine matching_first_event
```

⟨*Matching base: matching: TBP*⟩+≡
```
procedure (matching_get_method), deferred :: get_method
```

⟨*Matching base: interfaces*⟩+≡
```
abstract interface
   function matching_get_method (matching) result (method)
     import
     type(string_t) :: method
     class(matching_t), intent(in) :: matching
   end function matching_get_method
end interface
```

⟨*Matching base: matching: TBP*⟩+≡
```
procedure :: final => matching_final
```

⟨*Matching base: procedures*⟩+≡
```
subroutine matching_final (matching)
  class(matching_t), intent(in) :: matching
end subroutine matching_final
```

31.1.1 Matching implementations

⟨*Matching base: public*⟩+≡
```
public :: MATCH_MLM, MATCH_CKKW, MATCH_POWHEG
```

⟨*Matching base: parameters*⟩≡

```
integer, parameter :: MATCH_MLM = 1
integer, parameter :: MATCH_CKKW = 2
integer, parameter :: MATCH_POWHEG = 3
integer, parameter :: MATCH_UNDEFINED = 17
```

A dictionary

⟨*Matching base: public*⟩+≡

```
public :: matching_method
```

⟨*Matching base: interfaces*⟩+≡

```
interface matching_method
   module procedure matching_method_of_string
   module procedure matching_method_to_string
end interface
```

⟨*Matching base: procedures*⟩+≡

```
elemental function matching_method_of_string (string) result (i)
  integer :: i
  type(string_t), intent(in) :: string
  select case (char (string))
  case ("MLM")
     i = MATCH_MLM
  case ("CKKW")
     i = MATCH_CKKW
  case ("POWHEG")
     i = MATCH_POWHEG
  case default
     i = MATCH_UNDEFINED
  end select
end function matching_method_of_string

elemental function matching_method_to_string (i) result (string)
  type(string_t) :: string
  integer, intent(in) :: i
  select case (i)
  case (MATCH_MLM)
     string = "MLM"
  case (MATCH_CKKW)
     string = "CKKW"
  case (MATCH_POWHEG)
     string = "POWHEG"
  case default
     string = "UNDEFINED"
  end select
end function matching_method_to_string
```

31.2 MLM Matching

⟨mlm_matching.f90⟩≡

⟨*File header*⟩

```
module mlm_matching
```

```
⟨Use kinds with double⟩
⟨Use strings⟩
  use io_units
  use constants
  use format_utils, only: write_separator
  use diagnostics
  use file_utils
  use lorentz
  use subevents, only: PRT_OUTGOING
  use particles
  use variables
  use shower_base
  use ktclus
  use matching_base

⟨Standard module head⟩

⟨MLM matching: public⟩

⟨MLM matching: types⟩

contains

⟨MLM matching: procedures⟩

end module mlm_matching
```

⟨*MLM matching: public*⟩≡

```
  public :: mlm_matching_settings_t
```

⟨*MLM matching: types*⟩≡

```
  type :: mlm_matching_settings_t
     real(default) :: mlm_Qcut_ME = one
     real(default) :: mlm_Qcut_PS = one
     real(default) :: mlm_ptmin, mlm_etamax, mlm_Rmin, mlm_Emin
     real(default) :: mlm_ETclusfactor = 0.2_default
     real(default) :: mlm_ETclusminE = five
     real(default) :: mlm_etaclusfactor = one
     real(default) :: mlm_Rclusfactor = one
     real(default) :: mlm_Eclusfactor = one
     integer :: kt_imode_hadronic = 4313
     integer :: kt_imode_leptonic = 1111
     integer :: mlm_nmaxMEjets = 0
   contains
   ⟨MLM matching: mlm matching settings: TBP⟩
  end type mlm_matching_settings_t
```

⟨*MLM matching: mlm matching settings: TBP*⟩≡

```
  procedure :: init => mlm_matching_settings_init
```

⟨*MLM matching: procedures*⟩≡

```
  subroutine mlm_matching_settings_init (settings, var_list)
    class(mlm_matching_settings_t), intent(out) :: settings
    type(var_list_t), intent(in) :: var_list
```

```
    settings%mlm_Qcut_ME = &
         var_list%get_rval (var_str ("mlm_Qcut_ME"))
    settings%mlm_Qcut_PS = &
         var_list%get_rval (var_str ("mlm_Qcut_PS"))
    settings%mlm_ptmin = &
         var_list%get_rval (var_str ("mlm_ptmin"))
    settings%mlm_etamax = &
         var_list%get_rval (var_str ("mlm_etamax"))
    settings%mlm_Rmin = &
         var_list%get_rval (var_str ("mlm_Rmin"))
    settings%mlm_Emin = &
         var_list%get_rval (var_str ("mlm_Emin"))
    settings%mlm_nmaxMEjets = &
         var_list%get_ival (var_str ("mlm_nmaxMEjets"))

    settings%mlm_ETclusfactor = &
         var_list%get_rval (var_str ("mlm_ETclusfactor"))
    settings%mlm_ETclusminE = &
         var_list%get_rval (var_str ("mlm_ETclusminE"))
    settings%mlm_etaclusfactor = &
         var_list%get_rval (var_str ("mlm_etaclusfactor"))
    settings%mlm_Rclusfactor = &
         var_list%get_rval (var_str ("mlm_Rclusfactor"))
    settings%mlm_Eclusfactor = &
         var_list%get_rval (var_str ("mlm_Eclusfactor"))
  end subroutine mlm_matching_settings_init
```

⟨*MLM matching: mlm matching settings: TBP*⟩+≡

```
  procedure :: write => mlm_matching_settings_write
```

⟨*MLM matching: procedures*⟩+≡

```
  subroutine mlm_matching_settings_write (settings, unit)
    class(mlm_matching_settings_t), intent(in) :: settings
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(3x,A,ES19.12)") &
         "mlm_Qcut_ME                 = ", settings%mlm_Qcut_ME
    write (u, "(3x,A,ES19.12)") &
         "mlm_Qcut_PS                 = ", settings%mlm_Qcut_PS
    write (u, "(3x,A,ES19.12)") &
         "mlm_ptmin                   = ", settings%mlm_ptmin
    write (u, "(3x,A,ES19.12)") &
         "mlm_etamax                  = ", settings%mlm_etamax
    write (u, "(3x,A,ES19.12)") &
         "mlm_Rmin                    = ", settings%mlm_Rmin
    write (u, "(3x,A,ES19.12)") &
         "mlm_Emin                    = ", settings%mlm_Emin
    write (u, "(3x,A,1x,I0)") &
         "mlm_nmaxMEjets              = ", settings%mlm_nmaxMEjets
    write (u, "(3x,A,ES19.12)") &
         "mlm_ETclusfactor  (D=0.2)   = ", settings%mlm_ETclusfactor
    write (u, "(3x,A,ES19.12)") &
         "mlm_ETclusminE    (D=5.0)   = ", settings%mlm_ETclusminE
```

```
    write (u, "(3x,A,ES19.12)") &
         "mlm_etaclusfactor (D=1.0)    = ", settings%mlm_etaClusfactor
    write (u, "(3x,A,ES19.12)") &
         "mlm_Rclusfactor   (D=1.0)    = ", settings%mlm_RClusfactor
    write (u, "(3x,A,ES19.12)") &
         "mlm_Eclusfactor   (D=1.0)    = ", settings%mlm_EClusfactor
end subroutine mlm_matching_settings_write
```

This is a container for the (colored) parton momenta as well as the jet momenta.

⟨*MLM matching: public*⟩+≡

```
public :: mlm_matching_t
```

⟨*MLM matching: types*⟩+≡

```
type, extends (matching_t) :: mlm_matching_t
   type(vector4_t), dimension(:), allocatable, public :: P_ME
   type(vector4_t), dimension(:), allocatable, public :: P_PS
   type(vector4_t), dimension(:), allocatable, private :: JETS_ME
   type(vector4_t), dimension(:), allocatable, private :: JETS_PS
   type(mlm_matching_settings_t) :: settings
 contains
 ⟨MLM matching: mlm matching: TBP⟩
end type mlm_matching_t
```

⟨*MLM matching: mlm matching: TBP*⟩≡

```
procedure :: init => mlm_matching_init
```

⟨*MLM matching: procedures*⟩+≡

```
subroutine mlm_matching_init (matching, var_list, process_name)
  class(mlm_matching_t), intent(out) :: matching
⟨default matching init⟩
end subroutine mlm_matching_init
```

⟨*MLM matching: mlm matching: TBP*⟩+≡

```
procedure :: write => mlm_matching_write
```

⟨*MLM matching: procedures*⟩+≡

```
subroutine mlm_matching_write (matching, unit)
  class(mlm_matching_t), intent(in) :: matching
  integer, intent(in), optional :: unit
  integer :: i, u
  u = given_output_unit (unit);  if (u < 0)  return

  write (u, "(1x,A)") "MLM matching:"
  call matching%settings%write (u)
  write (u, "(3x,A)") "Momenta of ME partons:"
  if (allocated (matching%P_ME)) then
     do i = 1, size (matching%P_ME)
        write (u, "(4x)", advance = "no")
        call vector4_write (matching%P_ME(i), unit = u)
     end do
  else
     write (u, "(5x,A)")  "[empty]"
  end if
  call write_separator (u)
```

```
    write (u, "(3x,A)")  "Momenta of ME jets:"
    if (allocated (matching%JETS_ME)) then
       do i = 1, size (matching%JETS_ME)
          write (u, "(4x)", advance = "no")
          call vector4_write (matching%JETS_ME(i), unit = u)
       end do
    else
       write (u, "(5x,A)")  "[empty]"
    end if
    call write_separator (u)
    write(u, "(3x,A)")  "Momenta of shower partons:"
    if (allocated (matching%P_PS)) then
       do i = 1, size (matching%P_PS)
          write (u, "(4x)", advance = "no")
          call vector4_write (matching%P_PS(i), unit = u)
       end do
    else
       write (u, "(5x,A)")  "[empty]"
    end if
    call write_separator (u)
    write (u, "(3x,A)")  "Momenta of shower jets:"
    if (allocated (matching%JETS_PS)) then
       do i = 1, size (matching%JETS_PS)
          write (u, "(4x)", advance = "no")
          call vector4_write (matching%JETS_PS(i), unit = u)
       end do
    else
       write (u, "(5x,A)")  "[empty]"
    end if
    call write_separator (u)
  end subroutine mlm_matching_write
```

⟨*MLM matching: mlm matching: TBP*⟩+≡

```
  procedure :: get_method => mlm_matching_get_method
```

⟨*MLM matching: procedures*⟩+≡

```
  function mlm_matching_get_method (matching) result (method)
     type(string_t) :: method
     class(mlm_matching_t), intent(in) :: matching
     method = matching_method (MATCH_MLM)
  end function mlm_matching_get_method
```

⟨*MLM matching: mlm matching: TBP*⟩+≡

```
  procedure :: before_shower => mlm_matching_before_shower
```

⟨*MLM matching: procedures*⟩+≡

```
  subroutine mlm_matching_before_shower &
         (matching, particle_set, vetoed)
    class(mlm_matching_t), intent(inout) :: matching
    type(particle_set_t), intent(inout) :: particle_set
    logical, intent(out) :: vetoed
    vetoed = .false.
  end subroutine mlm_matching_before_shower
```

⟨*MLM matching: mlm matching: TBP*⟩+≡

```
procedure :: after_shower => mlm_matching_after_shower
```

⟨*MLM matching: procedures*⟩+≡

```
subroutine mlm_matching_after_shower (matching, particle_set, vetoed)
  class(mlm_matching_t), intent(inout) :: matching
  type(particle_set_t), intent(inout) :: particle_set
  logical, intent(out) :: vetoed
  call msg_debug (D_MATCHING, "mlm_matching_after_shower")
  call matching%shower%get_final_colored_ME_momenta (matching%P_ME)
  call matching%fill_P_PS (particle_set)
  !!! MLM stage 3 -> reconstruct and possibly reject
  call matching%apply (vetoed)
  if (allocated (matching%P_ME))  deallocate (matching%P_ME)
  if (allocated (matching%P_PS))  deallocate (matching%P_PS)
  if (allocated (matching%JETS_ME))  deallocate (matching%JETS_ME)
  if (allocated (matching%JETS_PS))  deallocate (matching%JETS_PS)
end subroutine mlm_matching_after_shower
```

Transfer partons after parton shower to `matching%P_PS`

⟨*MLM matching: mlm matching: TBP*⟩+≡

```
procedure :: fill_P_PS => mlm_matching_fill_P_PS
```

⟨*MLM matching: procedures*⟩+≡

```
subroutine mlm_matching_fill_P_PS (matching, particle_set)
  class(mlm_matching_t), intent(inout) :: matching
  type(particle_set_t), intent(in) :: particle_set
  integer :: i, j, n_jets_PS
  integer, dimension(2) :: col
  type(particle_t) :: tempprt
  real(double) :: eta
  type(vector4_t) :: p_tmp

  !!! loop over particles and extract final colored ones with eta<etamax
  n_jets_PS = 0
  do i = 1, particle_set%get_n_tot ()
     if (signal_is_pending ()) return
     tempprt = particle_set%get_particle (i)
     if (tempprt%get_status () /= PRT_OUTGOING) cycle
     col = tempprt%get_color ()
     if (all (col == 0)) cycle
! TODO: (bcn 2015-04-28) where is the corresponding part for lepton colliders?
     if (matching%is_hadron_collision) then
        p_tmp = tempprt%get_momentum ()
        if (energy (p_tmp) - longitudinal_part (p_tmp) < 1.E-10_default .or. &
             energy (p_tmp) + longitudinal_part (p_tmp) < 1.E-10_default) then
           eta = pseudorapidity (p_tmp)
        else
           eta = rapidity (p_tmp)
        end if
        if (eta > matching%settings%mlm_etaClusfactor * &
             matching%settings%mlm_etamax)  then
           if (debug_active (D_MATCHING)) then
              call msg_debug (D_MATCHING, "Rejecting this particle")
              call tempprt%write ()
```

```
            end if
            cycle
         end if
      end if
      n_jets_PS = n_jets_PS + 1
   end do

   allocate (matching%P_PS(1:n_jets_PS))
   call msg_debug (D_MATCHING, "n_jets_ps", n_jets_ps)

   j = 1
   do i = 1, particle_set%get_n_tot ()
      tempprt = particle_set%get_particle (i)
      if (tempprt%get_status () /= PRT_OUTGOING) cycle
      col = tempprt%get_color ()
      if (all(col == 0)) cycle
! TODO: (bcn 2015-04-28) where is the corresponding part for lepton colliders?
      if (matching%is_hadron_collision) then
         p_tmp = tempprt%get_momentum ()
         if (energy (p_tmp) - longitudinal_part (p_tmp) < 1.E-10_default .or. &
              energy (p_tmp) + longitudinal_part (p_tmp) < 1.E-10_default) then
            eta = pseudorapidity (p_tmp)
         else
            eta = rapidity (p_tmp)
         end if
         if (eta > matching%settings%mlm_etaClusfactor * &
              matching%settings%mlm_etamax) cycle
      end if
      matching%P_PS(j) = tempprt%get_momentum ()
      j = j + 1
   end do
end subroutine mlm_matching_fill_P_PS
```

⟨*MLM matching: mlm matching: TBP*⟩+≡

```
procedure :: apply => mlm_matching_apply
```

⟨*MLM matching: procedures*⟩+≡

```
subroutine mlm_matching_apply (matching, vetoed)
   class(mlm_matching_t), intent(inout) :: matching
   logical, intent(out) :: vetoed
   integer :: i, j
   integer :: n_jets_ME, n_jets_PS, n_jets_PS_atycut
   real(double) :: ycut
   real(double), dimension(:, :), allocatable :: PP
   real(double), dimension(:), allocatable :: Y
   real(double), dimension(:,:), allocatable :: P_JETS
   real(double), dimension(:,:), allocatable :: P_ME
   integer, dimension(:), allocatable :: JET
   integer :: NJET, NSUB
   integer :: imode
!!! TODO: (bcn 2014-03-26) Why is ECUT hard coded to 1?
!!! It is the denominator of the KT measure. Candidate for removal
   real(double) :: ECUT = 1._double
   integer :: ip1,ip2
```

```
      ! KTCLUS COMMON BLOCK
      INTEGER NMAX,NUM,HIST
      PARAMETER (NMAX=512)
      DOUBLE PRECISION P,KT,KTP,KTS,ETOT,RSQ,KTLAST
      COMMON /KTCOMM/ETOT,RSQ,P(9,NMAX),KTP(NMAX,NMAX),KTS(NMAX), &
           KT(NMAX),KTLAST(NMAX),HIST(NMAX),NUM

      vetoed = .true.
      if (signal_is_pending ())  return

      ⟨Set n_jets_ME/PS from matching (or equal zero)⟩

      ⟨Jet clustering for partons after matrix element⟩

      ⟨Jet clustering for partons after shower⟩

      ⟨Veto: too many or not enough jets after PS⟩

      ⟨Cluster ME jets with PS jets one at a time⟩

      vetoed = .false.
  999 continue
    end subroutine mlm_matching_apply
```

⟨*Set* n_jets_ME/PS *from* matching *(or equal zero)*⟩≡

```
  if (allocated (matching%P_ME)) then
     ! print *, "number of partons after ME: ", size(matching%P_ME)
     n_jets_ME = size (matching%P_ME)
  else
     n_jets_ME = 0
  end if
  if (allocated (matching%p_PS)) then
     ! print *, "number of partons after PS: ", size(matching%p_PS)
     n_jets_PS = size (matching%p_PS)
  else
     n_jets_PS = 0
  end if
```

⟨*Jet clustering for partons after matrix element*⟩≡

```
  if (n_jets_ME > 0) then
     ycut = (matching%settings%mlm_ptmin)**2
     allocate (PP(1:4, 1:N_jets_ME))
     do i = 1, n_jets_ME
        PP(1:3,i) = matching%p_ME(i)%p(1:3)
        PP(4,i) = matching%p_ME(i)%p(0)
     end do

     ⟨Set imode for lepton or hadron collisions⟩

     allocate (P_ME(1:4,1:n_jets_ME))
     allocate (JET(1:n_jets_ME))
     allocate (Y(1:n_jets_ME))
```

```
      if (signal_is_pending ())  return
      call KTCLUR (imode, PP, n_jets_ME, &
           dble (matching%settings%mlm_Rclusfactor * matching%settings%mlm_Rmin), ECUT, y, *999)
      call KTRECO (1, PP, n_jets_ME, ECUT, ycut, ycut, P_ME, JET, &
           NJET, NSUB, *999)

      n_jets_ME = NJET
      if (NJET > 0) then
         allocate (matching%JETS_ME (1:NJET))
         do i = 1, NJET
            matching%JETS_ME(i) = vector4_moving (REAL(P_ME(4,i), default), &
                 vector3_moving([REAL(P_ME(1,i), default), &
                 REAL(P_ME(2,i), default), REAL(P_ME(3,i), default)]))
         end do
      end if
      deallocate (P_ME)
      deallocate (JET)
      deallocate (Y)
      deallocate (PP)
   end if
```

⟨*Jet clustering for partons after shower*⟩≡

```
   if (n_jets_PS > 0) then
      ycut = (matching%settings%mlm_ptmin + max (matching%settings%mlm_ETclusminE, &
           matching%settings%mlm_ETclusfactor * matching%settings%mlm_ptmin))**2
      allocate (PP(1:4, 1:n_jets_PS))
      do i = 1, n_jets_PS
         PP(1:3,i) = matching%p_PS(i)%p(1:3)
         PP(4,i) = matching%p_PS(i)%p(0)
      end do

      ⟨Set imode for lepton or hadron collisions⟩

      allocate (P_JETS(1:4,1:n_jets_PS))
      allocate (JET(1:n_jets_PS))
      allocate (Y(1:n_jets_PS))

      if (signal_is_pending ()) return
      call KTCLUR (imode, PP, n_jets_PS, &
           dble (matching%settings%mlm_Rclusfactor * matching%settings%mlm_Rmin), &
           ECUT, y, *999)
      call KTRECO (1, PP, n_jets_PS, ECUT, ycut, ycut, P_JETS, JET, &
           NJET, NSUB, *999)
      n_jets_PS_atycut = NJET
      if (n_jets_ME == matching%settings%mlm_nmaxMEjets .and. NJET > 0) then
         ! print *, " resetting ycut to ", Y(matching%settings%mlm_nmaxMEjets)
         ycut = y(matching%settings%mlm_nmaxMEjets)
         call KTRECO (1, PP, n_jets_PS, ECUT, ycut, ycut, P_JETS, JET, &
              NJET, NSUB, *999)
      end if

      ! !Sample of code for a FastJet interface
      ! palg = 1d0         ! 1.0d0 = kt, 0.0d0 = Cam/Aachen, -1.0d0 = anti-kt
      ! R = 0.7_double     ! radius parameter
      ! f = 0.75_double    ! overlap threshold
```

```
      ! !call fastjetppgenkt(PP,n,R,palg,P_JETS,NJET)   ! KT-Algorithm
      ! !call fastjetsiscone(PP,n,R,f,P_JETS,NJET)      ! SiSCone-Algorithm

      if (NJET > 0) then
         allocate (matching%JETS_PS(1:NJET))
         do i = 1, NJET
            matching%JETS_PS(i) = vector4_moving (REAL(P_JETS(4,i), default), &
                 vector3_moving([REAL(P_JETS(1,i), default), &
                 REAL(P_JETS(2,i), default), REAL(P_JETS(3,i), default)]))
         end do
      end if

      deallocate (P_JETS)
      deallocate (JET)
      deallocate (Y)
   else
      n_jets_PS_atycut = 0
   end if
```

⟨*Set* imode *for lepton or hadron collisions*⟩≡

```
   if (matching%is_hadron_collision) then
      imode = matching%settings%kt_imode_hadronic
   else
      imode = matching%settings%kt_imode_leptonic
   end if
```

⟨*Veto: too many or not enough jets after PS*⟩≡

```
   if (n_jets_PS_atycut < n_jets_ME) then
      ! print *, "DISCARDING: Not enough PS jets: ", n_jets_PS_atycut
      return
   end if
   if (n_jets_PS_atycut > n_jets_ME .and. n_jets_ME /= matching%settings%mlm_nmaxMEjets) then
      ! print *, "DISCARDING: Too many PS jets: ", n_jets_PS_atycut
      return
   end if
```

⟨*Cluster ME jets with PS jets one at a time*⟩≡

```
   if (allocated(matching%JETS_PS)) then
      ! print *, "number of jets after PS: ", size(matching%JETS_PS)
      n_jets_PS = size (matching%JETS_PS)
   else
      n_jets_PS = 0
   end if
   if (n_jets_ME > 0 .and. n_jets_PS > 0) then
      n_jets_PS = size (matching%JETS_PS)
      if (allocated (PP))  deallocate(PP)
      allocate (PP(1:4, 1:n_jets_PS + 1))
      do i = 1, n_jets_PS
         if (signal_is_pending ()) return
         PP(1:3,i) = matching%JETS_PS(i)%p(1:3)
         PP(4,i) = matching%JETS_PS(i)%p(0)
      end do
      if (allocated (Y))  deallocate(Y)
      allocate (Y(1:n_jets_PS + 1))
      y = zero
      do i = 1, n_jets_ME
```

```
      PP(1:3,n_jets_PS + 2 - i) = matching%JETS_ME(i)%p(1:3)
      PP(4,n_jets_PS + 2 - i) = matching%JETS_ME(i)%p(0)
      !!! This makes more sense than hardcoding
      ! call KTCLUS (4313, PP, (n_jets_PS + 2 - i), 1.0_double, Y, *999)
      call KTCLUR (imode, PP, (n_jets_PS + 2 - i), &
        dble (matching%settings%mlm_Rclusfactor * matching%settings%mlm_Rmin), &
        ECUT, y, *999)
      if (0.99 * y(n_jets_PS + 1 - (i - 1)).gt.ycut) then
         ! print *, "DISCARDING: Jet ", i, " not clusterd"
         return
      end if
      !!! search for and remove PS jet clustered with ME Jet
      ip1 = HIST(n_jets_PS + 2 - i) / NMAX
      ip2 = mod(hist(n_jets_PS + 2 - i), NMAX)
      if ((ip2 /= n_jets_PS + 2 - i) .or. (ip1 <= 0)) then
         ! print *, "DISCARDING: Jet ", i, " not clustered ", ip1, ip2, &
         !      hist(n_jets_PS + 2 - i)
         return
      else
         ! print *, "PARTON clustered", ip1, ip2, hist(n_jets_PS + 2 - i)
         PP(:,IP1) = zero
         do j = IP1, n_jets_PS - i
            PP(:, j) = PP(:,j + 1)
         end do
      end if
   end do
end if
```

31.3 CKKW matching

This module contains the CKKW matching.

The type `ckkw_pseudo_shower_weights_t` gives the (relative) weights for different clusterings of the final particles, as given in Eq. (2.7) of hep-ph/0503281v1. Each particle has a binary labelling (power of 2) (first particle = 1, second particle = 2, third particle = 4, ...). Each recombination therefore corresponds to an integer, that is not a power of 2. Fur multiple subsequent recombinations, no different weights for different sequences of clustering are stored. It is assumed that the weight of a multiply recombined state is a combination of the states with one fewer recombination and that these states' contributions are proportional to their weights. For a $2->n$ event, the weights array thus has the size $2^{(2+n)-1}$. The `weights_by_type` array gives the weights depending on the type of the particle, the first index is the same as for weights, the second index gives the type of the new mother particle:

0: uncolored (γ, Z, W, Higgs)

1: colored (quark)

2: gluon

3: squark

4: gluino

`alphaS` gives the value for $alpha_s$ used in the generation of the matrix element. This is needed for the reweighting using the values for a running $alpha_s$ at the scales of the clusterings.

⟨`ckkw_matching.f90`⟩≡
```
⟨File header⟩

module ckkw_matching

⟨Use kinds with double⟩
⟨Use strings⟩
  use io_units
  use constants
  use format_utils, only: write_separator
  use diagnostics
  use physics_defs
  use lorentz
  use particles
  use rng_base
  use shower_base
  use shower_partons
  use shower_core
  use variables
  use matching_base

⟨Standard module head⟩

⟨CKKW matching: public⟩

⟨CKKW matching: types⟩

contains

⟨CKKW matching: procedures⟩

end module ckkw_matching
```

The fundamental CKKW matching parameter are defined here:

⟨*CKKW matching: public*⟩≡
```
  public :: ckkw_matching_settings_t
```

⟨*CKKW matching: types*⟩≡
```
  type :: ckkw_matching_settings_t
     real(default) :: alphaS = 0.118_default
     real(default) :: Qmin = one
     integer :: n_max_jets = 0
   contains
   ⟨CKKW matching: ckkw matching settings: TBP⟩
  end type ckkw_matching_settings_t
```

This is empty for the moment.

⟨*CKKW matching: ckkw matching settings: TBP*⟩≡
```
  procedure :: init => ckkw_matching_settings_init
```

⟨*CKKW matching: procedures*⟩≡

```
subroutine ckkw_matching_settings_init (settings, var_list)
  class(ckkw_matching_settings_t), intent(out) :: settings
  type(var_list_t), intent(in) :: var_list
  settings%alphaS = 1.0_default
  settings%Qmin = 1.0_default
  settings%n_max_jets = 3
end subroutine ckkw_matching_settings_init
```

⟨*CKKW matching: ckkw matching settings: TBP*⟩+≡

```
procedure :: write => ckkw_matching_settings_write
```

⟨*CKKW matching: procedures*⟩+≡

```
subroutine ckkw_matching_settings_write (settings, unit)
  class(ckkw_matching_settings_t), intent(in) :: settings
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)")  "CKKW matching settings:"
  call write_separator (u)
  write (u, "(3x,A,1x,ES19.12)") &
       "alphaS       = ", settings%alphaS
  write (u, "(3x,A,1x,ES19.12)") &
       "Qmin         = ", settings%Qmin
  write (u, "(3x,A,1x,I0)") &
       "n_max_jets   = ", settings%n_max_jets
end subroutine ckkw_matching_settings_write
```

⟨*CKKW matching: public*⟩+≡

```
public :: ckkw_pseudo_shower_weights_t
```

⟨*CKKW matching: types*⟩+≡

```
type :: ckkw_pseudo_shower_weights_t
   real(default) :: alphaS
   real(default), dimension(:), allocatable :: weights
   real(default), dimension(:,:), allocatable :: weights_by_type
 contains
```

⟨*CKKW matching: ckkw pseudo shower weights: TBP*⟩

```
end type ckkw_pseudo_shower_weights_t
```

⟨*CKKW matching: ckkw pseudo shower weights: TBP*⟩≡

```
procedure :: init => ckkw_pseudo_shower_weights_init
```

⟨*CKKW matching: procedures*⟩+≡

```
subroutine ckkw_pseudo_shower_weights_init (weights)
  class(ckkw_pseudo_shower_weights_t), intent(out) :: weights
  weights%alphaS = zero
end subroutine ckkw_pseudo_shower_weights_init
```

⟨*CKKW matching: ckkw pseudo shower weights: TBP*⟩+≡

```
procedure :: write => ckkw_pseudo_shower_weights_write
```

⟨*CKKW matching: procedures*⟩+≡

```
subroutine ckkw_pseudo_shower_weights_write (weights, unit)
  class(ckkw_pseudo_shower_weights_t), intent(in) :: weights
  integer, intent(in), optional :: unit
  integer :: s, i, u
  u = given_output_unit (unit); if (u < 0) return
  s = size (weights%weights)
  write (u, "(1x,A)")  "CKKW (pseudo) shower weights: "
  do i = 1, s
     write (u, "(3x,I0,2(ES19.12))")  i, weights%weights(i), &
          weights%weights_by_type(i,:)
  end do
  write (u, "(3x,A,1x,I0)")  "alphaS =", weights%alphaS
end subroutine ckkw_pseudo_shower_weights_write
```

Generate fake ckkw weights. This can be dropped, once information from the matrix element generation is available.

⟨*CKKW matching: ckkw pseudo shower weights: TBP*⟩+≡

```
procedure :: fake => ckkw_pseudo_shower_weights_fake
```

⟨*CKKW matching: procedures*⟩+≡

```
pure subroutine ckkw_pseudo_shower_weights_fake (weights, particle_set)
  class(ckkw_pseudo_shower_weights_t), intent(inout) :: weights
  type(particle_set_t), intent(in) :: particle_set
  integer :: i, j, n
  type(vector4_t) :: momentum
  n = 2**particle_set%n_tot
  if (allocated (weights%weights)) then
     deallocate (weights%weights)
  end if
  allocate (weights%weights (1:n))
  do i = 1, n
     momentum = vector4_null
     do j = 1, particle_set%n_tot
        if (btest (i,j-1)) then
           momentum = momentum + particle_set%prt(j)%p
        end if
     end do
     if (momentum**1 > 0.0) then
        weights%weights(i) = 1.0 / (momentum**2)
     end if
  end do
  ! equally distribute the weights by type
  if (allocated (weights%weights_by_type)) then
     deallocate (weights%weights_by_type)
  end if
  allocate (weights%weights_by_type (1:n, 0:4))
  do i = 1, n
     do j = 0, 4
        weights%weights_by_type(i,j) = 0.2 * weights%weights(i)
     end do
  end do
end subroutine ckkw_pseudo_shower_weights_fake
```

⟨*CKKW matching: public*⟩+≡
```
public :: ckkw_matching_t
```

⟨*CKKW matching: types*⟩+≡
```
type, extends (matching_t) :: ckkw_matching_t
   type(ckkw_matching_settings_t) :: settings
   type(ckkw_pseudo_shower_weights_t) :: weights
 contains
   ⟨CKKW matching: ckkw matching: TBP⟩
end type ckkw_matching_t
```

⟨*CKKW matching: ckkw matching: TBP*⟩≡
```
procedure :: init => ckkw_matching_init
```

⟨*CKKW matching: procedures*⟩+≡
```
subroutine ckkw_matching_init (matching, var_list, process_name)
  class(ckkw_matching_t), intent(out) :: matching
⟨default matching init⟩
end subroutine ckkw_matching_init
```

⟨*CKKW matching: ckkw matching: TBP*⟩+≡
```
procedure :: write => ckkw_matching_write
```

⟨*CKKW matching: procedures*⟩+≡
```
subroutine ckkw_matching_write (matching, unit)
  class(ckkw_matching_t), intent(in) :: matching
  integer, intent(in), optional :: unit
  call matching%settings%write (unit)
  call matching%weights%write (unit)
end subroutine ckkw_matching_write
```

⟨*CKKW matching: ckkw matching: TBP*⟩+≡
```
procedure :: get_method => ckkw_matching_get_method
```

⟨*CKKW matching: procedures*⟩+≡
```
function ckkw_matching_get_method (matching) result (method)
   type(string_t) :: method
   class(ckkw_matching_t), intent(in) :: matching
   method = matching_method (MATCH_CKKW)
end function ckkw_matching_get_method
```

⟨*CKKW matching: ckkw matching: TBP*⟩+≡
```
procedure :: before_shower => ckkw_matching_before_shower
```

⟨*CKKW matching: procedures*⟩+≡
```
subroutine ckkw_matching_before_shower &
       (matching, particle_set, vetoed)
  class(ckkw_matching_t), intent(inout) :: matching
  type(particle_set_t), intent(inout) :: particle_set
  logical, intent(out) :: vetoed
  call matching%weights%init ()
  call matching%weights%fake (particle_set)
  select type (shower => matching%shower)
  type is (shower_t)
```

```
      call ckkw_matching_apply (shower%partons, &
           matching%settings, &
           matching%weights, matching%rng, vetoed)
   class default
      call msg_bug ("CKKW matching only works with WHIZARD shower.")
   end select
end subroutine ckkw_matching_before_shower
```

⟨*CKKW matching: public*⟩+≡

```
public :: ckkw_matching_apply
```

⟨*CKKW matching: procedures*⟩+≡

```
subroutine ckkw_matching_apply (partons, settings, weights, rng, vetoed)
   type(parton_pointer_t), dimension(:), intent(inout), allocatable :: &
        partons
   type(ckkw_matching_settings_t), intent(in) :: settings
   type(ckkw_pseudo_shower_weights_t), intent(in) :: weights
   class(rng_t), intent(inout), allocatable :: rng
   logical, intent(out) :: vetoed

   real(default), dimension(:), allocatable :: scales
   real(double) :: weight, sf
   real(default) :: rand
   integer :: i, n_partons

   if (signal_is_pending ()) return
   weight = one

   n_partons = size (partons)

   do i = 1, n_partons
      call partons(i)%p%write ()
   end do

   !!! the pseudo parton shower is already simulated by shower_add_interaction
   !!! get the respective clustering scales
   allocate (scales (1:n_partons))
   do i = 1, n_partons
      if (.not. associated (partons(i)%p)) cycle
      if (partons(i)%p%type == INTERNAL) then
         scales(i) = two * min (partons(i)%p%child1%momentum%p(0),  &
                                partons(i)%p%child2%momentum%p(0))**2 * &
              (1.0 - (space_part (partons(i)%p%child1%momentum) * &
               space_part (partons(i)%p%child2%momentum)) / &
              (space_part (partons(i)%p%child1%momentum)**1 * &
               space_part (partons(i)%p%child2%momentum)**1))
         scales(i) = sqrt (scales(i))
         partons(i)%p%ckkwscale = scales(i)
         print *, scales(i)
      end if
   end do

   print *, " scales finished"
   !!! if (highest multiplicity) -> reweight with PDF(mu_F) / PDF(mu_cut)
```

```
do i = 1, n_partons
   call partons(i)%p%write ()
end do

!!! Reweight and possibly veto the whole event

!!! calculate the relative alpha_S weight

!! calculate the Sudakov weights for internal lines
!! calculate the Sudakov weights for external lines
do i = 1, n_partons
   if (signal_is_pending ()) return
   if (.not. associated (partons(i)%p)) cycle
   if (partons(i)%p%type == INTERNAL) then
      !!! get type
      !!! check that all particles involved are colored
      if ((partons(i)%p%is_colored () .or. &
           partons(i)%p%ckkwtype > 0) .and. &
           (partons(i)%p%child1%is_colored () .or. &
           partons(i)%p%child1%ckkwtype > 0) .and. &
           (partons(i)%p%child1%is_colored () .or. &
           partons(i)%p%child1%ckkwtype > 0)) then
         print *, "reweight with alphaS(" , partons(i)%p%ckkwscale, &
              ") for particle ", partons(i)%p%nr
         if (partons(i)%p%belongstoFSR) then
            print *, "FSR"
            weight = weight * D_alpha_s_fsr (partons(i)%p%ckkwscale**2, &
                 partons(i)%p%settings) / settings%alphas
         else
            print *, "ISR"
            weight = weight * &
                 D_alpha_s_isr (partons(i)%p%ckkwscale**2, &
                 partons(i)%p%settings) / settings%alphas
         end if
      else
         print *, "no reweight with alphaS for ", partons(i)%p%nr
      end if
      if (partons(i)%p%child1%type == INTERNAL) then
         print *, "internal line from ", &
              partons(i)%p%child1%ckkwscale, &
              " to ", partons(i)%p%ckkwscale, &
              " for type ", partons(i)%p%child1%ckkwtype
         if (partons(i)%p%child1%ckkwtype == 0) then
            sf = 1.0
         else if (partons(i)%p%child1%ckkwtype == 1) then
            sf = SudakovQ (partons(i)%p%child1%ckkwscale, &
                 partons(i)%p%ckkwscale, &
                 partons(i)%p%settings, .true., rng)
            print *, "SFQ = ", sf
         else if (partons(i)%p%child1%ckkwtype == 2) then
            sf = SudakovG (partons(i)%p%child1%ckkwscale, &
                 partons(i)%p%ckkwscale, &
                 partons(i)%p%settings, .true., rng)
            print *, "SFG = ", sf
```

```
   else
      print *, "SUSY not yet implemented"
   end if
   weight = weight * min (one, sf)
else
   print *, "external line from ", settings%Qmin, &
        partons(i)%p%ckkwscale
   if (partons(i)%p%child1%is_quark ()) then
      sf = SudakovQ (settings%Qmin, &
           partons(i)%p%ckkwscale, &
           partons(i)%p%settings, .true., rng)
      print *, "SFQ = ", sf
   else if (partons(i)%p%child1%is_gluon ()) then
      sf = SudakovG (settings%Qmin, &
           partons(i)%p%ckkwscale, &
           partons(i)%p%settings, .true., rng)
      print *, "SFG = ", sf
   else
      print *, "not yet implemented (", &
           partons(i)%p%child2%type, ")"
      sf = one
   end if
   weight = weight * min (one, sf)
end if
if (partons(i)%p%child2%type == INTERNAL) then
   print *, "internal line from ", partons(i)%p%child2%ckkwscale, &
        " to ", partons(i)%p%ckkwscale, &
        " for type ", partons(i)%p%child2%ckkwtype
   if (partons(i)%p%child2%ckkwtype == 0) then
      sf = 1.0
   else if (partons(i)%p%child2%ckkwtype == 1) then
      sf = SudakovQ (partons(i)%p%child2%ckkwscale, &
           partons(i)%p%ckkwscale, &
           partons(i)%p%settings, .true., rng)
      print *, "SFQ = ", sf
   else if (partons(i)%p%child2%ckkwtype == 2) then
      sf = SudakovG (partons(i)%p%child2%ckkwscale, &
           partons(i)%p%ckkwscale, &
           partons(i)%p%settings, .true., rng)
      print *, "SFG = ", sf
   else
      print *, "SUSY not yet implemented"
   end if
   weight = weight * min (one, sf)
else
   print *, "external line from ", settings%Qmin, &
        partons(i)%p%ckkwscale
   if (partons(i)%p%child2%is_quark ()) then
      sf = SudakovQ (settings%Qmin, &
           partons(i)%p%ckkwscale, &
           partons(i)%p%settings, .true., rng)
      print *, "SFQ = ", sf
   else if (partons(i)%p%child2%is_gluon ()) then
      sf = SudakovG (settings%Qmin, &
```

```
                    partons(i)%p%ckkwscale, &
                    partons(i)%p%settings, .true., rng)
                print *, "SFG = ", sf
             else
                print *, "not yet implemented (", &
                    partons(i)%p%child2%type, ")"
                sf = one
             end if
             weight = weight * min (one, sf)
          end if
       end if
    end do

    call rng%generate (rand)

    print *, "final weight: ", weight

    !!!!!!! WRONG
    vetoed = .false.
    ! vetoed = (rand > weight)
    if (vetoed) then
       return
    end if

    !!! finally perform the parton shower
    !!! veto emissions that are too hard

    deallocate (scales)
  end subroutine ckkw_matching_apply
```

⟨*CKKW matching: ckkw matching: TBP*⟩+≡

```
  procedure :: after_shower => ckkw_matching_after_shower
```

⟨*CKKW matching: procedures*⟩+≡

```
  subroutine ckkw_matching_after_shower (matching, particle_set, vetoed)
    class(ckkw_matching_t), intent(inout) :: matching
       type(particle_set_t), intent(inout) :: particle_set
       logical, intent(out) :: vetoed
       vetoed = .false.
  end subroutine ckkw_matching_after_shower
```

⟨*CKKW matching: procedures*⟩+≡

```
  function GammaQ (smallq, largeq, settings, fsr) result (gamma)
    real(default), intent(in) :: smallq, largeq
    type(shower_settings_t), intent(in) :: settings
    logical, intent(in) :: fsr
    real(default) :: gamma
    gamma = (8._default / three) / (pi * smallq)
    gamma = gamma * (log(largeq / smallq) - 0.75)
    if (fsr) then
       gamma = gamma * D_alpha_s_fsr (smallq**2, settings)
    else
       gamma = gamma * D_alpha_s_isr (smallq**2, settings)
    end if
```

```
end function GammaQ
```

⟨*CKKW matching: procedures*⟩+≡

```
function GammaG (smallq, largeq, settings, fsr) result (gamma)
  real(default), intent(in) :: smallq, largeq
  type(shower_settings_t), intent(in) :: settings
  logical, intent(in) :: fsr
  real(default) :: gamma
  gamma = 6._default / (pi * smallq)
  gamma = gamma *( log(largeq / smallq) - 11.0 / 12.0)
  if (fsr) then
     gamma = gamma * D_alpha_s_fsr (smallq**2, settings)
  else
     gamma = gamma * D_alpha_s_isr (smallq**2, settings)
  end if
end function GammaG
```

⟨*CKKW matching: procedures*⟩+≡

```
function GammaF (smallq, settings, fsr) result (gamma)
  real(default), intent(in) :: smallq
  type(shower_settings_t), intent(in) :: settings
  logical, intent(in) :: fsr
  real(default) :: gamma
  gamma = number_of_flavors (smallq, settings%max_n_flavors, &
       settings%min_virtuality) / (three * pi * smallq)
  if (fsr) then
     gamma = gamma * D_alpha_s_fsr (smallq**2, settings)
  else
     gamma = gamma * D_alpha_s_isr (smallq**2, settings)
  end if
end function GammaF
```

⟨*CKKW matching: procedures*⟩+≡

```
function SudakovQ (Q1, Q, settings, fsr, rng) result (sf)
  real(default), intent(in) :: Q1, Q
  type(shower_settings_t), intent(in) :: settings
  class(rng_t), intent(inout), allocatable :: rng
  logical, intent(in) :: fsr
  real(default) :: sf
  real(default) :: integral
  integer, parameter :: NTRIES = 100
  integer :: i
  real(default) :: rand
  integral = zero
  do i = 1, NTRIES
     call rng%generate (rand)
     integral = integral + GammaQ (Q1 + rand * (Q - Q1), Q, settings, fsr)
  end do
  integral = integral / NTRIES
  sf = exp (-integral)
end function SudakovQ
```

⟨*CKKW matching: procedures*⟩+≡

```
function SudakovG (Q1, Q, settings, fsr, rng) result (sf)
  real(default), intent(in) :: Q1, Q
  type(shower_settings_t), intent(in) :: settings
  logical, intent(in) :: fsr
  real(default) :: sf
  real(default) :: integral
  class(rng_t), intent(inout), allocatable :: rng
  integer, parameter :: NTRIES = 100
  integer :: i
  real(default) :: rand
  integral = zero
  do i = 1, NTRIES
     call rng%generate (rand)
     integral = integral + &
          GammaG (Q1 + rand * (Q - Q1), Q, settings, fsr) + &
          GammaF (Q1 + rand * (Q - Q1), settings, fsr)
  end do
  integral = integral / NTRIES
  sf = exp (-integral)
end function SudakovG
```

31.4 POWHEG

This module generates radiation according to the POWHEG Sudakov form factor

$$\Delta^{f_b}(\Phi_n, p_{\mathrm{T}}) = \prod_{\alpha_r \in \{\alpha_r | f_b\}} \Delta^{f_b}_{\alpha_r}(\Phi_n, p_{\mathrm{T}}), \tag{31.1}$$

with

$$\Delta^{f_b}_{\alpha_r}(\Phi_n, p_{\mathrm{T}}) = \exp\left\{ -\left[\int d\Phi_{\mathrm{rad}} \, \frac{R(\Phi_{n+1})}{B^{f_b}(\Phi_n)} \, \theta(k_{\mathrm{T}}(\Phi_{n+1}) - p_{\mathrm{T}}) \right]_{\alpha_r}^{\bar{\mathbf{\Phi}}_n^{\alpha_r} = \Phi_n} \right\} \tag{31.2}$$

We expect that an underlying Born flavor structure f_b has been generated with a probability proportional to its contribution to the $\tilde{B}$ at the given kinematic point.

⟨powheg_matching.f90⟩≡

```
⟨File header⟩

module powheg_matching

  use, intrinsic :: iso_fortran_env

⟨Use kinds⟩
⟨Use strings⟩
  use diagnostics
  use constants, only: ZERO, ONE, TWO, FIVE, TINY_07, PI, TWOPI
  use numeric_utils
  use io_units, only: given_output_unit, free_unit
  use format_utils, only: write_separator
```

```
use format_defs, only: FMT_16, FMT_19
use string_utils, only: str
use os_interface, only: os_file_exist
use physics_defs, only: CA, BORN, NLO_REAL
use lorentz
use sm_qcd, only: qcd_t, alpha_qcd_from_scale_t, alpha_qcd_from_lambda_t
use sm_physics, only: Li2
use subevents, only: PRT_INCOMING, PRT_OUTGOING
use colors
use particles
use grids
use solver
use rng_base
use variables

use phs_fks, only: phs_fks_generator_t, compute_dalitz_bounds, beta_emitter
use phs_fks, only: phs_point_set_t, phs_identifier_t, phs_fks_t
use phs_fks, only: I_XI, I_Y, FSR_SIMPLE, FSR_MASSIVE
use matching_base
use pcm, only: pcm_nlo_t, pcm_instance_nlo_t
```

⟨*Standard module head*⟩

⟨*POWHEG matching: public*⟩

⟨*POWHEG matching: parameters*⟩

⟨*POWHEG matching: types*⟩

⟨*POWHEG matching: interfaces*⟩

```
contains
```

⟨*POWHEG matching: procedures*⟩

```
end module powheg_matching
```

31.4.1 Base types for settings and data

`lambda` enters for now as the lowest scale $2\Lambda^{(}5)_{\overline{\mathrm{MS}}}$ where the radiation α_s^{rad} is still larger than the true α_s.

⟨*POWHEG matching: public*⟩≡

```
public :: powheg_settings_t
```

⟨*POWHEG matching: types*⟩≡

```
type :: powheg_settings_t
   real(default) :: pt2_min = zero
   real(default) :: lambda = zero
   integer :: n_init = 0
   integer :: size_grid_xi = 0
   integer :: size_grid_y = 0
   integer :: upper_bound_func = UBF_SIMPLE
   logical :: rebuild_grids = .false.
   logical :: test_sudakov = .false.
```

```
    logical :: disable_sudakov = .false.
    logical :: singular_jacobian = .false.
contains
 ⟨POWHEG matching: powheg settings: TBP⟩
end type powheg_settings_t
```

These are the possible values for `upper_bound_func` and will be used to decide which ubf object is allocated.

⟨*POWHEG matching: parameters*⟩≡

```
integer, parameter :: UBF_SIMPLE = 1
integer, parameter :: UBF_EEQQ = 2
integer, parameter :: UBF_MASSIVE = 3
```

⟨*POWHEG matching: powheg settings: TBP*⟩≡

```
procedure :: init => powheg_settings_init
```

⟨*POWHEG matching: procedures*⟩≡

```
subroutine powheg_settings_init (settings, var_list)
  class(powheg_settings_t), intent(out) :: settings
  type(var_list_t), intent(in) :: var_list
  settings%pt2_min = &
       var_list%get_rval (var_str ("powheg_pt_min"))**2
  settings%size_grid_xi = &
       var_list%get_ival (var_str ("powheg_grid_size_xi"))
  settings%size_grid_y = &
       var_list%get_ival (var_str ("powheg_grid_size_y"))
  settings%n_init = &
       var_list%get_ival (var_str ("powheg_grid_sampling_points"))
  settings%lambda = var_list%get_rval (var_str ("powheg_lambda"))
  settings%rebuild_grids = &
       var_list%get_lval (var_str ("?powheg_rebuild_grids"))
  settings%singular_jacobian = &
       var_list%get_lval (var_str ("?powheg_use_singular_jacobian"))
  settings%test_sudakov = &
       var_list%get_lval (var_str ("?powheg_test_sudakov"))
  settings%disable_sudakov = &
       var_list%get_lval (var_str ("?powheg_disable_sudakov"))
end subroutine powheg_settings_init
```

⟨*POWHEG matching: powheg settings: TBP*⟩+≡

```
procedure :: write => powheg_settings_write
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_settings_write (powheg_settings, unit)
  class(powheg_settings_t), intent(in) :: powheg_settings
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1X,A)") "POWHEG settings:"
  write (u, "(3X,A," // FMT_16 //")") "pt2_min = ", powheg_settings%pt2_min
  write (u, "(3X,A," // FMT_16 //")") "lambda = ", powheg_settings%lambda
  write (u, "(3X,A,I12)") "n_init = ", powheg_settings%n_init
  write (u, "(3X,A,I12)") "size_grid_xi = ", powheg_settings%size_grid_xi
  write (u, "(3X,A,I12)") "size_grid_y = ", powheg_settings%size_grid_y
```

```
    write (u, "(3X,A,I12)") "upper_bound_func = ", powheg_settings%upper_bound_func
  end subroutine powheg_settings_write
```

⟨*POWHEG matching: public*⟩+≡

```
  public :: radiation_t
```

⟨*POWHEG matching: types*⟩+≡

```
  type :: radiation_t
    real(default) :: xi, y, phi, pt2
    integer :: alr
    logical :: valid = .false.
  contains
  ⟨POWHEG matching: radiation: TBP⟩
  end type radiation_t
```

⟨*POWHEG matching: radiation: TBP*⟩≡

```
  procedure :: write => radiation_write
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine radiation_write (radiation, unit)
    class(radiation_t), intent(in) :: radiation
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(1X, A)") "Radiation:"
    write (u, "(3X, A," // FMT_16 // ")") "xi = ", radiation%xi
    write (u, "(3X, A," // FMT_16 // ")") "y = ", radiation%y
    write (u, "(3X, A," // FMT_16 // ")") "phi = ", radiation%phi
    write (u, "(3X, A," // FMT_16 // ")") "pt2 = ", radiation%pt2
    write (u, "(3X, A, I12)") "alr = ", radiation%alr
  end subroutine radiation_write
```

`lambda2_gen` ≠ `lambda` and is used in the upper bounding functions.

⟨*POWHEG matching: public*⟩+≡

```
  public :: process_deps_t
```

⟨*POWHEG matching: types*⟩+≡

```
  type :: process_deps_t
     real(default) :: lambda2_gen, sqrts
     integer :: n_alr
     logical :: cm_frame = .true.
     type(phs_identifier_t), dimension(:), allocatable :: phs_identifiers
     integer, dimension(:), allocatable :: alr_to_i_phs
     integer :: i_born
     integer, dimension(:), allocatable :: i_real
  contains
   ⟨POWHEG matching: process deps: TBP⟩
  end type process_deps_t
```

⟨*POWHEG matching: process deps: TBP*⟩≡

```
  procedure :: write => process_deps_write
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine process_deps_write (process_deps, unit)
  class(process_deps_t), intent(in) :: process_deps
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1X,A)") "Process dependencies:"
  write (u, "(3X,A," // FMT_19 // ")") "lambda2_gen = ", process_deps%lambda2_gen
  write (u, "(3X,A, I12)") "n_alr = ", process_deps%n_alr
end subroutine process_deps_write
```

⟨*POWHEG matching: public*⟩+≡

```
public :: event_deps_t
```

⟨*POWHEG matching: types*⟩+≡

```
type :: event_deps_t
   real(default) :: s_hat
   type(phs_point_set_t) :: p_born_cms
   type(phs_point_set_t) :: p_born_lab
   type(phs_point_set_t) :: p_real_cms
   type(phs_point_set_t) :: p_real_lab
   real(default) :: sqme_born
contains
 ⟨POWHEG matching: event deps: TBP⟩
end type event_deps_t
```

⟨*POWHEG matching: event deps: TBP*⟩≡

```
procedure :: write => event_deps_write
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine event_deps_write (event_deps, unit)
  class(event_deps_t), intent(in) :: event_deps
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1X,A)") "Event dependencies:"
  write (u, "(3X,A," // FMT_19 // ")") "s_hat = ", event_deps%s_hat
  write (u, "(3X,A," // FMT_19 // ")") "sqme_born = ", event_deps%sqme_born
end subroutine event_deps_write
```

⟨*POWHEG matching: event deps: TBP*⟩+≡

```
procedure :: update => event_deps_update
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine event_deps_update (event_deps, sqme_born, p_born, lt_lab_to_cms)
  class(event_deps_t), intent(inout) :: event_deps
  real(default), intent(in) :: sqme_born
  type(vector4_t), dimension(:), intent(in) :: p_born
  type(lorentz_transformation_t), intent(in), optional :: lt_lab_to_cms
  integer :: n_born
  event_deps%sqme_born = sqme_born
  n_born = size (p_born)
  if (debug_active (D_MATCHING)) then
     if (n_born /= event_deps%p_born_lab%get_n_particles (1)) then
```

```
          call msg_fatal &
               ("event_deps_update: number of born_momenta has changed")
       end if
    end if
    call event_deps%p_born_lab%set_momenta (1, p_born)
    call event_deps%set_cms (lt_lab_to_cms)
  end subroutine event_deps_update
```

This has to be changed when we have sorted out the handling of ISR, partonic vs hadronic cms as well as decays in POWHEG:

⟨*POWHEG matching: event deps: TBP*⟩+≡

```
  procedure :: set_cms => event_deps_set_cms
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine event_deps_set_cms (event_deps, lt_lab_to_cms)
    class(event_deps_t), intent(inout) :: event_deps
    type(lorentz_transformation_t), intent(in), optional :: lt_lab_to_cms
    associate (p => event_deps%p_born_lab%phs_point(1)%p)
       event_deps%s_hat = (p(1) + p(2))**2
       if (present (lt_lab_to_cms)) then
          event_deps%p_born_cms%phs_point(1)%p = lt_lab_to_cms * p
       else
          event_deps%p_born_cms%phs_point(1)%p = p
       end if
    end associate
  end subroutine event_deps_set_cms
```

⟨*POWHEG matching: types*⟩+≡

```
  type :: veto_counter_t
    integer :: n_ubf = 0
    integer :: n_first_fail = 0
    integer :: n_alpha_s = 0
    integer :: n_xi_max = 0
    integer :: n_norm = 0
    integer :: n_sqme = 0
    integer :: veto_ubf = 0
    integer :: veto_alpha_s = 0
    integer :: veto_xi_max = 0
    integer :: veto_norm = 0
    integer :: veto_sqme = 0
    integer :: n_veto_fail = 0
  contains
  ⟨POWHEG matching: veto counter: TBP⟩
  end type veto_counter_t
```

⟨*POWHEG matching: veto counter: TBP*⟩≡

```
  procedure :: record_ubf => veto_counter_record_ubf
```

⟨*POWHEG matching: procedures*⟩+≡

```
  pure subroutine veto_counter_record_ubf (counter, vetoed)
    class(veto_counter_t), intent(inout) :: counter
    logical, intent(in) :: vetoed
    counter%n_ubf = counter%n_ubf + 1
```

```
    if (vetoed) counter%veto_ubf = counter%veto_ubf + 1
  end subroutine veto_counter_record_ubf
```

⟨*POWHEG matching: veto counter: TBP*⟩+≡

```
  procedure :: record_first_fail => veto_counter_record_first_fail
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine veto_counter_record_first_fail (counter)
    class(veto_counter_t), intent(inout) :: counter
    counter%n_first_fail = counter%n_first_fail + 1
  end subroutine veto_counter_record_first_fail
```

⟨*POWHEG matching: veto counter: TBP*⟩+≡

```
  procedure :: record_alpha_s => veto_counter_record_alpha_s
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine veto_counter_record_alpha_s (counter, vetoed)
    class(veto_counter_t), intent(inout) :: counter
    logical, intent(in) :: vetoed
    counter%n_alpha_s = counter%n_alpha_s + 1
    if (vetoed) counter%veto_alpha_s = counter%veto_alpha_s + 1
  end subroutine veto_counter_record_alpha_s
```

⟨*POWHEG matching: veto counter: TBP*⟩+≡

```
  procedure :: record_xi_max => veto_counter_record_xi_max
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine veto_counter_record_xi_max (counter, vetoed)
    class(veto_counter_t), intent(inout) :: counter
    logical, intent(in) :: vetoed
    counter%n_xi_max = counter%n_xi_max + 1
    if (vetoed) counter%veto_xi_max = counter%veto_xi_max + 1
  end subroutine veto_counter_record_xi_max
```

⟨*POWHEG matching: veto counter: TBP*⟩+≡

```
  procedure :: record_norm => veto_counter_record_norm
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine veto_counter_record_norm (counter, vetoed)
    class(veto_counter_t), intent(inout) :: counter
    logical, intent(in) :: vetoed
    counter%n_norm = counter%n_norm + 1
    if (vetoed) counter%veto_norm = counter%veto_norm + 1
  end subroutine veto_counter_record_norm
```

⟨*POWHEG matching: veto counter: TBP*⟩+≡

```
  procedure :: record_sqme => veto_counter_record_sqme
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine veto_counter_record_sqme (counter, vetoed)
    class(veto_counter_t), intent(inout) :: counter
    logical, intent(in) :: vetoed
    counter%n_sqme = counter%n_sqme + 1
```

```
    if (vetoed) counter%veto_sqme = counter%veto_sqme + 1
  end subroutine veto_counter_record_sqme
```

⟨*POWHEG matching: veto counter: TBP*⟩+≡

```
  procedure :: record_fail => veto_counter_record_fail
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine veto_counter_record_fail (counter)
    class(veto_counter_t), intent(inout) :: counter
    counter%n_veto_fail = counter%n_veto_fail + 1
  end subroutine veto_counter_record_fail
```

⟨*POWHEG matching: veto counter: TBP*⟩+≡

```
  procedure :: write => veto_counter_write
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine veto_counter_write (counter, unit)
    class(veto_counter_t), intent(in) :: counter
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit); if (u < 0) return
    write (u, "(A,I12)") "Nr. of ubf-veto calls: ", counter%n_ubf
    write (u, "(A,I12)") "Nr. of ubf-vetos: ", counter%veto_ubf
    if (counter%n_ubf > 0) &
       write (u, "(A,F4.2)") "Fraction of vetoed points: ", &
                             one*counter%veto_ubf / counter%n_ubf
    call write_separator (u)

    write (u, "(A,I12)") "Nr. of alpha_s-veto calls: ", counter%n_alpha_s
    write (u, "(A,I12)") "Nr. of alpha_s-vetos: ", counter%veto_alpha_s
    if (counter%n_alpha_s > 0) &
       write (u, "(A,F4.2)") "Fraction of vetoed points: ", &
                             one*counter%veto_alpha_s / counter%n_alpha_s
    call write_separator (u)

    write (u, "(A,I12)") "Nr. of xi_max-veto calls: ", counter%n_xi_max
    write (u, "(A,I12)") "Nr. of xi_max-vetos: ", counter%veto_xi_max
    if (counter%n_alpha_s > 0) &
       write (u, "(A,F4.2)") "Fraction of vetoed points: ", &
                             one*counter%veto_xi_max / counter%n_xi_max
    call write_separator (u)

    write (u, "(A,I0)") "Nr. of norm-veto calls: ", counter%n_norm
    write (u, "(A,I0)") "Nr. of norm-vetos: ", counter%veto_norm
    if (counter%n_norm > 0) &
       write (u, "(A,F4.2)") "Fraction of vetoed points: ", &
                             one*counter%veto_norm / counter%n_norm
    call write_separator (u)

    write (u, "(A,I0)") "Nr. of sqme-veto calls: ", counter%n_sqme
    write (u, "(A,I0)") "Nr. of sqme-vetos: ", counter%veto_sqme
    if (counter%n_sqme > 0) &
       write (u, "(A,F4.2)") "Fraction of vetoed points: ", &
                             one*counter%veto_sqme / counter%n_sqme
```

```
    call write_separator (u)
    write (u, "(A,I0)") "Nr. of upper-bound failures: ", &
                        counter%n_veto_fail
  end subroutine veto_counter_write
```

31.4.2 Upper bounding functions and sudakovs

Abstract version

This contains the pieces that depend on the radiation region α_r

⟨*POWHEG matching: public*⟩+≡

```
  public :: sudakov_t
```

⟨*POWHEG matching: types*⟩+≡

```
  type, abstract, extends (solver_function_t) :: sudakov_t
     type(process_deps_t), pointer :: process_deps => null()
     type(event_deps_t), pointer :: event_deps => null()
     type(powheg_settings_t), pointer :: powheg_settings => null()
     type(phs_fks_generator_t), pointer :: phs_fks_generator => null()
     type(qcd_t), pointer :: qcd => null()
     class(rng_t), pointer :: rng => null()
     real(default) :: xi2_max = zero
     real(default) :: norm_max = zero
     real(default) :: current_pt2_max = zero
     real(default) :: last_log = zero
     real(default) :: random = zero
     type(veto_counter_t) :: veto_counter
     integer :: i_phs = 0
  contains
   ⟨POWHEG matching: sudakov: TBP⟩
  end type sudakov_t
```

⟨*POWHEG matching: sudakov: TBP*⟩≡

```
  procedure :: write => sudakov_write
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine sudakov_write (sudakov, unit)
    class(sudakov_t), intent(in) :: sudakov
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    write (u, "(3X,A," // FMT_19 // ")")  "xi2_max = ", sudakov%xi2_max
    write (u, "(3X,A," // FMT_19 // ")")  "norm_max = ", sudakov%norm_max
    write (u, "(3X,A," // FMT_19 // ")")  &
         "current_pt2_max = ", sudakov%current_pt2_max
    write (u, "(3X,A," // FMT_19 // ")")  "last_log = ", sudakov%last_log
    write (u, "(3X,A," // FMT_19 // ")")  "random = ", sudakov%random
  end subroutine sudakov_write
```

To allow for arrays of this class

⟨*POWHEG matching: public*⟩+≡

```
  public :: sudakov_wrapper_t
```

⟨*POWHEG matching: types*⟩+≡
```
type :: sudakov_wrapper_t
   class(sudakov_t), allocatable :: s
end type sudakov_wrapper_t
```

⟨*POWHEG matching: sudakov: TBP*⟩+≡
```
procedure :: init => sudakov_init
```

⟨*POWHEG matching: procedures*⟩+≡
```
subroutine sudakov_init (sudakov, process_deps, event_deps, &
       powheg_settings, phs_fks_generator, qcd, rng)
  class(sudakov_t), intent(out) :: sudakov
  type(process_deps_t), target, intent(in) :: process_deps
  type(event_deps_t), target, intent(in) :: event_deps
  type(powheg_settings_t), target, intent(in) :: powheg_settings
  type(phs_fks_generator_t), target, intent(in) :: phs_fks_generator
  type(qcd_t), target, intent(in) :: qcd
  class(rng_t), target, intent(in) :: rng
  sudakov%process_deps => process_deps
  sudakov%event_deps => event_deps
  sudakov%powheg_settings => powheg_settings
  sudakov%phs_fks_generator => phs_fks_generator
  sudakov%qcd => qcd
  sudakov%rng => rng
end subroutine sudakov_init
```

This has to be done after the grids are initialized.

⟨*POWHEG matching: sudakov: TBP*⟩+≡
```
procedure :: set_normalization => sudakov_set_normalization
```

⟨*POWHEG matching: procedures*⟩+≡
```
pure subroutine sudakov_set_normalization (sudakov, norm_max)
  class(sudakov_t), intent(inout) :: sudakov
  real(default), intent(in) :: norm_max
  sudakov%norm_max = norm_max
end subroutine sudakov_set_normalization
```

⟨*POWHEG matching: sudakov: TBP*⟩+≡
```
procedure :: update => sudakov_update
```

⟨*POWHEG matching: procedures*⟩+≡
```
pure subroutine sudakov_update (sudakov, xi2_max)
  class(sudakov_t), intent(inout) :: sudakov
  real(default), intent(in) :: xi2_max
  sudakov%xi2_max = xi2_max
end subroutine sudakov_update
```

`upper_bound_func` does *not* contain the normalization N which is given by the grids. In the notation of `1002.2581`, it is thus $\frac{1}{N}U(\xi, y)$

⟨*POWHEG matching: sudakov: TBP*⟩+≡
```
procedure (sudakov_upper_bound_func), deferred :: upper_bound_func
```

⟨*POWHEG matching: interfaces*⟩≡

```
abstract interface
   pure function sudakov_upper_bound_func (sudakov, xi, y, alpha_s) result (u)
     import
     real(default) :: u
     class(sudakov_t), intent(in) :: sudakov
     real(default), intent(in) :: xi, y, alpha_s
   end function sudakov_upper_bound_func
end interface
```

Similar to the upper_bound_func, this is $-\frac{1}{N}\log\Delta(p_T^2)$ where

$$\Delta^{(U)}(p_T) = \exp - \int U(\xi, y)\theta(k_T - p_T)\, d\xi dy d\phi \tag{31.3}$$

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
procedure (sudakov_log_integrated_ubf), deferred :: log_integrated_ubf
```

⟨*POWHEG matching: interfaces*⟩+≡

```
abstract interface
   pure function sudakov_log_integrated_ubf (sudakov, pt2) result (y)
     import
     real(default) :: y
     class(sudakov_t), intent(in) :: sudakov
     real(default), intent(in) :: pt2
   end function sudakov_log_integrated_ubf
end interface
```

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
procedure (sudakov_generate_xi_and_y_and_phi), deferred :: generate_xi_and_y_and_phi
```

⟨*POWHEG matching: interfaces*⟩+≡

```
abstract interface
   subroutine sudakov_generate_xi_and_y_and_phi (sudakov, r)
     import
     class(sudakov_t), intent(inout) :: sudakov
     type(radiation_t), intent(inout) :: r
   end subroutine sudakov_generate_xi_and_y_and_phi
end interface
```

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
procedure (sudakov_kt2), deferred :: kt2
```

⟨*POWHEG matching: interfaces*⟩+≡

```
abstract interface
   function sudakov_kt2 (sudakov, xi, y) result (kt2)
     import
     real(default) :: kt2
     class(sudakov_t), intent(in) :: sudakov
     real(default), intent(in) :: xi, y
   end function sudakov_kt2
end interface
```

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
procedure (sudakov_kt2_max), deferred :: kt2_max
```

⟨*POWHEG matching: interfaces*⟩+≡

```
abstract interface
   pure function sudakov_kt2_max (sudakov, s_hat) result (kt2_max)
      import
      real(default) :: kt2_max
      class(sudakov_t), intent(in) :: sudakov
      real(default), intent(in) :: s_hat
  end function sudakov_kt2_max
end interface
```

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
procedure (sudakov_reweight_ubf), deferred :: reweight_ubf
```

⟨*POWHEG matching: interfaces*⟩+≡

```
abstract interface
   function sudakov_reweight_ubf (sudakov, pt2) result (accepted)
     import
     logical :: accepted
     class(sudakov_t), intent(inout) :: sudakov
     real(default), intent(in) :: pt2
  end function sudakov_reweight_ubf
end interface
```

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
procedure (sudakov_reweight_xi_max), deferred :: reweight_xi_max
```

⟨*POWHEG matching: interfaces*⟩+≡

```
abstract interface
   function sudakov_reweight_xi_max (sudakov, xi) result (accepted)
     import
     logical :: accepted
     class(sudakov_t), intent(in) :: sudakov
     real(default), intent(in) :: xi
   end function sudakov_reweight_xi_max
end interface
```

In the generation of p_T^2 via `log_integrated_ubf`, we use the simplified version α_s^{rad} while the grids take the improved version.

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
procedure :: alpha_s => sudakov_alpha_s
```

⟨*POWHEG matching: procedures*⟩+≡

```
function sudakov_alpha_s (sudakov, kT2, use_correct) result (a)
  real(default) :: a
  class(sudakov_t), intent(in) :: sudakov
  real(default), intent(in) :: kT2
  logical, intent(in), optional :: use_correct
  logical :: yorn
  yorn = .false.; if (present (use_correct)) yorn = use_correct
  if (yorn) then
     a = get_alpha (sudakov%qcd, kT2)
```

```
    else
       a = sudakov%alpha_s_rad (kT2)
    end if
  end function sudakov_alpha_s
```

We have to solve the equation

$$\frac{\log \Delta^{(U)}(p_T)}{\log \Delta^{(U)}(p_T^{\max}} = \log \Delta^{(U)}(p_T) = \log r_1$$

iteratively for p_T. If the current emission is not accepted, in the next step it is $\log \Delta^{(U)}(p_T^{\max}) = \log r_1$, so that we have to solve the equation

$$\log \Delta^{(U)}(p_T) = \log r_1 + \log r_2$$

using the second random number r_2.

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
  procedure :: generate_pt2 => sudakov_generate_pt2
```

⟨*POWHEG matching: procedures*⟩+≡

```
  function sudakov_generate_pt2 (sudakov) result (pt2)
    real(default) :: pt2
    class(sudakov_t), intent(inout) :: sudakov
    logical :: success
    success = .false.
    if (sudakov%current_pt2_max > sudakov%powheg_settings%pt2_min) then
       call sudakov%rng%generate (sudakov%random)
       sudakov%last_log = sudakov%last_log + log(sudakov%random)
       pt2 = solve_interval (sudakov, &
            sudakov%powheg_settings%pt2_min, &
            sudakov%current_pt2_max, success, &
            0.001_default)
       !sudakov%last_log = sudakov%norm_max * sudakov%log_integrated_ubf (pt2)
              !sudakov%last_log + &
    end if
    if (.not. success) then
       pt2 = sudakov%powheg_settings%pt2_min
    end if
  end function sudakov_generate_pt2
```

This could be activated `if (debug_active (MATCHING))`.

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
  procedure :: check_solution_interval => sudakov_check_solution_interval
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine sudakov_check_solution_interval (sudakov)
    class(sudakov_t), intent(inout) :: sudakov
    real(default) :: r
    real(default), parameter :: dr = 0.05
    real(default) :: pt2
    logical :: success
    r = 0._default
    do
       r = r + dr
```

```
      sudakov%random = r
      pt2 = solve_interval (sudakov, &
        sudakov%powheg_settings%pt2_min, &
        sudakov%current_pt2_max, success, &
        0.001_default)
     if (success) then
        print *, 'r: ', r, ' zero found'
     else
        print *, 'r: ', r, 'no zero found'
     end if
     if (r >= 1._default) exit
   end do
end subroutine sudakov_check_solution_interval
```

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
procedure :: generate_emission => sudakov_generate_emission
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine sudakov_generate_emission (sudakov, r)
  class(sudakov_t), intent(inout) :: sudakov
  type(radiation_t), intent(inout) :: r
  logical :: accepted
  sudakov%current_pt2_max = r%pt2
  call sudakov%generate_xi_and_y_and_phi (r)
  !sudakov%last_log = sudakov%norm_max * &
       !sudakov%log_integrated_ubf (sudakov%current_pt2_max)
  call msg_debug2 (D_MATCHING, "sudakov_generate_emission")
  call msg_debug2 (D_MATCHING, "sqrt (sudakov%current_pt2_max)", &
       sqrt (sudakov%current_pt2_max))
  call msg_debug2 (D_MATCHING, "sudakov%last_log", sudakov%last_log)
  LOOP_UNTIL_ACCEPTED: do
     if (signal_is_pending ())  return
     r%valid = .false.
     r%pt2 = sudakov%generate_pt2 ()
     call msg_debug2 (D_MATCHING, "sudakov_generate_emission: after generate_pt2")
     call msg_debug2 (D_MATCHING, "sqrt (r%pt2)", sqrt (r%pt2))
     call msg_debug2 (D_MATCHING, "sudakov%last_log", sudakov%last_log)
     if (r%pt2 <= sudakov%powheg_settings%pt2_min) then
        exit
     end if
     accepted = sudakov%reweight_ubf (r%pt2)
     call sudakov%veto_counter%record_ubf (.not. accepted)
     if (.not. accepted) then
        sudakov%current_pt2_max = r%pt2
        cycle
     end if
     accepted = sudakov%reweight_alpha_s (r%pt2)
     call sudakov%veto_counter%record_alpha_s (.not. accepted)
     if (.not. accepted) then
        sudakov%current_pt2_max = r%pt2
        cycle
     end if
     call sudakov%generate_xi_and_y_and_phi (r)
     accepted = sudakov%reweight_xi_max (r%xi)
```

```
      call sudakov%veto_counter%record_xi_max (.not. accepted)
      if (.not. accepted) then
         sudakov%current_pt2_max = r%pt2
         cycle
      end if
      if (debug_active (D_MATCHING)) then
         call assert_equal (OUTPUT_UNIT, r%pt2, &
              sudakov%kt2 (r%xi, r%y), &
              "sudakov_generate_xi_and_y_and_phi: pt2 inconsistency")
         ! for this we have to recompute z?
         !call msg_bug ()
      end if
      r%valid = .true.
      exit
    end do LOOP_UNTIL_ACCEPTED
  end subroutine sudakov_generate_emission
```

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
  procedure :: evaluate => sudakov_evaluate
```

⟨*POWHEG matching: procedures*⟩+≡

```
  function sudakov_evaluate (solver_f, x) result (f)
    complex(default) :: f
    class(sudakov_t), intent(in) :: solver_f
    real(default), intent(in) :: x
    f = solver_f%last_log + solver_f%norm_max * solver_f%log_integrated_ubf (x)
    !f = log (solver_f%random) + solver_f%norm_max * solver_f%log_integrated_ubf (x) &
         !- solver_f%last_log
  end function sudakov_evaluate
```

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
  procedure :: associated_emitter => sudakov_associated_emitter
```

⟨*POWHEG matching: procedures*⟩+≡

```
  elemental function sudakov_associated_emitter (sudakov) result (emitter)
    integer :: emitter
    class(sudakov_t), intent(in) :: sudakov
    emitter = sudakov%process_deps%phs_identifiers(sudakov%i_phs)%emitter
  end function sudakov_associated_emitter
```

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
  procedure :: set_i_phs => sudakov_set_i_phs
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine sudakov_set_i_phs (sudakov, alr)
    class(sudakov_t), intent(inout) :: sudakov
    integer, intent(in) :: alr
    sudakov%i_phs = sudakov%process_deps%alr_to_i_phs(alr)
  end subroutine sudakov_set_i_phs
```

Simple FSR

This corresponds to Appendix C of 1002.2581

⟨*POWHEG matching: public*⟩+≡
```
public :: sudakov_simple_fsr_t
```

⟨*POWHEG matching: types*⟩+≡
```
type, extends (sudakov_t) :: sudakov_simple_fsr_t
contains
 ⟨POWHEG matching: sudakov simple fsr: TBP⟩
end type sudakov_simple_fsr_t
```

The simplest upper bounding function for final-state radiation is

$$\texttt{upper_bound_func} = \frac{U(\xi, y)}{N} = \frac{\alpha_s}{\xi(1-y)} \tag{31.4}$$

⟨*POWHEG matching: sudakov simple fsr: TBP*⟩≡
```
procedure :: upper_bound_func => sudakov_simple_fsr_upper_bound_func
```

⟨*POWHEG matching: procedures*⟩+≡
```
pure function sudakov_simple_fsr_upper_bound_func (sudakov, xi, y, alpha_s) result (u)
  real(default) :: u
  class(sudakov_simple_fsr_t), intent(in) :: sudakov
  real(default), intent(in) :: xi, y, alpha_s
  u = alpha_s / (xi * (1 - y))
end function sudakov_simple_fsr_upper_bound_func
```

The above upper bounding function corresponds to the transverse momentum scale

$$k_T^2 = \frac{s}{2}\xi^2(1-y). \tag{31.5}$$

⟨*POWHEG matching: sudakov simple fsr: TBP*⟩+≡
```
procedure :: kt2 => sudakov_simple_fsr_kt2
```

⟨*POWHEG matching: procedures*⟩+≡
```
function sudakov_simple_fsr_kt2 (sudakov, xi, y) result (kt2)
  real(default) :: kt2
  class(sudakov_simple_fsr_t), intent(in) :: sudakov
  real(default), intent(in) :: xi, y
  kt2 = sudakov%phs_fks_generator%real_kinematics%kt2 &
     (sudakov%i_phs, sudakov%associated_emitter (), FSR_SIMPLE, xi, y)
end function sudakov_simple_fsr_kt2
```

For massless emitters, the upper bound on the radiated energy is

$$t_{\max} = \xi_{\max}^2 \hat{s}$$

⟨*POWHEG matching: sudakov simple fsr: TBP*⟩+≡
```
procedure :: kt2_max => sudakov_simple_fsr_kt2_max
```

⟨*POWHEG matching: procedures*⟩+≡

```
pure function sudakov_simple_fsr_kt2_max (sudakov, s_hat) result (kt2_max)
   real(default) :: kt2_max
   class(sudakov_simple_fsr_t), intent(in) :: sudakov
   real(default), intent(in) :: s_hat
   kt2_max = sudakov%xi2_max * s_hat
end function sudakov_simple_fsr_kt2_max
```

This is

$$-\frac{\log\Delta^{(U)}(p_T)}{N} = \frac{\pi}{b_0}\theta\left(\xi^2_{\max} - \frac{p_T^2}{s}\right)\left[\log\frac{\xi^2_{\max}s}{\Lambda^2}\log\frac{\log\xi^2_{\max}s/\Lambda^2}{\log p_T^2/\Lambda^2} - \log\frac{\xi^2_{\max}s}{p_T^2}\right] \tag{31.6}$$

with $p^2_{\mathrm{T,max}} = \xi^2_{\max}s$.

⟨*POWHEG matching: sudakov simple fsr: TBP*⟩+≡

```
procedure :: log_integrated_ubf => sudakov_simple_fsr_log_integrated_ubf
```

⟨*POWHEG matching: procedures*⟩+≡

```
pure function sudakov_simple_fsr_log_integrated_ubf (sudakov, pt2) result (y)
  real(default) :: y
  class(sudakov_simple_fsr_t), intent(in) :: sudakov
  real(default), intent(in) :: pt2
  real(default) :: xm2s, xm2sl, pt2l
  logical :: within_boundaries
  within_boundaries = pt2 / sudakov%event_deps%s_hat <= sudakov%xi2_max &
       .and. pt2 >= sudakov%powheg_settings%pt2_min
  if (within_boundaries) then
     xm2s = sudakov%xi2_max * sudakov%event_deps%s_hat
     xm2sl = xm2s / sudakov%process_deps%lambda2_gen
     pt2l = pt2 / sudakov%process_deps%lambda2_gen
     y = pi / b0rad * (log (xm2sl) * &
          log (log (xm2sl) / log (pt2l)) - &
          log (xm2s / pt2))
  else
     y = 0
  end if
end function sudakov_simple_fsr_log_integrated_ubf
```

No further veto needed for this upper bounding function.

⟨*POWHEG matching: sudakov simple fsr: TBP*⟩+≡

```
procedure :: reweight_ubf => sudakov_simple_fsr_reweight_ubf
```

⟨*POWHEG matching: procedures*⟩+≡

```
function sudakov_simple_fsr_reweight_ubf (sudakov, pt2) result (accepted)
  logical :: accepted
  class(sudakov_simple_fsr_t), intent(inout) :: sudakov
  real(default), intent(in) :: pt2
  accepted = .true.
end function sudakov_simple_fsr_reweight_ubf
```

⟨*POWHEG matching: sudakov simple fsr: TBP*⟩+≡

```
procedure :: reweight_xi_max => sudakov_simple_fsr_reweight_xi_max
```

⟨*POWHEG matching: procedures*⟩+≡

```
function sudakov_simple_fsr_reweight_xi_max (sudakov, xi) result (accepted)
   logical :: accepted
   class(sudakov_simple_fsr_t), intent(in) :: sudakov
   real(default), intent(in) :: xi
   accepted = .true.
end function sudakov_simple_fsr_reweight_xi_max
```

This depends on the choice of p_T and is tested in the assertion.

⟨*POWHEG matching: sudakov simple fsr: TBP*⟩+≡

```
procedure :: generate_xi_and_y_and_phi => sudakov_simple_fsr_generate_xi_and_y_and_phi
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine sudakov_simple_fsr_generate_xi_and_y_and_phi (sudakov, r)
  class(sudakov_simple_fsr_t), intent(inout) :: sudakov
  type(radiation_t), intent(inout) :: r
  real(default) :: s
  s = sudakov%event_deps%s_hat
  call sudakov%generate_xi (r)
  r%y = one - (two * r%pt2) / (s * r%xi**2)
  call sudakov%rng%generate (sudakov%random)
  r%phi = sudakov%random * twopi
end subroutine sudakov_simple_fsr_generate_xi_and_y_and_phi
```

Generate $\xi \in [\frac{p_T}{\sqrt{s}}, \xi_{\max}]$ with a density $1/\xi$

⟨*POWHEG matching: sudakov simple fsr: TBP*⟩+≡

```
procedure :: generate_xi => sudakov_simple_fsr_generate_xi
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine sudakov_simple_fsr_generate_xi (sudakov, r)
  class(sudakov_simple_fsr_t), intent(inout) :: sudakov
  type(radiation_t), intent(inout) :: r
  real(default) :: s, xi2_max
  s = sudakov%event_deps%s_hat
  xi2_max = sudakov%xi2_max
  call sudakov%rng%generate (sudakov%random)
  r%xi = exp (((one - sudakov%random) * log (r%pt2 / s) + &
       sudakov%random * log (xi2_max)) / two)
end subroutine sudakov_simple_fsr_generate_xi
```

Dijet production at lepton colliders

In the POWHEG method paper, this is done for $e^+e^- \to q\bar{q}$. There $k_{\max} = q^0/2 = \sqrt{s}/2$. We slightly extend this to

⟨*POWHEG matching: public*⟩+≡

```
public :: sudakov_eeqq_fsr_t
```

⟨*POWHEG matching: types*⟩+≡

```
type, extends (sudakov_t) :: sudakov_eeqq_fsr_t
contains
 ⟨POWHEG matching: sudakov eeqq fsr: TBP⟩
end type sudakov_eeqq_fsr_t
```

This k_T measure is the same as the simple FSR up to $\mathcal{O}(\theta^4)$ when $y = \cos\theta$.

⟨*POWHEG matching: sudakov eeqq fsr: TBP*⟩≡
```
procedure :: kt2 => sudakov_eeqq_fsr_kt2
```

⟨*POWHEG matching: procedures*⟩+≡
```
function sudakov_eeqq_fsr_kt2 (sudakov, xi, y) result (kt2)
  real(default) :: kt2
  class(sudakov_eeqq_fsr_t), intent(in) :: sudakov
  real(default), intent(in) :: xi, y
  kt2 = sudakov%event_deps%s_hat / 2 * xi**2 * (1 - y**2) / 2
  ! TODO: (bcn 2015-07-13) call here phs_fks_generator%real_kinematics%kt2
end function sudakov_eeqq_fsr_kt2
```

Same expression as for `sudakov_simple_fsr_kt2_max`

⟨*POWHEG matching: sudakov eeqq fsr: TBP*⟩+≡
```
procedure :: kt2_max => sudakov_eeqq_fsr_kt2_max
```

⟨*POWHEG matching: procedures*⟩+≡
```
pure function sudakov_eeqq_fsr_kt2_max (sudakov, s_hat) result (kt2_max)
   real(default) :: kt2_max
   class(sudakov_eeqq_fsr_t), intent(in) :: sudakov
   real(default), intent(in) :: s_hat
   kt2_max = sudakov%xi2_max * s_hat
end function sudakov_eeqq_fsr_kt2_max
```

This covers also the singularity at $(\xi, y) \to (1, -1)$ that occurs for a massless recoiling system.

⟨*POWHEG matching: sudakov eeqq fsr: TBP*⟩+≡
```
procedure :: upper_bound_func => sudakov_eeqq_fsr_upper_bound_func
```

⟨*POWHEG matching: procedures*⟩+≡
```
pure function sudakov_eeqq_fsr_upper_bound_func (sudakov, xi, y, alpha_s) result (u)
  real(default) :: u
  class(sudakov_eeqq_fsr_t), intent(in) :: sudakov
  real(default), intent(in) :: xi, y, alpha_s
  u = alpha_s / (xi * (1 - y**2))
end function sudakov_eeqq_fsr_upper_bound_func
```

⟨*POWHEG matching: sudakov eeqq fsr: TBP*⟩+≡
```
procedure :: log_integrated_ubf => sudakov_eeqq_fsr_log_integrated_ubf
```

⟨*POWHEG matching: procedures*⟩+≡
```
pure function sudakov_eeqq_fsr_log_integrated_ubf (sudakov, pt2) result (y)
  real(default) :: y
  class(sudakov_eeqq_fsr_t), intent(in) :: sudakov
  real(default), intent(in) :: pt2
  logical :: within_boundaries
  within_boundaries = pt2 / sudakov%event_deps%s_hat <= sudakov%xi2_max &
       .and. pt2 >= sudakov%powheg_settings%pt2_min
  if (within_boundaries) then
     !xm2s = sudakov%xi2_max * sudakov%event_deps%s_hat
     !xm2sl = xm2s / sudakov%process_deps%lambda2_gen
     !pt2l = pt2 / sudakov%process_deps%lambda2_gen
     !y = pi / b0rad * (log (xm2sl) * &
```

```
            !log (log (xm2sl) / log (pt2l)) - &
            !log (xm2s / pt2))
      else
         y = 0
      end if
    end function sudakov_eeqq_fsr_log_integrated_ubf
```

⟨*POWHEG matching: sudakov eeqq fsr: TBP*⟩+≡

```
    procedure :: reweight_ubf => sudakov_eeqq_fsr_reweight_ubf
```

⟨*POWHEG matching: procedures*⟩+≡

```
    function sudakov_eeqq_fsr_reweight_ubf (sudakov, pt2) result (accepted)
      logical :: accepted
      class(sudakov_eeqq_fsr_t), intent(inout) :: sudakov
      real(default), intent(in) :: pt2
      accepted = .false.
    end function sudakov_eeqq_fsr_reweight_ubf
```

⟨*POWHEG matching: sudakov eeqq fsr: TBP*⟩+≡

```
    procedure :: reweight_xi_max => sudakov_eeqq_fsr_reweight_xi_max
```

⟨*POWHEG matching: procedures*⟩+≡

```
    function sudakov_eeqq_fsr_reweight_xi_max (sudakov, xi) result (accepted)
       logical :: accepted
       class(sudakov_eeqq_fsr_t), intent(in) :: sudakov
       real(default), intent(in) :: xi
       accepted = .true.
    end function sudakov_eeqq_fsr_reweight_xi_max
```

⟨*POWHEG matching: sudakov eeqq fsr: TBP*⟩+≡

```
    procedure :: generate_xi_and_y_and_phi => sudakov_eeqq_fsr_generate_xi_and_y_and_phi
```

⟨*POWHEG matching: procedures*⟩+≡

```
    subroutine sudakov_eeqq_fsr_generate_xi_and_y_and_phi (sudakov, r)
      class(sudakov_eeqq_fsr_t), intent(inout) :: sudakov
      type(radiation_t), intent(inout) :: r
      real(default) :: s
      s = sudakov%event_deps%s_hat
      !r%xi = sudakov%generate_xi (r)
      !r%y = one - (two * r%pt2) / (s * r%xi**2)
      call sudakov%rng%generate (sudakov%random)
      r%phi = sudakov%random * twopi
    end subroutine sudakov_eeqq_fsr_generate_xi_and_y_and_phi
```

Massive FSR

⟨*POWHEG matching: public*⟩+≡

```
    public :: sudakov_massive_fsr_t
```

⟨*POWHEG matching: types*⟩+≡

```
type, extends (sudakov_t) :: sudakov_massive_fsr_t
  real(default) :: z, z1, z2 = 0._default
  real(default) :: xi_1, xi_min, xi_m = 0._default
  real(default) :: xi_max_extended = 1._default
contains
⟨POWHEG matching: sudakov massive fsr: TBP⟩
end type sudakov_massive_fsr_t
```

During the radiation generation, an alternative expression for $\xi_{\max}$,

$$\xi_{\max} = 1 - \frac{(m + m_{\mathrm{rec}})^2}{q^2},$$

is used, which corresponds to an extenden Dalitz region. Phase space points outside of the original Dalitz region will be vetoed afterwards.

⟨*POWHEG matching: sudakov massive fsr: TBP*⟩≡

```
procedure :: compute_xi_max_extended &
   => sudakov_massive_fsr_compute_xi_max_extended
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine sudakov_massive_fsr_compute_xi_max_extended (sudakov, i_phs)
   class(sudakov_massive_fsr_t), intent(inout) :: sudakov
   integer, intent(in) :: i_phs
   real(default) :: m, mrec
   real(default) :: q0
   type(vector4_t) :: p
   q0 = sqrt(sudakov%event_deps%s_hat)
   p = sudakov%event_deps%p_born_lab%get_momentum (1, sudakov%associated_emitter())
   m = p**1
   mrec = sqrt ((q0 - p%p(0))**2 - p%p(1)**2 - p%p(2)**2 - p%p(3)**2)
   sudakov%xi_max_extended = one - (m + mrec)**2 / q0**2
end subroutine sudakov_massive_fsr_compute_xi_max_extended
```

For massive emitters, the radiation variable ξ is constructed as follows. First,

$$\xi_{\min}(k_T^2) = \frac{\sqrt{k_T^2\left(k_T^2 z_2^2 + 8\bar{p}^0 q(1-z_2)\right)} - k_T^2 z_2}{2q^2(1-z_2)} \tag{31.7}$$

is computed. Then ξ_1 is computed according to the same equation with $z_2 \leftrightarrow z_1$. Finally, ξ is generated according to

$$\xi = \frac{1}{q^2}\exp\left[\log\left(\xi_{\min}q^2 - k_T^2\right) + r\log\frac{\xi_m q^2 - k_T^2}{\xi_{\min}q^2 - k_T^2} + k_T^2\right], \tag{31.8}$$

where $\xi_m = \min\left(\xi_{\max}, \xi_1\right)$.

⟨*POWHEG matching: sudakov massive fsr: TBP*⟩+≡

```
procedure :: generate_xi => sudakov_massive_fsr_generate_xi
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine sudakov_massive_fsr_generate_xi (sudakov, r)
  class(sudakov_massive_fsr_t), intent(inout) :: sudakov
  type(radiation_t), intent(inout) :: r
  real(default) :: pt2, q0, q02
```

```
    real(default) :: E_em, xi_max
    real(default) :: xi_1, xi_min, xi_m
    pt2 = r%pt2
    E_em = sudakov%event_deps%p_born_lab%get_energy &
       (1, sudakov%associated_emitter())
    q02 = sudakov%event_deps%s_hat; q0 = sqrt(q02)
    !xi_max = sqrt (sudakov%xi2_max)
    xi_max = sudakov%xi_max_extended
    associate (z1 => sudakov%z1, z2 => sudakov%z2)
       xi_1 = (sqrt(pt2 * (pt2 * z1**2 + 8 * E_em * q0 * (one - z1))) - pt2 * z1) / &
         (two * q02 * (one - z1))
       xi_min = (sqrt(pt2 * (pt2 * z2**2 + 8 * E_em * q0 * (one - z2))) - pt2 * z2) / &
         (two * q02 * (one - z2))
    end associate
    xi_m = min (xi_max, xi_1)
    call sudakov%rng%generate (sudakov%random)
    r%xi = (exp (log(xi_min * q02 - pt2) + sudakov%random * &
          log((xi_m * q02 - pt2) / (xi_min * q02 - pt2))) + pt2) / q02
  end subroutine sudakov_massive_fsr_generate_xi
```

⟨*POWHEG matching: sudakov massive fsr: TBP*⟩+≡

```
  procedure :: generate_xi_and_y_and_phi => sudakov_massive_fsr_generate_xi_and_y_and_phi
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine sudakov_massive_fsr_generate_xi_and_y_and_phi (sudakov, r)
    class(sudakov_massive_fsr_t), intent(inout) :: sudakov
    type(radiation_t), intent(inout) :: r
    real(default) :: q0
    real(default) :: m2, mrec2, k0_rec_max
    real(default) :: E_em
    type(vector4_t) :: p_emitter

    q0 = sqrt (sudakov%event_deps%s_hat)
    p_emitter = sudakov%event_deps%p_born_lab%get_momentum &
       (1, sudakov%associated_emitter())
    associate (p => p_emitter%p)
      mrec2 = (q0 - p(0))**2 - p(1)**2 - p(2)**2 - p(3)**2
      E_em = p(0)
    end associate
    m2 = p_emitter**2
    call compute_dalitz_bounds (q0, m2, mrec2, sudakov%z1, sudakov%z2, k0_rec_max)
    call sudakov%generate_xi (r)

    sudakov%z = (two * r%pt2 * E_em - r%xi**2 * q0**3) / &
       (r%pt2 * r%xi * q0 - r%xi**2 * q0**3)
    sudakov%xi2_max = - (q0**2 * sudakov%z**2 - two * q0 * k0_rec_max * sudakov%z + mrec2) / &
       (q0**2 * sudakov%z * (one - sudakov%z))
    sudakov%xi2_max = sudakov%xi2_max**2
    r%y = two * (sudakov%z2 - sudakov%z) / (sudakov%z2 - sudakov%z1) - one
    call sudakov%rng%generate (sudakov%random)
    r%phi = sudakov%random * twopi
  end subroutine sudakov_massive_fsr_generate_xi_and_y_and_phi
```

Computes the hardness scale:

$$K_T^2 = \frac{\xi^2 q^2 (1-z)}{2\bar{p}^0_{\rm em} - z\xi q} \tag{31.9}$$

⟨*POWHEG matching: sudakov massive fsr: TBP*⟩+≡
```
procedure :: kt2 => sudakov_massive_fsr_kt2
```

⟨*POWHEG matching: procedures*⟩+≡
```
function sudakov_massive_fsr_kt2 (sudakov, xi, y) result (kt2)
  real(default) :: kt2
  class(sudakov_massive_fsr_t), intent(in) :: sudakov
  real(default), intent(in) :: xi, y
  kt2 = sudakov%phs_fks_generator%real_kinematics%kt2 &
     (sudakov%i_phs, sudakov%associated_emitter(), FSR_MASSIVE, xi, y)
end function sudakov_massive_fsr_kt2
```

For massive emitters, the upper bound on the radiated p_T is

$$t_{\rm max} = \frac{\xi^2_{\rm max} q^3 (1-z_2)}{2 * \bar{p}^0 - z_2 \xi_{\rm max} q}$$

⟨*POWHEG matching: sudakov massive fsr: TBP*⟩+≡
```
procedure :: kt2_max => sudakov_massive_fsr_kt2_max
```

⟨*POWHEG matching: procedures*⟩+≡
```
pure function sudakov_massive_fsr_kt2_max (sudakov, s_hat) result (kt2_max)
   real(default) :: kt2_max
   class(sudakov_massive_fsr_t), intent(in) :: sudakov
   real(default), intent(in) :: s_hat
   real(default) :: q, E_em, xi_max, z2
   q = sqrt(s_hat)
   E_em = sudakov%event_deps%p_born_lab%get_energy &
      (1, sudakov%associated_emitter())
   xi_max = sudakov%xi_max_extended
   z2 = sudakov%z2
   kt2_max = (xi_max**2 * q**3 * (one - z2)) / (two * E_em - z2 * xi_max * q)
end function sudakov_massive_fsr_kt2_max
```

The upper bounding function for massive emitters is (disregarding a possible factor of α_s)

$$U(\xi, y) \sim \frac{\sqrt{s}}{\bar{p}_{\rm em}} \frac{1}{\xi(1-z)} \tag{31.10}$$

⟨*POWHEG matching: sudakov massive fsr: TBP*⟩+≡
```
procedure :: upper_bound_func => sudakov_massive_fsr_upper_bound_func
```

⟨*POWHEG matching: procedures*⟩+≡
```
pure function sudakov_massive_fsr_upper_bound_func (sudakov, xi, y, alpha_s) result (u)
  real(default) :: u
  class(sudakov_massive_fsr_t), intent(in) :: sudakov
  real(default), intent(in) :: xi, y, alpha_s
  real(default) :: q, p_em
  q = sqrt (sudakov%event_deps%s_hat)
  p_em = space_part_norm &
```

```
      (sudakov%event_deps%p_born_lab%get_momentum (1, sudakov%associated_emitter()))
    u = alpha_s * q / p_em * one / (xi * (one - sudakov%z))
  end function sudakov_massive_fsr_upper_bound_func
```

The integrated upper-bounding function for massive final-state emitters is given by

$$I(t) = \frac{q}{\bar{p}_{\rm em}} \left[\log\xi \log\left[(1-z_2)\frac{q}{k_T^2}\right] + \frac{1}{2}\log^2\xi + G(-k_T^2, q^2, \xi) - G(2\bar{p}_{\rm em}, -q, \xi) \right]_{\xi_{\rm min}}^{\min(\xi_1(\mathrm{k}_\mathrm{T}^2),\xi_{\rm max}}) \\ + \frac{q}{\bar{p}_{\rm em}} \theta(\xi_{\rm max} - \xi_1(k_T^2)) \log\frac{\xi_{\rm max}}{\xi_1(k_T^2)} \log\frac{1-z_2}{1-z_1},$$

where the function $G(a, b, \xi)$ is given by

$$G(a, b, \xi) = \log(a + b\xi) \log\left(1 - \frac{a + b\xi}{a}\right) + Li_2\left(\frac{a + b\xi}{a}\right), \tag{31.11}$$

for $a < 0$ and by

$$G(a, b, \xi) = \log\left|\frac{b\xi}{a}\right| \log a - Li_2\left(-\frac{b\xi}{a}\right) + \frac{\pi^2}{6}, \tag{31.12}$$

for $a > 0$.

⟨*POWHEG matching: sudakov massive fsr: TBP*⟩+≡

```
  procedure :: log_integrated_ubf => sudakov_massive_fsr_log_integrated_ubf
```

⟨*POWHEG matching: procedures*⟩+≡

```
  pure function sudakov_massive_fsr_log_integrated_ubf (sudakov, pt2) result (y)
    real(default) :: y
    class(sudakov_massive_fsr_t), intent(in) :: sudakov
    real(default), intent(in) :: pt2
    real(default) :: xi, xi_max, xi_1, xi_min
    real(default) :: q0, p_em, E_em
    real(default) :: y1, y2
    q0 = sqrt (sudakov%event_deps%s_hat)
    E_em = sudakov%event_deps%p_born_lab%get_energy &
       (1, sudakov%associated_emitter())
    p_em = space_part_norm &
       (sudakov%event_deps%p_born_lab%get_momentum (1, sudakov%associated_emitter()))
    xi_max = sudakov%xi_max_extended
    associate (z1 => sudakov%z1, z2 => sudakov%z2)
       xi_1 = (sqrt (pt2 * (pt2 * z1**2 + 8 * E_em * q0 * (one - z1))) - pt2 * z1) / &
          (two * q0**2 * (one - z1))
       xi_min = (sqrt (pt2 * (pt2 * z2**2 + 8 * E_em * q0 * (one - z2))) - pt2 * z2) / &
          (two * q0**2 * (one - z2))
       xi = min (xi_1, xi_max)
       y1 = log(xi) * log((one - z2) * q0 / pt2) + log(xi)**2 / two + &
          G_FSR(-pt2,q0**2, xi) - G_FSR(two * E_em, -q0, xi)
       xi = xi_min
       y2 = log(xi) * log((one - z2) * q0 / pt2) + log(xi)**2/two + &
          G_FSR(-pt2, q0**2, xi) - G_FSR(two * E_em, -q0, xi)
       y = y1 - y2
       if (xi_max > xi_1) &
```

```
        y = y + log(xi_max / xi_1) * log((one - z2) / (one - z1))
      y = twopi * q0 / p_em * y
    end associate
  end function sudakov_massive_fsr_log_integrated_ubf
```

No further ubf veto needed for now.

⟨*POWHEG matching: sudakov massive fsr: TBP*⟩+≡

```
  procedure :: reweight_ubf => sudakov_massive_fsr_reweight_ubf
```

⟨*POWHEG matching: procedures*⟩+≡

```
  function sudakov_massive_fsr_reweight_ubf (sudakov, pt2) result (accepted)
    logical :: accepted
    class(sudakov_massive_fsr_t), intent(inout) :: sudakov
    real(default), intent(in) :: pt2
    accepted = .true.
  end function sudakov_massive_fsr_reweight_ubf
```

⟨*POWHEG matching: sudakov massive fsr: TBP*⟩+≡

```
  procedure :: reweight_xi_max => sudakov_massive_fsr_reweight_xi_max
```

⟨*POWHEG matching: procedures*⟩+≡

```
  function sudakov_massive_fsr_reweight_xi_max (sudakov, xi) result (accepted)
    logical :: accepted
    class(sudakov_massive_fsr_t), intent(in) :: sudakov
    real(default), intent(in) :: xi
    accepted = xi < sqrt (sudakov%xi2_max)
  end function sudakov_massive_fsr_reweight_xi_max
```

Auxiliary functions

Implements the function $G(a, b, \xi)$ given in eq. (31.12) and eq. (31.11).

⟨*POWHEG matching: procedures*⟩+≡

```
  elemental function G_FSR (a,b,xi)
    real(default) :: G_FSR
    real(default), intent(in) :: a, b, xi
    if (a > 0) then
       G_FSR = G_FSR_Plus (a,b,xi)
    else if (a < 0) then
       G_FSR = G_FSR_Minus (a,b,xi)
    !!! a == 0 ?
    end if
  end function G_FSR

  elemental function G_FSR_Minus (a,b,xi)
    real(default) :: G_FSR_Minus
    real(default), intent(in) :: a, b, xi
    G_FSR_Minus = log(a+b*xi)*log(one - (a+b*xi)/a) + Li2((a+b*xi)/a)
  end function G_FSR_Minus

  elemental function G_FSR_Plus (a,b,xi)
    real(default) :: G_FSR_Plus
    real(default), intent(in) :: a, b, xi
```

```
    G_FSR_Plus = log(abs(b*xi/a))*log(a) - Li2(-b*xi/a) + pi**2/6
  end function G_FSR_Plus
```

31.4.3 Main POWHEG class

⟨*POWHEG matching: public*⟩+≡

```
  public :: powheg_matching_t
```

⟨*POWHEG matching: types*⟩+≡

```
  type, extends(matching_t) :: powheg_matching_t
     type(grid_t) :: grid
     type(phs_fks_generator_t) :: phs_fks_generator
     type(powheg_settings_t) :: settings
     type(event_deps_t) :: event_deps
     type(process_deps_t) :: process_deps
     type(sudakov_wrapper_t), dimension(:), allocatable :: sudakov
     integer :: n_emissions = 0
     logical :: active = .true.
   contains
   ⟨POWHEG matching: powheg matching: TBP⟩
  end type powheg_matching_t
```

⟨*POWHEG matching: powheg matching: TBP*⟩≡

```
  procedure :: get_method => powheg_matching_get_method
```

⟨*POWHEG matching: procedures*⟩+≡

```
  function powheg_matching_get_method (matching) result (method)
     type(string_t) :: method
     class(powheg_matching_t), intent(in) :: matching
     method = matching_method (MATCH_POWHEG)
  end function powheg_matching_get_method
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
  procedure :: before_shower => powheg_matching_before_shower
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine powheg_matching_before_shower &
         (matching, particle_set, vetoed)
    class(powheg_matching_t), intent(inout) :: matching
    type(particle_set_t), intent(inout) :: particle_set
    logical, intent(out) :: vetoed
    if (signal_is_pending ())  return
    if (.not. matching%active)  return
    call matching%update (particle_set)
    if (matching%settings%test_sudakov) then
       call matching%test_sudakov ()
       stop
    end if
    if (.not. matching%settings%disable_sudakov) then
       call matching%generate_emission (particle_set = particle_set)
    end if
    vetoed = .false.
  end subroutine powheg_matching_before_shower
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: first_event => powheg_matching_first_event
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_first_event (matching)
  class(powheg_matching_t),  intent(inout), target :: matching
  associate (instance => matching%process_instance)
     matching%process_deps%cm_frame = instance%is_cm_frame (1)
  end associate
  call matching%setup_grids ()
end subroutine powheg_matching_first_event
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: after_shower => powheg_matching_after_shower
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_after_shower (matching, particle_set, vetoed)
  class(powheg_matching_t), intent(inout) :: matching
  type(particle_set_t), intent(inout) :: particle_set
  logical, intent(out) :: vetoed
  vetoed = .false.
end subroutine powheg_matching_after_shower
```

Output

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: display_grid_startup_message => &
                     powheg_display_grid_startup_message
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_display_grid_startup_message (powheg)
  class(powheg_matching_t), intent(in) :: powheg
  real(default) :: points_per_cell
  write (msg_buffer, "(A,A,A)") "POWHEG: Generating grid for process '", &
                            char (powheg%process_name), "'"
  call msg_message ()
  associate (settings => powheg%settings)
     write (msg_buffer, "(A,I10)") "Number of xi-points: ", &
                                   settings%size_grid_xi
     call msg_message ()
     write (msg_buffer, "(A,I10)") "Number of y-points: ", &
                                   settings%size_grid_y
     call msg_message ()
     write (msg_buffer, "(A,I10,A)") "Using ", settings%n_init , &
                                    " sampling points"
     call msg_message ()
     points_per_cell =  settings%n_init*one / &
                       (settings%size_grid_xi * settings%size_grid_y)
     write (msg_buffer, "(A,F10.2,A)") "Average: ", points_per_cell, &
                                     " points per cell"
     call msg_message ()
     call msg_message ("Progress:")
  end associate
end subroutine powheg_display_grid_startup_message
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: write => powheg_write
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_write (matching, unit)
  class(powheg_matching_t), intent(in) :: matching
  integer, intent(in), optional :: unit
  integer :: u, alr
  u = given_output_unit (unit);  if (u < 0)  return
  call write_separator (u, 2)
  write (u, "(1X,A)") "POWHEG Emission Generator"
  write (u, "(1X,A)") "Process name: " // char (matching%process_name)
  if (allocated (matching%rng)) then
     call matching%rng%write (u)
  else
     write (u, "(1X,A)") "RNG not allocated"
  end if
  if (associated (matching%qcd)) then
     call matching%qcd%write (u)
  else
     write (u, "(1X,A)") "QCD not associated"
  end if
  call matching%settings%write (u)
  call matching%event_deps%write (u)
  call matching%process_deps%write (u)
  do alr = 1, size (matching%sudakov)
     call write_separator (u)
     write (u, "(1X,A,I12,A)") "sudakov (alr = ", alr, ")"
     call matching%sudakov(alr)%s%write (u)
  end do
  call write_separator (u, 2)
end subroutine powheg_write
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: final => powheg_matching_final
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_final (matching)
  class(powheg_matching_t), intent(in) :: matching
  integer :: u, alr
  type(string_t) :: filename
  u = free_unit ()
  filename = matching%process_name // "_veto.log"
  open (file=char(filename), unit=u, action='write')
  write (u, '(A)') "Summary of POWHEG veto procedure"
  do alr = 1, matching%process_deps%n_alr
     write(u,'(A,I0)') 'alr: ', alr
     call matching%sudakov(alr)%s%veto_counter%write (u)
     call write_separator (u)
  end do
  write (u,'(A,I0)') "Total number of events which radiate a gluon: ", &
                     matching%n_emissions
  close (u)
end subroutine powheg_matching_final
```

Initialization and Finalization

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: setup_grids => powheg_matching_setup_grids
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_setup_grids (matching)
  class(powheg_matching_t), intent(inout), target :: matching
  call matching%prepare_for_events ()
  if (matching%requires_new_grids ()) then
     call matching%fill_grids ()
     call matching%save_grids ()
  else
     call matching%load_grids ()
  end if
  call matching%grid%compute_and_write_mean_and_max ()
  call matching%import_norms_from_grid ()
end subroutine powheg_matching_setup_grids
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: setup_sudakovs => powheg_matching_setup_sudakovs
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_setup_sudakovs (powheg)
  class(powheg_matching_t), intent(inout), target :: powheg
  integer :: alr
  logical :: is_fsr, is_massive
  integer :: ubf_type
  allocate (powheg%sudakov (powheg%process_deps%n_alr))
  is_fsr = .true.
  do alr = 1, powheg%process_deps%n_alr
     if (is_fsr) then
        ubf_type = powheg%settings%upper_bound_func
        select type (pcm => powheg%process_instance%pcm)
        class is (pcm_instance_nlo_t)
           select type (config => pcm%config)
           type is (pcm_nlo_t)
              is_massive = powheg%phs_fks_generator%is_massive &
                 (config%region_data%get_emitter (alr))
           end select
        end select
        if (is_massive) ubf_type = UBF_MASSIVE
        select case (ubf_type)
        case (UBF_SIMPLE)
           allocate (sudakov_simple_fsr_t :: powheg%sudakov(alr)%s)
        case (UBF_EEQQ)
           allocate (sudakov_eeqq_fsr_t :: powheg%sudakov(alr)%s)
        case (UBF_MASSIVE)
           allocate (sudakov_massive_fsr_t :: powheg%sudakov(alr)%s)
        case default
           call msg_fatal ("powheg_setup_sudakovs: Please choose upper bounding function!")
        end select
     else
        call msg_fatal ("powheg_setup_sudakovs: ISR not implemented yet!")
     end if
```

```
      call powheg%sudakov(alr)%s%init (powheg%process_deps, &
           powheg%event_deps, powheg%settings, &
           powheg%phs_fks_generator, powheg%qcd, powheg%rng)

      call powheg%sudakov(alr)%s%set_i_phs (alr)
    end do
  end subroutine powheg_matching_setup_sudakovs
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
  procedure :: init => powheg_matching_init
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine powheg_matching_init (matching, var_list, process_name)
    class(powheg_matching_t), intent(out) :: matching
  ⟨default matching init⟩
  end subroutine powheg_matching_init
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
  generic :: update => update_momenta, &
                       update_particle_set
  procedure :: update_momenta => powheg_matching_update_momenta
  procedure :: update_particle_set => powheg_matching_update_particle_set
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine powheg_matching_update_momenta (powheg, p_born)
    class(powheg_matching_t), intent(inout) :: powheg
    type(vector4_t), dimension(:), intent(in) :: p_born
    type(lorentz_transformation_t) :: lt_lab_to_cms
    real(default) :: sqme_born

    sqme_born = powheg%process_instance%get_sqme (powheg%process_deps%i_born)
    if (.not. powheg%process_deps%cm_frame) then
       lt_lab_to_cms = powheg%process_instance%get_boost_to_cms (1)
       call powheg%update_event_deps (sqme_born, p_born, lt_lab_to_cms)
    else
       call powheg%update_event_deps (sqme_born, p_born)
    end if
  end subroutine powheg_matching_update_momenta

  subroutine powheg_matching_update_particle_set (powheg, particle_set)
    class(powheg_matching_t), intent(inout) :: powheg
    type(particle_set_t), intent(in) :: particle_set
    integer, dimension(:), allocatable :: indices
    logical, dimension(:), allocatable :: in_out_mask
    integer :: i
    allocate (in_out_mask (particle_set%get_n_tot()))
    do i = 1, particle_set%get_n_tot()
       in_out_mask(i) = particle_set%prt(i)%get_status () == PRT_INCOMING &
            .or. particle_set%prt(i)%get_status () == PRT_OUTGOING
    end do
    allocate (indices (size (particle_set%get_indices (in_out_mask))))
    indices = particle_set%get_indices (in_out_mask)
    call powheg%update_momenta (particle_set%get_momenta (indices))
```

```
end subroutine powheg_matching_update_particle_set
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: update_event_deps => powheg_matching_update_event_deps
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_update_event_deps (powheg, sqme_born, p_born, lt_lab_to_cms)
  class(powheg_matching_t), intent(inout) :: powheg
  real(default), intent(in) :: sqme_born
  type(vector4_t), dimension(:), intent(in) :: p_born
  type(lorentz_transformation_t), intent(in), optional :: lt_lab_to_cms
  call powheg%event_deps%update (sqme_born, p_born, lt_lab_to_cms)
end subroutine powheg_matching_update_event_deps
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: boost_preal_to_lab_frame => powheg_matching_boost_preal_to_lab_frame
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_boost_preal_to_lab_frame (powheg, i_phs)
  class(powheg_matching_t), intent(inout) :: powheg
  type(lorentz_transformation_t) :: lt_cms_to_lab
  integer, intent(in) :: i_phs
  associate (event_deps => powheg%event_deps)
     if (powheg%process_deps%cm_frame) then
        event_deps%p_real_lab%phs_point(i_phs) = event_deps%p_real_cms%phs_point(i_phs)
     else
        lt_cms_to_lab = powheg%process_instance%get_boost_to_lab (1)
        event_deps%p_real_lab%phs_point(i_phs) = &
           lt_cms_to_lab * event_deps%p_real_cms%phs_point(i_phs)
     end if
  end associate
end subroutine powheg_matching_boost_preal_to_lab_frame
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: reweight_matrix_elements => powheg_matching_reweight_matrix_elements
```

⟨*POWHEG matching: procedures*⟩+≡

```
function powheg_matching_reweight_matrix_elements (powheg, r) result (accepted)
  logical :: accepted
  class(powheg_matching_t), intent(inout) :: powheg
  type(radiation_t), intent(in) :: r
  integer :: emitter, i_phs
  real(default) :: sqme_real_x_jacobian, sqme_born
  real(default) :: norm, ubf, ubound, random, weight
  real(default) :: alpha_s
  call msg_debug (D_MATCHING, "reweight_matrix_elements")
  select type (pcm => powheg%process_instance%pcm)
  class is (pcm_instance_nlo_t)
     call powheg%rng%generate (random)
     i_phs = powheg%process_deps%alr_to_i_phs (r%alr)
     select type (config => pcm%config)
     type is (pcm_nlo_t)
        emitter = config%region_data%get_emitter (r%alr)
     end select
```

```
      call powheg%phs_fks_generator%generate_fsr_from_xi_and_y (r%xi, r%y, &
         r%phi, emitter, i_phs, powheg%event_deps%p_born_cms%get_momenta(1), &
         powheg%event_deps%p_real_cms%phs_point(i_phs))
      call powheg%boost_preal_to_lab_frame (i_phs)
      call powheg%copy_momenta ()
      norm = powheg%norm_from_xi_and_y (r)
      associate (s => powheg%sudakov(r%alr)%s)
        alpha_s = s%alpha_s (s%kt2 (r%xi, r%y), use_correct=.true.)
        ubf = s%upper_bound_func (r%xi, r%y, alpha_s)
        sqme_real_x_jacobian = powheg%compute_sqme_real (r%alr, alpha_s)
        sqme_born = powheg%event_deps%sqme_born
        ubound = sqme_born * ubf * norm
        weight = sqme_real_x_jacobian / ubound
        if (weight > 1) call s%veto_counter%record_fail()
        if (debug_active (D_MATCHING)) then
           if (weight < 0) call msg_warning ("R/B < 0!")
        end if
        accepted = random < weight
      end associate
      if (debug_active (D_MATCHING)) then
         print *, '  r%alr =    ',   r%alr
         print *, '  r%xi =    ', r%xi
         print *, '  r%y =    ', r%y
         print *, '  emitter =    ', emitter
         print *, '  random =    ', random
         print *, '  sqme_real_x_jacobian =    ', sqme_real_x_jacobian
         print *, '  sqme_born =    ', sqme_born
         print *, '  ubf =    ', ubf
         print *, '  norm =    ',   norm
         print *, '  ubound =    ', ubound
         print *, '  matrix element  accepted =    ', accepted
      end if
    end select
  end function powheg_matching_reweight_matrix_elements
```

Generation algorithm and grid initialization

`compute_sqme_real` is the projected real matrix element $R_{\alpha_r} = S_{\alpha_r} R$ whereby the current α_r is implied by the `emitter`. Furthermore, it is multiplied by the Jacobian.

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
  procedure :: compute_sqme_real => powheg_matching_compute_sqme_real
```

⟨*POWHEG matching: procedures*⟩+≡

```
  function powheg_matching_compute_sqme_real (powheg, alr, alpha_s) result (sqme)
    real(default) :: sqme
    class(powheg_matching_t), intent(inout) :: powheg
    integer, intent(in) :: alr
    real(default), intent(in) :: alpha_s
    integer :: i_phs, i_term
    select type (pcm => powheg%process_instance%pcm)
    class is (pcm_instance_nlo_t)
       i_phs = powheg%process_deps%alr_to_i_phs (alr)
```

```
      i_term = powheg%process_deps%i_real(i_phs)
      associate (instance => powheg%process_instance)
         call instance%compute_sqme_rad (i_term, i_phs, .false., alpha_s)
         sqme = instance%get_sqme (i_term)
      end associate
   end select
end function powheg_matching_compute_sqme_real
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: set_scale => powheg_matching_set_scale
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_set_scale (powheg, pT2)
  class(powheg_matching_t), intent(inout) :: powheg
  real(default), intent(in) :: pT2
  call powheg%process_instance%set_fac_scale (sqrt(pT2))
end subroutine powheg_matching_set_scale
```

For each underlying Born f_b, there is a number of radiation regions. A radiation region rr may correspond multiple α_rs. The phase space only depends upon the radiation region kinematics rr and not on the specific α_r. α_r can be picked in the set $\{\alpha_r | f_b, \mathrm{rr}\}$ proportional to their R_{α_r}. For now, we simplify things though and just work with the α_r.

The following is valid for one underlying Born.

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: fill_grids => powheg_matching_fill_grids
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_fill_grids (powheg)
  class(powheg_matching_t), intent(inout) :: powheg
  real(default), dimension(3) :: radiation_variables
  real(default) :: f_alr, xi, y, norm, real_me, ubf
  integer :: alr
  integer :: n, n_points
  real(default) :: alpha_s
  call msg_debug (D_MATCHING, "powheg_fill_grids")
  call powheg%display_grid_startup_message()
  n_points = powheg%settings%n_init
  call msg_debug (D_MATCHING, "n_points", n_points)
  UNTIL_ACCEPTED: do
     EVALUATE_GRID_POINTS: do n = 1, n_points
        if (signal_is_pending ())  return
        call powheg%prepare_momenta_for_fill_grids (radiation_variables)
        do alr = 1, powheg%process_deps%n_alr
           call powheg%generate_xi_and_y_for_grids &
              (radiation_variables, alr, xi, y)
           associate (s => powheg%sudakov(alr)%s)
              alpha_s = s%alpha_s (s%kt2(xi, y), use_correct=.true.)
              ubf = s%upper_bound_func (xi, y, alpha_s)
           end associate
           real_me = powheg%compute_sqme_real (alr, alpha_s)
           norm = real_me / (powheg%event_deps%sqme_born * ubf)
           f_alr = (one * alr) / powheg%process_deps%n_alr - tiny_07
```

```
            call powheg%grid%update_maxima &
               ([radiation_variables(I_XI:I_Y), f_alr], norm)
            call msg_show_progress (n, n_points)
            if (debug2_active (D_MATCHING))  call show_vars ()
         end do
      end do EVALUATE_GRID_POINTS
      if (powheg%grid%is_non_zero_everywhere () .or. &
           n_points <= 0) then
         return
      else
         n_points = powheg%settings%n_init / 5
         write (msg_buffer, '(A,I12,A)') 'POWHEG: Number of points for grid ' // &
              'initialization was not enough. Run continues with ', &
              n_points, ' additional points to fill empty segments.'
         call msg_warning ()
      end if
   end do UNTIL_ACCEPTED

contains

   subroutine show_vars ()
     if (norm > 1E5_default) then
        call msg_debug2 (D_MATCHING, "alr", alr)
        call msg_debug2 (D_MATCHING, "f_alr", f_alr)
        call msg_debug2 (D_MATCHING, "radiation_variables(1)", &
             radiation_variables(1))
        call msg_debug2 (D_MATCHING, "radiation_variables(2)", &
             radiation_variables(2))
        call msg_debug2 (D_MATCHING, "radiation_variables(3)", &
             radiation_variables(3))
        call msg_debug2 (D_MATCHING, "xi", xi)
        call msg_debug2 (D_MATCHING, "y", y)
        call msg_debug2 (D_MATCHING, "powheg%sudakov(alr)%s%kt2(xi,y)", &
             powheg%sudakov(alr)%s%kt2(xi,y))
        call msg_debug2 (D_MATCHING, "powheg%event_deps%sqme_born", &
             powheg%event_deps%sqme_born)
        call msg_debug2 (D_MATCHING, "alpha_s", alpha_s)
        call msg_debug2 (D_MATCHING, "real_me", real_me)
        call msg_debug2 (D_MATCHING, "ubf", ubf)
        call msg_debug2 (D_MATCHING, "norm", norm)
        call msg_debug2 (D_MATCHING, "")
     end if
   end subroutine show_vars

end subroutine powheg_matching_fill_grids
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: generate_xi_and_y_for_grids => powheg_matching_generate_xi_and_y_for_grids
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_generate_xi_and_y_for_grids (powheg, &
   radiation_randoms, alr, xi, y)
  class(powheg_matching_t), intent(inout) :: powheg
  integer, intent(in) :: alr
```

```
    real(default), dimension(:), intent(in) :: radiation_randoms
    real(default), intent(out) :: xi, y
    integer :: emitter, i_phs
    select type (pcm => powheg%process_instance%pcm)
    class is (pcm_instance_nlo_t)
       associate (fks => powheg%phs_fks_generator)
          i_phs = powheg%process_deps%alr_to_i_phs (alr)
          emitter = powheg%process_deps%phs_identifiers(i_phs)%emitter
          associate (generator => powheg%phs_fks_generator)
             call generator%prepare_generation (radiation_randoms, &
                i_phs, emitter, powheg%event_deps%p_born_cms%phs_point(1)%p, &
                powheg%process_deps%phs_identifiers)
             call generator%generate_fsr (emitter, i_phs, &
                powheg%event_deps%p_born_cms%phs_point(1)%p, &
                powheg%event_deps%p_real_cms%phs_point(i_phs))
             call generator%get_radiation_variables (i_phs, xi, y)
          end associate
          call powheg%boost_preal_to_lab_frame (i_phs)
          call powheg%copy_momenta ()
       end associate
     end select
  end subroutine powheg_matching_generate_xi_and_y_for_grids
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
  procedure :: prepare_momenta_for_fill_grids => powheg_matching_prepare_momenta_for_fill_grids
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine powheg_matching_prepare_momenta_for_fill_grids (powheg, &
     radiation_randoms)
    real(default), dimension(3), intent(out) :: radiation_randoms
    class(powheg_matching_t), intent(inout) :: powheg
    integer :: alr, i_phs, emitter
    select type (pcm => powheg%process_instance%pcm)
    class is (pcm_instance_nlo_t)
       associate ( &
            fks => powheg%phs_fks_generator, &
            process => powheg%process_instance%process)
          do
             call powheg%process_instance%generate_weighted_event (1)
             call powheg%update (pcm%get_momenta (i_phs = 1, &
                born_phsp = .true., cms = .false.))
             call powheg%rng%generate (radiation_randoms)
             do alr = 1, powheg%process_deps%n_alr
                i_phs = powheg%process_deps%alr_to_i_phs (alr)
                emitter = powheg%sudakov(alr)%s%associated_emitter ()
                call fks%prepare_generation (radiation_randoms, i_phs, &
                   emitter, powheg%event_deps%p_born_lab%get_momenta(1), &
                   powheg%process_deps%phs_identifiers)
                call powheg%update_sudakovs (alr, i_phs, emitter, fks%real_kinematics%y(i_phs))
             end do
             if (powheg%above_pt2_min ()) exit
          end do
       end associate
    end select
```

```
end subroutine powheg_matching_prepare_momenta_for_fill_grids
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: above_pt2_min => powheg_matching_above_pt2_min
```

⟨*POWHEG matching: procedures*⟩+≡

```
function powheg_matching_above_pt2_min (powheg) result (above)
  logical :: above
  class(powheg_matching_t), intent(in) :: powheg
  integer :: alr, i_phs
  real(default) :: xi, y
  above = .true.
  select type (pcm => powheg%process_instance%pcm)
  class is (pcm_instance_nlo_t)
     associate (fks => powheg%phs_fks_generator)
       do alr = 1, powheg%process_deps%n_alr
          i_phs = powheg%process_deps%alr_to_i_phs(alr)
          call fks%get_radiation_variables (i_phs, xi, y)
          above = powheg%sudakov(alr)%s%kt2 (xi, y) >= powheg%settings%pt2_min
          if (.not. above) exit
       end do
     end associate
  end select
end function powheg_matching_above_pt2_min
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: update_sudakovs => powheg_matching_update_sudakovs
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_update_sudakovs (powheg, alr, i_phs, emitter, y)
  class(powheg_matching_t), intent(inout) :: powheg
  integer, intent(in) :: alr, i_phs, emitter
  real(default), intent(in) :: y
  real(default) :: q0, m2, mrec2, k0_rec_max
  type(vector4_t) :: p_emitter
  select type (s => powheg%sudakov(alr)%s)
  type is (sudakov_massive_fsr_t)
     q0 = sqrt (s%event_deps%s_hat)
     p_emitter = s%event_deps%p_born_lab%get_momentum (1, emitter)
     associate (p => p_emitter%p)
        mrec2 = (q0 - p(0))**2 - p(1)**2 - p(2)**2 - p(3)**2
     end associate
     m2 = p_emitter**2
     call compute_dalitz_bounds (q0, m2, mrec2, s%z1, s%z2, k0_rec_max)
     s%z = s%z2 - (s%z2 - s%z1) * (one + y) / two
  end select
end subroutine powheg_matching_update_sudakovs
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: import_norms_from_grid => powheg_matching_import_norms_from_grid
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_import_norms_from_grid (powheg)
  class(powheg_matching_t), intent(inout) :: powheg
```

```
    integer :: alr
    real(default) :: norm_max
    do alr = 1, powheg%process_deps%n_alr
       norm_max = powheg%grid%get_maximum_in_3d (alr)
       call powheg%sudakov(alr)%s%set_normalization (norm_max)
    end do
  end subroutine powheg_matching_import_norms_from_grid
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
  procedure :: save_grids => powheg_matching_save_grids
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine powheg_matching_save_grids (powheg)
    class(powheg_matching_t), intent(inout) :: powheg
    type(string_t) :: filename, n_points
    n_points = str (powheg%settings%n_init)
    filename = powheg%process_name // "_" // n_points // "_powheg_grids.dat"
    call powheg%grid%save_to_file (char (filename))
  end subroutine powheg_matching_save_grids
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
  procedure :: load_grids => powheg_matching_load_grids
```

⟨*POWHEG matching: procedures*⟩+≡

```
  subroutine powheg_matching_load_grids (powheg)
    class(powheg_matching_t), intent(inout) :: powheg
    type(string_t) :: filename, n_points
    n_points = str (powheg%settings%n_init)
    filename = powheg%process_name // "_" // n_points // "_powheg_grids.dat"
    call powheg%grid%load_from_file (char (filename))
    write (msg_buffer, "(A,A,A)") "POWHEG: using grids from file '", &
                              char (filename), "'"
    call msg_message ()
  end subroutine powheg_matching_load_grids
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
  procedure :: requires_new_grids => powheg_matching_requires_new_grids
```

⟨*POWHEG matching: procedures*⟩+≡

```
  function powheg_matching_requires_new_grids (powheg) result (requires)
    logical :: requires
    class(powheg_matching_t), intent(in) :: powheg
    type(string_t) :: filename, n_points
    n_points = str (powheg%settings%n_init)
    filename = powheg%process_name // "_" // n_points // "_powheg_grids.dat"
    requires = .not. os_file_exist (filename) .or. powheg%settings%rebuild_grids
  end function powheg_matching_requires_new_grids
```

By keeping the radiation with the largest `pt2`, we are effectively implementing the highest bid procedure. This means that we generate values (f_B)

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
  procedure :: generate_emission => powheg_matching_generate_emission
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_generate_emission (powheg, particle_set, pt2_generated)
  class(powheg_matching_t), intent(inout) :: powheg
  type(particle_set_t), intent(inout), optional :: particle_set
  real(default), intent(out), optional :: pt2_generated
  type(radiation_t) :: r, r_max
  real(default) :: xi2_max
  integer :: alr, emitter
  logical :: accepted
  type(vector4_t), dimension(:), allocatable :: p_real_max
  if (signal_is_pending ())  return
  r_max%pt2 = zero
  r_max%alr = 0
  call msg_debug (D_MATCHING, "powheg_matching_generate_emission")
  select type (pcm => powheg%process_instance%pcm)
  class is (pcm_instance_nlo_t)
     select type (config => pcm%config)
     type is (pcm_nlo_t)
        allocate (p_real_max (config%region_data%n_legs_real))
     end select
     do alr = 1, powheg%process_deps%n_alr
        if (signal_is_pending ())  return
        associate (sudakov => powheg%sudakov(alr)%s)
          xi2_max = pcm%get_xi_max (alr)**2
          call sudakov%update (xi2_max)
          select type (sudakov)
          type is (sudakov_massive_fsr_t)
             call sudakov%compute_xi_max_extended (sudakov%i_phs)
          end select
          r%alr = alr
          r%pt2 = sudakov%kt2_max (powheg%event_deps%s_hat)
          sudakov%last_log = 0
          call msg_debug (D_MATCHING, "Starting evolution at r%pt2", r%pt2)
          PT_EVOLUTION: do
             if (signal_is_pending ())  return
             call sudakov%generate_emission (r)
             if (signal_is_pending ())  return
             if (r%valid) then
                accepted = powheg%reweight_norm (r)
                call sudakov%veto_counter%record_norm (.not. accepted)
                if (.not. accepted) cycle PT_EVOLUTION
                accepted = powheg%reweight_matrix_elements (r)
                call sudakov%veto_counter%record_sqme (.not. accepted)
                if (.not. accepted) cycle PT_EVOLUTION
             end if
             exit
          end do PT_EVOLUTION
          if (r%pt2 > r_max%pt2 .and. r%valid) then
             r_max = r
             p_real_max = powheg%event_deps%p_real_lab%get_momenta (sudakov%i_phs)
          end if
        end associate
     end do
     if (r_max%pt2 > powheg%settings%pt2_min) then
```

```
         powheg%n_emissions = powheg%n_emissions + 1
         call powheg%set_scale (r_max%pt2)
         if (present (particle_set)) then
            select type (config => pcm%config)
            type is (pcm_nlo_t)
               emitter = config%region_data%get_emitter (r_max%alr)
               call powheg%build_particle_set (particle_set, &
                  powheg%event_deps%p_born_lab%get_momenta (1), &
                  p_real_max, emitter)
            end select
         end if
         if (present (pt2_generated)) pt2_generated = r_max%pt2
      else
         call powheg%set_scale (powheg%settings%pt2_min)
         if (present (pt2_generated)) pt2_generated = powheg%settings%pt2_min
      end if
   end select
end subroutine powheg_matching_generate_emission
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: build_particle_set => powheg_matching_build_particle_set
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_build_particle_set &
     (powheg, particle_set, p_born, p_real, emitter)
  class(powheg_matching_t), intent(inout) :: powheg
  type(particle_set_t), intent(inout) :: particle_set
  type(vector4_t), dimension(:), intent(in) :: p_born, p_real
  integer, intent(in) :: emitter
  integer, dimension(:), allocatable :: flv_radiated
  real(default) :: r_col
  select type (pcm => powheg%process_instance%pcm)
  class is (pcm_instance_nlo_t)
     select type (config => pcm%config)
     type is (pcm_nlo_t)
        allocate (flv_radiated (size (config%region_data%get_flv_states_real (1))))
        flv_radiated = config%region_data%get_flv_states_real (1)
     end select
     call powheg%rng%generate (r_col)
     call particle_set%build_radiation (p_real, emitter, flv_radiated, &
          powheg%process_instance%process%get_model_ptr (), r_col)
  end select
end subroutine powheg_matching_build_particle_set
```

Only massless for now

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: reweight_norm => powheg_matching_reweight_norm
```

⟨*POWHEG matching: procedures*⟩+≡

```
function powheg_matching_reweight_norm (powheg, r) result (accepted)
  logical :: accepted
  class(powheg_matching_t), intent(inout) :: powheg
  type(radiation_t), intent(in) :: r
  real(default) :: random, norm_max, norm_true
```

```
    call msg_debug2 (D_MATCHING, "reweight_norm")
    call powheg%rng%generate (random)
    associate (s => powheg%sudakov(r%alr)%s)
       norm_true = powheg%norm_from_xi_and_y (r)
       norm_max = s%norm_max
    end associate
    accepted = random < norm_true / norm_max
    if (debug2_active (D_MATCHING)) then
       print *, '  r%alr =    ', r%alr
       print *, '  random =    ', random
       print *, '  norm_true =    ', norm_true
       print *, '  norm_max =    ', norm_max
       print *, '  norm accepted =    ', accepted
    end if
    if (debug_active (D_MATCHING)) then
       if (.not. (zero < r%xi .and. &
                  r%xi < sqrt(powheg%sudakov(r%alr)%s%xi2_max))) then
          call msg_bug ("powheg_matching_reweight_norm: xi is out of bounds")
       end if
       if (norm_true > norm_max) then
          call msg_bug ("powheg_matching_reweight_norm: norm shouldnt be larger than norm_max")
       end if
    end if
  end function powheg_matching_reweight_norm
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
  procedure :: norm_from_xi_and_y => powheg_matching_norm_from_xi_and_y
```

⟨*POWHEG matching: procedures*⟩+≡

```
  function powheg_matching_norm_from_xi_and_y (powheg, r) result (norm_true)
    real(default) :: norm_true
    class(powheg_matching_t), intent(inout) :: powheg
    type(radiation_t), intent(in) :: r
    real(default) :: f_alr
    real(default), dimension(2) :: rands
    real(default) :: beta
    f_alr = (one * r%alr) / powheg%process_deps%n_alr - tiny_07
    rands(I_XI) = r%xi / sqrt (powheg%sudakov(r%alr)%s%xi2_max)
    select type (s => powheg%sudakov(r%alr)%s)
    type is (sudakov_simple_fsr_t)
       rands(I_Y) = (one - r%y) / two
    type is (sudakov_massive_fsr_t)
       beta = beta_emitter (sqrt (powheg%event_deps%s_hat), &
          powheg%event_deps%p_born_lab%get_momentum (1, s%associated_emitter()))
       rands(I_Y) = - log((one - r%y * beta) / (one + beta)) / &
          log((one + beta) / (one - beta))
    end select
    norm_true = powheg%grid%get_value ([rands, f_alr])
  end function powheg_matching_norm_from_xi_and_y
```

31.4.4 α_s and its reweighting

The main point to ensure here is that the simple fixed-flavor-1-loop expression $\alpha_s^{\rm rad}$ is larger than the more accurate α_s such that we can use a reweighting veto and use $\alpha_s^{\rm rad}$ for the generation of the emission. This can be done by setting

$$\alpha_s^{\rm rad}(\mu_0) = \alpha_s(\mu_0) \tag{31.13}$$

whereby μ_0^2 is the `scale_to_relate2` that is taken to be $p_{T,\rm min}^2$.

⟨*POWHEG matching: powheg matching: TBP*⟩+≡
```
procedure :: prepare_for_events => powheg_matching_prepare_for_events
```

⟨*POWHEG matching: procedures*⟩+≡
```
subroutine powheg_matching_prepare_for_events (matching)
  class(powheg_matching_t), intent(inout), target :: matching
  call msg_debug (D_MATCHING, "powheg_matching_prepare_for_events")
  call matching%setup_nlo_environment ()
  call matching%grid%init ([matching%settings%size_grid_xi, &
     matching%settings%size_grid_y, matching%process_deps%n_alr])
  call matching%compute_lambda2_gen ()
  call matching%setup_sudakovs ()
end subroutine powheg_matching_prepare_for_events
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡
```
procedure :: compute_lambda2_gen => powheg_matching_compute_lambda2_gen
```

⟨*POWHEG matching: procedures*⟩+≡
```
subroutine powheg_matching_compute_lambda2_gen (matching)
  class(powheg_matching_t), intent(inout) :: matching
  real(default) :: scale_to_relate2, alpha_s
  scale_to_relate2 = matching%settings%pt2_min
  alpha_s = get_alpha (matching%qcd, scale_to_relate2)
  matching%process_deps%lambda2_gen = exp (- one / (b0rad * alpha_s)) * &
       scale_to_relate2
end subroutine powheg_matching_compute_lambda2_gen
```

⟨*POWHEG matching: powheg matching: TBP*⟩+≡
```
procedure :: setup_nlo_environment => powheg_matching_setup_nlo_environment
```

⟨*POWHEG matching: procedures*⟩+≡
```
subroutine powheg_matching_setup_nlo_environment (matching)
  class(powheg_matching_t), intent(inout) :: matching
  integer :: n_tot_born, n_tot_real
  integer :: i, i_real, i_real_fin, i_term
  integer :: n_phs, nlo_type
  call msg_debug (D_MATCHING, "powheg_matching_setup_nlo_environment")
  select type (pcm => matching%process_instance%pcm)
  class is (pcm_instance_nlo_t)
     matching%process_deps%sqrts = matching%process_instance%get_sqrts ()
     select type (config => pcm%config)
     type is (pcm_nlo_t)
        matching%process_deps%n_alr = config%region_data%n_regions
        n_tot_born = config%region_data%n_legs_born
        n_tot_real = config%region_data%n_legs_real
        call config%setup_phs_generator (pcm, &
```

```
            matching%phs_fks_generator, matching%process_deps%sqrts, &
            singular_jacobian = matching%settings%singular_jacobian)
      end select
      associate (process => matching%process_instance%process)
          i_real = process%get_associated_real_component (1)
          i_real_fin = process%get_associated_real_fin (1)
      end associate
      associate (process_deps => matching%process_deps)
         select type (phs => matching%process_instance%term(i_real)%k_term%phs)
         type is (phs_fks_t)
            n_phs = size (phs%phs_identifiers)
            allocate (process_deps%phs_identifiers (n_phs))
            process_deps%phs_identifiers = phs%phs_identifiers
         end select
         allocate (matching%process_deps%alr_to_i_phs &
            (size (pcm%real_kinematics%alr_to_i_phs)))
         process_deps%alr_to_i_phs = pcm%real_kinematics%alr_to_i_phs
         allocate (process_deps%i_real (n_phs))
         i = 1
         do i_term = 1, size (matching%process_instance%term)
            nlo_type = matching%process_instance%term(i_term)%nlo_type
            if (nlo_type == BORN) then
               process_deps%i_born = i_term
            else if (nlo_type == NLO_REAL) then
               if (matching%process_instance%term(i_term)%k_term%emitter >= 0) then
                  process_deps%i_real(i) = i_term
                  i = i + 1
               end if
            end if
         end do
      end associate
      call matching%event_deps%p_born_lab%init (n_tot_born, 1)
      call matching%event_deps%p_born_cms%init (n_tot_born, 1)
      call matching%event_deps%p_real_lab%init (n_tot_real, n_phs)
      call matching%event_deps%p_real_cms%init (n_tot_real, n_phs)
   end select
end subroutine powheg_matching_setup_nlo_environment
```

Copy momenta from `event_deps` to `real_kinematics`. So far this is only valid if the center-of-mass system is equal to the lab frame, i.e. for FSR processes without beamstrahlung or structure functions.

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: copy_momenta => powheg_matching_copy_momenta
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_matching_copy_momenta (matching)
   class(powheg_matching_t), intent(inout) :: matching
   integer :: i_phs
   select type (pcm => matching%process_instance%pcm)
   class is (pcm_instance_nlo_t)
      !!! Too much redundance!
      do i_phs = 1, matching%event_deps%p_real_cms%get_n_phs ()
         call pcm%real_kinematics%p_real_cms%set_momenta &
            (i_phs, matching%event_deps%p_real_cms%get_momenta (i_phs))
```

```
            call pcm%real_kinematics%p_real_lab%set_momenta &
               (i_phs,  matching%event_deps%p_real_lab%get_momenta (i_phs))
         end do
      end select
   end subroutine powheg_matching_copy_momenta
```

qcd%alpha%get should implement a variable-flavor result and optionally return n_flavors that are active at the scale...

⟨*POWHEG matching: procedures*⟩+≡

```
   function get_alpha (qcd, scale2) result (alpha_s)
     real(default) :: alpha_s
     class(qcd_t), intent(in) :: qcd
     real(default), intent(in) :: scale2
     integer :: nf, order
     ! TODO: (bcn 2015-01-30) implement variable flavor alpha_s
     alpha_s = qcd%alpha%get (sqrt(scale2))
     select type (alpha => qcd%alpha)
     type is (alpha_qcd_from_scale_t)
        nf = alpha%nf
        order = alpha%order
     type is (alpha_qcd_from_lambda_t)
        nf = alpha%nf
        order = alpha%order
     class default
        call msg_warning ("get_alpha: QCD type is not running!" // &
             "Assuming 5-flavors and LO (1-loop) running!")
        nf = 5
        order = 0
     end select
     if (order > 0) alpha_s = improve_nll_accuracy (alpha_s, nf)
   end function get_alpha
```

See Eq. (4.31) in 0709.2092. Should be used everywhere in the Sudakov exponent.

⟨*POWHEG matching: procedures*⟩+≡

```
   pure function improve_nll_accuracy (alpha_s, n_flavors) result (alpha_s_imp)
     real(default) :: alpha_s_imp
     real(default), intent(in) :: alpha_s
     integer, intent(in) :: n_flavors
       alpha_s_imp = alpha_s * (one + alpha_s / (two*pi) * &
            ((67.0_default/18 - pi**2/6) * CA - five/9 * n_flavors))
   end function improve_nll_accuracy
```

This is fixed to $n_f = 5$ for radiation generation. It will be reweighted to the more precise α_s.

⟨*POWHEG matching: parameters*⟩+≡

```
   real(default), parameter :: b0rad = (33 - 2 * 5) / (12 * pi)
```

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
   procedure :: alpha_s_rad => sudakov_alpha_s_rad
```

⟨*POWHEG matching: procedures*⟩+≡

```
   elemental function sudakov_alpha_s_rad (sudakov, scale2) result (y)
```

```
    real(default) :: y
    class(sudakov_t), intent(in) :: sudakov
    real(default), intent(in) :: scale2
    y = one / (b0rad * log (scale2 / sudakov%process_deps%lambda2_gen))
  end function sudakov_alpha_s_rad
```

⟨*POWHEG matching: sudakov: TBP*⟩+≡

```
  procedure :: reweight_alpha_s => sudakov_reweight_alpha_s
```

⟨*POWHEG matching: procedures*⟩+≡

```
  function sudakov_reweight_alpha_s (sudakov, pt2) result (accepted)
    logical :: accepted
    class(sudakov_t), intent(inout) :: sudakov
    real(default), intent(in) :: pt2
    real(default) :: alpha_s_true, alpha_s_rad
    logical :: alpha_s_equal
    call msg_debug2 (D_MATCHING, "reweight_alpha_s")
    alpha_s_true = get_alpha (sudakov%qcd, pt2)
    alpha_s_rad = sudakov%alpha_s_rad (pt2)
    call sudakov%rng%generate (sudakov%random)
    alpha_s_equal = nearly_equal (alpha_s_true, alpha_s_rad)
    accepted = alpha_s_equal .or. sudakov%random < alpha_s_true / alpha_s_rad
    if (debug2_active (D_MATCHING)) then
       print *, '  sudakov%random =    ', sudakov%random
       print *, '  alpha_s_true =    ', alpha_s_true
       print *, '  alpha_s_rad =    ', alpha_s_rad
       print *, '  alpha_s accepted =    ', accepted
       if (alpha_s_rad < alpha_s_true .and. .not. alpha_s_equal) then
          print *, 'pt2 =    ', pt2
          print *, 'sudakov%process_deps%lambda2_gen =    ', &
               sudakov%process_deps%lambda2_gen
          call msg_fatal ("sudakov_reweight_alpha_s: This should never happen. &
                          &Have you chosen a running alpha_s?")
       end if
    end if
  end function sudakov_reweight_alpha_s
```

31.4.5 Unit tests

Test module, followed by the corresponding implementation module.

⟨powheg_matching_ut.f90⟩≡

```
  ⟨File header⟩

  module powheg_matching_ut
    use unit_tests
    use powheg_matching_uti

  ⟨Standard module head⟩

  ⟨POWHEG matching: public test⟩

  contains
```

```
  ⟨POWHEG matching: test driver⟩

  end module powheg_matching_ut
```

⟨powheg_matching_uti.f90⟩≡

```
  ⟨File header⟩

  module powheg_matching_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use constants, only: zero, one
    use lorentz
    use physics_defs, only: LAMBDA_QCD_REF
    use sm_qcd
    use subevents, only: PRT_INCOMING, PRT_OUTGOING
    use model_data
    use particles
    use rng_base
    use variables
    use shower_base
    use shower_core

    use powheg_matching

    use rng_base_ut, only: rng_test_factory_t

  ⟨Standard module head⟩

  ⟨POWHEG matching: test declarations⟩

  contains

  ⟨POWHEG matching: tests⟩

  end module powheg_matching_uti
```

API: driver for the unit tests below.

⟨POWHEG matching: public test⟩≡

```
    public :: powheg_test
```

⟨POWHEG matching: test driver⟩≡

```
    subroutine powheg_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨POWHEG matching: execute tests⟩
    end subroutine powheg_test
```

Initialization

No Powheg unit tests so far

Compare generated emission with Sudakov form factor

This is a nontrivial test of the generation algorithm and should be independent of the used upper bounding function (as long as all singularities are included).

⟨*POWHEG matching: powheg matching: TBP*⟩+≡

```
procedure :: test_sudakov => powheg_test_sudakov
```

⟨*POWHEG matching: procedures*⟩+≡

```
subroutine powheg_test_sudakov (powheg)
  class(powheg_matching_t), intent(inout) :: powheg
  integer :: n_calls1, n_calls2
  integer, parameter :: n_bins = 20
  real(default) :: sqme_real_x_jacobian, sqme_born
  type(vector4_t), dimension(:), allocatable :: p_born
  real(default), dimension(3) :: random
  real(default) :: xi, y
  integer :: i_call, i_bin, alr, emitter
  real(default) :: alpha_s, kT2, weight
  real(default) :: pt2_min, s, random_jacobian
  real(default), dimension(n_bins) :: histo1, histo2, histo1sq, histo2sq
  real(default), dimension(n_bins) :: tmp
  integer :: i_strip, n_in_strip, n_strips
  real(default), dimension(n_bins) :: average, average_sq, error
  real(default), dimension(n_bins) :: &
       sudakov_0, sudakov_p, sudakov_m, rel_error
  integer :: u

  p_born = powheg%event_deps%p_born_lab%get_momenta (1)
  sqme_born = powheg%event_deps%sqme_born
  s = powheg%event_deps%s_hat
  pt2_min = powheg%settings%pt2_min
  n_calls1 = 100000; n_calls2 = 1000000
  histo1 = zero; histo2 = zero; histo1sq = zero; histo2sq = zero
  n_strips = 10

  call compute_integrals ()
  call generate_emissions ()
  call write_to_screen_and_file ()

contains

⟨POWHEG matching: powheg test sudakov: procedures⟩

end subroutine powheg_test_sudakov
```

⟨*POWHEG matching: powheg test sudakov: procedures*⟩≡

```
pure function binning (i) result (pt2)
  real(default) :: pt2
  integer, intent(in) :: i
  !pt2 = pt2_min + (s-pt2_min) * (i-1) / (n_bins-1)
  pt2 = pt2_min * exp (log (s / pt2_min) * (i-1) / (n_bins-1))
end function
```

⟨*POWHEG matching: powheg test sudakov: procedures*⟩+≡

```
subroutine compute_integrals ()
  write (msg_buffer, "(A)") "POWHEG: test_sudakov: Computing integrals"
  call msg_message ()
  select type (pcm => powheg%process_instance%pcm)
  class is (pcm_instance_nlo_t)
     associate (fks => powheg%phs_fks_generator)
       do i_call = 1, n_calls1
          do alr = 1, powheg%process_deps%n_alr
             call powheg%rng%generate (random)
             select type (config => pcm%config)
             type is (pcm_nlo_t)
                emitter = config%region_data%get_emitter (alr)
             end select
             !!! The sudakov test works only with lepton collisions without beam spectria
             !!! so we can identify the cms and lab momenta.
             powheg%event_deps%p_real_lab = powheg%event_deps%p_real_cms
             call powheg%copy_momenta ()
             call fks%get_radiation_variables (emitter, xi, y)
             kT2 = powheg%sudakov(alr)%s%kt2(xi, y)
             if (kT2 >= pt2_min .and. xi < one - tiny_07) then
                alpha_s = get_alpha (powheg%qcd, kT2)
                sqme_real_x_jacobian = powheg%compute_sqme_real (alr, alpha_s)
                random_jacobian = pcm%real_kinematics%jac_rand (emitter)
                weight = sqme_real_x_jacobian * random_jacobian / sqme_born
                do i_bin = 1, n_bins
                   if (kT2 > binning(i_bin)) then
                      histo1(i_bin) = histo1(i_bin) + weight
                      histo1sq(i_bin) = histo1sq(i_bin) + weight**2
                   end if
                end do
             end if
             ! Do not cycle since there is a Heaviside in the exponent
          end do
          call msg_show_progress (i_call, n_calls1)
       end do
     end associate
  end select
  average = histo1 / n_calls1
  average_sq = histo1sq / n_calls1
  error = sqrt ((average_sq - average**2) / n_calls1)
  sudakov_0 = exp(-average)
  sudakov_p = exp(-(average + error))
  sudakov_m = exp(-(average - error))
  rel_error = (sudakov_0 - sudakov_p + sudakov_m - sudakov_0) / &
       (2 * sudakov_0) * 100
end subroutine compute_integrals
```

⟨*POWHEG matching: powheg test sudakov: procedures*⟩+≡

```
subroutine generate_emissions ()
  write (msg_buffer, "(A)") "POWHEG: test_sudakov: Generating emissions"
  call msg_message ()
  do i_strip = 1, n_strips
     tmp = 0
```

```
      n_in_strip = n_calls2 / n_strips
      do i_call = 1, n_in_strip
         if (signal_is_pending ())  return
         call powheg%generate_emission (pt2_generated = kT2)
         do i_bin = 1, n_bins
            if (kT2 > binning(i_bin)) then
               tmp(i_bin) = tmp(i_bin) + 1
            end if
         end do
      end do
      tmp = one - (one * tmp) / n_in_strip
      histo2 = histo2 + tmp
      histo2sq = histo2sq + tmp**2
      call msg_show_progress (i_strip, n_strips)
   end do
   average = histo2 / n_strips
   average_sq = histo2sq / n_strips
   error = sqrt ((average_sq - average**2) / n_strips)
end subroutine generate_emissions
```

⟨*POWHEG matching: powheg test sudakov: procedures*⟩+≡

```
subroutine write_to_screen_and_file ()
   u = free_unit ()
   open (file='sudakov.dat', unit=u, action='write')
   print *, 'exp(-Integrated R/B)-distribution: '
   print *, 'pT2  sudakov_+  sudakov_0  sudakov_-  rel_err[%]: '
   do i_bin = 1, n_bins
      print *, binning(i_bin), &
           sudakov_p(i_bin), sudakov_0(i_bin), sudakov_m(i_bin), &
           rel_error(i_bin)
      write (u, "(6(" // FMT_16 // ",2X))") binning(i_bin), &
           sudakov_p(i_bin), sudakov_0(i_bin), sudakov_m(i_bin), &
           average(i_bin), error(i_bin)
   end do
   close (u)
   print *, '******************************'
   print *, 'Noemission probability: '
   do i_bin = 1, n_bins
      print *, binning (i_bin), average (i_bin), error(i_bin)
   end do
end subroutine write_to_screen_and_file
```

Chapter 32

Event Implementation

With a process object and the associated methods at hand, we can generate events for elementary processes and, by subsequent transformation, for complete physical processes.

We have the following modules:

event_transforms Abstract base type for transforming a physical process with process instance and included evaluators, etc., into a new object. The following modules extend this base type.

decays Combine the elementary process with elementary decay processes and thus transform the elementary event into a decayed event, still at the parton level.

showers Create QED/QCD showers out of the partons that are emitted by elementary processes. This should be interleaved with showering of radiated particles (structure functions) and multiple interactions.

hadrons (not implemented yet) Apply hadronization to the partonic events, interleaved with hadron decays. (The current setup relies on hadronizing partonic events externally.)

tau_decays (not implemented yet) Let τ leptons decay taking full spin correlations into account.

events Combine all pieces to generate full events.

eio_raw Raw I/O for complete events.

32.1 Abstract Event Transforms

⟨`event_transforms.f90`⟩≡
  ⟨*File header*⟩

```
module event_transforms
```

  ⟨*Use kinds*⟩
  ⟨*Use strings*⟩

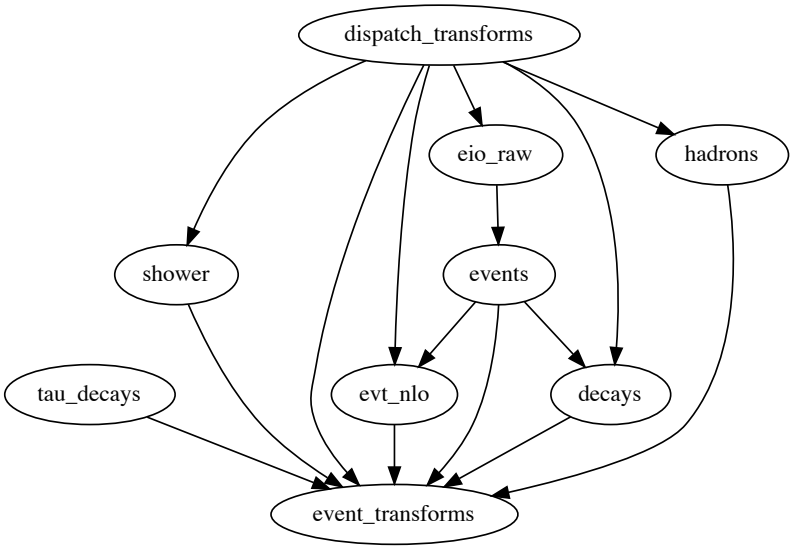

Figure 32.1: Module dependencies in `src/transforms`.

```
  use io_units
  use format_utils, only: write_separator
  use diagnostics
  use model_data
  use interactions
  use particles
  use subevents
  use rng_base
  use quantum_numbers, only: quantum_numbers_t
  use process, only: process_t
  use instances, only: process_instance_t
  use process_stacks
```

⟨*Standard module head*⟩

⟨*Event transforms: public*⟩

⟨*Event transforms: types*⟩

⟨*Event transforms: interfaces*⟩

```
contains
```

⟨*Event transforms: procedures*⟩

```
end module event_transforms
```

32.1.1 Abstract base type

Essentially, all methods are abstract, but some get minimal base versions. We know that there will be a random-number generator at top level, and that we will relate to an elementary process.

The model is stored separately. It may contain modified setting that differ from the model instance stored in the process object.

Each event transform contains a particle set that it can fill for further use. There is a flag that indicates this.

We will collect event transforms in a list, therefore we include `previous` and `next` pointers.

⟨*Event transforms: public*⟩≡
```
public :: evt_t
```

⟨*Event transforms: types*⟩≡
```
type, abstract :: evt_t
   type(process_t), pointer :: process => null ()
   type(process_instance_t), pointer :: process_instance => null ()
   class(model_data_t), pointer :: model => null ()
   class(rng_t), allocatable :: rng
   integer :: rejection_count = 0
   logical :: particle_set_exists = .false.
   type(particle_set_t) :: particle_set
   class(evt_t), pointer :: previous => null ()
   class(evt_t), pointer :: next => null ()
   real(default) :: weight = 0._default
   logical :: only_weighted_events = .false.
 contains
 ⟨Event transforms: evt: TBP⟩
end type evt_t
```

Finalizer. In any case, we finalize the r.n.g. The process instance is a pointer and should not be finalized here.

⟨*Event transforms: evt: TBP*⟩≡
```
procedure :: final => evt_final
procedure :: base_final => evt_final
```

⟨*Event transforms: procedures*⟩≡
```
subroutine evt_final (evt)
  class(evt_t), intent(inout) :: evt
  if (allocated (evt%rng))  call evt%rng%final ()
  if (evt%particle_set_exists) &
       call evt%particle_set%final ()
end subroutine evt_final
```

Print out the type of the `evt`.

⟨*Event transforms: evt: TBP*⟩+≡
```
procedure (evt_write_name), deferred :: write_name
```

⟨*Event transforms: interfaces*⟩≡
```
abstract interface
   subroutine evt_write_name (evt, unit)
     import
     class(evt_t), intent(in) :: evt
```

```
      integer, intent(in), optional :: unit
    end subroutine evt_write_name
end interface
```

⟨*Event transforms: evt: TBP*⟩+≡

```
procedure (evt_write), deferred :: write
```

⟨*Event transforms: interfaces*⟩+≡

```
abstract interface
   subroutine evt_write (evt, unit, verbose, more_verbose, testflag)
     import
     class(evt_t), intent(in) :: evt
     integer, intent(in), optional :: unit
     logical, intent(in), optional :: verbose, more_verbose, testflag
   end subroutine evt_write
end interface
```

Output. We can print r.n.g. info.

⟨*Event transforms: evt: TBP*⟩+≡

```
procedure :: base_write => evt_base_write
```

⟨*Event transforms: procedures*⟩+≡

```
subroutine evt_base_write (evt, unit, testflag, show_set)
  class(evt_t), intent(in) :: evt
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: testflag, show_set
  integer :: u
  logical :: show
  u = given_output_unit (unit)
  show = .true.;  if (present (show_set))  show = show_set
  if (associated (evt%process)) then
     write (u, "(3x,A,A,A)")   "Associated process: '", &
          char (evt%process%get_id ()), "'"
  end if
  if (allocated (evt%rng)) then
     call evt%rng%write (u, 1)
     write (u, "(3x,A,I0)")  "Number of tries = ", evt%rejection_count
  end if
  if (show) then
     if (evt%particle_set_exists) then
        call write_separator (u)
        call evt%particle_set%write (u, testflag = testflag)
     end if
  end if
end subroutine evt_base_write
```

Connect the transform with a process instance (and thus with the associated process). Use this to allocate the master random-number generator.

This is not an initializer; we may initialize the transform by implementation-specific methods.

⟨*Event transforms: evt: TBP*⟩+≡

```
procedure :: connect => evt_connect
procedure :: base_connect => evt_connect
```

⟨*Event transforms: procedures*⟩+≡

```
subroutine evt_connect (evt, process_instance, model, process_stack)
  class(evt_t), intent(inout), target :: evt
  type(process_instance_t), intent(in), target :: process_instance
  class(model_data_t), intent(in), target :: model
  type(process_stack_t), intent(in), optional :: process_stack
  evt%process => process_instance%process
  evt%process_instance => process_instance
  evt%model => model
  call evt%process%make_rng (evt%rng)
end subroutine evt_connect
```

Reset internal state.

⟨*Event transforms: evt: TBP*⟩+≡

```
procedure :: reset => evt_reset
procedure :: base_reset => evt_reset
```

⟨*Event transforms: procedures*⟩+≡

```
subroutine evt_reset (evt)
  class(evt_t), intent(inout) :: evt
  evt%rejection_count = 0
  evt%particle_set_exists = .false.
end subroutine evt_reset
```

Prepare for a new event: reset internal state, if necessary. We provide MCI and term index of the parent process.

⟨*Event transforms: evt: TBP*⟩+≡

```
procedure (evt_prepare_new_event), deferred :: prepare_new_event
```

⟨*Event transforms: interfaces*⟩+≡

```
interface
   subroutine evt_prepare_new_event (evt, i_mci, i_term)
     import
     class(evt_t), intent(inout) :: evt
     integer, intent(in) :: i_mci, i_term
   end subroutine evt_prepare_new_event
end interface
```

Generate a weighted event, using a valid initiator event in the process instance, and the random-number generator. The returned event probability should be a number between zero and one that we can use for rejection.

⟨*Event transforms: evt: TBP*⟩+≡

```
procedure (evt_generate_weighted), deferred :: generate_weighted
```

⟨*Event transforms: interfaces*⟩+≡

```
abstract interface
   subroutine evt_generate_weighted (evt, probability)
     import
     class(evt_t), intent(inout) :: evt
     real(default), intent(inout) :: probability
   end subroutine evt_generate_weighted
end interface
```

The unweighted event generation routine is actually implemented. It uses the random-number generator for simple rejection. Of course, the implementation may override this and implement a different way of generating an unweighted event.

⟨*Event transforms: evt: TBP*⟩+≡

```
procedure :: generate_unweighted => evt_generate_unweighted
procedure :: base_generate_unweighted => evt_generate_unweighted
```

⟨*Event transforms: procedures*⟩+≡

```
subroutine evt_generate_unweighted (evt)
  class(evt_t), intent(inout) :: evt
  real(default) :: p, x
  evt%rejection_count = 0
  REJECTION: do
     evt%rejection_count = evt%rejection_count + 1
     call evt%generate_weighted (p)
     if (signal_is_pending ())  return
     call evt%rng%generate (x)
     if (x < p)  exit REJECTION
  end do REJECTION
end subroutine evt_generate_unweighted
```

Make a particle set. This should take the most recent evaluator (or whatever stores the event), factorize the density matrix if necessary, and store as a particle set.

If applicable, the factorization should make use of the `factorization_mode` and `keep_correlations` settings.

The values `r`, if set, should control the factorization in more detail, e.g., bypassing the random-number generator.

⟨*Event transforms: evt: TBP*⟩+≡

```
procedure (evt_make_particle_set), deferred :: make_particle_set
```

⟨*Event transforms: interfaces*⟩+≡

```
interface
   subroutine evt_make_particle_set &
        (evt, factorization_mode, keep_correlations, r)
     import
     class(evt_t), intent(inout) :: evt
     integer, intent(in) :: factorization_mode
     logical, intent(in) :: keep_correlations
     real(default), dimension(:), intent(in), optional :: r
   end subroutine evt_make_particle_set
end interface
```

Copy an existing particle set into the event record. This bypasses all methods to evaluate the internal state, but may be sufficient for further processing.

⟨*Event transforms: evt: TBP*⟩+≡

```
procedure :: set_particle_set => evt_set_particle_set
```

⟨*Event transforms: procedures*⟩+≡

```
subroutine evt_set_particle_set (evt, particle_set, i_mci, i_term)
  class(evt_t), intent(inout) :: evt
  type(particle_set_t), intent(in) :: particle_set
```

```
    integer, intent(in) :: i_term, i_mci
    call evt%prepare_new_event (i_mci, i_term)
    evt%particle_set = particle_set
    evt%particle_set_exists = .true.
  end subroutine evt_set_particle_set
```

This procedure can help in the previous task, if the particles are available in the form of an interaction object. (We need two interactions, one with color summed over, and one with the probability distributed among flows.)

We use the two values from the random number generator for factorizing the state. For testing purposes, we can provide those numbers explicitly.

⟨*Event transforms: evt: TBP*⟩+≡

```
  procedure :: factorize_interactions => evt_factorize_interactions
```

⟨*Event transforms: procedures*⟩+≡

```
  subroutine evt_factorize_interactions &
       (evt, int_matrix, int_flows, factorization_mode, &
       keep_correlations, r, qn_select)
    class(evt_t), intent(inout) :: evt
    type(interaction_t), intent(in), target :: int_matrix, int_flows
    integer, intent(in) :: factorization_mode
    logical, intent(in) :: keep_correlations
    real(default), dimension(:), intent(in), optional :: r
    type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_select
    real(default), dimension(2) :: x
    if (present (r)) then
       if (size (r) == 2) then
          x = r
       else
          call msg_bug ("event factorization: size of r array must be 2")
       end if
    else
       call evt%rng%generate (x)
    end if
    call evt%particle_set%init (evt%particle_set_exists, &
         int_matrix, int_flows, factorization_mode, x, &
         keep_correlations, keep_virtual=.true., qn_select = qn_select)
    evt%particle_set_exists = .true.
  end subroutine evt_factorize_interactions
```

⟨*Event transforms: public*⟩+≡

```
  public :: make_factorized_particle_set
```

⟨*Event transforms: procedures*⟩+≡

```
  subroutine make_factorized_particle_set (evt, factorization_mode, &
         keep_correlations, r, ii_term, qn_select)
    class(evt_t), intent(inout) :: evt
    integer, intent(in) :: factorization_mode
    logical, intent(in) :: keep_correlations
    real(default), dimension(:), intent(in), optional :: r
    integer, intent(in), optional :: ii_term
    type(quantum_numbers_t), dimension(:), intent(in), optional :: qn_select
    integer :: i_term
    type(interaction_t), pointer :: int_matrix, int_flows
```

```
    if (evt%process_instance%is_complete_event ()) then
       if (present (ii_term)) then
          i_term = ii_term
       else
          i_term = evt%process_instance%select_i_term ()
       end if
       int_matrix => evt%process_instance%get_matrix_int_ptr (i_term)
       int_flows  => evt%process_instance%get_flows_int_ptr (i_term)
       call evt%factorize_interactions (int_matrix, int_flows, &
            factorization_mode, keep_correlations, r, qn_select)
       call evt%tag_incoming ()
    else
       call msg_bug ("Event factorization: event is incomplete")
    end if
  end subroutine make_factorized_particle_set
```

Mark the incoming particles as incoming in the particle set. This is necessary because in the interaction objects they are usually marked as virtual.

In the inquiry functions we set the term index to one; the indices of beams and incoming particles should be identical for all process terms.

We use the initial elementary process for obtaining the indices. Thus, we implicitly assume that the beam and incoming indices stay the same across event transforms. If this is not true for a transform (say, MPI), it should override this method.

⟨*Event transforms: evt: TBP*⟩+≡

```
  procedure :: tag_incoming => evt_tag_incoming
```

⟨*Event transforms: procedures*⟩+≡

```
  subroutine evt_tag_incoming (evt)
    class(evt_t), intent(inout) :: evt
    integer :: i_term, n_in
    integer, dimension(:), allocatable :: beam_index, in_index
    n_in = evt%process%get_n_in ()
    i_term = 1
    allocate (beam_index (n_in))
    call evt%process_instance%get_beam_index (i_term, beam_index)
    call evt%particle_set%reset_status (beam_index, PRT_BEAM)
    allocate (in_index (n_in))
    call evt%process_instance%get_in_index (i_term, in_index)
    call evt%particle_set%reset_status (in_index, PRT_INCOMING)
  end subroutine evt_tag_incoming
```

32.1.2 Implementation: Trivial transform

This transform contains just a pointer to process and process instance. The `generate` methods do nothing.

⟨*Event transforms: public*⟩+≡

```
  public :: evt_trivial_t
```

⟨*Event transforms: types*⟩+≡

```
  type, extends (evt_t) :: evt_trivial_t
   contains
```

```
  ⟨Event transforms: evt trivial: TBP⟩
end type evt_trivial_t
```

⟨*Event transforms: evt trivial: TBP*⟩≡

```
procedure :: write_name => evt_trivial_write_name
```

⟨*Event transforms: procedures*⟩+≡

```
subroutine evt_trivial_write_name (evt, unit)
  class(evt_trivial_t), intent(in) :: evt
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Event transform: trivial (hard process)"
end subroutine evt_trivial_write_name
```

The finalizer is trivial. Some output:

⟨*Event transforms: evt trivial: TBP*⟩+≡

```
procedure :: write => evt_trivial_write
```

⟨*Event transforms: procedures*⟩+≡

```
subroutine evt_trivial_write (evt, unit, verbose, more_verbose, testflag)
  class(evt_trivial_t), intent(in) :: evt
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose, more_verbose, testflag
  integer :: u
  u = given_output_unit (unit)
  call write_separator (u, 2)
  call evt%write_name (u)
  call write_separator (u)
  call evt%base_write (u, testflag = testflag)
end subroutine evt_trivial_write
```

Nothing to do here:

⟨*Event transforms: evt trivial: TBP*⟩+≡

```
procedure :: prepare_new_event => evt_trivial_prepare_new_event
```

⟨*Event transforms: procedures*⟩+≡

```
subroutine evt_trivial_prepare_new_event (evt, i_mci, i_term)
  class(evt_trivial_t), intent(inout) :: evt
  integer, intent(in) :: i_mci, i_term
  call evt%reset ()
end subroutine evt_trivial_prepare_new_event
```

The weighted generator is, surprisingly, trivial.

⟨*Event transforms: evt trivial: TBP*⟩+≡

```
procedure :: generate_weighted => evt_trivial_generate_weighted
```

⟨*Event transforms: procedures*⟩+≡

```
subroutine evt_trivial_generate_weighted (evt, probability)
  class(evt_trivial_t), intent(inout) :: evt
  real(default), intent(inout) :: probability
  probability = 1
end subroutine evt_trivial_generate_weighted
```

This routine makes a particle set, using the associated process instance as-is.

⟨*Event transforms: evt trivial: TBP*⟩+≡
```
procedure :: make_particle_set => evt_trivial_make_particle_set
```

⟨*Event transforms: procedures*⟩+≡
```
subroutine evt_trivial_make_particle_set &
     (evt, factorization_mode, keep_correlations, r)
  class(evt_trivial_t), intent(inout) :: evt
  integer, intent(in) :: factorization_mode
  logical, intent(in) :: keep_correlations
  real(default), dimension(:), intent(in), optional :: r
  call make_factorized_particle_set (evt, factorization_mode, &
       keep_correlations, r)
  evt%particle_set_exists = .true.
end subroutine evt_trivial_make_particle_set
```

32.1.3 Unit tests

Test module, followed by the corresponding implementation module.

⟨event_transforms_ut.f90⟩≡
```
⟨File header⟩

module event_transforms_ut
  use unit_tests
  use event_transforms_uti

⟨Standard module head⟩

⟨Event transforms: public test⟩

contains

⟨Event transforms: test driver⟩

end module event_transforms_ut
```

⟨event_transforms_uti.f90⟩≡
```
⟨File header⟩

module event_transforms_uti

⟨Use kinds⟩
⟨Use strings⟩
  use format_utils, only: write_separator
  use os_interface
  use sm_qcd
  use model_data
  use state_matrices, only: FM_IGNORE_HELICITY
  use interactions, only: reset_interaction_counter
  use process_libraries
  use rng_base
  use mci_base
  use mci_midpoint
```

```
  use phs_base
  use phs_single
  use prc_core
  use prc_test, only: prc_test_create_library

  use process, only: process_t
  use instances, only: process_instance_t

  use event_transforms

  use rng_base_ut, only: rng_test_factory_t
```

⟨*Standard module head*⟩

⟨*Event transforms: test declarations*⟩

```
contains
```

⟨*Event transforms: tests*⟩

```
end module event_transforms_uti
```

API: driver for the unit tests below.

⟨*Event transforms: public test*⟩≡

```
  public :: event_transforms_test
```

⟨*Event transforms: test driver*⟩≡

```
  subroutine event_transforms_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```

⟨*Event transforms: execute tests*⟩

```
  end subroutine event_transforms_test
```

Test trivial event transform

The trivial transform, as an instance of the abstract transform, does nothing but to trigger event generation for an elementary process.

⟨*Event transforms: execute tests*⟩≡

```
  call test (event_transforms_1, "event_transforms_1", &
       "trivial event transform", &
       u, results)
```

⟨*Event transforms: test declarations*⟩≡

```
  public :: event_transforms_1
```

⟨*Event transforms: tests*⟩≡

```
  subroutine event_transforms_1 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    type(qcd_t) :: qcd
    class(rng_factory_t), allocatable :: rng_factory
    class(model_data_t), pointer :: model
    type(process_library_t), target :: lib
```

```
type(string_t) :: libname, procname1, run_id
class(mci_t), allocatable :: mci_template
class(phs_config_t), allocatable :: phs_config_template
real(default) :: sqrts
type(process_t), allocatable, target :: process
type(process_instance_t), allocatable, target :: process_instance
class(evt_t), allocatable :: evt
integer :: factorization_mode
logical :: keep_correlations

write (u, "(A)")  "* Test output: event_transforms_1"
write (u, "(A)")  "*   Purpose: handle trivial transform"
write (u, "(A)")

write (u, "(A)")  "* Initialize environment and parent process"
write (u, "(A)")

call os_data_init (os_data)
allocate (rng_test_factory_t :: rng_factory)

libname = "event_transforms_1_lib"
procname1 = "event_transforms_1_p"
run_id = "event_transforms_1"

call prc_test_create_library (libname, lib, &
     scattering = .true., procname1 = procname1)
call reset_interaction_counter ()

allocate (model)
call model%init_test ()

allocate (process)
call process%init (procname1, run_id, &
     lib, os_data, qcd, rng_factory, model)
call process%setup_test_cores ()

allocate (mci_midpoint_t :: mci_template)
allocate (phs_single_config_t :: phs_config_template)
call process%init_component &
   (1, .true., mci_template, phs_config_template)

sqrts = 1000
call process%setup_beams_sqrts (sqrts, i_core = 1)
call process%configure_phs ()
call process%setup_mci ()
call process%setup_terms ()

allocate (process_instance)
call process_instance%init (process)
call process_instance%integrate (1, n_it=1, n_calls=100)
call process%final_integration (1)
call process_instance%final ()
deallocate (process_instance)
```

```
  allocate (process_instance)
  call process_instance%init (process)
  call process_instance%setup_event_data ()
  call process_instance%init_simulation (1)

  write (u, "(A)")  "* Initialize trivial event transform"
  write (u, "(A)")

  allocate (evt_trivial_t :: evt)
  model => process%get_model_ptr ()
  call evt%connect (process_instance, model)

  write (u, "(A)")  "* Generate event and subsequent transform"
  write (u, "(A)")

  call process_instance%generate_unweighted_event (1)
  call process_instance%evaluate_event_data ()

  call evt%prepare_new_event (1, 1)
  call evt%generate_unweighted ()

  call write_separator (u, 2)
  call evt%write (u)
  call write_separator (u, 2)

  write (u, "(A)")
  write (u, "(A)")  "* Obtain particle set"
  write (u, "(A)")

  factorization_mode = FM_IGNORE_HELICITY
  keep_correlations = .false.

  call evt%make_particle_set (factorization_mode, keep_correlations)

  call write_separator (u, 2)
  call evt%write (u)
  call write_separator (u, 2)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call evt%final ()
  call process_instance%final ()
  deallocate (process_instance)

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: event_transforms_1"

end subroutine event_transforms_1
```

32.2 Hadronization interface

⟨hadrons.f90⟩≡
⟨*File header*⟩

```
module hadrons
```

⟨*Use kinds with double*⟩
⟨*Use strings*⟩

```
  use io_units
  use format_utils, only: write_separator
  use diagnostics
  use sm_qcd
  use lorentz
  use subevents, only: PRT_OUTGOING
  use particles
  use variables
  use model_data
  use models
  use rng_base
  use hep_common
  use event_transforms
  use shower_base
  use shower_pythia6
  use process
```

⟨*Standard module head*⟩

⟨*Hadrons: public*⟩

⟨*Hadrons: parameters*⟩

⟨*Hadrons: types*⟩

⟨*Hadrons: interfaces*⟩

```
contains
```

⟨*Hadrons: procedures*⟩

```
end module hadrons
```

32.2.1 Hadronization implementations

⟨*Hadrons: public*⟩≡

```
  public :: HADRONS_UNDEFINED, HADRONS_WHIZARD, HADRONS_PYTHIA6, HADRONS_PYTHIA8
```

⟨*Hadrons: parameters*⟩≡

```
  integer, parameter :: HADRONS_UNDEFINED = 0
  integer, parameter :: HADRONS_WHIZARD = 1
  integer, parameter :: HADRONS_PYTHIA6 = 2
  integer, parameter :: HADRONS_PYTHIA8 = 3
```

A dictionary

⟨*Hadrons: public*⟩+≡

```
    public :: hadrons_method
```

⟨*Hadrons: interfaces*⟩≡

```
    interface hadrons_method
       module procedure hadrons_method_of_string
       module procedure hadrons_method_to_string
    end interface
```

⟨*Hadrons: procedures*⟩≡

```
    elemental function hadrons_method_of_string (string) result (i)
      integer :: i
      type(string_t), intent(in) :: string
      select case (char(string))
      case ("WHIZARD")
         i = HADRONS_WHIZARD
      case ("PYTHIA6")
         i = HADRONS_PYTHIA6
      case ("PYTHIA8")
         i = HADRONS_PYTHIA8
      case default
         i = HADRONS_UNDEFINED
      end select
    end function hadrons_method_of_string

    elemental function hadrons_method_to_string (i) result (string)
      type(string_t) :: string
      integer, intent(in) :: i
      select case (i)
      case (HADRONS_WHIZARD)
         string = "WHIZARD"
      case (HADRONS_PYTHIA6)
         string = "PYTHIA6"
      case (HADRONS_PYTHIA8)
         string = "PYTHIA8"
      case default
         string = "UNDEFINED"
      end select
    end function hadrons_method_to_string
```

32.2.2 Hadronization settings

These are the general settings and parameters for the different shower methods.

⟨*Hadrons: public*⟩+≡

```
    public :: hadron_settings_t
```

⟨*Hadrons: types*⟩≡

```
    type :: hadron_settings_t
       logical :: active = .false.
       integer :: method = HADRONS_UNDEFINED
       real(default) :: enhanced_fraction = 0
       real(default) :: enhanced_width = 0
     contains
     ⟨Hadrons: hadron settings: TBP⟩
    end type hadron_settings_t
```

Read in the hadronization settings.

⟨*Hadrons: hadron settings: TBP*⟩≡
```
procedure :: init => hadron_settings_init
```

⟨*Hadrons: procedures*⟩+≡
```
subroutine hadron_settings_init (hadron_settings, var_list)
  class(hadron_settings_t), intent(out) :: hadron_settings
  type(var_list_t), intent(in) :: var_list
  hadron_settings%active = &
       var_list%get_lval (var_str ("?hadronization_active"))
  hadron_settings%method = hadrons_method_of_string ( &
       var_list%get_sval (var_str ("$hadronization_method")))
  hadron_settings%enhanced_fraction = &
       var_list%get_rval (var_str ("hadron_enhanced_fraction"))
  hadron_settings%enhanced_width = &
       var_list%get_rval (var_str ("hadron_enhanced_width"))
end subroutine hadron_settings_init
```

⟨*Hadrons: hadron settings: TBP*⟩+≡
```
procedure :: write => hadron_settings_write
```

⟨*Hadrons: procedures*⟩+≡
```
subroutine hadron_settings_write (settings, unit)
  class(hadron_settings_t), intent(in) :: settings
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  write (u, "(1x,A)")  "Hadronization settings:"
  call write_separator (u)
  write (u, "(1x,A)")  "Master switches:"
  write (u, "(3x,A,1x,L1)") &
       "active                = ", settings%active
  write (u, "(1x,A)")  "General settings:"
  if (settings%active) then
     write (u, "(3x,A)") &
          "hadron_method         =  " // &
          char (hadrons_method_to_string (settings%method))
  else
     write (u, "(3x,A)") " [Hadronization off]"
  end if
  write (u, "(1x,A)")  "pT generation parameters"
  write (u, "(3x,A,1x,ES19.12)") &
       "enhanced_fraction     = ", settings%enhanced_fraction
  write (u, "(3x,A,1x,ES19.12)") &
       "enhanced_width        = ", settings%enhanced_width
end subroutine hadron_settings_write
```

32.2.3 Abstract Hadronization Type

The `model` is the fallback model including all hadrons

⟨*Hadrons: types*⟩+≡
```
type, abstract :: hadrons_t
```

```
      class(rng_t), allocatable :: rng
      type(shower_settings_t) :: shower_settings
      type(hadron_settings_t) :: hadron_settings
      type(model_t), pointer :: model => null()
    contains
    ⟨Hadrons: hadrons: TBP⟩
  end type hadrons_t
```

⟨*Hadrons: hadrons: TBP*⟩≡

```
  procedure (hadrons_init), deferred :: init
```

⟨*Hadrons: interfaces*⟩+≡

```
  abstract interface
    subroutine hadrons_init &
         (hadrons, shower_settings, hadron_settings, model_hadrons)
      import
      class(hadrons_t), intent(out) :: hadrons
      type(shower_settings_t), intent(in) :: shower_settings
      type(hadron_settings_t), intent(in) :: hadron_settings
      type(model_t), target, intent(in) :: model_hadrons
    end subroutine hadrons_init
  end interface
```

⟨*Hadrons: hadrons: TBP*⟩+≡

```
  procedure (hadrons_hadronize), deferred :: hadronize
```

⟨*Hadrons: interfaces*⟩+≡

```
  abstract interface
     subroutine hadrons_hadronize (hadrons, particle_set, valid)
       import
       class(hadrons_t), intent(inout) :: hadrons
       type(particle_set_t), intent(in) :: particle_set
       logical, intent(out) :: valid
     end subroutine hadrons_hadronize
  end interface
```

⟨*Hadrons: hadrons: TBP*⟩+≡

```
  procedure (hadrons_make_particle_set), deferred :: make_particle_set
```

⟨*Hadrons: interfaces*⟩+≡

```
  abstract interface
     subroutine hadrons_make_particle_set (hadrons, particle_set, &
          model, valid)
       import
       class(hadrons_t), intent(in) :: hadrons
       type(particle_set_t), intent(inout) :: particle_set
       class(model_data_t), intent(in), target :: model
       logical, intent(out) :: valid
     end subroutine hadrons_make_particle_set
  end interface
```

⟨*Hadrons: hadrons: TBP*⟩+≡

```
  procedure :: import_rng => hadrons_import_rng
```

⟨*Hadrons: procedures*⟩+≡
```
pure subroutine hadrons_import_rng (hadrons, rng)
  class(hadrons_t), intent(inout) :: hadrons
  class(rng_t), intent(inout), allocatable :: rng
  call move_alloc (from = rng, to = hadrons%rng)
end subroutine hadrons_import_rng
```

32.2.4 WHIZARD Hadronization Type

Hadronization can be (incompletely) performed through WHIZARD's internal routine.

⟨*Hadrons: public*⟩+≡
```
public :: hadrons_hadrons_t
```

⟨*Hadrons: types*⟩+≡
```
type, extends (hadrons_t) :: hadrons_hadrons_t
   contains
     ⟨Hadrons: hadrons hadrons: TBP⟩
  end type hadrons_hadrons_t
```

⟨*Hadrons: hadrons hadrons: TBP*⟩≡
```
procedure :: init => hadrons_hadrons_init
```

⟨*Hadrons: procedures*⟩+≡
```
subroutine hadrons_hadrons_init &
     (hadrons, shower_settings, hadron_settings, model_hadrons)
  class(hadrons_hadrons_t), intent(out) :: hadrons
  type(shower_settings_t), intent(in) :: shower_settings
  type(hadron_settings_t), intent(in) :: hadron_settings
  type(model_t), intent(in), target :: model_hadrons
  hadrons%model => model_hadrons
  hadrons%shower_settings = shower_settings
  hadrons%hadron_settings = hadron_settings
  call msg_message &
       ("Hadronization: WHIZARD model for hadronization and decays")
end subroutine hadrons_hadrons_init
```

⟨*Hadrons: hadrons hadrons: TBP*⟩+≡
```
procedure :: hadronize => hadrons_hadrons_hadronize
```

⟨*Hadrons: procedures*⟩+≡
```
subroutine hadrons_hadrons_hadronize (hadrons, particle_set, valid)
  class(hadrons_hadrons_t), intent(inout) :: hadrons
  type(particle_set_t), intent(in) :: particle_set
  logical, intent(out) :: valid
  integer, dimension(:), allocatable :: cols, acols, octs
  integer :: n
  if (signal_is_pending ()) return
  call msg_debug (D_TRANSFORMS, "hadrons_hadrons_hadronize")
  call particle_set%write (6, compressed=.true.)
  n = particle_set%get_n_tot ()
  allocate (cols (n), acols (n), octs (n))
```

```
    call extract_color_systems (particle_set, cols, acols, octs)
    print *, "size(cols)  = ", size (cols)
    if (size(cols) > 0) then
       print *, "cols  = ", cols
    end if
    print *, "size(acols) = ", size(acols)
    if (size(acols) > 0) then
       print *, "acols = ", acols
    end if
    print *, "size(octs)  = ", size(octs)
    if (size (octs) > 0) then
       print *, "octs  = ", octs
    end if
    !!! if all arrays are empty, i.e. zero particles found, nothing to do
  end subroutine hadrons_hadrons_hadronize
```

This type contains a flavor selector for the creation of hadrons, including parameters for the special handling of baryons.

⟨*Hadrons: public*⟩+≡

```
  public :: had_flav_t
```

⟨*Hadrons: types*⟩+≡

```
  type had_flav_t
  end type had_flav_t
```

This is the type for the ends of Lund strings.

⟨*Hadrons: public*⟩+≡

```
  public :: lund_end
```

⟨*Hadrons: types*⟩+≡

```
  type lund_end
     logical :: from_pos
     integer :: i_end
     integer :: i_max
     integer :: id_had
     integer :: i_pos_old
     integer :: i_neg_old
     integer :: i_pos_new
     integer :: i_neg_new
     real(default) :: px_old
     real(default) :: py_old
     real(default) :: px_new
     real(default) :: py_new
     real(default) :: px_had
     real(default) :: py_had
     real(default) :: m_had
     real(default) :: mT2_had
     real(default) :: z_had
     real(default) :: gamma_old
     real(default) :: gamma_new
     real(default) :: x_pos_old
     real(default) :: x_pos_new
     real(default) :: x_pos_had
     real(default) :: x_neg_old
```

```
      real(default) :: x_neg_new
      real(default) :: x_neg_had
      type(had_flav_t) :: old_flav
      type(had_flav_t) :: new_flav
      type(vector4_t) :: p_had
      type(vector4_t) :: p_pre
   end type lund_end
```

Generator for transverse momentum for the fragmentation.

⟨*Hadrons: public*⟩+≡

```
   public :: lund_pt_t
```

⟨*Hadrons: types*⟩+≡

```
   type lund_pt_t
      real(default) :: sigma_min
      real(default) :: sigma_q
      real(default) :: enhanced_frac
      real(default) :: enhanced_width
      real(default) :: sigma_to_had
      class(rng_t), allocatable :: rng
    contains
    ⟨Hadrons: lund pT: TBP⟩
   end type lund_pt_t
```

⟨*Hadrons: lund pT: TBP*⟩≡

```
   procedure :: init => lund_pt_init
```

⟨*Hadrons: procedures*⟩+≡

```
   subroutine lund_pt_init (lund_pt, settings)
     class (lund_pt_t), intent(out) :: lund_pt
     type(hadron_settings_t), intent(in) :: settings
   end subroutine lund_pt_init
```

⟨*Hadrons: hadrons hadrons: TBP*⟩+≡

```
   procedure :: make_particle_set => hadrons_hadrons_make_particle_set
```

⟨*Hadrons: procedures*⟩+≡

```
   subroutine hadrons_hadrons_make_particle_set &
          (hadrons, particle_set, model, valid)
     class(hadrons_hadrons_t), intent(in) :: hadrons
     type(particle_set_t), intent(inout) :: particle_set
     class(model_data_t), intent(in), target :: model
     logical, intent(out) :: valid
     if (signal_is_pending ()) return
     valid = .false.
     if (valid) then
     else
        call msg_fatal ("WHIZARD hadronization not yet implemented")
     end if
   end subroutine hadrons_hadrons_make_particle_set
```

⟨*Hadrons: procedures*⟩+≡

```
   subroutine extract_color_systems (p_set, cols, acols, octs)
```

```
    type(particle_set_t), intent(in) :: p_set
    integer, dimension(:), allocatable, intent(out) :: cols, acols, octs
    logical, dimension(:), allocatable :: mask
    integer :: i, n, n_cols, n_acols, n_octs
    n = p_set%get_n_tot ()
    allocate (mask (n))
    do i = 1, n
       mask(i) = p_set%prt(i)%col%get_col () /= 0 .and. &
            p_set%prt(i)%col%get_acl () == 0 .and. &
            p_set%prt(i)%get_status () == PRT_OUTGOING
    end do
    n_cols = count (mask)
    allocate (cols (n_cols))
    cols = p_set%get_indices (mask)
    do i = 1, n
       mask(i) = p_set%prt(i)%col%get_col () == 0 .and. &
            p_set%prt(i)%col%get_acl () /= 0 .and. &
            p_set%prt(i)%get_status () == PRT_OUTGOING
    end do
    n_acols = count (mask)
    allocate (acols (n_acols))
    acols = p_set%get_indices (mask)
    do i = 1, n
       mask(i) = p_set%prt(i)%col%get_col () /= 0 .and. &
            p_set%prt(i)%col%get_acl () /= 0 .and. &
            p_set%prt(i)%get_status () == PRT_OUTGOING
    end do
    n_octs = count (mask)
    allocate (octs (n_octs))
    octs = p_set%get_indices (mask)
  end subroutine extract_color_systems
```

32.2.5 PYTHIA6 Hadronization Type

Hadronization via PYTHIA6 is at another option for hadronization within WHIZARD.

⟨*Hadrons: public*⟩+≡

```
  public :: hadrons_pythia6_t
```

⟨*Hadrons: types*⟩+≡

```
  type, extends (hadrons_t) :: hadrons_pythia6_t
   contains
   ⟨Hadrons: hadrons pythia6: TBP⟩
  end type hadrons_pythia6_t
```

⟨*Hadrons: hadrons pythia6: TBP*⟩≡

```
  procedure :: init => hadrons_pythia6_init
```

⟨*Hadrons: procedures*⟩+≡

```
  subroutine hadrons_pythia6_init &
       (hadrons, shower_settings, hadron_settings, model_hadrons)
    class(hadrons_pythia6_t), intent(out) :: hadrons
    type(shower_settings_t), intent(in) :: shower_settings
    type(hadron_settings_t), intent(in) :: hadron_settings
```

```
    type(model_t), intent(in), target :: model_hadrons
    logical :: pygive_not_set_by_shower
    hadrons%model => model_hadrons
    hadrons%shower_settings = shower_settings
    hadrons%hadron_settings = hadron_settings
    pygive_not_set_by_shower = .not. (shower_settings%method == PS_PYTHIA6 &
         .and. (shower_settings%isr_active .or. shower_settings%fsr_active))
    if (pygive_not_set_by_shower) then
       call pythia6_set_verbose (shower_settings%verbose)
       call pythia6_set_config (shower_settings%pythia6_pygive)
    end if
    call msg_message &
         ("Hadronization: Using PYTHIA6 interface for hadronization and decays")
  end subroutine hadrons_pythia6_init
```

Assume that the event record is still in the PYTHIA COMMON BLOCKS transferred there by the WHIZARD or PYTHIA6 shower routines.

⟨*Hadrons: hadrons pythia6: TBP*⟩+≡

```
  procedure :: hadronize => hadrons_pythia6_hadronize
```

⟨*Hadrons: procedures*⟩+≡

```
  subroutine hadrons_pythia6_hadronize (hadrons, particle_set, valid)
    class(hadrons_pythia6_t), intent(inout) :: hadrons
    type(particle_set_t), intent(in) :: particle_set
    logical, intent(out) :: valid
    integer :: N, NPAD, K
    real(double) :: P, V
    common /PYJETS/ N, NPAD, K(4000,5), P(4000,5), V(4000,5)
    save /PYJETS/
    if (signal_is_pending ()) return
    call msg_debug (D_TRANSFORMS, "hadrons_pythia6_hadronize")
    call pygive ("MSTP(111)=1")    !!! Switch on hadronization and decays
    call pygive ("MSTJ(1)=1")      !!! String fragmentation
    call pygive ("MSTJ(21)=2")     !!! String fragmentation keeping resonance momentum
    call pygive ("MSTJ(28)=0")     !!! Switch off tau decays
    if (debug_active (D_TRANSFORMS)) then
       call msg_debug (D_TRANSFORMS, "N", N)
       call pylist(2)
       print *, ' line 7 : ', k(7,1:5), p(7,1:5)
    end if
    call pyedit (12)
    call pythia6_set_last_treated_line (N)
    call pyexec ()
    call pyedit (12)
    valid = .true.
  end subroutine hadrons_pythia6_hadronize
```

⟨*Hadrons: hadrons pythia6: TBP*⟩+≡

```
  procedure :: make_particle_set => hadrons_pythia6_make_particle_set
```

⟨*Hadrons: procedures*⟩+≡

```
  subroutine hadrons_pythia6_make_particle_set &
         (hadrons, particle_set, model, valid)
    class(hadrons_pythia6_t), intent(in) :: hadrons
```

```
    type(particle_set_t), intent(inout) :: particle_set
    class(model_data_t), intent(in), target :: model
    logical, intent(out) :: valid
    if (signal_is_pending ()) return
    valid = pythia6_handle_errors ()
    if (valid) then
       call pythia6_combine_with_particle_set &
            (particle_set, model, hadrons%model, hadrons%shower_settings)
    end if
  end subroutine hadrons_pythia6_make_particle_set
```

32.2.6 Pythia8 Hadronization Type

⟨*Hadrons: public*⟩+≡

```
  public :: hadrons_pythia8_t
```

⟨*Hadrons: types*⟩+≡

```
  type,extends (hadrons_t) :: hadrons_pythia8_t
  contains
   ⟨Hadrons: hadrons pythia8: TBP⟩
  end type hadrons_pythia8_t
```

⟨*Hadrons: hadrons pythia8: TBP*⟩≡

```
  procedure :: init => hadrons_pythia8_init
```

⟨*Hadrons: procedures*⟩+≡

```
  subroutine hadrons_pythia8_init &
       (hadrons, shower_settings, hadron_settings, model_hadrons)
    class(hadrons_pythia8_t), intent(out) :: hadrons
    type(shower_settings_t), intent(in) :: shower_settings
    type(hadron_settings_t), intent(in) :: hadron_settings
    type(model_t), intent(in), target :: model_hadrons
    logical :: options_not_set_by_shower
    hadrons%shower_settings = shower_settings
    hadrons%hadron_settings = hadron_settings
    options_not_set_by_shower = .not. (shower_settings%method == PS_PYTHIA8 &
         .and. (shower_settings%isr_active .or. shower_settings%fsr_active))
    if (options_not_set_by_shower) then
       !call pythia8_set_verbose (settings%verbose)
       !call pythia8_set_config (settings%pythia8_config)
       !call pythia8_set_config_file (settings%pythia8_config_file)
    end if
    call msg_message &
         ("Using Pythia8 interface for hadronization and decays")
  end subroutine hadrons_pythia8_init
```

⟨*Hadrons: hadrons pythia8: TBP*⟩+≡

```
  procedure :: hadronize => hadrons_pythia8_hadronize
```

⟨*Hadrons: procedures*⟩+≡

```
  subroutine hadrons_pythia8_hadronize (hadrons, particle_set, valid)
    class(hadrons_pythia8_t), intent(inout) :: hadrons
    type(particle_set_t), intent(in) :: particle_set
```

```
    logical, intent(out) :: valid
    ! call pythia8_hadronize
    valid = .true.
  end subroutine hadrons_pythia8_hadronize
```

⟨*Hadrons: hadrons pythia8: TBP*⟩+≡

```
  procedure :: make_particle_set => hadrons_pythia8_make_particle_set
```

⟨*Hadrons: procedures*⟩+≡

```
  subroutine hadrons_pythia8_make_particle_set &
         (hadrons, particle_set, model, valid)
    class(hadrons_pythia8_t), intent(in) :: hadrons
    type(particle_set_t), intent(inout) :: particle_set
    class(model_data_t), intent(in), target :: model
    logical, intent(out) :: valid
    ! call pythia8_combine_particle_set
    valid = .true.
  end subroutine hadrons_pythia8_make_particle_set
```

32.2.7 Hadronization Event Transform

This is the type for the hadronization event transform. It does not depend on the specific hadronization implementation of `hadrons_t`.

⟨*Hadrons: public*⟩+≡

```
  public :: evt_hadrons_t
```

⟨*Hadrons: types*⟩+≡

```
  type, extends (evt_t) :: evt_hadrons_t
     class(hadrons_t), allocatable :: hadrons
     type(model_t), pointer :: model_hadrons => null()
     type(qcd_t), pointer :: qcd_t => null()
     logical :: is_first_event
   contains
   ⟨Hadrons: evt hadrons: TBP⟩
  end type evt_hadrons_t
```

Initialize the parameters. The `model_hadrons` is supposed to be the SM variant that contains all hadrons that may be generated in the shower.

⟨*Hadrons: evt hadrons: TBP*⟩≡

```
  procedure :: init => evt_hadrons_init
```

⟨*Hadrons: procedures*⟩+≡

```
  subroutine evt_hadrons_init (evt, model_hadrons)
    class(evt_hadrons_t), intent(out) :: evt
    type(model_t), intent(in), target :: model_hadrons
    evt%model_hadrons => model_hadrons
    evt%is_first_event = .true.
  end subroutine evt_hadrons_init
```

⟨*Hadrons: evt hadrons: TBP*⟩+≡

```
  procedure :: write_name => evt_hadrons_write_name
```

⟨*Hadrons: procedures*⟩+≡

```
subroutine evt_hadrons_write_name (evt, unit)
  class(evt_hadrons_t), intent(in) :: evt
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Event transform: hadronization"
end subroutine evt_hadrons_write_name
```

Output.

⟨*Hadrons: evt hadrons: TBP*⟩+≡

```
procedure :: write => evt_hadrons_write
```

⟨*Hadrons: procedures*⟩+≡

```
subroutine evt_hadrons_write (evt, unit, verbose, more_verbose, testflag)
  class(evt_hadrons_t), intent(in) :: evt
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose, more_verbose, testflag
  integer :: u
  u = given_output_unit (unit)
  call write_separator (u, 2)
  call evt%write_name (u)
  call write_separator (u)
  call evt%base_write (u, testflag = testflag, show_set = .false.)
  if (evt%particle_set_exists)  &
       call evt%particle_set%write &
       (u, summary = .true., compressed = .true., testflag = testflag)
  call write_separator (u)
  call evt%hadrons%shower_settings%write (u)
  call write_separator (u)
  call evt%hadrons%hadron_settings%write (u)
end subroutine evt_hadrons_write
```

⟨*Hadrons: evt hadrons: TBP*⟩+≡

```
procedure :: first_event => evt_hadrons_first_event
```

⟨*Hadrons: procedures*⟩+≡

```
subroutine evt_hadrons_first_event (evt)
  class(evt_hadrons_t), intent(inout) :: evt
  call msg_debug (D_TRANSFORMS, "evt_hadrons_first_event")
  associate (settings => evt%hadrons%shower_settings)
     settings%hadron_collision = .false.
     !!! !!! !!! Workaround for PGF90 16.1
     !!! if (all (evt%particle_set%prt(1:2)%flv%get_pdg_abs () <= 39)) then
     if (evt%particle_set%prt(1)%flv%get_pdg_abs () <= 39 .and. &
         evt%particle_set%prt(2)%flv%get_pdg_abs () <= 39) then
        settings%hadron_collision = .false.
     !!! else if (all (evt%particle_set%prt(1:2)%flv%get_pdg_abs () >= 100)) then
     else if (evt%particle_set%prt(1)%flv%get_pdg_abs () >= 100 .and. &
              evt%particle_set%prt(2)%flv%get_pdg_abs () >= 100) then
        settings%hadron_collision = .true.
     else
        call msg_fatal ("evt_hadrons didn't recognize beams setup")
     end if
```

```
      call msg_debug (D_TRANSFORMS, "hadron_collision", settings%hadron_collision)
      if (.not. (settings%isr_active .or. settings%fsr_active)) then
         call msg_fatal ("Hadronization without shower is not supported")
      end if
    end associate
    evt%is_first_event = .false.
  end subroutine evt_hadrons_first_event
```

Here we take the particle set from the previous event transform and apply the hadronization. The result is stored in the `evt%hadrons` object. We always return a probability of unity as we don't have the analytic weight of the hadronization. Invalid events have to be discarded by the caller which is why we mark the particle set as invalid.

⟨*Hadrons: evt hadrons: TBP*⟩+≡

```
  procedure :: generate_weighted => evt_hadrons_generate_weighted
```

⟨*Hadrons: procedures*⟩+≡

```
  subroutine evt_hadrons_generate_weighted (evt, probability)
    class(evt_hadrons_t), intent(inout) :: evt
    real(default), intent(inout) :: probability
    logical :: valid
    if (signal_is_pending ())  return
    evt%particle_set = evt%previous%particle_set
    if (evt%is_first_event) then
       call evt%first_event ()
    end if
    call evt%hadrons%hadronize (evt%particle_set, valid)
    probability = 1
    evt%particle_set_exists = valid
  end subroutine evt_hadrons_generate_weighted
```

The factorization parameters are irrelevant.

⟨*Hadrons: evt hadrons: TBP*⟩+≡

```
  procedure :: make_particle_set => evt_hadrons_make_particle_set
```

⟨*Hadrons: procedures*⟩+≡

```
  subroutine evt_hadrons_make_particle_set &
      (evt, factorization_mode, keep_correlations, r)
    class(evt_hadrons_t), intent(inout) :: evt
    integer, intent(in) :: factorization_mode
    logical, intent(in) :: keep_correlations
    real(default), dimension(:), intent(in), optional :: r
    logical :: valid
    call evt%hadrons%make_particle_set (evt%particle_set, evt%model, valid)
    evt%particle_set_exists = evt%particle_set_exists .and. valid
  end subroutine evt_hadrons_make_particle_set
```

Create RNG instances, spawned by the process object.

⟨*Hadrons: evt hadrons: TBP*⟩+≡

```
  procedure :: make_rng => evt_hadrons_make_rng
```

⟨*Hadrons: procedures*⟩+≡
```
    subroutine evt_hadrons_make_rng (evt, process)
      class(evt_hadrons_t), intent(inout) :: evt
      type(process_t), intent(inout) :: process
      class(rng_t), allocatable :: rng
      call process%make_rng (rng)
      call evt%hadrons%import_rng (rng)
    end subroutine evt_hadrons_make_rng
```

⟨*Hadrons: evt hadrons: TBP*⟩+≡
```
    procedure :: prepare_new_event => evt_hadrons_prepare_new_event
```

⟨*Hadrons: procedures*⟩+≡
```
    subroutine evt_hadrons_prepare_new_event (evt, i_mci, i_term)
      class(evt_hadrons_t), intent(inout) :: evt
      integer, intent(in) :: i_mci, i_term
      call evt%reset ()
    end subroutine evt_hadrons_prepare_new_event
```

32.3 Tau decays

⟨tau_decays.f90⟩≡
```
  ⟨File header⟩

  module tau_decays

  ⟨Use kinds⟩
    use io_units
    use format_utils, only: write_separator
    use sm_qcd
    use model_data
    use models
    use event_transforms

  ⟨Standard module head⟩

  ⟨Tau decays: public⟩

  ⟨Tau decays: types⟩

  contains

  ⟨Tau decays: procedures⟩

  end module tau_decays
```

32.3.1 Tau Decays Event Transform

This is the type for the tau decay event transform.

⟨*Tau decays: public*⟩≡
```
    public :: evt_tau_decays_t
```

⟨*Tau decays: types*⟩≡

```
type, extends (evt_t) :: evt_tau_decays_t
   type(model_t), pointer :: model_hadrons => null()
   type(qcd_t), pointer :: qcd_t => null()
 contains
 ⟨Tau decays: evt tau decays: TBP⟩
end type evt_tau_decays_t
```

⟨*Tau decays: evt tau decays: TBP*⟩≡

```
procedure :: write_name => evt_tau_decays_write_name
```

⟨*Tau decays: procedures*⟩≡

```
subroutine evt_tau_decays_write_name (evt, unit)
  class(evt_tau_decays_t), intent(in) :: evt
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Event transform: tau decays"
end subroutine evt_tau_decays_write_name
```

Output.

⟨*Tau decays: evt tau decays: TBP*⟩+≡

```
procedure :: write => evt_tau_decays_write
```

⟨*Tau decays: procedures*⟩+≡

```
subroutine evt_tau_decays_write (evt, unit, verbose, more_verbose, testflag)
  class(evt_tau_decays_t), intent(in) :: evt
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose, more_verbose, testflag
  integer :: u
  u = given_output_unit (unit)
  call write_separator (u, 2)
  call evt%write_name (u)
  call write_separator (u)
  call evt%base_write (u, testflag = testflag, show_set = .false.)
  if (evt%particle_set_exists)  &
       call evt%particle_set%write &
       (u, summary = .true., compressed = .true., testflag = testflag)
  call write_separator (u)
end subroutine evt_tau_decays_write
```

Here we take the particle set from the previous event transform and apply the tau decays. What probability should be given back, the product of branching ratios of the corresponding tau decays?

⟨*Tau decays: evt tau decays: TBP*⟩+≡

```
procedure :: generate_weighted => evt_tau_decays_generate_weighted
```

⟨*Tau decays: procedures*⟩+≡

```
subroutine evt_tau_decays_generate_weighted (evt, probability)
  class(evt_tau_decays_t), intent(inout) :: evt
  real(default), intent(inout) :: probability
  logical :: valid
  evt%particle_set = evt%previous%particle_set
  !!! To be checked or expanded
```

```
    probability = 1
    valid = .true.
    evt%particle_set_exists = valid
  end subroutine evt_tau_decays_generate_weighted
```

The factorization parameters are irrelevant.

⟨*Tau decays: evt tau decays: TBP*⟩+≡

```
  procedure :: make_particle_set => evt_tau_decays_make_particle_set
```

⟨*Tau decays: procedures*⟩+≡

```
  subroutine evt_tau_decays_make_particle_set &
      (evt, factorization_mode, keep_correlations, r)
    class(evt_tau_decays_t), intent(inout) :: evt
    integer, intent(in) :: factorization_mode
    logical, intent(in) :: keep_correlations
    real(default), dimension(:), intent(in), optional :: r
    logical :: valid
    !!! to be checked and expanded
    valid = .true.
    evt%particle_set_exists = evt%particle_set_exists .and. valid
  end subroutine evt_tau_decays_make_particle_set
```

⟨*Tau decays: evt tau decays: TBP*⟩+≡

```
  procedure :: prepare_new_event => evt_tau_decays_prepare_new_event
```

⟨*Tau decays: procedures*⟩+≡

```
  subroutine evt_tau_decays_prepare_new_event (evt, i_mci, i_term)
    class(evt_tau_decays_t), intent(inout) :: evt
    integer, intent(in) :: i_mci, i_term
    call evt%reset ()
  end subroutine evt_tau_decays_prepare_new_event
```

32.4 Decays

⟨decays.f90⟩≡

```
  ⟨File header⟩

  module decays

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use io_units
    use format_utils, only: write_indent, write_separator
    use format_defs, only: FMT_15
    use numeric_utils
    use diagnostics
    use flavors
    use helicities
    use quantum_numbers
    use interactions
    use evaluators
```

```
  use model_data
  use rng_base
  use selectors
  use parton_states
  use process, only: process_t
  use instances, only: process_instance_t, pacify
  use process_stacks
  use event_transforms
```

⟨*Standard module head*⟩

⟨*Decays: public*⟩

⟨*Decays: types*⟩

⟨*Decays: interfaces*⟩

```
contains
```

⟨*Decays: procedures*⟩

```
end module decays
```

32.4.1 Final-State Particle Configuration

A final-state particle may be either stable or unstable. Here is an empty abstract type as the parent of both, with holds just the flavor information.

⟨*Decays: types*⟩≡

```
type, abstract :: any_config_t
   private
 contains
 ⟨Decays: any config: TBP⟩
end type any_config_t
```

Finalizer, depends on the implementation.

⟨*Decays: any config: TBP*⟩≡

```
procedure (any_config_final), deferred :: final
```

⟨*Decays: interfaces*⟩≡

```
interface
   subroutine any_config_final (object)
     import
     class(any_config_t), intent(inout) :: object
   end subroutine any_config_final
end interface
```

The output is also deferred:

⟨*Decays: any config: TBP*⟩+≡

```
procedure (any_config_write), deferred :: write
```

⟨*Decays: interfaces*⟩+≡

```
interface
   subroutine any_config_write (object, unit, indent, verbose)
     import
     class(any_config_t), intent(in) :: object
     integer, intent(in), optional :: unit, indent
     logical, intent(in), optional :: verbose
   end subroutine any_config_write
end interface
```

This is a container for a stable or unstable particle configurator. We need this wrapper for preparing arrays that mix stable and unstable particles.

⟨*Decays: types*⟩+≡

```
type :: particle_config_t
   private
   class(any_config_t), allocatable :: c
end type particle_config_t
```

32.4.2 Final-State Particle

In theory, for the particle instance we only need to consider the unstable case. However, it is more straightforward to treat configuration and instance on the same footing, and to introduce a wrapper for particle objects as above. This also works around a compiler bug in gfortran.

⟨*Decays: types*⟩+≡

```
type, abstract :: any_t
   private
 contains
 ⟨Decays: any: TBP⟩
end type any_t
```

Finalizer, depends on the implementation.

⟨*Decays: any: TBP*⟩≡

```
procedure (any_final), deferred :: final
```

⟨*Decays: interfaces*⟩+≡

```
interface
   subroutine any_final (object)
     import
     class(any_t), intent(inout) :: object
   end subroutine any_final
end interface
```

The output is also deferred:

⟨*Decays: any: TBP*⟩+≡

```
procedure (any_write), deferred :: write
```

⟨*Decays: interfaces*⟩+≡

```
interface
   subroutine any_write (object, unit, indent)
     import
```

```
      class(any_t), intent(in) :: object
      integer, intent(in), optional :: unit, indent
   end subroutine any_write
end interface
```

This is a container for a stable or unstable outgoing particle. We need this wrapper for preparing arrays that mix stable and unstable particles.

⟨*Decays: types*⟩+≡

```
type :: particle_out_t
   private
   class(any_t), allocatable :: c
end type particle_out_t
```

32.4.3 Decay Term Configuration

A decay term is a distinct final state, corresponding to a process term. Each decay process may give rise to several terms with, possibly, differing flavor content.

⟨*Decays: types*⟩+≡

```
type :: decay_term_config_t
   private
   type(particle_config_t), dimension(:), allocatable :: prt
 contains
 ⟨Decays: decay term config: TBP⟩
end type decay_term_config_t
```

Finalizer, recursive.

⟨*Decays: decay term config: TBP*⟩≡

```
procedure :: final => decay_term_config_final
```

⟨*Decays: procedures*⟩≡

```
recursive subroutine decay_term_config_final (object)
  class(decay_term_config_t), intent(inout) :: object
  integer :: i
  if (allocated (object%prt)) then
     do i = 1, size (object%prt)
        if (allocated (object%prt(i)%c))  call object%prt(i)%c%final ()
     end do
  end if
end subroutine decay_term_config_final
```

Output, with optional indentation

⟨*Decays: decay term config: TBP*⟩+≡

```
procedure :: write => decay_term_config_write
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_term_config_write (object, unit, indent, verbose)
  class(decay_term_config_t), intent(in) :: object
  integer, intent(in), optional :: unit, indent
  logical, intent(in), optional :: verbose
  integer :: i, j, u, ind
```

```
    logical :: verb
    u = given_output_unit (unit)
    ind = 0;  if (present (indent))  ind = indent
    verb = .true.;  if (present (verbose))  verb = verbose
    call write_indent (u, ind)
    write (u, "(1x,A)", advance="no")  "Final state:"
    do i = 1, size (object%prt)
       select type (prt_config => object%prt(i)%c)
       type is (stable_config_t)
          write (u, "(1x,A)", advance="no") &
               char (prt_config%flv(1)%get_name ())
          do j = 2, size (prt_config%flv)
             write (u, "(':',A)", advance="no") &
                  char (prt_config%flv(j)%get_name ())
          end do
       type is (unstable_config_t)
          write (u, "(1x,A)", advance="no") &
               char (prt_config%flv%get_name ())
       end select
    end do
    write (u, *)
    if (verb) then
       do i = 1, size (object%prt)
          call object%prt(i)%c%write (u, ind)
       end do
    end if
  end subroutine decay_term_config_write
```

Initialize, given a set of flavors. For each flavor, we must indicate whether the particle is stable. The second index of the flavor array runs over alternatives for each decay product; alternatives are allowed only if the decay product is itself stable.

⟨*Decays: decay term config: TBP*⟩+≡

```
  procedure :: init => decay_term_config_init
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine decay_term_config_init &
       (term, flv, stable, model, process_stack)
    class(decay_term_config_t), intent(out) :: term
    type(flavor_t), dimension(:,:), intent(in) :: flv
    logical, dimension(:), intent(in) :: stable
    class(model_data_t), intent(in), target :: model
    type(process_stack_t), intent(in), optional :: process_stack
    type(string_t), dimension(:), allocatable :: decay
    integer :: i
    allocate (term%prt (size (flv, 1)))
    do i = 1, size (flv, 1)
       associate (prt => term%prt(i))
         if (stable(i)) then
            allocate (stable_config_t :: prt%c)
         else
            allocate (unstable_config_t :: prt%c)
         end if
         select type (prt_config => prt%c)
```

```
         type is (stable_config_t)
            call prt_config%init (flv(i,:))
         type is (unstable_config_t)
            if (all (flv(i,:) == flv(i,1))) then
               call prt_config%init (flv(i,1))
               call flv(i,1)%get_decays (decay)
               call prt_config%init_decays (decay, model, process_stack)
            else
               call prt_config%write ()
               call msg_fatal ("Decay configuration: &
                    &unstable product must be unique")
            end if
         end select
      end associate
   end do
end subroutine decay_term_config_init
```

Recursively compute widths and branching ratios for all unstable particles.

⟨*Decays: decay term config: TBP*⟩+≡

```
procedure :: compute => decay_term_config_compute
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_term_config_compute (term)
  class(decay_term_config_t), intent(inout) :: term
  integer :: i
  do i = 1, size (term%prt)
     select type (unstable_config => term%prt(i)%c)
     type is (unstable_config_t)
        call unstable_config%compute ()
     end select
  end do
end subroutine decay_term_config_compute
```

32.4.4 Decay Term

A decay term instance is selected when we generate an event for the associated process instance. When evaluated, it triggers further decays down the chain.

Only unstable products are allocated as child particles.

⟨*Decays: types*⟩+≡

```
type :: decay_term_t
   private
   type(decay_term_config_t), pointer :: config => null ()
   type(particle_out_t), dimension(:), allocatable :: particle_out
 contains
 ⟨Decays: decay term: TBP⟩
end type decay_term_t
```

Finalizer.

⟨*Decays: decay term: TBP*⟩≡

```
procedure :: final => decay_term_final
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_term_final (object)
  class(decay_term_t), intent(inout) :: object
  integer :: i
  if (allocated (object%particle_out)) then
     do i = 1, size (object%particle_out)
        call object%particle_out(i)%c%final ()
     end do
  end if
end subroutine decay_term_final
```

Output.

⟨*Decays: decay term: TBP*⟩+≡

```
procedure :: write => decay_term_write
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_term_write (object, unit, indent)
  class(decay_term_t), intent(in) :: object
  integer, intent(in), optional :: unit, indent
  integer :: i, u, ind
  u = given_output_unit (unit)
  ind = 0;  if (present (indent))  ind = indent
  call object%config%write (u, ind, verbose = .false.)
  do i = 1, size (object%particle_out)
     call object%particle_out(i)%c%write (u, ind)
  end do
end subroutine decay_term_write
```

Recursively write the embedded process instances.

⟨*Decays: decay term: TBP*⟩+≡

```
procedure :: write_process_instances => decay_term_write_process_instances
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_term_write_process_instances (term, unit, verbose)
  class(decay_term_t), intent(in) :: term
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer :: i
  do i = 1, size (term%particle_out)
     select type (unstable => term%particle_out(i)%c)
     type is (unstable_t)
        call unstable%write_process_instances (unit, verbose)
     end select
  end do
end subroutine decay_term_write_process_instances
```

Initialization, using the configuration object. We allocate particle objects in parallel to the particle configuration objects which we use to initialize them, one at a time.

⟨*Decays: decay term: TBP*⟩+≡

```
procedure :: init => decay_term_init
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_term_init (term, config)
  class(decay_term_t), intent(out) :: term
  type(decay_term_config_t), intent(in), target :: config
  integer :: i
  term%config => config
  allocate (term%particle_out (size (config%prt)))
  do i = 1, size (config%prt)
     select type (prt_config => config%prt(i)%c)
     type is (stable_config_t)
        allocate (stable_t :: term%particle_out(i)%c)
        select type (stable => term%particle_out(i)%c)
        type is (stable_t)
           call stable%init (prt_config)
        end select
     type is (unstable_config_t)
        allocate (unstable_t :: term%particle_out(i)%c)
        select type (unstable => term%particle_out(i)%c)
        type is (unstable_t)
           call unstable%init (prt_config)
        end select
     end select
  end do
end subroutine decay_term_init
```

Implement a RNG instance, spawned by the process object.

⟨*Decays: decay term: TBP*⟩+≡

```
procedure :: make_rng => decay_term_make_rng
```

⟨*Decays: procedures*⟩+≡

```
subroutine decay_term_make_rng (term, process)
  class(decay_term_t), intent(inout) :: term
  type(process_t), intent(inout) :: process
  class(rng_t), allocatable :: rng
  integer :: i
  do i = 1, size (term%particle_out)
     select type (unstable => term%particle_out(i)%c)
     type is (unstable_t)
        call process%make_rng (rng)
        call unstable%import_rng (rng)
     end select
  end do
end subroutine decay_term_make_rng
```

Link the interactions for unstable decay products to the interaction of the parent process.

⟨*Decays: decay term: TBP*⟩+≡

```
procedure :: link_interactions => decay_term_link_interactions
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_term_link_interactions (term, trace)
  class(decay_term_t), intent(inout) :: term
  type(interaction_t), intent(in), target :: trace
  integer :: i
```

```
    do i = 1, size (term%particle_out)
       select type (unstable => term%particle_out(i)%c)
       type is (unstable_t)
          call unstable%link_interactions (i, trace)
       end select
    end do
  end subroutine decay_term_link_interactions
```

Recursively generate a decay chain, for each of the unstable particles in the final state.

⟨*Decays: decay term: TBP*⟩+≡

```
  procedure :: select_chain => decay_term_select_chain
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine decay_term_select_chain (term)
    class(decay_term_t), intent(inout) :: term
    integer :: i
    do i = 1, size (term%particle_out)
       select type (unstable => term%particle_out(i)%c)
       type is (unstable_t)
          call unstable%select_chain ()
       end select
    end do
  end subroutine decay_term_select_chain
```

Recursively generate a decay event, for each of the unstable particles in the final state.

⟨*Decays: decay term: TBP*⟩+≡

```
  procedure :: generate => decay_term_generate
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine decay_term_generate (term)
    class(decay_term_t), intent(inout) :: term
    integer :: i
    do i = 1, size (term%particle_out)
       select type (unstable => term%particle_out(i)%c)
       type is (unstable_t)
          call unstable%generate ()
       end select
    end do
  end subroutine decay_term_generate
```

32.4.5 Decay Root Configuration

At the root of a decay chain, there is a parent process. The decay root stores a pointer to the parent process and the set of decay configurations.

⟨*Decays: public*⟩≡

```
  public :: decay_root_config_t
```

⟨*Decays: types*⟩+≡

```
type :: decay_root_config_t
   private
   type(string_t) :: process_id
   type(process_t), pointer :: process => null ()
   class(model_data_t), pointer :: model => null ()
   type(decay_term_config_t), dimension(:), allocatable :: term_config
 contains
 ⟨Decays: decay root config: TBP⟩
end type decay_root_config_t
```

The finalizer is recursive since there may be cascade decays.

⟨*Decays: decay root config: TBP*⟩≡

```
procedure :: final => decay_root_config_final
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_root_config_final (object)
  class(decay_root_config_t), intent(inout) :: object
  integer :: i
  if (allocated (object%term_config)) then
     do i = 1, size (object%term_config)
        call object%term_config(i)%final ()
     end do
  end if
end subroutine decay_root_config_final
```

The output routine is also recursive, and it contains an adjustable indentation.

⟨*Decays: decay root config: TBP*⟩+≡

```
procedure :: write => decay_root_config_write
procedure :: write_header => decay_root_config_write_header
procedure :: write_terms => decay_root_config_write_terms
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_root_config_write (object, unit, indent, verbose)
  class(decay_root_config_t), intent(in) :: object
  integer, intent(in), optional :: unit, indent
  logical, intent(in), optional :: verbose
  integer :: u, ind
  u = given_output_unit (unit)
  ind = 0;  if (present (indent))  ind = indent
  call write_indent (u, ind)
  write (u, "(1x,A)")  "Final-state decay tree:"
  call object%write_header (unit, indent)
  call object%write_terms (unit, indent, verbose)
end subroutine decay_root_config_write

subroutine decay_root_config_write_header (object, unit, indent)
  class(decay_root_config_t), intent(in) :: object
  integer, intent(in), optional :: unit, indent
  integer :: u, ind
  u = given_output_unit (unit)
  ind = 0;  if (present (indent))  ind = indent
  call write_indent (u, ind)
  if (associated (object%process)) then
```

```
      write (u, 3)  "process ID      =", char (object%process_id), "*"
   else
      write (u, 3)  "process ID      =", char (object%process_id)
   end if
3  format (3x,A,2(1x,A))
end subroutine decay_root_config_write_header

recursive subroutine decay_root_config_write_terms &
     (object, unit, indent, verbose)
   class(decay_root_config_t), intent(in) :: object
   integer, intent(in), optional :: unit, indent
   logical, intent(in), optional :: verbose
   integer :: i, u, ind
   logical :: verb
   u = given_output_unit (unit)
   ind = 0;  if (present (indent))  ind = indent
   verb = .true.;  if (present (verbose))  verb = verbose
   if (verb .and. allocated (object%term_config)) then
      do i = 1, size (object%term_config)
         call object%term_config(i)%write (u, ind + 1)
      end do
   end if
end subroutine decay_root_config_write_terms
```

Initialize for a named process and (optionally) a pre-determined number of terms.

⟨*Decays: decay root config: TBP*⟩+≡

```
procedure :: init => decay_root_config_init
```

⟨*Decays: procedures*⟩+≡

```
subroutine decay_root_config_init (decay, model, process_id, n_terms)
   class(decay_root_config_t), intent(out) :: decay
   class(model_data_t), intent(in), target :: model
   type(string_t), intent(in) :: process_id
   integer, intent(in), optional :: n_terms
   decay%model => model
   decay%process_id = process_id
   if (present (n_terms)) then
      allocate (decay%term_config (n_terms))
   end if
end subroutine decay_root_config_init
```

Declare a decay term, given an array of flavors.

⟨*Decays: decay root config: TBP*⟩+≡

```
procedure :: init_term => decay_root_config_init_term
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_root_config_init_term &
     (decay, i, flv, stable, model, process_stack)
   class(decay_root_config_t), intent(inout) :: decay
   integer, intent(in) :: i
   type(flavor_t), dimension(:,:), intent(in) :: flv
   logical, dimension(:), intent(in) :: stable
   class(model_data_t), intent(in), target :: model
```

```
    type(process_stack_t), intent(in), optional :: process_stack
    call decay%term_config(i)%init (flv, stable, model, process_stack)
  end subroutine decay_root_config_init_term
```

Connect the decay root configuration with a process object (which should represent the parent process). This includes initialization, therefore intent(out).

The flavor state is retrieved from the process term object. However, we have to be careful: the flavor object points to the model instance that is stored in the process object. This model instance may not contain the current setting for unstable particles and decay. Therefore, we assign the model directly.

If the `process_instance` argument is provided, we use this for the flavor state. This applies to the decay root only, where the process can be entangled with a beam setup, and the latter contains beam remnants as further outgoing particles. These must be included in the set of outgoing flavors, since the decay application is also done on the connected state.

Infer stability from the particle properties, using the first row in the set of flavor states. For unstable particles, we look for decays, recursively, available from the process stack (if present).

For the unstable particles, we have to check whether their masses match between the production and the decay. Fortunately, both versions are available for comparison.

⟨*Decays: decay root config: TBP*⟩+≡

```
  procedure :: connect => decay_root_config_connect
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine decay_root_config_connect &
      (decay, process, model, process_stack, process_instance)
    class(decay_root_config_t), intent(out) :: decay
    type(process_t), intent(in), target :: process
    class(model_data_t), intent(in), target :: model
    type(process_stack_t), intent(in), optional :: process_stack
    type(process_instance_t), intent(in), optional, target :: process_instance
    type(connected_state_t), pointer :: connected_state
    type(interaction_t), pointer :: int
    type(flavor_t), dimension(:,:), allocatable :: flv
    logical, dimension(:), allocatable :: stable
    real(default), dimension(:), allocatable :: m_prod, m_dec
    !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
    integer :: i, j
    call decay%init (model, process%get_id (), process%get_n_terms ())
    do i = 1, size (decay%term_config)
       if (present (process_instance)) then
          connected_state => process_instance%get_connected_state_ptr (i)
          int => connected_state%get_matrix_int_ptr ()
          call interaction_get_flv_out (int, flv)
       else
          call process%get_term_flv_out (i, flv)
       end if
       allocate (m_prod (size (flv(:,1)%get_mass ())))
       !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
       do j = 1, size (flv(:,1)%get_mass ())
          m_prod(j) = flv(j,1)%get_mass ()
       end do
```

```
      !!! m_prod = flv(:,1)%get_mass ()
      call flv%set_model (model)
      allocate (m_dec (size (flv(:,1)%get_mass ())))
      !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
      do j = 1, size (flv(:,1)%get_mass ())
         m_dec(j) = flv(j,1)%get_mass ()
      end do
      !!! m_dec = flv(:,1)%get_mass ()
      allocate (stable (size (flv, 1)))
      !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
      do j = 1, size (flv, 1)
         stable(j) = flv(j,1)%is_stable ()
      end do
      !!! stable = flv(:,1)%is_stable ()
      call check_masses ()
      call decay%init_term (i, flv, stable, model, process_stack)
      deallocate (flv, stable, m_prod, m_dec)
   end do
   decay%process => process
contains
   subroutine check_masses ()
     integer :: i
     logical :: ok
     ok = .true.
     do i = 1, size (m_prod)
        if (.not. stable(i)) then
           if (.not. nearly_equal (m_prod(i), m_dec(i))) then
              write (msg_buffer, "(A,A,A)"  "particle '", &
                   char (flv(i,1)%get_name ()), "':"
              call msg_message
              write (msg_buffer, &
                   "(2x,A,1x," // FMT_15 // ",3x,A,1x," // FMT_15 // ")") &
                   "m_prod =", m_prod(i), "m_dec =", m_dec(i)
              call msg_message
              ok = .false.
           end if
        end if
     end do
     if (.not. ok)  call msg_fatal &
          ("Particle mass mismatch between production and decay")
   end subroutine check_masses
end subroutine decay_root_config_connect
```

Recursively compute widths, errors, and branching ratios.

⟨*Decays: decay root config: TBP*⟩+≡

```
procedure :: compute => decay_root_config_compute
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_root_config_compute (decay)
  class(decay_root_config_t), intent(inout) :: decay
  integer :: i
  do i = 1, size (decay%term_config)
     call decay%term_config(i)%compute ()
  end do
```

```
end subroutine decay_root_config_compute
```

32.4.6 Decay Root Instance

This is the common parent type for decay and decay root. The process instance points to the parent process. The model pointer is separate because particle settings may be updated w.r.t. the parent process object.

⟨*Decays: types*⟩+≡

```
type, abstract :: decay_gen_t
   private
   type(decay_term_t), dimension(:), allocatable :: term
   type(process_instance_t), pointer :: process_instance => null ()
   integer :: selected_mci = 0
   integer :: selected_term = 0
 contains
 ⟨Decays: decay gen: TBP⟩
end type decay_gen_t
```

The decay root represents the parent process. When an event is generated, the generator selects the term to which the decay chain applies (if possible).

The process instance is just a pointer.

⟨*Decays: public*⟩+≡

```
public :: decay_root_t
```

⟨*Decays: types*⟩+≡

```
type, extends (decay_gen_t) :: decay_root_t
   private
   type(decay_root_config_t), pointer :: config => null ()
 contains
 ⟨Decays: decay root: TBP⟩
end type decay_root_t
```

The finalizer has to recursively finalize the terms, but we can skip the process instance which is not explicitly allocated.

⟨*Decays: decay gen: TBP*⟩≡

```
procedure :: base_final => decay_gen_final
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_gen_final (object)
  class(decay_gen_t), intent(inout) :: object
  integer :: i
  if (allocated (object%term)) then
     do i = 1, size (object%term)
        call object%term(i)%final ()
     end do
  end if
end subroutine decay_gen_final
```

No extra finalization for the decay root.

⟨*Decays: decay root: TBP*⟩≡

```
procedure :: final => decay_root_final
```

⟨*Decays: procedures*⟩+≡

```
subroutine decay_root_final (object)
  class(decay_root_t), intent(inout) :: object
  call object%base_final ()
end subroutine decay_root_final
```

Output.

⟨*Decays: decay root: TBP*⟩+≡

```
procedure :: write => decay_root_write
```

⟨*Decays: procedures*⟩+≡

```
subroutine decay_root_write (object, unit)
  class(decay_root_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  if (associated (object%config)) then
     call object%config%write (unit, verbose = .false.)
  else
     write (u, "(1x,A)")  "Final-state decay tree: [not configured]"
  end if
  if (object%selected_mci > 0) then
     write (u, "(3x,A,I0)")  "Selected MCI    = ", object%selected_mci
  else
     write (u, "(3x,A)")  "Selected MCI    = [undefined]"
  end if
  if (object%selected_term > 0) then
     write (u, "(3x,A,I0)")  "Selected term   = ", object%selected_term
     call object%term(object%selected_term)%write (u, 1)
  else
     write (u, "(3x,A)")  "Selected term   = [undefined]"
  end if
end subroutine decay_root_write
```

Write the process instances, recursively.

⟨*Decays: decay gen: TBP*⟩+≡

```
procedure :: write_process_instances => decay_gen_write_process_instances
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_gen_write_process_instances (decay, unit, verbose)
  class(decay_gen_t), intent(in) :: decay
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  logical :: verb
  verb = .true.;  if (present (verbose))  verb = verbose
  if (associated (decay%process_instance)) then
     if (verb) then
        call decay%process_instance%write (unit)
     else
        call decay%process_instance%write_header (unit)
     end if
  end if
  if (decay%selected_term > 0) then
     call decay%term(decay%selected_term)%write_process_instances (unit, verb)
```

```
    end if
  end subroutine decay_gen_write_process_instances
```

Generic initializer. All can be done recursively.

⟨*Decays: decay gen: TBP*⟩+≡

```
  procedure :: base_init => decay_gen_init
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine decay_gen_init (decay, term_config)
    class(decay_gen_t), intent(out) :: decay
    type(decay_term_config_t), dimension(:), intent(in), target :: term_config
    integer :: i
    allocate (decay%term (size (term_config)))
    do i = 1, size (decay%term)
       call decay%term(i)%init (term_config(i))
    end do
  end subroutine decay_gen_init
```

Specific initializer. We assign the configuration object, which should correspond to a completely initialized decay configuration tree. We also connect to an existing process instance. Then, we recursively link the child interactions to the parent process.

⟨*Decays: decay root: TBP*⟩+≡

```
  procedure :: init => decay_root_init
```

⟨*Decays: procedures*⟩+≡

```
  subroutine decay_root_init (decay_root, config, process_instance)
    class(decay_root_t), intent(out) :: decay_root
    type(decay_root_config_t), intent(in), target :: config
    type(process_instance_t), intent(in), target :: process_instance
    call decay_root%base_init (config%term_config)
    decay_root%config => config
    decay_root%process_instance => process_instance
    call decay_root%make_term_rng (config%process)
    call decay_root%link_term_interactions ()
  end subroutine decay_root_init
```

Explicitly set/get mci and term indices. (Used in unit test.)

⟨*Decays: decay gen: TBP*⟩+≡

```
  procedure :: set_mci => decay_gen_set_mci
  procedure :: set_term => decay_gen_set_term
  procedure :: get_mci => decay_gen_get_mci
  procedure :: get_term => decay_gen_get_term
```

⟨*Decays: procedures*⟩+≡

```
  subroutine decay_gen_set_mci (decay, i)
    class(decay_gen_t), intent(inout) :: decay
    integer, intent(in) :: i
    decay%selected_mci = i
  end subroutine decay_gen_set_mci

  subroutine decay_gen_set_term (decay, i)
    class(decay_gen_t), intent(inout) :: decay
```

```
    integer, intent(in) :: i
    decay%selected_term = i
  end subroutine decay_gen_set_term

  function decay_gen_get_mci (decay) result (i)
    class(decay_gen_t), intent(inout) :: decay
    integer :: i
    i = decay%selected_mci
  end function decay_gen_get_mci

  function decay_gen_get_term (decay) result (i)
    class(decay_gen_t), intent(inout) :: decay
    integer :: i
    i = decay%selected_term
  end function decay_gen_get_term
```

Implement random-number generators for unstable decay selection in all terms. This is not recursive.

We also make use of the fact that `process` is a pointer; the (state of the RNG factory inside the) target process will be modified by the rng-spawning method, but not the pointer.

⟨*Decays: decay gen: TBP*⟩+≡

```
  procedure :: make_term_rng => decay_gen_make_term_rng
```

⟨*Decays: procedures*⟩+≡

```
  subroutine decay_gen_make_term_rng (decay, process)
    class(decay_gen_t), intent(inout) :: decay
    type(process_t), intent(in), pointer :: process
    integer :: i
    do i = 1, size (decay%term)
       call decay%term(i)%make_rng (process)
    end do
  end subroutine decay_gen_make_term_rng
```

Recursively link interactions of the enclosed decay terms to the corresponding terms in the current process instance.

Note: A bug in nagfor requires the extra `i_term` variable.

⟨*Decays: decay gen: TBP*⟩+≡

```
  procedure :: link_term_interactions => decay_gen_link_term_interactions
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine decay_gen_link_term_interactions (decay)
    class(decay_gen_t), intent(inout) :: decay
    integer :: i, i_term
    type(interaction_t), pointer :: trace
    associate (instance => decay%process_instance)
      do i = 1, size (decay%term)
         i_term = i
         trace => instance%get_trace_int_ptr (i_term)
         call decay%term(i_term)%link_interactions (trace)
      end do
    end associate
  end subroutine decay_gen_link_term_interactions
```

Select a decay chain: decay modes and process components.

⟨*Decays: decay root: TBP*⟩+≡
```
procedure :: select_chain => decay_root_select_chain
```

⟨*Decays: procedures*⟩+≡
```
subroutine decay_root_select_chain (decay_root)
  class(decay_root_t), intent(inout) :: decay_root
  if (decay_root%selected_term > 0) then
     call decay_root%term(decay_root%selected_term)%select_chain ()
  else
     call msg_bug ("Decays: no term selected for parent process")
  end if
end subroutine decay_root_select_chain
```

Generate a decay tree, i.e., for the selected term in the parent process, recursively generate a decay event for all unstable particles.

Factor out the trace of the connected state of the parent process. This trace should not be taken into account for unweighting the decay chain, since it was already used for unweighting the parent event, or it determines the overall event weight.

⟨*Decays: decay root: TBP*⟩+≡
```
procedure :: generate => decay_root_generate
```

⟨*Decays: procedures*⟩+≡
```
subroutine decay_root_generate (decay_root)
  class(decay_root_t), intent(inout) :: decay_root
  type(connected_state_t), pointer :: connected_state
  if (decay_root%selected_term > 0) then
     connected_state => decay_root%process_instance%get_connected_state_ptr &
          (decay_root%selected_term)
     call connected_state%normalize_matrix_by_trace ()
     call decay_root%term(decay_root%selected_term)%generate ()
  else
     call msg_bug ("Decays: no term selected for parent process")
  end if
end subroutine decay_root_generate
```

32.4.7 Decay Configuration

A decay configuration describes a distinct decay mode of a particle. Each decay mode may include several terms, which correspond to the terms in the associated process. In addition to the base type, the decay configuration object contains the integral of the parent process and the selector for the MCI group inside this process.

The flavor component should be identical to the flavor component of the parent particle (`unstable` object).

⟨*Decays: types*⟩+≡
```
type, extends (decay_root_config_t) :: decay_config_t
   private
   type(flavor_t) :: flv
   real(default) :: weight = 0
```

```
    real(default) :: integral = 0
    real(default) :: abs_error = 0
    real(default) :: rel_error = 0
    type(selector_t) :: mci_selector
  contains
  ⟨Decays: decay config: TBP⟩
end type decay_config_t
```

The output routine extends the decay-root writer by listing numerical component values.

⟨*Decays: decay config: TBP*⟩≡

```
procedure :: write => decay_config_write
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_config_write (object, unit, indent, verbose)
  class(decay_config_t), intent(in) :: object
  integer, intent(in), optional :: unit, indent
  logical, intent(in), optional :: verbose
  integer :: u, ind
  u = given_output_unit (unit)
  ind = 0;  if (present (indent))  ind = indent
  call write_indent (u, ind)
  write (u, "(1x,A)")  "Decay:"
  call object%write_header (unit, indent)
  call write_indent (u, ind)
  write (u, 2)  "branching ratio =", object%weight * 100
  call write_indent (u, ind)
  write (u, 1)  "partial width   =", object%integral
  call write_indent (u, ind)
  write (u, 1)  "error (abs)     =", object%abs_error
  call write_indent (u, ind)
  write (u, 1)  "error (rel)     =", object%rel_error
1 format (3x,A,ES19.12)
2 format (3x,A,F11.6,1x,'%')
  call object%write_terms (unit, indent, verbose)
end subroutine decay_config_write
```

Connect a decay configuration with a process object (which should represent the decay). This includes initialization, therefore intent(out). We first connect the process itself, then do initializations that are specific for this decay.

Infer stability from the particle properties, using the first row in the set of flavor states. Once we can deal with predetermined decay chains, they should be used instead.

⟨*Decays: decay config: TBP*⟩+≡

```
procedure :: connect => decay_config_connect
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_config_connect &
     (decay, process, model, process_stack, process_instance)
  class(decay_config_t), intent(out) :: decay
  type(process_t), intent(in), target :: process
  class(model_data_t), intent(in), target :: model
  type(process_stack_t), intent(in), optional :: process_stack
```

```
    type(process_instance_t), intent(in), optional, target :: process_instance
    real(default), dimension(:), allocatable :: integral_mci
    integer :: i, n_mci
    call decay%decay_root_config_t%connect (process, model, process_stack)
    decay%integral = process%get_integral ()
    decay%abs_error = process%get_error ()
    if (process%cm_frame ()) then
       call msg_fatal ("Decay process " // char (process%get_id ()) &
            // ": unusable because rest frame is fixed.")
    end if
    n_mci = process%get_n_mci ()
    allocate (integral_mci (n_mci))
    do i = 1, n_mci
       integral_mci(i) = process%get_integral_mci (i)
    end do
    call decay%mci_selector%init (integral_mci)
  end subroutine decay_config_connect
```

Set the flavor entry, which repeats the flavor of the parent unstable particle.

⟨*Decays: decay config: TBP*⟩+≡

```
  procedure :: set_flv => decay_config_set_flv
```

⟨*Decays: procedures*⟩+≡

```
  subroutine decay_config_set_flv (decay, flv)
    class(decay_config_t), intent(inout) :: decay
    type(flavor_t), intent(in) :: flv
    decay%flv = flv
  end subroutine decay_config_set_flv
```

Compute embedded branchings and the relative error. This method does not apply to the decay root.

⟨*Decays: decay config: TBP*⟩+≡

```
  procedure :: compute => decay_config_compute
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine decay_config_compute (decay)
    class(decay_config_t), intent(inout) :: decay
    call decay%decay_root_config_t%compute ()
    if (.not. vanishes (decay%integral)) then
       decay%rel_error = decay%abs_error / decay%integral
    else
       decay%rel_error = 0
    end if
  end subroutine decay_config_compute
```

32.4.8 Decay Instance

The decay contains a collection of terms. One of them is selected when the decay is evaluated. This is similar to the decay root, but we implement it independently.

The process instance object is allocated via a pointer, so it automatically behaves as a target.

⟨*Decays: types*⟩+≡

```
type, extends (decay_gen_t) :: decay_t
   private
   type(decay_config_t), pointer :: config => null ()
   class(rng_t), allocatable :: rng
 contains
 ⟨Decays: decay: TBP⟩
end type decay_t
```

The finalizer is recursive.

⟨*Decays: decay: TBP*⟩≡

```
procedure :: final => decay_final
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_final (object)
  class(decay_t), intent(inout) :: object
  integer :: i
  call object%base_final ()
  do i = 1, object%config%process%get_n_mci ()
     call object%process_instance%final_simulation (i)
  end do
  call object%process_instance%final ()
  deallocate (object%process_instance)
end subroutine decay_final
```

Output.

⟨*Decays: decay: TBP*⟩+≡

```
procedure :: write => decay_write
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine decay_write (object, unit, indent, recursive)
  class(decay_t), intent(in) :: object
  integer, intent(in), optional :: unit, indent, recursive
  integer :: u, ind
  u = given_output_unit (unit)
  ind = 0;  if (present (indent))  ind = indent
  call object%config%write (unit, indent, verbose = .false.)
  if (allocated (object%rng)) then
     call object%rng%write (u, ind + 1)
  end if
  call write_indent (u, ind)
  if (object%selected_mci > 0) then
     write (u, "(3x,A,I0)")  "Selected MCI    = ", object%selected_mci
  else
     write (u, "(3x,A)")  "Selected MCI    = [undefined]"
  end if
  call write_indent (u, ind)
  if (object%selected_term > 0) then
     write (u, "(3x,A,I0)")  "Selected term   = ", object%selected_term
     call object%term(object%selected_term)%write (u, ind + 1)
  else
     write (u, "(3x,A)")  "Selected term   = [undefined]"
```

```
    end if
  end subroutine decay_write
```

Initializer. Base initialization is done recursively. Then, we prepare the current process instance and allocate a random-number generator for term selection. For all unstable particles, we also allocate a r.n.g. as spawned by the current process.

⟨*Decays: decay: TBP*⟩+≡

```
  procedure :: init => decay_init
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine decay_init (decay, config)
    class(decay_t), intent(out) :: decay
    type(decay_config_t), intent(in), target :: config
    integer :: i
    call decay%base_init (config%term_config)
    decay%config => config
    allocate (decay%process_instance)
    call decay%process_instance%init (decay%config%process)
    call decay%process_instance%setup_event_data (decay%config%model)
    do i = 1, decay%config%process%get_n_mci ()
       call decay%process_instance%init_simulation (i)
    end do
    call decay%config%process%make_rng (decay%rng)
    call decay%make_term_rng (decay%config%process)
  end subroutine decay_init
```

Link interactions to the parent process. `i_prt` is the index of the current outgoing particle in the parent interaction, for which we take the trace evaluator. We link it to the beam particle in the beam interaction of the decay process instance. Then, repeat the procedure for the outgoing particles.

⟨*Decays: decay: TBP*⟩+≡

```
  procedure :: link_interactions => decay_link_interactions
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine decay_link_interactions (decay, i_prt, trace)
    class(decay_t), intent(inout) :: decay
    integer, intent(in) :: i_prt
    type(interaction_t), intent(in), target :: trace
    type(interaction_t), pointer :: beam_int
    integer :: n_in, n_vir
    beam_int => decay%process_instance%get_beam_int_ptr ()
    n_in = trace%get_n_in ()
    n_vir = trace%get_n_vir ()
    call beam_int%set_source_link (1, trace, &
         n_in + n_vir + i_prt)
    call decay%link_term_interactions ()
  end subroutine decay_link_interactions
```

Determine a decay chain. For each unstable particle we select one of the possible decay modes, and for each decay process we select one of the possible decay

MCI components, calling the random-number generators. We do not generate momenta, yet.

⟨*Decays: decay: TBP*⟩+≡
```
procedure :: select_chain => decay_select_chain
```

⟨*Decays: procedures*⟩+≡
```
recursive subroutine decay_select_chain (decay)
  class(decay_t), intent(inout) :: decay
  real(default) :: x
  integer :: i
  call decay%rng%generate (x)
  decay%selected_mci = decay%config%mci_selector%select (x)
  call decay%process_instance%choose_mci (decay%selected_mci)
  decay%selected_term = decay%process_instance%select_i_term ()
  do i = 1, size (decay%term)
     call decay%term(i)%select_chain ()
  end do
end subroutine decay_select_chain
```

Generate a decay. We first receive the beam momenta from the parent process (assuming that this is properly linked), then call the associated process object for a new event.

Factor out the trace of the helicity density matrix of the isolated state (the one that will be used for the decay chain). The trace is taken into account for unweighting the individual decay event and should therefore be ignored for unweighting the correlated decay chain afterwards.

⟨*Decays: decay: TBP*⟩+≡
```
procedure :: generate => decay_generate
```

⟨*Decays: procedures*⟩+≡
```
recursive subroutine decay_generate (decay)
  class(decay_t), intent(inout) :: decay
  type(isolated_state_t), pointer :: isolated_state
  integer :: i
  call decay%process_instance%receive_beam_momenta ()
  call decay%process_instance%generate_unweighted_event (decay%selected_mci)
  if (signal_is_pending ())  return
  call decay%process_instance%evaluate_event_data ()
  isolated_state => &
       decay%process_instance%get_isolated_state_ptr (decay%selected_term)
  call isolated_state%normalize_matrix_by_trace ()
  do i = 1, size (decay%term)
     call decay%term(i)%generate ()
     if (signal_is_pending ())  return
  end do
end subroutine decay_generate
```

32.4.9 Stable Particles

This is a stable particle. The flavor can be ambiguous (e.g., partons).

⟨*Decays: types*⟩+≡
```
type, extends (any_config_t) :: stable_config_t
```

```
    private
    type(flavor_t), dimension(:), allocatable :: flv
  contains
  ⟨Decays: stable config: TBP⟩
end type stable_config_t
```

The finalizer is empty:

⟨*Decays: stable config: TBP*⟩≡

```
procedure :: final => stable_config_final
```

⟨*Decays: procedures*⟩+≡

```
subroutine stable_config_final (object)
  class(stable_config_t), intent(inout) :: object
end subroutine stable_config_final
```

Output.

⟨*Decays: stable config: TBP*⟩+≡

```
procedure :: write => stable_config_write
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine stable_config_write (object, unit, indent, verbose)
  class(stable_config_t), intent(in) :: object
  integer, intent(in), optional :: unit, indent
  logical, intent(in), optional :: verbose
  integer :: u, i, ind
  u = given_output_unit (unit)
  ind = 0;  if (present (indent))  ind = indent
  call write_indent (u, ind)
  write (u, "(1x,'+',1x,A)", advance = "no")  "Stable:"
  write (u, "(1x,A)", advance = "no")  char (object%flv(1)%get_name ())
  do i = 2, size (object%flv)
     write (u, "(':',A)", advance = "no") &
          char (object%flv(i)%get_name ())
  end do
  write (u, *)
end subroutine stable_config_write
```

Initializer. We are presented with an array of flavors, but there may be double entries which we remove, so we store only the distinct flavors.

⟨*Decays: stable config: TBP*⟩+≡

```
procedure :: init => stable_config_init
```

⟨*Decays: procedures*⟩+≡

```
subroutine stable_config_init (config, flv)
  class(stable_config_t), intent(out) :: config
  type(flavor_t), dimension(:), intent(in) :: flv
  integer, dimension (size (flv)) :: pdg
  logical, dimension (size (flv)) :: mask
  integer :: i
  pdg = flv%get_pdg ()
  mask(1) = .true.
  forall (i = 2 : size (pdg))
     mask(i) = all (pdg(i) /= pdg(1:i-1))
  end forall
```

```
    allocate (config%flv (count (mask)))
    config%flv = pack (flv, mask)
  end subroutine stable_config_init
```

Here is the corresponding object instance. Except for the pointer to the configuration, there is no content.

⟨*Decays: types*⟩+≡

```
  type, extends (any_t) :: stable_t
     private
     type(stable_config_t), pointer :: config => null ()
   contains
   ⟨Decays: stable: TBP⟩
  end type stable_t
```

The finalizer does nothing.

⟨*Decays: stable: TBP*⟩≡

```
  procedure :: final => stable_final
```

⟨*Decays: procedures*⟩+≡

```
  subroutine stable_final (object)
    class(stable_t), intent(inout) :: object
  end subroutine stable_final
```

We can delegate output to the configuration object.

⟨*Decays: stable: TBP*⟩+≡

```
  procedure :: write => stable_write
```

⟨*Decays: procedures*⟩+≡

```
  subroutine stable_write (object, unit, indent)
    class(stable_t), intent(in) :: object
    integer, intent(in), optional :: unit, indent
    call object%config%write (unit, indent)
  end subroutine stable_write
```

Initializer: just assign the configuration.

⟨*Decays: stable: TBP*⟩+≡

```
  procedure :: init => stable_init
```

⟨*Decays: procedures*⟩+≡

```
  subroutine stable_init (stable, config)
    class(stable_t), intent(out) :: stable
    type(stable_config_t), intent(in), target :: config
    stable%config => config
  end subroutine stable_init
```

32.4.10 Unstable Particles

A branching configuration enables us to select among distinct decay modes of a particle. We store the particle flavor (with its implicit link to a model), an array of decay configurations, and a selector object.

The total width, absolute and relative error are stored as `integral`, `abs_error`, and `rel_error`, respectively.

The flavor must be unique in this case.

⟨*Decays: public*⟩+≡
```
public :: unstable_config_t
```

⟨*Decays: types*⟩+≡
```
type, extends (any_config_t) :: unstable_config_t
   private
   type(flavor_t) :: flv
   real(default) :: integral = 0
   real(default) :: abs_error = 0
   real(default) :: rel_error = 0
   type(selector_t) :: selector
   type(decay_config_t), dimension(:), allocatable :: decay_config
 contains
 ⟨Decays: unstable config: TBP⟩
end type unstable_config_t
```

Finalizer. The branching configuration can be a recursive structure.

⟨*Decays: unstable config: TBP*⟩≡
```
procedure :: final => unstable_config_final
```

⟨*Decays: procedures*⟩+≡
```
recursive subroutine unstable_config_final (object)
  class(unstable_config_t), intent(inout) :: object
  integer :: i
  if (allocated (object%decay_config)) then
     do i = 1, size (object%decay_config)
        call object%decay_config(i)%final ()
     end do
  end if
end subroutine unstable_config_final
```

Output. Since this may be recursive, we include indentation.

⟨*Decays: unstable config: TBP*⟩+≡
```
procedure :: write => unstable_config_write
```

⟨*Decays: procedures*⟩+≡
```
recursive subroutine unstable_config_write (object, unit, indent, verbose)
  class(unstable_config_t), intent(in) :: object
  integer, intent(in), optional :: unit, indent
  logical, intent(in), optional :: verbose
  integer :: u, i, ind
  logical :: verb
  u = given_output_unit (unit)
  ind = 0;  if (present (indent))  ind = indent
  verb = .true.;  if (present (verbose))  verb = verbose
  call write_indent (u, ind)
  write (u, "(1x,'+',1x,A,1x,A)")  "Unstable:", &
       char (object%flv%get_name ())
  call write_indent (u, ind)
  write (u, 1)  "total width =", object%integral
  call write_indent (u, ind)
```

```
      write (u, 1)  "error (abs) =", object%abs_error
      call write_indent (u, ind)
      write (u, 1)  "error (rel) =", object%rel_error
  1   format (5x,A,ES19.12)
      if (verb .and. allocated (object%decay_config)) then
         do i = 1, size (object%decay_config)
            call object%decay_config(i)%write (u, ind + 1)
         end do
      end if
    end subroutine unstable_config_write
```

Initializer. For the unstable particle, the flavor is unique.

⟨*Decays: unstable config: TBP*⟩+≡

```
    procedure :: init => unstable_config_init
```

⟨*Decays: procedures*⟩+≡

```
    subroutine unstable_config_init (unstable, flv, set_decays, model)
      class(unstable_config_t), intent(out) :: unstable
      type(flavor_t), intent(in) :: flv
      logical, intent(in), optional :: set_decays
      class(model_data_t), intent(in), optional, target :: model
      type(string_t), dimension(:), allocatable :: decay
      unstable%flv = flv
      if (present (set_decays)) then
         call unstable%flv%get_decays (decay)
         call unstable%init_decays (decay, model)
      end if
    end subroutine unstable_config_init
```

Further initialization: determine the number of decay modes. We can assume that the flavor of the particle has been set already.

If the process stack is given, we can delve recursively into actually assigning decay processes. Otherwise, we just initialize with decay process names.

⟨*Decays: unstable config: TBP*⟩+≡

```
    procedure :: init_decays => unstable_config_init_decays
```

⟨*Decays: procedures*⟩+≡

```
    recursive subroutine unstable_config_init_decays &
         (unstable, decay_id, model, process_stack)
      class(unstable_config_t), intent(inout) :: unstable
      type(string_t), dimension(:), intent(in) :: decay_id
      class(model_data_t), intent(in), target :: model
      type(process_stack_t), intent(in), optional :: process_stack
      integer :: i
      allocate (unstable%decay_config (size (decay_id)))
      do i = 1, size (decay_id)
         associate (decay => unstable%decay_config(i))
           if (present (process_stack)) then
              call decay%connect (process_stack%get_process_ptr (decay_id(i)), &
                   model, process_stack)
           else
              call decay%init (model, decay_id(i))
           end if
           call decay%set_flv (unstable%flv)
```

```
      end associate
    end do
  end subroutine unstable_config_init_decays
```

Explicitly connect a specific decay with a process. This is used only in unit tests.

⟨*Decays: unstable config: TBP*⟩+≡

```
  procedure :: connect_decay => unstable_config_connect_decay
```

⟨*Decays: procedures*⟩+≡

```
  subroutine unstable_config_connect_decay (unstable, i, process, model)
    class(unstable_config_t), intent(inout) :: unstable
    integer, intent(in) :: i
    type(process_t), intent(in), target :: process
    class(model_data_t), intent(in), target :: model
    associate (decay => unstable%decay_config(i))
      call decay%connect (process, model)
    end associate
  end subroutine unstable_config_connect_decay
```

Compute the total width and branching ratios, initializing the decay selector.

⟨*Decays: unstable config: TBP*⟩+≡

```
  procedure :: compute => unstable_config_compute
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine unstable_config_compute (unstable)
    class(unstable_config_t), intent(inout) :: unstable
    integer :: i
    do i = 1, size (unstable%decay_config)
       call unstable%decay_config(i)%compute ()
    end do
    unstable%integral = sum (unstable%decay_config%integral)
    if (unstable%integral <= 0) then
       call unstable%write ()
       call msg_fatal ("Decay configuration: computed total width is zero")
    end if
    unstable%abs_error = sqrt (sum (unstable%decay_config%abs_error ** 2))
    unstable%rel_error = unstable%abs_error / unstable%integral
    call unstable%selector%init (unstable%decay_config%integral)
    do i = 1, size (unstable%decay_config)
       unstable%decay_config(i)%weight &
            = unstable%selector%get_weight (i)
    end do
  end subroutine unstable_config_compute
```

Now we define the instance of an unstable particle.

⟨*Decays: public*⟩+≡

```
  public :: unstable_t
```

⟨*Decays: types*⟩+≡

```
  type, extends (any_t) :: unstable_t
     private
     type(unstable_config_t), pointer :: config => null ()
```

```
    class(rng_t), allocatable :: rng
    integer :: selected_decay = 0
    type(decay_t), dimension(:), allocatable :: decay
  contains
  ⟨Decays: unstable: TBP⟩
end type unstable_t
```

Recursive finalizer.

⟨*Decays: unstable: TBP*⟩≡

```
procedure :: final => unstable_final
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine unstable_final (object)
  class(unstable_t), intent(inout) :: object
  integer :: i
  if (allocated (object%decay)) then
     do i = 1, size (object%decay)
        call object%decay(i)%final ()
     end do
  end if
end subroutine unstable_final
```

Output.

⟨*Decays: unstable: TBP*⟩+≡

```
procedure :: write => unstable_write
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine unstable_write (object, unit, indent)
  class(unstable_t), intent(in) :: object
  integer, intent(in), optional :: unit, indent
  integer :: u, ind
  u = given_output_unit (unit)
  ind = 0;  if (present (indent))  ind = indent
  call object%config%write (u, ind, verbose=.false.)
  if (allocated (object%rng)) then
     call object%rng%write (u, ind + 2)
  end if
  call write_indent (u, ind)
  if (object%selected_decay > 0) then
     write (u, "(5x,A,I0)") "Sel. decay  = ", object%selected_decay
     call object%decay(object%selected_decay)%write (u, ind + 1)
  else
     write (u, "(5x,A)")  "Sel. decay  = [undefined]"
  end if
end subroutine unstable_write
```

Write the embedded process instances.

⟨*Decays: unstable: TBP*⟩+≡

```
procedure :: write_process_instances => unstable_write_process_instances
```

⟨*Decays: procedures*⟩+≡

```
recursive subroutine unstable_write_process_instances &
     (unstable, unit, verbose)
  class(unstable_t), intent(in) :: unstable
```

```
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: verbose
    if (unstable%selected_decay > 0) then
       call unstable%decay(unstable%selected_decay)% &
            write_process_instances (unit, verbose)
    end if
  end subroutine unstable_write_process_instances
```

Initialization, using the configuration object.

⟨*Decays: unstable: TBP*⟩+≡

```
  procedure :: init => unstable_init
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine unstable_init (unstable, config)
    class(unstable_t), intent(out) :: unstable
    type(unstable_config_t), intent(in), target :: config
    integer :: i
    unstable%config => config
    allocate (unstable%decay (size (config%decay_config)))
    do i = 1, size (config%decay_config)
       call unstable%decay(i)%init (config%decay_config(i))
    end do
  end subroutine unstable_init
```

Recursively link interactions to the parent process. `i_prt` is the index of the current outgoing particle in the parent interaction.

⟨*Decays: unstable: TBP*⟩+≡

```
  procedure :: link_interactions => unstable_link_interactions
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine unstable_link_interactions (unstable, i_prt, trace)
    class(unstable_t), intent(inout) :: unstable
    integer, intent(in) :: i_prt
    type(interaction_t), intent(in), target :: trace
    integer :: i
    do i = 1, size (unstable%decay)
       call unstable%decay(i)%link_interactions (i_prt, trace)
    end do
  end subroutine unstable_link_interactions
```

Import the random-number generator state.

⟨*Decays: unstable: TBP*⟩+≡

```
  procedure :: import_rng => unstable_import_rng
```

⟨*Decays: procedures*⟩+≡

```
  subroutine unstable_import_rng (unstable, rng)
    class(unstable_t), intent(inout) :: unstable
    class(rng_t), intent(inout), allocatable :: rng
    call move_alloc (from = rng, to = unstable%rng)
  end subroutine unstable_import_rng
```

Generate a decay chain. First select a decay mode, then call the `select_chain` method of the selected mode.

⟨*Decays: unstable: TBP*⟩+≡
```
procedure :: select_chain => unstable_select_chain
```

⟨*Decays: procedures*⟩+≡
```
recursive subroutine unstable_select_chain (unstable)
  class(unstable_t), intent(inout) :: unstable
  real(default) :: x
  call unstable%rng%generate (x)
  unstable%selected_decay = unstable%config%selector%select (x)
  call unstable%decay(unstable%selected_decay)%select_chain ()
end subroutine unstable_select_chain
```

Generate a decay event.

⟨*Decays: unstable: TBP*⟩+≡
```
procedure :: generate => unstable_generate
```

⟨*Decays: procedures*⟩+≡
```
recursive subroutine unstable_generate (unstable)
  class(unstable_t), intent(inout) :: unstable
  call unstable%decay(unstable%selected_decay)%generate ()
end subroutine unstable_generate
```

32.4.11 Decay Chain

While the decay configuration tree and the decay tree are static entities (during a simulation run), the decay chain is dynamically generated for each event. The reason is that with the possibility of several decay modes for each particle, and several terms for each process, the total number of distinct decay chains is not under control.

Each entry in the decay chain is a connected parton state. The origin of the chain is a connected state in the parent process (not part of the chain itself). For each decay, mode and term chosen, we convolute this with the isolated (!) state of the current decay, to generate a new connected state. We accumulate this chain by recursively traversing the allocated decay tree. Whenever a particle decays, it becomes virtual and is replaced by its decay product, while all other particles stay in the parton state as spectators.

Technically, we implement the decay chain as a stack structure and include information from the associated decay object for easier debugging. This is a decay chain entry:

⟨*Decays: types*⟩+≡
```
type, extends (connected_state_t) :: decay_chain_entry_t
   private
   integer :: index = 0
   type(decay_config_t), pointer :: config => null ()
   integer :: selected_mci = 0
   integer :: selected_term = 0
   type(decay_chain_entry_t), pointer :: previous => null ()
end type decay_chain_entry_t
```

This is the complete chain; we need just a pointer to the last entry. We also include a pointer to the master process instance, which serves as the seed for the decay chain.

The evaluator `correlated_trace` traces over all quantum numbers for the final spin-correlated (but color-summed) evaluator of the decay chain. This allows us to compute the probability for a momentum configuration, given that all individual density matrices (of the initial process and the subsequent decays) have been normalized to one.

Note: This trace is summed over color, so color is treated exactly when computing spin correlations. However, we do not keep non-diagonal color correlations. When an event is accepted, we compute probabilities for all color states and can choose one of them.

⟨*Decays: public*⟩+≡
```
public :: decay_chain_t
```

⟨*Decays: types*⟩+≡
```
type :: decay_chain_t
   private
   type(process_instance_t), pointer :: process_instance => null ()
   integer :: selected_term = 0
   type(evaluator_t) :: correlated_trace
   type(decay_chain_entry_t), pointer :: last => null ()
 contains
 ⟨Decays: decay chain: TBP⟩
end type decay_chain_t
```

The finalizer recursively deletes and deallocates the entries.

⟨*Decays: decay chain: TBP*⟩≡
```
procedure :: final => decay_chain_final
```

⟨*Decays: procedures*⟩+≡
```
subroutine decay_chain_final (object)
  class(decay_chain_t), intent(inout) :: object
  type(decay_chain_entry_t), pointer :: entry
  do while (associated (object%last))
     entry => object%last
     object%last => entry%previous
     call entry%final ()
     deallocate (entry)
  end do
  call object%correlated_trace%final ()
end subroutine decay_chain_final
```

Doing output recursively allows us to display the chain in chronological order.

⟨*Decays: decay chain: TBP*⟩+≡
```
procedure :: write => decay_chain_write
```

⟨*Decays: procedures*⟩+≡
```
subroutine decay_chain_write (object, unit)
  class(decay_chain_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
```

```
    call write_separator (u, 2)
    write (u, "(1x,A)")  "Decay chain:"
    call write_entries (object%last)
    call write_separator (u, 2)
    write (u, "(1x,A)")  "Evaluator (correlated trace of the decay chain):"
    call write_separator (u)
    call object%correlated_trace%write (u)
    call write_separator (u, 2)
  contains
    recursive subroutine write_entries (entry)
      type(decay_chain_entry_t), intent(in), pointer :: entry
      if (associated (entry)) then
         call write_entries (entry%previous)
         call write_separator (u, 2)
         write (u, "(1x,A,I0)")  "Decay #", entry%index
         call entry%config%write_header (u)
         write (u, "(3x,A,I0)")  "Selected MCI    = ", entry%selected_mci
         write (u, "(3x,A,I0)")  "Selected term   = ", entry%selected_term
         call entry%config%term_config(entry%selected_term)%write (u, indent=1)
         call entry%write (u)
      end if
    end subroutine write_entries
  end subroutine decay_chain_write
```

Build a decay chain, recursively following the selected decays and terms in a decay tree. Before start, we finalize the chain, deleting any previous contents.

⟨*Decays: decay chain: TBP*⟩+≡

```
  procedure :: build => decay_chain_build
```

⟨*Decays: procedures*⟩+≡

```
  subroutine decay_chain_build (chain, decay_root)
    class(decay_chain_t), intent(inout), target :: chain
    type(decay_root_t), intent(in) :: decay_root
    type(quantum_numbers_mask_t), dimension(:), allocatable :: qn_mask
    type(interaction_t), pointer :: int_last_decay
    call chain%final ()
    if (decay_root%selected_term > 0) then
       chain%process_instance => decay_root%process_instance
       chain%selected_term = decay_root%selected_term
       call chain%build_term_entries (decay_root%term(decay_root%selected_term))
    end if
    int_last_decay => chain%last%get_matrix_int_ptr ()
    allocate (qn_mask (int_last_decay%get_n_tot ()))
    call qn_mask%init (mask_f = .true., mask_c = .true., mask_h = .true.)
    call chain%correlated_trace%init_qn_sum (int_last_decay, qn_mask)
  end subroutine decay_chain_build
```

Build the entries that correspond to a decay term. We have to scan all unstable particles.

⟨*Decays: decay chain: TBP*⟩+≡

```
  procedure :: build_term_entries => decay_chain_build_term_entries
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine decay_chain_build_term_entries (chain, term)
```

```
    class(decay_chain_t), intent(inout) :: chain
    type(decay_term_t), intent(in) :: term
    integer :: i
    do i = 1, size (term%particle_out)
       select type (unstable => term%particle_out(i)%c)
       type is (unstable_t)
          if (unstable%selected_decay > 0) then
             call chain%build_decay_entries &
                  (unstable%decay(unstable%selected_decay))
          end if
       end select
    end do
  end subroutine decay_chain_build_term_entries
```

Build the entries that correspond to a specific decay. The decay term should have been determined, so we allocate a decay chain entry and fill it, then proceed to child decays.

For the first entry, we convolute the connected state of the parent process instance with the isolated state of the current decay (which does not contain an extra beam entry for the parent). For subsequent entries, we take the previous entry as first factor.

In principle, each chain entry (as a parton state) is capable of holding a subevent object and associated expressions. We currently do not make use of that feature.

Before generating the decays, factor out the trace of the helicity density matrix of the parent parton state. This trace has been used for unweighting the original event (unweighted case) or it determines the overall weight, so it should not be taken into account in the decay chain generation.

⟨*Decays: decay chain: TBP*⟩+≡

```
  procedure :: build_decay_entries => decay_chain_build_decay_entries
```

⟨*Decays: procedures*⟩+≡

```
  recursive subroutine decay_chain_build_decay_entries (chain, decay)
    class(decay_chain_t), intent(inout) :: chain
    type(decay_t), intent(in) :: decay
    type(decay_chain_entry_t), pointer :: entry
    type(connected_state_t), pointer :: previous_state
    type(isolated_state_t), pointer :: current_decay
    type(helicity_t) :: hel
    type(quantum_numbers_t) :: qn_filter_conn
    allocate (entry)
    if (associated (chain%last)) then
       entry%previous => chain%last
       entry%index = entry%previous%index + 1
       previous_state => entry%previous%connected_state_t
    else
       entry%index = 1
       previous_state => &
            chain%process_instance%get_connected_state_ptr (chain%selected_term)
    end if
    entry%config => decay%config
    entry%selected_mci = decay%selected_mci
    entry%selected_term = decay%selected_term
```

```
    current_decay => decay%process_instance%get_isolated_state_ptr &
         (decay%selected_term)
    call entry%setup_connected_trace &
         (current_decay, previous_state%get_trace_int_ptr (), resonant=.true.)
    if (entry%config%flv%has_decay_helicity ()) then
       call hel%init (entry%config%flv%get_decay_helicity ())
       call qn_filter_conn%init (hel)
       call entry%setup_connected_matrix &
            (current_decay, previous_state%get_matrix_int_ptr (), &
            resonant=.true., qn_filter_conn = qn_filter_conn)
       call entry%setup_connected_flows &
            (current_decay, previous_state%get_flows_int_ptr (), &
            resonant=.true., qn_filter_conn = qn_filter_conn)
    else
       call entry%setup_connected_matrix &
            (current_decay, previous_state%get_matrix_int_ptr (), &
            resonant=.true.)
       call entry%setup_connected_flows &
            (current_decay, previous_state%get_flows_int_ptr (), &
            resonant=.true.)
    end if
    chain%last => entry
    call chain%build_term_entries (decay%term(decay%selected_term))
  end subroutine decay_chain_build_decay_entries
```

Recursively fill the decay chain with momenta and evaluate the matrix elements. Since all evaluators should have correct source entries at this point, momenta are automatically retrieved from the appropriate process instance.

Like we did above for the parent process, factor out the trace for each subsequent decay (the helicity density matrix in the isolated state, which is taken for the convolution).

⟨*Decays: decay chain: TBP*⟩+≡

```
  procedure :: evaluate => decay_chain_evaluate
```

⟨*Decays: procedures*⟩+≡

```
  subroutine decay_chain_evaluate (chain)
    class(decay_chain_t), intent(inout) :: chain
    call evaluate (chain%last)
    call chain%correlated_trace%receive_momenta ()
    call chain%correlated_trace%evaluate ()
  contains
    recursive subroutine evaluate (entry)
      type(decay_chain_entry_t), intent(inout), pointer :: entry
      if (associated (entry)) then
         call evaluate (entry%previous)
         call entry%receive_kinematics ()
         call entry%evaluate_trace ()
         call entry%evaluate_event_data ()
      end if
    end subroutine evaluate
  end subroutine decay_chain_evaluate
```

Return the probability of a decay chain. This is given as the trace of the density

matrix with intermediate helicity correlations, normalized by the product of the uncorrelated density matrix traces. This works only if an event has been evaluated and the `correlated_trace` evaluator is filled. By definition, this evaluator has only one matrix element, and this must be real.

⟨*Decays: decay chain: TBP*⟩+≡
```
procedure :: get_probability => decay_chain_get_probability
```

⟨*Decays: procedures*⟩+≡
```
function decay_chain_get_probability (chain) result (x)
  class(decay_chain_t), intent(in) :: chain
  real(default) :: x
  x = real (chain%correlated_trace%get_matrix_element (1))
end function decay_chain_get_probability
```

32.4.12 Decay as Event Transform

The `evt_decay` object combines decay configuration, decay tree, and chain in a single object, as an implementation of the `evt` (event transform) abstract type.

⟨*Decays: public*⟩+≡
```
public :: evt_decay_t
```

⟨*Decays: types*⟩+≡
```
type, extends (evt_t) :: evt_decay_t
   private
   type(decay_root_config_t) :: decay_root_config
   type(decay_root_t) :: decay_root
   type(decay_chain_t) :: decay_chain
 contains
 ⟨Decays: evt decay: TBP⟩
end type evt_decay_t
```

⟨*Decays: evt decay: TBP*⟩≡
```
procedure :: write_name => evt_decay_write_name
```

⟨*Decays: procedures*⟩+≡
```
subroutine evt_decay_write_name (evt, unit)
  class(evt_decay_t), intent(in) :: evt
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Event transform: partonic decays"
end subroutine evt_decay_write_name
```

Output. We display the currently selected decay tree, which includes configuration data, and the decay chain, i.e., the evaluators.

⟨*Decays: evt decay: TBP*⟩+≡
```
procedure :: write => evt_decay_write
```

⟨*Decays: procedures*⟩+≡
```
subroutine evt_decay_write (evt, unit, verbose, more_verbose, testflag)
  class(evt_decay_t), intent(in) :: evt
  integer, intent(in), optional :: unit
```

```
    logical, intent(in), optional :: verbose, more_verbose, testflag
    logical :: verb, verb2
    integer :: u
    u = given_output_unit (unit)
    verb = .true.;  if (present (verbose))  verb = verbose
    verb2 = .false.;  if (present (more_verbose))  verb2 = more_verbose
    call write_separator (u, 2)
    call evt%write_name (u)
    call write_separator (u, 2)
    call evt%base_write (u, testflag = testflag)
    if (verb) then
       call write_separator (u)
       call evt%decay_root%write (u)
       if (verb2) then
          call evt%decay_chain%write (u)
          call evt%decay_root%write_process_instances (u, verb)
       end if
    else
       call write_separator (u, 2)
    end if
  end subroutine evt_decay_write
```

Connect with a process instance and process. This initializes the decay configuration. The process stack is used to look for process objects that implement daughter decays.

When all processes are assigned, configure the decay tree instance, using the decay tree configuration. First obtain the branching ratios, then allocate the decay tree. This is done once for all events.

⟨*Decays: evt decay: TBP*⟩+≡

```
  procedure :: connect => evt_decay_connect
```

⟨*Decays: procedures*⟩+≡

```
  subroutine evt_decay_connect (evt, process_instance, model, process_stack)
    class(evt_decay_t), intent(inout), target :: evt
    type(process_instance_t), intent(in), target :: process_instance
    class(model_data_t), intent(in), target :: model
    type(process_stack_t), intent(in), optional :: process_stack
    call evt%base_connect (process_instance, model)
    call evt%decay_root_config%connect (process_instance%process, &
         model, process_stack, process_instance)
    call evt%decay_root_config%compute ()
    call evt%decay_root%init (evt%decay_root_config, evt%process_instance)
  end subroutine evt_decay_connect
```

Prepare a new event: Select a decay chain and build the corresponding chain object.

⟨*Decays: evt decay: TBP*⟩+≡

```
  procedure :: prepare_new_event => evt_decay_prepare_new_event
```

⟨*Decays: procedures*⟩+≡

```
  subroutine evt_decay_prepare_new_event (evt, i_mci, i_term)
    class(evt_decay_t), intent(inout) :: evt
    integer, intent(in) :: i_mci, i_term
```

```
    call evt%reset ()
    evt%decay_root%selected_mci = i_mci
    evt%decay_root%selected_term = i_term
    call evt%decay_root%select_chain ()
    call evt%decay_chain%build (evt%decay_root)
  end subroutine evt_decay_prepare_new_event
```

Generate a weighted event and assign the resulting weight (probability). We use a chain initialized by the preceding subroutine, fill it with momenta and evaluate.

⟨*Decays: evt decay: TBP*⟩+≡

```
  procedure :: generate_weighted => evt_decay_generate_weighted
```

⟨*Decays: procedures*⟩+≡

```
  subroutine evt_decay_generate_weighted (evt, probability)
    class(evt_decay_t), intent(inout) :: evt
    real(default), intent(inout) :: probability
    call evt%decay_root%generate ()
    if (signal_is_pending ())  return
    call evt%decay_chain%evaluate ()
    probability = evt%decay_chain%get_probability ()
  end subroutine evt_decay_generate_weighted
```

To create a usable event, we have to transform the interaction into a particle set; this requires factorization for the correlated density matrix, according to the factorization mode.

⟨*Decays: evt decay: TBP*⟩+≡

```
  procedure :: make_particle_set => evt_decay_make_particle_set
```

⟨*Decays: procedures*⟩+≡

```
  subroutine evt_decay_make_particle_set &
       (evt, factorization_mode, keep_correlations, r)
    class(evt_decay_t), intent(inout) :: evt
    integer, intent(in) :: factorization_mode
    logical, intent(in) :: keep_correlations
    real(default), dimension(:), intent(in), optional :: r
    type(interaction_t), pointer :: int_matrix, int_flows
    type(decay_chain_entry_t), pointer :: last_entry
    last_entry => evt%decay_chain%last
    int_matrix => last_entry%get_matrix_int_ptr ()
    int_flows  => last_entry%get_flows_int_ptr ()
    call evt%factorize_interactions (int_matrix, int_flows, &
         factorization_mode, keep_correlations, r)
    call evt%tag_incoming ()
  end subroutine evt_decay_make_particle_set
```

Auxiliary

Eliminate numerical noise for the associated process instances.

⟨*Decays: public*⟩+≡

```
  public :: pacify
```

⟨*Decays: interfaces*⟩+≡

```
interface pacify
   module procedure pacify_decay
   module procedure pacify_decay_gen
   module procedure pacify_term
   module procedure pacify_unstable
end interface pacify
```

⟨*Decays: procedures*⟩+≡

```
subroutine pacify_decay (evt)
  class(evt_decay_t), intent(inout) :: evt
  call pacify_decay_gen (evt%decay_root)
end subroutine pacify_decay

recursive subroutine pacify_decay_gen (decay)
  class(decay_gen_t), intent(inout) :: decay
  if (associated (decay%process_instance)) then
     call pacify (decay%process_instance)
  end if
  if (decay%selected_term > 0) then
     call pacify_term (decay%term(decay%selected_term))
  end if
end subroutine pacify_decay_gen

recursive subroutine pacify_term (term)
  class(decay_term_t), intent(inout) :: term
  integer :: i
  do i = 1, size (term%particle_out)
     select type (unstable => term%particle_out(i)%c)
     type is (unstable_t);  call pacify_unstable (unstable)
     end select
  end do
end subroutine pacify_term

recursive subroutine pacify_unstable (unstable)
  class(unstable_t), intent(inout) :: unstable
  if (unstable%selected_decay > 0) then
     call pacify_decay_gen (unstable%decay(unstable%selected_decay))
  end if
end subroutine pacify_unstable
```

Prepare specific configurations for use in unit tests.

⟨*Decays: unstable config: TBP*⟩+≡

```
procedure :: init_test_case1
procedure :: init_test_case2
```

⟨*Decays: procedures*⟩+≡

```
subroutine init_test_case1 (unstable, i, flv, integral, relerr, model)
  class(unstable_config_t), intent(inout) :: unstable
  integer, intent(in) :: i
  type(flavor_t), dimension(:,:), intent(in) :: flv
  real(default), intent(in) :: integral
  real(default), intent(in) :: relerr
  class(model_data_t), intent(in), target :: model
  associate (decay => unstable%decay_config(i))
```

```
    allocate (decay%term_config (1))
    call decay%init_term (1, flv, stable = [.true., .true.], model=model)
    decay%integral = integral
    decay%abs_error = integral * relerr
  end associate
end subroutine init_test_case1

subroutine init_test_case2 (unstable, flv1, flv21, flv22, model)
  class(unstable_config_t), intent(inout) :: unstable
  type(flavor_t), dimension(:,:), intent(in) :: flv1, flv21, flv22
  class(model_data_t), intent(in), target :: model
  associate (decay => unstable%decay_config(1))
    decay%integral = 1.e-3_default
    decay%abs_error = decay%integral * .01_default

    allocate (decay%term_config (1))
    call decay%init_term (1, flv1, stable = [.false., .true.], model=model)

    select type (w => decay%term_config(1)%prt(1)%c)
    type is (unstable_config_t)

       associate (w_decay => w%decay_config(1))
         w_decay%integral = 2._default
         allocate (w_decay%term_config (1))
         call w_decay%init_term (1, flv21, stable = [.true., .true.], &
              model=model)
       end associate
       associate (w_decay => w%decay_config(2))
         w_decay%integral = 1._default
         allocate (w_decay%term_config (1))
         call w_decay%init_term (1, flv22, stable = [.true., .true.], &
              model=model)
       end associate
       call w%compute ()

    end select
  end associate
end subroutine init_test_case2
```

32.4.13 Unit tests

Test module, followed by the corresponding implementation module.

⟨decays_ut.f90⟩≡

```
⟨File header⟩

module decays_ut
  use unit_tests
  use decays_uti

⟨Standard module head⟩

⟨Decays: public test⟩
```

```
⟨Decays: public test auxiliary⟩

contains

⟨Decays: test driver⟩

end module decays_ut
```

⟨decays_uti.f90⟩≡

```
⟨File header⟩

module decays_uti

⟨Use kinds⟩
⟨Use strings⟩
  use os_interface
  use sm_qcd
  use model_data
  use state_matrices, only: FM_IGNORE_HELICITY
  use interactions, only: reset_interaction_counter
  use flavors
  use process_libraries
  use rng_base
  use mci_base
  use mci_midpoint
  use phs_base
  use phs_single
  use prc_core
  use prc_test, only: prc_test_create_library
  use process, only: process_t
  use instances, only: process_instance_t
  use process_stacks

  use decays

  use rng_base_ut, only: rng_test_t, rng_test_factory_t

⟨Standard module head⟩

⟨Decays: public test auxiliary⟩

⟨Decays: test declarations⟩

contains

⟨Decays: tests⟩

⟨Decays: test auxiliary⟩

end module decays_uti
```

API: driver for the unit tests below.

⟨*Decays: public test*⟩≡

```
public :: decays_test
```

⟨*Decays: test driver*⟩≡

```
subroutine decays_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨Decays: execute tests⟩
end subroutine decays_test
```

Testbed

As a variation of the `prepare_test_process` routine used elsewhere, we define here a routine that creates two processes (scattering $ss \to ss$ and decay $s \to f\bar{f}$), compiles and integrates them and prepares for event generation.

⟨*Decays: public test auxiliary*⟩≡

```
public :: prepare_testbed
```

⟨*Decays: test auxiliary*⟩≡

```
subroutine prepare_testbed &
     (lib, process_stack, prefix, os_data, &
      scattering, decay, decay_rest_frame)
  type(process_library_t), intent(out), target :: lib
  type(process_stack_t), intent(out) :: process_stack
  type(string_t), intent(in) :: prefix
  type(os_data_t), intent(in) :: os_data
  logical, intent(in) :: scattering, decay
  logical, intent(in), optional :: decay_rest_frame

  type(model_data_t), target :: model
  class(model_data_t), pointer :: model_copy
  type(string_t) :: libname, procname1, procname2, run_id
  type(qcd_t) :: qcd
  class(rng_factory_t), allocatable :: rng_factory
  type(process_entry_t), pointer :: process
  type(process_instance_t), allocatable, target :: process_instance
  class(mci_t), allocatable :: mci_template
  class(phs_config_t), allocatable :: phs_config_template
  type(field_data_t), pointer :: field_data
  real(default) :: sqrts

  libname = prefix // "_lib"
  procname1 = prefix // "_p"
  procname2 = prefix // "_d"
  run_id = prefix

  call model%init_test ()
  call model%set_par (var_str ("ff"), 0.4_default)
  call model%set_par (var_str ("mf"), &
       model%get_real (var_str ("ff")) * model%get_real (var_str ("ms")))

  if (scattering .and. decay) then
     field_data => model%get_field_ptr (25)
     call field_data%set (p_is_stable = .false.)
```

```
end if

call prc_test_create_library (libname, lib, &
     scattering = .true., decay = .true., &
     procname1 = procname1, procname2 = procname2)

call reset_interaction_counter ()

allocate (mci_midpoint_t :: mci_template)
allocate (phs_single_config_t :: phs_config_template)

if (scattering) then

   allocate (rng_test_factory_t :: rng_factory)
   allocate (model_copy)
   call model_copy%init (model%get_name (), &
        model%get_n_real (), &
        model%get_n_complex (), &
        model%get_n_field (), &
        model%get_n_vtx ())
   call model_copy%copy_from (model)

   allocate (process)
   call process%init (procname1, &
        run_id, lib, os_data, qcd, rng_factory, model_copy)
   call process%setup_test_cores ()
   call process%init_component &
      (1, .true., mci_template, phs_config_template)
   sqrts = 1000
   call process%setup_beams_sqrts (sqrts, i_core = 1)
   call process%configure_phs ()
   call process%setup_mci ()
   call process%setup_terms ()

   allocate (process_instance)
   call process_instance%init (process%process_t)
   call process_instance%integrate (1, n_it = 1, n_calls = 100)
   call process%final_integration (1)
   call process_instance%final ()
   deallocate (process_instance)

   call process%prepare_simulation (1)
   call process_stack%push (process)
end if

if (decay) then
   allocate (rng_test_factory_t :: rng_factory)
   allocate (model_copy)
   call model_copy%init (model%get_name (), &
        model%get_n_real (), &
        model%get_n_complex (), &
        model%get_n_field (), &
        model%get_n_vtx ())
   call model_copy%copy_from (model)
```

```
      allocate (process)
      call process%init (procname2, &
           run_id, lib, os_data, qcd, rng_factory, model_copy)
      call process%setup_test_cores ()
      call process%init_component &
         (1, .true., mci_template, phs_config_template)
      if (present (decay_rest_frame)) then
         call process%setup_beams_decay (rest_frame = decay_rest_frame, i_core = 1)
      else
         call process%setup_beams_decay (rest_frame = .not. scattering, i_core = 1)
      end if
      call process%configure_phs ()
      call process%setup_mci ()
      call process%setup_terms ()

      allocate (process_instance)
      call process_instance%init (process%process_t)
      call process_instance%integrate (1, n_it=1, n_calls=100)
      call process%final_integration (1)
      call process_instance%final ()
      deallocate (process_instance)

      call process%prepare_simulation (1)
      call process_stack%push (process)
   end if

   call model%final ()

end subroutine prepare_testbed
```

Simple decay configuration

We define a branching configuration with two decay modes. We set the integral values by hand, so we do not need to evaluate processes, yet.

⟨*Decays: execute tests*⟩≡

```
call test (decays_1, "decays_1", &
     "branching and decay configuration", &
     u, results)
```

⟨*Decays: test declarations*⟩≡

```
public :: decays_1
```

⟨*Decays: tests*⟩≡

```
subroutine decays_1 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_data_t), target :: model
  type(flavor_t) :: flv_h
  type(flavor_t), dimension(2,1) :: flv_hbb, flv_hgg
  type(unstable_config_t), allocatable :: unstable

  write (u, "(A)")  "* Test output: decays_1"
```

```
    write (u, "(A)")  "*   Purpose: Set up branching and decay configuration"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize environment"
    write (u, "(A)")

    call os_data_init (os_data)
    call model%init_sm_test ()

    call flv_h%init (25, model)
    call flv_hbb(:,1)%init ([5, -5], model)
    call flv_hgg(:,1)%init ([22, 22], model)

    write (u, "(A)")  "* Set up branching and decay"
    write (u, "(A)")

    allocate (unstable)
    call unstable%init (flv_h)
    call unstable%init_decays ([var_str ("h_bb"), var_str ("h_gg")], model)

    call unstable%init_test_case1 &
         (1, flv_hbb, 1.234e-3_default, .02_default, model)

    call unstable%init_test_case1 &
         (2, flv_hgg, 3.085e-4_default, .08_default, model)

    call unstable%compute ()
    call unstable%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call unstable%final ()
    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: decays_1"

  end subroutine decays_1
```

Cascade decay configuration

We define a branching configuration with one decay, which is followed by another branching.

⟨*Decays: execute tests*⟩+≡

```
  call test (decays_2, "decays_2", &
       "cascade decay configuration", &
       u, results)
```

⟨*Decays: test declarations*⟩+≡

```
  public :: decays_2
```

⟨*Decays: tests*⟩+≡

```
subroutine decays_2 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(model_data_t), target :: model
  type(flavor_t) :: flv_h, flv_wp, flv_wm
  type(flavor_t), dimension(2,1) :: flv_hww, flv_wud, flv_wen
  type(unstable_config_t), allocatable :: unstable

  write (u, "(A)")  "* Test output: decays_2"
  write (u, "(A)")  "*   Purpose: Set up cascade branching"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize environment"
  write (u, "(A)")

  call os_data_init (os_data)
  call model%init_sm_test ()

  call model%set_unstable (25, [var_str ("h_ww")])
  call model%set_unstable (24, [var_str ("w_ud"), var_str ("w_en")])

  call flv_h%init (25, model)
  call flv_hww(:,1)%init ([24, -24], model)
  call flv_wp%init (24, model)
  call flv_wm%init (-24, model)
  call flv_wud(:,1)%init ([2, -1], model)
  call flv_wen(:,1)%init ([-11, 12], model)


  write (u, "(A)")  "* Set up branching and decay"
  write (u, "(A)")

  allocate (unstable)
  call unstable%init (flv_h, set_decays=.true., model=model)

  call unstable%init_test_case2 (flv_hww, flv_wud, flv_wen, model)

  call unstable%compute ()
  call unstable%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call unstable%final ()
  call model%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: decays_2"

end subroutine decays_2
```

Decay and Process Object

We define a branching configuration with one decay and connect this with an actual process object.

⟨*Decays: execute tests*⟩+≡

```
call test (decays_3, "decays_3", &
     "associate process", &
     u, results)
```

⟨*Decays: test declarations*⟩+≡

```
public :: decays_3
```

⟨*Decays: tests*⟩+≡

```
subroutine decays_3 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  class(model_data_t), pointer :: model
  type(process_library_t), target :: lib
  type(string_t) :: prefix
  type(string_t) :: procname2
  type(process_stack_t) :: process_stack
  type(process_t), pointer :: process
  type(unstable_config_t), allocatable :: unstable
  type(flavor_t) :: flv

  write (u, "(A)")  "* Test output: decays_3"
  write (u, "(A)")  "*   Purpose: Connect a decay configuration &
       &with a process"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize environment and integrate process"
  write (u, "(A)")

  call os_data_init (os_data)

  prefix = "decays_3"
  call prepare_testbed &
       (lib, process_stack, prefix, os_data, &
       scattering=.false., decay=.true., decay_rest_frame=.false.)

  procname2 = prefix // "_d"
  process => process_stack%get_process_ptr (procname2)
  model => process%get_model_ptr ()
  call process%write (.false., u)

  write (u, "(A)")
  write (u, "(A)")  "* Set up branching and decay"
  write (u, "(A)")

  call flv%init (25, model)

  allocate (unstable)
  call unstable%init (flv)
  call unstable%init_decays ([procname2], model)
```

```
    write (u, "(A)")  "* Connect decay with process object"
    write (u, "(A)")

    call unstable%connect_decay (1, process, model)

    call unstable%compute ()
    call unstable%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call unstable%final ()
    call process_stack%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: decays_3"

  end subroutine decays_3
```

Decay and Process Object

Building upon the previous test, we set up a decay instance and generate a decay event.

⟨*Decays: execute tests*⟩+≡

```
  call test (decays_4, "decays_4", &
       "decay instance", &
       u, results)
```

⟨*Decays: test declarations*⟩+≡

```
  public :: decays_4
```

⟨*Decays: tests*⟩+≡

```
  subroutine decays_4 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    class(model_data_t), pointer :: model
    type(process_library_t), target :: lib
    type(string_t) :: prefix, procname2
    class(rng_t), allocatable :: rng
    type(process_stack_t) :: process_stack
    type(process_t), pointer :: process
    type(unstable_config_t), allocatable, target :: unstable
    type(flavor_t) :: flv
    type(unstable_t), allocatable :: instance

    write (u, "(A)")  "* Test output: decays_4"
    write (u, "(A)")  "*   Purpose: Create a decay process and evaluate &
         &an instance"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize environment, process, &
         &and decay configuration"
    write (u, "(A)")
```

```
    call os_data_init (os_data)

    prefix = "decays_4"
    call prepare_testbed &
         (lib, process_stack, prefix, os_data, &
         scattering=.false., decay=.true., decay_rest_frame = .false.)

    procname2 = prefix // "_d"
    process => process_stack%get_process_ptr (procname2)
    model => process%get_model_ptr ()

    call flv%init (25, model)

    allocate (unstable)
    call unstable%init (flv)
    call unstable%init_decays ([procname2], model)

    call model%set_unstable (25, [procname2])

    call unstable%connect_decay (1, process, model)

    call unstable%compute ()

    allocate (rng_test_t :: rng)

    allocate (instance)
    call instance%init (unstable)
    call instance%import_rng (rng)

    call instance%select_chain ()
    call instance%generate ()
    call instance%write (u)

    write (u, *)
    call instance%write_process_instances (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call instance%final ()
    call process_stack%final ()
    call unstable%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: decays_4"

  end subroutine decays_4
```

Decay with Parent Process

We define a scattering process $ss \to ss$ and subsequent decays $s \to f\bar{f}$.
⟨*Decays: execute tests*⟩+≡

```
    call test (decays_5, "decays_5", &
         "parent process and decay", &
         u, results)
```

⟨*Decays: test declarations*⟩+≡

```
    public :: decays_5
```

⟨*Decays: tests*⟩+≡

```
    subroutine decays_5 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      class(model_data_t), pointer :: model
      type(process_library_t), target :: lib
      type(string_t) :: prefix, procname1, procname2
      type(process_stack_t) :: process_stack
      type(process_t), pointer :: process
      type(process_instance_t), allocatable, target :: process_instance
      type(decay_root_config_t), target :: decay_root_config
      type(decay_root_t) :: decay_root
      type(decay_chain_t) :: decay_chain

      write (u, "(A)")  "* Test output: decays_5"
      write (u, "(A)")  "*   Purpose: Handle a process with subsequent decays"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize environment and parent process"
      write (u, "(A)")

      call os_data_init (os_data)

      prefix = "decays_5"
      procname1 = prefix // "_p"
      procname2 = prefix // "_d"
      call prepare_testbed &
           (lib, process_stack, prefix, os_data, &
           scattering=.true., decay=.true.)

      write (u, "(A)")  "* Initialize decay process"
      write (u, "(A)")

      process => process_stack%get_process_ptr (procname1)
      model => process%get_model_ptr ()
      call model%set_unstable (25, [procname2])

      write (u, "(A)")  "* Initialize decay tree configuration"
      write (u, "(A)")

      call decay_root_config%connect (process, model, process_stack)
      call decay_root_config%compute ()
      call decay_root_config%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Initialize decay tree"

      allocate (process_instance)
```

```
call process_instance%init (process)
call process_instance%setup_event_data ()
call process_instance%init_simulation (1)

call decay_root%init (decay_root_config, process_instance)

write (u, "(A)")
write (u, "(A)")  "* Select decay chain"
write (u, "(A)")

call decay_root%set_mci (1)
!!! Not yet implemented; there is only one term anyway:
! call process_instance%select_i_term (decay_root%selected_term)
call decay_root%set_term (1)
call decay_root%select_chain ()

call decay_chain%build (decay_root)

call decay_root%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate event"
write (u, "(A)")

call process_instance%generate_unweighted_event (decay_root%get_mci ())
call process_instance%evaluate_event_data ()

call decay_root%generate ()

call pacify (decay_root)

write (u, "(A)")  "* Process instances"
write (u, "(A)")

call decay_root%write_process_instances (u)

write (u, "(A)")
write (u, "(A)")  "* Generate decay chain"
write (u, "(A)")

call decay_chain%evaluate ()
call decay_chain%write (u)

write (u, *)
write (u, "(A,ES19.12)")  "chain probability =", &
     decay_chain%get_probability ()

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call decay_chain%final ()
call decay_root%final ()
call decay_root_config%final ()
call process_instance%final ()
```

```
    deallocate (process_instance)

    call process%final ()
    call process_stack%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: decays_5"

  end subroutine decays_5
```

Decay as Event Transform

Again, we define a scattering process $ss \to ss$ and subsequent decays $s \to f\bar{f}$.

⟨*Decays: execute tests*⟩+≡

```
  call test (decays_6, "decays_6", &
       "evt_decay object", &
       u, results)
```

⟨*Decays: test declarations*⟩+≡

```
  public :: decays_6
```

⟨*Decays: tests*⟩+≡

```
  subroutine decays_6 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    class(model_data_t), pointer :: model
    type(process_library_t), target :: lib
    type(string_t) :: prefix, procname1, procname2
    type(process_stack_t) :: process_stack
    type(process_t), pointer :: process
    type(process_instance_t), allocatable, target :: process_instance
    type(evt_decay_t), target :: evt_decay
    integer :: factorization_mode
    logical :: keep_correlations

    write (u, "(A)")  "* Test output: decays_6"
    write (u, "(A)")  "*   Purpose: Handle a process with subsequent decays"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize environment and parent process"
    write (u, "(A)")

    call os_data_init (os_data)

    prefix = "decays_6"
    procname1 = prefix // "_p"
    procname2 = prefix // "_d"
    call prepare_testbed &
         (lib, process_stack, prefix, os_data, &
         scattering=.true., decay=.true.)

    write (u, "(A)")  "* Initialize decay process"

    process => process_stack%get_process_ptr (procname1)
```

```
    model => process%get_model_ptr ()
    call model%set_unstable (25, [procname2])

    allocate (process_instance)
    call process_instance%init (process)
    call process_instance%setup_event_data ()
    call process_instance%init_simulation (1)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize decay object"

    call evt_decay%connect (process_instance, model, process_stack)

    write (u, "(A)")
    write (u, "(A)")  "* Generate scattering event"

    call process_instance%generate_unweighted_event (1)
    call process_instance%evaluate_event_data ()

    write (u, "(A)")
    write (u, "(A)")  "* Select decay chain and generate event"
    write (u, "(A)")

    call evt_decay%prepare_new_event (1, 1)
    call evt_decay%generate_unweighted ()

    factorization_mode = FM_IGNORE_HELICITY
    keep_correlations = .false.
    call evt_decay%make_particle_set (factorization_mode, keep_correlations)

    call evt_decay%write (u, verbose = .true.)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call evt_decay%final ()
    call process_instance%final ()
    deallocate (process_instance)

    call process_stack%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: decays_6"

  end subroutine decays_6
```

32.5 Shower

We might use matrix elements of LO and NLO to increase the accuracy of the shower in the sense of matching as well as merging.

⟨shower.f90⟩≡

⟨*File header*⟩

```
module shower
```

⟨*Use kinds*⟩
⟨*Use strings*⟩

```
  use io_units
  use format_utils, only: write_separator
  use system_defs, only: LF
  use os_interface
  use diagnostics
  use lorentz
  use pdf
  use subevents, only: PRT_BEAM_REMNANT, PRT_INCOMING, PRT_OUTGOING

  use shower_base
  use matching_base
  use powheg_matching, only: powheg_matching_t

  use sm_qcd
  use model_data
  use rng_base

  use event_transforms
  use models
  use hep_common
  use process, only: process_t
  use instances, only: process_instance_t
  use process_stacks
```

⟨*Standard module head*⟩

⟨*Shower: public*⟩

⟨*Shower: parameters*⟩

⟨*Shower: types*⟩

```
contains
```

⟨*Shower: procedures*⟩

```
end module shower
```

32.5.1 Configuration Parameters

POWHEG_TESTING allows to disable the parton shower for validation and testing of the POWHEG procedure.

⟨*Shower: parameters*⟩≡

```
    logical, parameter :: POWHEG_TESTING = .false.
```

32.5.2 Event Transform

The event transforms can do more than mere showering. Especially, it may reweight showered events to fixed-order matrix elements. The `model_hadrons` is supposed to be the SM variant that contains all hadrons that can be generated in the shower.

⟨*Shower: public*⟩≡
```
public :: evt_shower_t
```

⟨*Shower: types*⟩≡
```
type, extends (evt_t) :: evt_shower_t
   class(shower_base_t), allocatable :: shower
   class(matching_t), allocatable :: matching
   type(model_t), pointer :: model_hadrons => null ()
   type(qcd_t), pointer :: qcd => null()
   type(pdf_data_t) :: pdf_data
   type(os_data_t) :: os_data
   logical :: is_first_event
 contains
 ⟨Shower: evt shower: TBP⟩
end type evt_shower_t
```

⟨*Shower: evt shower: TBP*⟩≡
```
procedure :: write_name => evt_shower_write_name
```

⟨*Shower: procedures*⟩≡
```
subroutine evt_shower_write_name (evt, unit)
  class(evt_shower_t), intent(in) :: evt
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Event transform: shower"
end subroutine evt_shower_write_name
```

Output.

⟨*Shower: evt shower: TBP*⟩+≡
```
procedure :: write => evt_shower_write
```

⟨*Shower: procedures*⟩+≡
```
subroutine evt_shower_write (evt, unit, verbose, more_verbose, testflag)
  class(evt_shower_t), intent(in) :: evt
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose, more_verbose, testflag
  integer :: u
  u = given_output_unit (unit)
  call write_separator (u, 2)
  call evt%write_name (u)
  call write_separator (u)
  call evt%base_write (u, testflag = testflag, show_set = .false.)
  if (evt%particle_set_exists)  call evt%particle_set%write &
       (u, summary = .true., compressed = .true., testflag = testflag)
  call write_separator (u)
  call evt%shower%settings%write (u)
end subroutine evt_shower_write
```

⟨*Shower: evt shower: TBP*⟩+≡
```
procedure :: connect => evt_shower_connect
```
⟨*Shower: procedures*⟩+≡
```
subroutine evt_shower_connect &
     (evt, process_instance, model, process_stack)
  class(evt_shower_t), intent(inout), target :: evt
  type(process_instance_t), intent(in), target :: process_instance
  class(model_data_t), intent(in), target :: model
  type(process_stack_t), intent(in), optional :: process_stack
  call evt%base_connect (process_instance, model, process_stack)
  call evt%make_rng (evt%process)
  if (allocated (evt%matching)) then
     call evt%matching%connect (process_instance, model, evt%shower)
  end if
end subroutine evt_shower_connect
```

Initialize the event transformation. This will be executed once during dispatching. The `model_hadrons` is supposed to be the SM variant that contains all hadrons that may be generated in the shower.

⟨*Shower: evt shower: TBP*⟩+≡
```
procedure :: init => evt_shower_init
```
⟨*Shower: procedures*⟩+≡
```
subroutine evt_shower_init (evt, model_hadrons, os_data)
  class(evt_shower_t), intent(out) :: evt
  type(model_t), intent(in), target :: model_hadrons
  type(os_data_t), intent(in) :: os_data
  evt%os_data = os_data
  evt%model_hadrons => model_hadrons
  evt%is_first_event = .true.
end subroutine evt_shower_init
```

Create RNG instances, spawned by the process object.

⟨*Shower: evt shower: TBP*⟩+≡
```
procedure :: make_rng => evt_shower_make_rng
```
⟨*Shower: procedures*⟩+≡
```
subroutine evt_shower_make_rng (evt, process)
  class(evt_shower_t), intent(inout) :: evt
  type(process_t), intent(inout) :: process
  class(rng_t), allocatable :: rng
  call process%make_rng (rng)
  call evt%shower%import_rng (rng)
  if (allocated (evt%matching)) then
     call process%make_rng (rng)
     call evt%matching%import_rng (rng)
  end if
end subroutine evt_shower_make_rng
```

Things we want to do for a new event before the whole event transformation chain is evaluated.

⟨*Shower: evt shower: TBP*⟩+≡
```
procedure :: prepare_new_event => evt_shower_prepare_new_event
```

⟨*Shower: procedures*⟩+≡

```
subroutine evt_shower_prepare_new_event (evt, i_mci, i_term)
  class(evt_shower_t), intent(inout) :: evt
  integer, intent(in) :: i_mci, i_term
  call evt%reset ()
  call evt%shower%prepare_new_event ()
end subroutine evt_shower_prepare_new_event
```

⟨*Shower: evt shower: TBP*⟩+≡

```
procedure :: first_event => evt_shower_first_event
```

⟨*Shower: procedures*⟩+≡

```
subroutine evt_shower_first_event (evt)
  class(evt_shower_t), intent(inout) :: evt
  double precision :: pdftest
  call msg_debug (D_TRANSFORMS, "evt_shower_first_event")
  associate (settings => evt%shower%settings)
     settings%hadron_collision = .false.
     !!! !!! !!! Workaround for PGF90 v16.1
     !!! if (all (evt%particle_set%prt(1:2)%flv%get_pdg_abs () <= 39)) then
     if (evt%particle_set%prt(1)%flv%get_pdg_abs () <= 39 .and. &
         evt%particle_set%prt(2)%flv%get_pdg_abs () <= 39) then
        settings%hadron_collision = .false.
     !!! else if (all (evt%particle_set%prt(1:2)%flv%get_pdg_abs () >= 100)) then
     else if (evt%particle_set%prt(1)%flv%get_pdg_abs () >= 100 .and. &
              evt%particle_set%prt(2)%flv%get_pdg_abs () >= 100) then
        settings%hadron_collision = .true.
     else
        call msg_fatal ("evt_shower didn't recognize beams setup")
     end if
     call msg_debug (D_TRANSFORMS, "hadron_collision", settings%hadron_collision)
     if (allocated (evt%matching)) then
        evt%matching%is_hadron_collision = settings%hadron_collision
        call evt%matching%first_event ()
     end if
     if (.not. settings%hadron_collision .and. settings%isr_active) then
        call msg_fatal ("?ps_isr_active is only intended for hadron-collisions")
     end if
     if (evt%pdf_data%type == STRF_LHAPDF5) then
        if (settings%isr_active .and. settings%hadron_collision) then
           call GetQ2max (0, pdftest)
           if (pdftest < epsilon (pdftest)) then
              call msg_bug ("ISR QCD shower enabled, but LHAPDF not " // &
                   "initialized," // LF // "     aborting simulation")
              return
           end if
        end if
     else if (evt%pdf_data%type == STRF_PDF_BUILTIN .and. &
              settings%method == PS_PYTHIA6) then
        call msg_fatal ("Builtin PDFs cannot be used for PYTHIA showers," &
             // LF // "     aborting simulation")
        return
     end if
  end associate
```

```
    evt%is_first_event = .false.
  end subroutine evt_shower_first_event
```

Here we take the particle set from the previous event transform (assuming that there is always one) and apply the shower algorithm. The result is stored in the event transform of the current object. We always return a probability of unity as we don't have the analytic weight of the combination of shower, MLM matching and hadronization. A subdivision into multiple event transformations is under construction. Invalid or vetoed events have to be discarded by the caller which is why we mark the particle set as invalid. This procedure directly takes the (MLM) matching into account.

⟨*Shower: evt shower: TBP*⟩+≡

```
  procedure :: generate_weighted => evt_shower_generate_weighted
```

⟨*Shower: procedures*⟩+≡

```
  subroutine evt_shower_generate_weighted (evt, probability)
    class(evt_shower_t), intent(inout) :: evt
    real(default), intent(inout) :: probability
    logical :: valid, vetoed
    integer :: i_term
    real(default) :: fac_scale
    call msg_debug (D_TRANSFORMS, "evt_shower_generate_weighted")
    if (signal_is_pending ())  return
    i_term = 1
    evt%particle_set = evt%previous%particle_set
    valid = .true.;  vetoed = .false.
    fac_scale = evt%process_instance%get_fac_scale (i_term)
    if (evt%is_first_event) then
       call evt%first_event ()
    end if
    call evt%shower%import_particle_set &
         (evt%particle_set, evt%os_data, fac_scale)
    if (allocated (evt%matching)) then
       call evt%matching%before_shower (evt%particle_set, vetoed)
       if (msg_level(D_TRANSFORMS) >= DEBUG) then
          call msg_debug (D_TRANSFORMS, "Matching before generate emissions")
          call evt%matching%write ()
       end if
    end if
    if (.not. (vetoed .or. POWHEG_TESTING)) then
       if (evt%shower%settings%method == PS_PYTHIA6 .or. &
           evt%shower%settings%hadronization_active) then
          call assure_heprup (evt%particle_set)
       end if
       call evt%shower%generate_emissions (valid)
    end if
    probability = 1
    evt%particle_set_exists = valid .and. .not. vetoed
  end subroutine evt_shower_generate_weighted
```

Here, we fill the particle set with the partons from the shower. The factorization parameters are irrelevant. We make a sanity check that the initial energy lands

either in the outgoing particles or add to the beam remnant.

⟨*Shower: evt shower: TBP*⟩+≡

```
procedure :: make_particle_set => evt_shower_make_particle_set
```

⟨*Shower: procedures*⟩+≡

```
subroutine evt_shower_make_particle_set &
     (evt, factorization_mode, keep_correlations, r)
  class(evt_shower_t), intent(inout) :: evt
  integer, intent(in) :: factorization_mode
  logical, intent(in) :: keep_correlations
  real(default), dimension(:), intent(in), optional :: r
  type(vector4_t) :: sum_vec_in, sum_vec_out, sum_vec_beamrem, &
       sum_vec_beamrem_before
  logical :: vetoed, sane
  if (evt%particle_set_exists) then
     vetoed = .false.
     sum_vec_beamrem_before = sum (evt%particle_set%prt%p, &
          mask=evt%particle_set%prt%get_status () == PRT_BEAM_REMNANT)
     call evt%shower%make_particle_set (evt%particle_set, &
          evt%model, evt%model_hadrons)
     if (allocated (evt%matching)) then
        call evt%matching%after_shower (evt%particle_set, vetoed)
     end if
     if (debug_active (D_TRANSFORMS)) then
        call msg_debug (D_TRANSFORMS, &
             "Shower: obtained particle set after shower + matching")
        call evt%particle_set%write (summary = .true., compressed = .true.)
     end if
     sum_vec_in = sum (evt%particle_set%prt%p, &
          mask=evt%particle_set%prt%get_status () == PRT_INCOMING)
     sum_vec_out = sum (evt%particle_set%prt%p, &
          mask=evt%particle_set%prt%get_status () == PRT_OUTGOING)
     sum_vec_beamrem = sum (evt%particle_set%prt%p, &
          mask=evt%particle_set%prt%get_status () == PRT_BEAM_REMNANT)
     sum_vec_beamrem = sum_vec_beamrem - sum_vec_beamrem_before
     sane = abs(sum_vec_out%p(0) - sum_vec_in%p(0)) < &
          sum_vec_in%p(0) / 10 .or. &
          abs((sum_vec_out%p(0) + sum_vec_beamrem%p(0)) - sum_vec_in%p(0)) < &
          sum_vec_in%p(0) / 10
     sane = .true.
     evt%particle_set_exists = .not. vetoed .and. sane
  end if
end subroutine evt_shower_make_particle_set
```

⟨*Shower: evt shower: TBP*⟩+≡

```
procedure :: contains_powheg_matching => evt_shower_contains_powheg_matching
```

⟨*Shower: procedures*⟩+≡

```
function evt_shower_contains_powheg_matching (evt) result (val)
   logical :: val
   class(evt_shower_t), intent(in) :: evt
   val = .false.
   if (allocated (evt%matching)) &
      val = evt%matching%get_method () == "POWHEG"
```

```
    end function evt_shower_contains_powheg_matching
```

⟨*Shower: evt shower: TBP*⟩+≡

```
    procedure :: disable_powheg_matching => evt_shower_disable_powheg_matching
```

⟨*Shower: procedures*⟩+≡

```
    subroutine evt_shower_disable_powheg_matching (evt)
       class(evt_shower_t), intent(inout) :: evt
       select type (matching => evt%matching)
       type is (powheg_matching_t)
          matching%active = .false.
       class default
          call msg_fatal ("Trying to disable powheg but no powheg matching is allocated!")
       end select
    end subroutine evt_shower_disable_powheg_matching
```

⟨*Shower: evt shower: TBP*⟩+≡

```
    procedure :: enable_powheg_matching => evt_shower_enable_powheg_matching
```

⟨*Shower: procedures*⟩+≡

```
    subroutine evt_shower_enable_powheg_matching (evt)
       class(evt_shower_t), intent(inout) :: evt
       select type (matching => evt%matching)
       type is (powheg_matching_t)
          matching%active = .true.
       class default
          call msg_fatal ("Trying to enable powheg but no powheg matching is allocated!")
       end select
    end subroutine evt_shower_enable_powheg_matching
```

⟨*Shower: evt shower: TBP*⟩+≡

```
    procedure :: final => evt_shower_final
```

⟨*Shower: procedures*⟩+≡

```
    subroutine evt_shower_final (evt)
      class(evt_shower_t), intent(inout) :: evt
      call evt%base_final ()
      if (allocated (evt%matching))  call evt%matching%final ()
    end subroutine evt_shower_final
```

32.5.3 Unit tests

Test module, followed by the corresponding implementation module.

⟨`shower_ut.f90`⟩≡

⟨*File header*⟩

```
  module shower_ut
    use unit_tests
    use shower_uti
```

⟨*Standard module head*⟩

⟨*Shower: public test*⟩

```
contains
```

⟨*Shower: test driver*⟩

```
end module shower_ut
```

⟨shower_uti.f90⟩≡

⟨*File header*⟩

```
module shower_uti
```

⟨*Use kinds*⟩
⟨*Use strings*⟩

```
  use format_utils, only: write_separator
  use os_interface
  use sm_qcd
  use physics_defs, only: BORN
  use model_data
  use state_matrices, only: FM_IGNORE_HELICITY
  use process_libraries
  use rng_base
  use rng_tao
  use mci_base
  use mci_midpoint
  use phs_base
  use phs_single
  use prc_core
  use prc_omega
  use variables
  use models
  use event_transforms
  use tauola_interface !NODEP!

  use process, only: process_t
  use instances, only: process_instance_t

  use pdf
  use shower_base
  use shower_core

  use shower
```

⟨*Standard module head*⟩

⟨*Shower: test declarations*⟩

```
contains
```

⟨*Shower: tests*⟩

```
end module shower_uti
```

API: driver for the unit tests below.

⟨*Shower: public test*⟩≡
```
public :: shower_test
```

⟨*Shower: test driver*⟩≡
```
subroutine shower_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨Shower: execute tests⟩
end subroutine shower_test
```

Testbed

This sequence sets up a two-jet process, ready for generating events.

⟨*Shower: tests*⟩≡
⟨*setup testbed*⟩

⟨*setup testbed*⟩≡
```
subroutine setup_testbed &
     (prefix, os_data, lib, model_list, process, process_instance)
  type(string_t), intent(in) :: prefix
  type(os_data_t), intent(out) :: os_data
  type(process_library_t), intent(out), target :: lib
  type(model_list_t), intent(out) :: model_list
  class(model_data_t), pointer :: model
  type(model_t), pointer :: model_tmp
  type(process_t), target, intent(out) :: process
  type(process_instance_t), target, intent(out) :: process_instance
  type(var_list_t), pointer :: model_vars
  type(string_t) :: model_name, libname, procname, run_id
  type(process_def_entry_t), pointer :: entry
  type(string_t), dimension(:), allocatable :: prt_in, prt_out
  type(qcd_t) :: qcd
  class(rng_factory_t), allocatable :: rng_factory
  class(prc_core_t), allocatable :: core_template
  class(mci_t), allocatable :: mci_template
  class(phs_config_t), allocatable :: phs_config_template
  real(default) :: sqrts

  model_name = "SM"
  libname = prefix // "_lib"
  procname = prefix // "p"
  run_id = "1"

  call os_data_init (os_data)
  allocate (rng_tao_factory_t :: rng_factory)
  allocate (model_tmp)
  call model_list%read_model (model_name, model_name // ".mdl", &
       os_data, model_tmp)
  model_vars => model_tmp%get_var_list_ptr ()
  call model_vars%set_real (var_str ("me"), 0._default, &
       is_known = .true.)
  model => model_tmp
```

```
call lib%init (libname)

allocate (prt_in (2), source = [var_str ("e-"), var_str ("e+")])
allocate (prt_out (2), source = [var_str ("d"), var_str ("dbar")])

allocate (entry)
call entry%init (procname, model, n_in = 2, n_components = 1)
call omega_make_process_component (entry, 1, &
     model_name, prt_in, prt_out, &
     report_progress=.true.)
call lib%append (entry)

call lib%configure (os_data)
call lib%write_makefile (os_data, force = .true.)
call lib%clean (os_data, distclean = .false.)
call lib%write_driver (force = .true.)
call lib%load (os_data)

call process%init (procname, run_id, lib, os_data, &
     qcd, rng_factory, model)

allocate (prc_omega_t :: core_template)
allocate (mci_midpoint_t :: mci_template)
allocate (phs_single_config_t :: phs_config_template)

model => process%get_model_ptr ()

select type (core_template)
type is (prc_omega_t)
   call core_template%set_parameters (model = model)
end select
call process%core_manager_register (BORN, 1, var_str ("omega"))
call process%allocate_cm_arrays (1)
call process%allocate_core (1, core_template)
call process%init_cores ()
call process%init_component &
   (1, .true., mci_template, phs_config_template)

sqrts = 1000
call process%setup_beams_sqrts (sqrts, i_core = 1)
call process%configure_phs ()
call process%setup_mci ()
call process%setup_terms ()

call process_instance%init (process)
call process_instance%integrate (1, 1, 1000)
call process%final_integration (1)

call process_instance%setup_event_data (i_core = 1)
call process_instance%init_simulation (1)
call process_instance%generate_weighted_event (1)
call process_instance%evaluate_event_data ()
```

```
end subroutine setup_testbed
```

Trivial Test

We generate a two-jet event and shower it using default settings, i.e. in disabled mode.

⟨*Shower: execute tests*⟩≡

```
call test (shower_1, "shower_1", &
     "disabled shower", &
     u, results)
```

⟨*Shower: test declarations*⟩≡

```
public :: shower_1
```

⟨*Shower: tests*⟩+≡

```
subroutine shower_1 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  type(process_library_t), target :: lib
  type(model_list_t) :: model_list
  class(model_data_t), pointer :: model
  type(model_t), pointer :: model_hadrons
  type(process_t), target :: process
  type(process_instance_t), target :: process_instance
  type(pdf_data_t) :: pdf_data
  integer :: factorization_mode
  logical :: keep_correlations
  class(evt_t), allocatable, target :: evt_trivial
  class(evt_t), allocatable, target :: evt_shower
  type(shower_settings_t) :: settings
  type(taudec_settings_t) :: taudec_settings

  write (u, "(A)")  "* Test output: shower_1"
  write (u, "(A)")  "*   Purpose: Two-jet event with disabled shower"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize environment"
  write (u, "(A)")

  call syntax_model_file_init ()
  call os_data_init (os_data)
  call model_list%read_model &
       (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"), &
       os_data, model_hadrons)
  call setup_testbed (var_str ("shower_1"), &
       os_data, lib, model_list, process, process_instance)

  write (u, "(A)")  "* Set up trivial transform"
  write (u, "(A)")

  allocate (evt_trivial_t :: evt_trivial)
  model => process%get_model_ptr ()
  call evt_trivial%connect (process_instance, model)
  call evt_trivial%prepare_new_event (1, 1)
```

```
  call evt_trivial%generate_unweighted ()

  factorization_mode = FM_IGNORE_HELICITY
  keep_correlations = .false.
  call evt_trivial%make_particle_set (factorization_mode, keep_correlations)

  select type (evt_trivial)
  type is (evt_trivial_t)
     call evt_trivial%write (u)
     call write_separator (u, 2)
  end select

  write (u, "(A)")
  write (u, "(A)")  "* Set up shower event transform"
  write (u, "(A)")

  allocate (evt_shower_t :: evt_shower)
  select type (evt_shower)
  type is (evt_shower_t)
     call evt_shower%init (model_hadrons, os_data)
     allocate (shower_t :: evt_shower%shower)
     call evt_shower%shower%init (settings, taudec_settings, pdf_data)
     call evt_shower%connect (process_instance, model)
  end select

  evt_trivial%next => evt_shower
  evt_shower%previous => evt_trivial

  call evt_shower%prepare_new_event (1, 1)
  call evt_shower%generate_unweighted ()
  call evt_shower%make_particle_set (factorization_mode, keep_correlations)

  select type (evt_shower)
  type is (evt_shower_t)
     call evt_shower%write (u)
     call write_separator (u, 2)
  end select

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call evt_shower%final ()
  call evt_trivial%final ()
  call process_instance%final ()
  call process%final ()
  call lib%final ()
  call model_hadrons%final ()
  deallocate (model_hadrons)
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: shower_1"

end subroutine shower_1
```

FSR Shower

We generate a two-jet event and shower it with the Whizard FSR shower.

⟨*Shower: execute tests*⟩+≡

```
    call test (shower_2, "shower_2", &
         "final-state shower", &
         u, results)
```

⟨*Shower: test declarations*⟩+≡

```
    public :: shower_2
```

⟨*Shower: tests*⟩+≡

```
    subroutine shower_2 (u)
      integer, intent(in) :: u
      type(os_data_t) :: os_data
      type(process_library_t), target :: lib
      type(model_list_t) :: model_list
      type(model_t), pointer :: model_hadrons
      class(model_data_t), pointer :: model
      type(process_t), target :: process
      type(process_instance_t), target :: process_instance
      integer :: factorization_mode
      logical :: keep_correlations
      type(pdf_data_t) :: pdf_data
      class(evt_t), allocatable, target :: evt_trivial
      class(evt_t), allocatable, target :: evt_shower
      type(shower_settings_t) :: settings
      type(taudec_settings_t) :: taudec_settings

      write (u, "(A)")  "* Test output: shower_2"
      write (u, "(A)")  "*   Purpose: Two-jet event with FSR shower"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize environment"
      write (u, "(A)")

      call syntax_model_file_init ()
      call os_data_init (os_data)
      call model_list%read_model &
           (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"), &
           os_data, model_hadrons)
      call setup_testbed (var_str ("shower_2"), &
           os_data, lib, model_list, process, process_instance)
      model => process%get_model_ptr ()

      write (u, "(A)")  "* Set up trivial transform"
      write (u, "(A)")

      allocate (evt_trivial_t :: evt_trivial)
      call evt_trivial%connect (process_instance, model)
      call evt_trivial%prepare_new_event (1, 1)
      call evt_trivial%generate_unweighted ()
```

```
  factorization_mode = FM_IGNORE_HELICITY
  keep_correlations = .false.
  call evt_trivial%make_particle_set (factorization_mode, keep_correlations)

  select type (evt_trivial)
  type is (evt_trivial_t)
     call evt_trivial%write (u)
     call write_separator (u, 2)
  end select

  write (u, "(A)")
  write (u, "(A)")  "* Set up shower event transform"
  write (u, "(A)")

  settings%fsr_active = .true.

  allocate (evt_shower_t :: evt_shower)
  select type (evt_shower)
  type is (evt_shower_t)
     call evt_shower%init (model_hadrons, os_data)
     allocate (shower_t :: evt_shower%shower)
     call evt_shower%shower%init (settings, taudec_settings, pdf_data)
     call evt_shower%connect (process_instance, model)
  end select

  evt_trivial%next => evt_shower
  evt_shower%previous => evt_trivial

  call evt_shower%prepare_new_event (1, 1)
  call evt_shower%generate_unweighted ()
  call evt_shower%make_particle_set (factorization_mode, keep_correlations)

  select type (evt_shower)
  type is (evt_shower_t)
     call evt_shower%write (u, testflag = .true.)
     call write_separator (u, 2)
  end select

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call evt_shower%final ()
  call evt_trivial%final ()
  call process_instance%final ()
  call process%final ()
  call lib%final ()
  call model_hadrons%final ()
  deallocate (model_hadrons)
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: shower_2"

end subroutine shower_2
```

32.6 Fixed Order NLO Events

This section deals with the generation of weighted event samples which take into account next-to-leading order corrections. An approach generating unweighted events is not possible here, because negative weights might occur due to subtraction. Note that the events produced this way are not physical in the sense that they will not keep NLO-accuracy when interfaced to a parton shower. They are rather useful for theoretical consistency checks and a fast estimate of NLO effects.

We generate NLO events in the following way: First, the integration is carried out using the complete divergence-subtracted NLO matrix element. In the subsequent simulation, N-particle kinematics are generated using $\mathcal{B}+\mathcal{V}+\mathcal{C}$ as weight. After that, the program loops over all singular regions and for each of them generates an event with $N+1$-particle kinematics. The weight for those events corresponds to the real matrix element $\mathcal{R}^\alpha$ evaluated at the α-region's emitter's phase space point, multiplied with S_α. This procedure is implemented using the `evt_nlo` transform.

⟨`evt_nlo.f90`⟩≡

```
⟨File header⟩

module evt_nlo

⟨Use kinds⟩
⟨Use strings⟩
  use io_units, only: given_output_unit
  use constants
  use lorentz
  use diagnostics
  use physics_defs, only: NLO_REAL
  use sm_qcd
  use model_data
  use particles
  use instances, only: process_instance_t
  ! TODO (cw-2016-09-16): Ideally, only pcm_base
  use pcm, only: pcm_nlo_t, pcm_instance_nlo_t
  use process_stacks
  use event_transforms

  use phs_fks, only: phs_fks_t, phs_fks_generator_t
  use phs_fks, only: phs_identifier_t, phs_point_set_t
  use resonances, only: resonance_contributors_t
  use fks_regions, only: region_data_t

⟨Standard module head⟩

⟨Evt Nlo: public⟩

⟨Evt Nlo: public parameters⟩
```

⟨*Evt Nlo: types*⟩

```
contains
```

⟨*Evt Nlo: procedures*⟩

```
end module evt_nlo
```

⟨*Evt Nlo: types*⟩≡

```
type :: nlo_event_deps_t
   logical :: cm_frame = .true.
   type(phs_point_set_t) :: p_born_cms
   type(phs_point_set_t) :: p_born_lab
   type(phs_point_set_t) :: p_real_cms
   type(phs_point_set_t) :: p_real_lab
   type(resonance_contributors_t), dimension(:), allocatable :: contributors
   type(phs_identifier_t), dimension(:), allocatable :: phs_identifiers
   integer, dimension(:), allocatable :: alr_to_i_con
   integer :: n_phs = 0
end type nlo_event_deps_t
```

This event transformation is for the generation of fixed-order NLO events. It takes an event with Born kinematics and creates $N_\alpha + 1$ modified weighted events. The first one has Born kinematics and its weight is the sum of Born, Real and subtraction matrix elements. The other N_α events have a weight which is equal to the real matrix element, evaluated with the phase space corresponding to the emitter of the α-region. All NLO event objects share the same event transformation. For this reason, we save the particle set of the current α-region in the array `particle_set_radiated`. Otherwise it would be unretrievable if the usual particle set of the event object was used.@

⟨*Evt Nlo: public parameters*⟩≡

```
integer, parameter, public :: EVT_NLO_UNDEFINED = 0
integer, parameter, public :: EVT_NLO_SEPARATE_BORNLIKE = 1
integer, parameter, public :: EVT_NLO_SEPARATE_REAL = 2
integer, parameter, public :: EVT_NLO_COMBINED = 3
```

⟨*Evt Nlo: public*⟩≡

```
public :: evt_nlo_t
```

⟨*Evt Nlo: types*⟩+≡

```
type, extends (evt_t) :: evt_nlo_t
  type(phs_fks_generator_t) :: phs_fks_generator
  real(default) :: sqme_rad = zero
  integer :: i_evaluation = 0
  integer :: weight_multiplier = 1
  type(particle_set_t), dimension(:), allocatable :: particle_set_radiated
  type(qcd_t), pointer :: qcd => null ()
  type(nlo_event_deps_t) :: event_deps
  integer :: mode = EVT_NLO_UNDEFINED
  integer, dimension(:), allocatable :: &
     i_evaluation_to_i_phs, i_evaluation_to_emitter, &
     i_evaluation_to_i_term
  logical :: keep_failed_events = .false.
```

```
  integer :: selected_i_flv = 0
contains
⟨Evt Nlo: evt nlo: TBP⟩
end type evt_nlo_t
```

⟨*Evt Nlo: evt nlo: TBP*⟩≡

```
procedure :: write_name => evt_nlo_write_name
```

⟨*Evt Nlo: procedures*⟩≡

```
subroutine evt_nlo_write_name (evt, unit)
  class(evt_nlo_t), intent(in) :: evt
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)") "Event transform: NLO"
end subroutine evt_nlo_write_name
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: write => evt_nlo_write
```

⟨*Evt Nlo: procedures*⟩+≡

```
subroutine evt_nlo_write (evt, unit, verbose, more_verbose, testflag)
  class(evt_nlo_t), intent(in) :: evt
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose, more_verbose, testflag
end subroutine evt_nlo_write
```

Connects the event transform to the process. Here also the phase space is set up by making `real_kinematics` point to the corresponding object in the `pcm_instance`.

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: connect => evt_nlo_connect
```

⟨*Evt Nlo: procedures*⟩+≡

```
subroutine evt_nlo_connect (evt, process_instance, model, process_stack)
  class(evt_nlo_t), intent(inout), target :: evt
  type(process_instance_t), intent(in), target :: process_instance
  class(model_data_t), intent(in), target :: model
  type(process_stack_t), intent(in), optional :: process_stack
  call msg_debug (D_TRANSFORMS, "evt_nlo_connect")
  call evt%base_connect (process_instance, model, process_stack)
  select type (pcm => process_instance%pcm)
  class is (pcm_instance_nlo_t)
     select type (config => pcm%config)
     type is (pcm_nlo_t)
        call config%setup_phs_generator (pcm, evt%phs_fks_generator, &
           process_instance%get_sqrts ())
        call evt%set_i_evaluation_mappings (config%region_data, &
           pcm%real_kinematics%alr_to_i_phs)
     end select
  end select
  call evt%set_mode (process_instance)
  call evt%setup_general_event_kinematics (process_instance)
  if (evt%mode > EVT_NLO_SEPARATE_BORNLIKE) &
```

```
      call evt%setup_real_event_kinematics (process_instance)
    call msg_debug2 (D_TRANSFORMS, "evt_nlo_connect: success")
  end subroutine evt_nlo_connect
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
  procedure :: set_i_evaluation_mappings => evt_nlo_set_i_evaluation_mappings
```

⟨*Evt Nlo: procedures*⟩+≡

```
  subroutine evt_nlo_set_i_evaluation_mappings (evt, reg_data, alr_to_i_phs)
    class(evt_nlo_t), intent(inout) :: evt
    type(region_data_t), intent(in) :: reg_data
    integer, intent(in), dimension(:) :: alr_to_i_phs
    integer :: n_phs, alr
    integer :: i_evaluation, i_phs, emitter
    logical :: checked
    type :: registered_triple_t
      integer, dimension(2) :: phs_em
      type(registered_triple_t), pointer :: next => null ()
    end type registered_triple_t
    type(registered_triple_t), allocatable, target :: check_list
    i_evaluation = 1
    n_phs = reg_data%n_phs
    evt%weight_multiplier = n_phs + 1
    allocate (evt%i_evaluation_to_i_phs (n_phs), source = 0)
    allocate (evt%i_evaluation_to_emitter (n_phs), source = -1)
    allocate (evt%i_evaluation_to_i_term (0 : n_phs), source = 0)
    do alr = 1, reg_data%n_regions
       i_phs = alr_to_i_phs (alr)
       emitter = reg_data%regions(alr)%emitter
       call search_check_list (checked)
       if (.not. checked) then
          evt%i_evaluation_to_i_phs (i_evaluation) = i_phs
          evt%i_evaluation_to_emitter (i_evaluation) = emitter
          i_evaluation = i_evaluation + 1
       end if
    end do
    call fill_i_evaluation_to_i_term ()
    if (.not. (all (evt%i_evaluation_to_i_phs > 0) &
       .and. all (evt%i_evaluation_to_emitter > -1))) then
       call msg_fatal ("evt_nlo: Inconsistent mappings!")
    else
       if (debug2_active (D_TRANSFORMS)) then
          print *, 'evt_nlo Mappings, i_evaluation -> '
          print *, 'i_phs: ', evt%i_evaluation_to_i_phs
          print *, 'emitter: ', evt%i_evaluation_to_emitter
       end if
    end if
  contains
    subroutine fill_i_evaluation_to_i_term ()
      integer :: i_term, i_evaluation, term_emitter
      !!! First find subtraction component
      i_evaluation = 1
      do i_term = 1, evt%process%get_n_terms ()
         if (evt%process_instance%term(i_term)%nlo_type /= NLO_REAL) cycle
```

```
      term_emitter = evt%process_instance%term(i_term)%k_term%emitter
      if (term_emitter < 0) then
         evt%i_evaluation_to_i_term (0) = i_term
      else if (evt%i_evaluation_to_emitter(i_evaluation) == term_emitter) then
         evt%i_evaluation_to_i_term (i_evaluation) = i_term
         i_evaluation = i_evaluation + 1
      end if
    end do
  end subroutine fill_i_evaluation_to_i_term

  subroutine search_check_list (found)
    logical, intent(out) :: found
    type(registered_triple_t), pointer :: current_triple => null ()
    if (allocated (check_list)) then
       current_triple => check_list
       do
          if (all (current_triple%phs_em == [i_phs, emitter])) then
             found = .true.
             exit
          end if
          if (.not. associated (current_triple%next)) then
             allocate (current_triple%next)
             current_triple%next%phs_em = [i_phs, emitter]
             found = .false.
             exit
          else
             current_triple => current_triple%next
          end if
       end do
    else
       allocate (check_list)
       check_list%phs_em = [i_phs, emitter]
       found = .false.
    end if
  end subroutine search_check_list
end subroutine evt_nlo_set_i_evaluation_mappings
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
  procedure :: get_i_phs => evt_nlo_get_i_phs
```

⟨*Evt Nlo: procedures*⟩+≡

```
  function evt_nlo_get_i_phs (evt) result (i_phs)
    integer :: i_phs
    class(evt_nlo_t), intent(in) :: evt
    i_phs = evt%i_evaluation_to_i_phs (evt%i_evaluation)
  end function evt_nlo_get_i_phs
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
  procedure :: get_emitter => evt_nlo_get_emitter
```

⟨*Evt Nlo: procedures*⟩+≡

```
  function evt_nlo_get_emitter (evt) result (emitter)
    integer :: emitter
    class(evt_nlo_t), intent(in) :: evt
```

```
    emitter = evt%i_evaluation_to_emitter (evt%i_evaluation)
  end function evt_nlo_get_emitter
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
  procedure :: get_i_term => evt_nlo_get_i_term
```

⟨*Evt Nlo: procedures*⟩+≡

```
  function evt_nlo_get_i_term (evt) result (i_term)
    integer :: i_term
    class(evt_nlo_t), intent(in) :: evt
    if (evt%mode >= EVT_NLO_SEPARATE_REAL) then
       i_term = evt%i_evaluation_to_i_term (evt%i_evaluation)
    else
       i_term = evt%process_instance%get_first_active_i_term ()
    end if
  end function evt_nlo_get_i_term
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
  procedure :: copy_previous_particle_set => evt_nlo_copy_previous_particle_set
```

⟨*Evt Nlo: procedures*⟩+≡

```
  subroutine evt_nlo_copy_previous_particle_set (evt)
    class(evt_nlo_t), intent(inout) :: evt
    if (associated (evt%previous)) then
       evt%particle_set = evt%previous%particle_set
    else
       call msg_fatal ("evt_nlo requires one preceeding evt_trivial!")
    end if
  end subroutine evt_nlo_copy_previous_particle_set
```

The event transform has a variable which counts the number of times it has already been called for one generation point. If `i_evaluation` is zero, this means that `evt_nlo_generate` is called for the first time, so that the generation of an N-particle event is required. In all other cases, emission events are generated. Note that for the first event, the computed weights are added to `probability`, which at this point is equal to $\mathcal{B}+\mathcal{V}$, whereas for all other runs `probability` is replaced. To keep $<\sum w_i>= N\times\sigma$ as it is for weighted LO events, we have to multiply by $N_{\rm phs}+1$ since the cross section is distributed over the real and Born subevents.

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
  procedure :: generate_weighted => evt_nlo_generate_weighted
```

⟨*Evt Nlo: procedures*⟩+≡

```
  subroutine evt_nlo_generate_weighted (evt, probability)
    class(evt_nlo_t), intent(inout) :: evt
    real(default), intent(inout) :: probability
    real(default) :: weight
    call print_debug_info ()
    if (evt%mode > EVT_NLO_SEPARATE_BORNLIKE) then
       if (evt%i_evaluation == 0) then
          call evt%reset_phs_identifiers ()
          call evt%evaluate_real_kinematics ()
          weight = evt%compute_subtraction_weights ()
```

```
            if (evt%mode == EVT_NLO_SEPARATE_REAL) then
               probability = weight
            else
               probability = probability + weight
            end if
         else
            call evt%compute_real ()
            probability = evt%sqme_rad
         end if
         call msg_debug2 (D_TRANSFORMS, "event weight multiplier:", evt%weight_multiplier)
         probability = probability * evt%weight_multiplier
      end if
      call msg_debug (D_TRANSFORMS, "probability (after)", probability)
      evt%particle_set_exists = .true.
   contains
      function status_code_to_string (mode) result (smode)
         type(string_t) :: smode
         integer, intent(in) :: mode
         select case (mode)
         case (EVT_NLO_UNDEFINED)
            smode = var_str ("Undefined")
         case (EVT_NLO_SEPARATE_BORNLIKE)
            smode = var_str ("Born-like")
         case (EVT_NLO_SEPARATE_REAL)
            smode = var_str ("Real")
         case (EVT_NLO_COMBINED)
            smode = var_str ("Combined")
         end select
      end function status_code_to_string

      subroutine print_debug_info ()
         call msg_debug (D_TRANSFORMS, "evt_nlo_generate_weighted")
         call msg_debug (D_TRANSFORMS, char ("mode: " // status_code_to_string (evt%mode)))
         call msg_debug (D_TRANSFORMS, "probability (before)", probability)
         call msg_debug (D_TRANSFORMS, "evt%i_evaluation", evt%i_evaluation)
         if (debug2_active (D_TRANSFORMS)) then
            if (evt%mode > EVT_NLO_SEPARATE_BORNLIKE) then
               if (evt%i_evaluation == 0) then
                  print *, 'Evaluate subtraction component'
               else
                  print *, 'Evaluate radiation component'
               end if
            end if
         end if
      end subroutine print_debug_info
   end subroutine evt_nlo_generate_weighted
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
   procedure :: reset_phs_identifiers => evt_nlo_reset_phs_identifiers
```

⟨*Evt Nlo: procedures*⟩+≡

```
   subroutine evt_nlo_reset_phs_identifiers (evt)
      class(evt_nlo_t), intent(inout) :: evt
      evt%event_deps%phs_identifiers%evaluated = .false.
```

```
end subroutine evt_nlo_reset_phs_identifiers
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: make_particle_set => evt_nlo_make_particle_set
```

⟨*Evt Nlo: procedures*⟩+≡

```
subroutine evt_nlo_make_particle_set &
     (evt, factorization_mode, keep_correlations, r)
  class(evt_nlo_t), intent(inout) :: evt
  integer, intent(in) :: factorization_mode
  logical, intent(in) :: keep_correlations
  real(default), dimension(:), intent(in), optional :: r
  if (evt%mode >= EVT_NLO_SEPARATE_BORNLIKE) then
     select type (config => evt%process_instance%pcm%config)
     type is (pcm_nlo_t)
        if (evt%i_evaluation > 0) then
           call make_factorized_particle_set (evt, factorization_mode, &
                keep_correlations, r, evt%get_i_term (), &
                config%qn_real(:, evt%selected_i_flv))
        else
           call make_factorized_particle_set (evt, factorization_mode, &
                keep_correlations, r, evt%get_i_term (), &
                config%qn_born(:, evt%selected_i_flv))
        end if
     end select
  else
     call make_factorized_particle_set (evt, factorization_mode, &
          keep_correlations, r)
  end if
end subroutine evt_nlo_make_particle_set
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: keep_and_boost_born_particle_set => &
     evt_nlo_keep_and_boost_born_particle_set
```

⟨*Evt Nlo: procedures*⟩+≡

```
subroutine evt_nlo_keep_and_boost_born_particle_set (evt, i_event)
  class(evt_nlo_t), intent(inout) :: evt
  integer, intent(in) :: i_event
  evt%particle_set_radiated(i_event) = evt%particle_set
  if (evt%event_deps%cm_frame) then
     evt%event_deps%p_born_cms%phs_point(1) = &
        evt%particle_set%get_in_and_out_momenta ()
     evt%event_deps%p_born_lab%phs_point(1) = &
        evt%boost_to_lab (evt%event_deps%p_born_cms%phs_point(1))
     call evt%particle_set_radiated(i_event)%replace_incoming_momenta &
        (evt%event_deps%p_born_lab%phs_point(1)%p)
     call evt%particle_set_radiated(i_event)%replace_outgoing_momenta &
        (evt%event_deps%p_born_lab%phs_point(1)%p)
  end if
end subroutine evt_nlo_keep_and_boost_born_particle_set
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: evaluate_real_kinematics => evt_nlo_evaluate_real_kinematics
```

⟨*Evt Nlo: procedures*⟩+≡

```
subroutine evt_nlo_evaluate_real_kinematics (evt)
  class(evt_nlo_t), intent(inout) :: evt
  integer :: alr, i_phs, i_con, emitter
  real(default), dimension(3) :: x_rad
  logical :: use_contributors
  integer :: i_term

  select type (pcm => evt%process_instance%pcm)
  class is (pcm_instance_nlo_t)
     x_rad = pcm%real_kinematics%x_rad
     associate (event_deps => evt%event_deps)
        i_term = evt%get_i_term ()
        event_deps%p_born_lab%phs_point(1) = &
             evt%process_instance%term(i_term)%connected%matrix%get_momenta ()
        event_deps%p_born_cms%phs_point(1) &
             = evt%boost_to_cms (event_deps%p_born_lab%phs_point(1))
        call evt%phs_fks_generator%set_sqrts_hat &
             (event_deps%p_born_cms%get_energy (1, 1))
        use_contributors = allocated (event_deps%contributors)
        do alr = 1, pcm%get_n_regions ()
           i_phs = pcm%real_kinematics%alr_to_i_phs(alr)
           if (event_deps%phs_identifiers(i_phs)%evaluated) cycle
           emitter = event_deps%phs_identifiers(i_phs)%emitter
           associate (generator => evt%phs_fks_generator)
              !!! TODO: (cw-2016-12-30): Replace by n_in
              if (emitter <= 2) then
                 call generator%prepare_generation (x_rad, i_phs, emitter, &
                      event_deps%p_born_cms%phs_point(1)%p, event_deps%phs_identifiers)
                 call generator%generate_isr (i_phs, &
                      event_deps%p_born_lab%phs_point(1)%p, &
                      event_deps%p_real_lab%phs_point(i_phs))
                 event_deps%p_real_cms%phs_point(i_phs) &
                      = evt%boost_to_cms (event_deps%p_real_lab%phs_point(i_phs))
              else
                 if (use_contributors) then
                    i_con = event_deps%alr_to_i_con(alr)
                    call generator%prepare_generation (x_rad, i_phs, emitter, &
                         event_deps%p_born_cms%phs_point(1)%p, &
                         event_deps%phs_identifiers, event_deps%contributors, i_con)
                    call generator%generate_fsr (emitter, i_phs, i_con, &
                         event_deps%p_born_cms%phs_point(1)%p, &
                         event_deps%p_real_cms%phs_point(i_phs))
                 else
                    call generator%prepare_generation (x_rad, i_phs, emitter, &
                         event_deps%p_born_cms%phs_point(1)%p, event_deps%phs_identifiers)
                    call generator%generate_fsr (emitter, i_phs, &
                         event_deps%p_born_cms%phs_point(1)%p, &
                         event_deps%p_real_cms%phs_point(i_phs))
                 end if
                 event_deps%p_real_lab%phs_point(i_phs) &
                      = evt%boost_to_lab (event_deps%p_real_cms%phs_point(i_phs))
              end if
           end associate
```

```
            call pcm%set_momenta (event_deps%p_born_lab%phs_point(1)%p, &
                 event_deps%p_real_lab%phs_point(i_phs)%p, i_phs)
            call pcm%set_momenta (event_deps%p_born_cms%phs_point(1)%p, &
                 event_deps%p_real_cms%phs_point(i_phs)%p, i_phs, cms = .true.)
            event_deps%phs_identifiers(i_phs)%evaluated = .true.
         end do
      end associate
   end select
end subroutine evt_nlo_evaluate_real_kinematics
```

This routine calls the evaluation of the singular regions only for the subtraction terms.

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: compute_subtraction_weights => evt_nlo_compute_subtraction_weights
```

⟨*Evt Nlo: procedures*⟩+≡

```
function evt_nlo_compute_subtraction_weights (evt) result (weight)
  class(evt_nlo_t), intent(inout) :: evt
  real(default) :: weight
  integer :: i_phs, i_term
  call msg_debug (D_TRANSFORMS, "evt_nlo_compute_subtraction_weights")
  weight = zero
  select type (pcm => evt%process_instance%pcm)
  class is (pcm_instance_nlo_t)
     associate (event_deps => evt%event_deps)
        i_phs = 1; i_term = evt%i_evaluation_to_i_term(0)
        call evt%process_instance%compute_sqme_rad (i_term, i_phs, .true.)
        weight = weight + evt%process_instance%get_sqme (i_term)
     end associate
  end select
end function evt_nlo_compute_subtraction_weights
```

This routine calls the evaluation of the singular regions only for emission matrix elements.

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: compute_real => evt_nlo_compute_real
```

⟨*Evt Nlo: procedures*⟩+≡

```
subroutine evt_nlo_compute_real (evt)
  class(evt_nlo_t), intent(inout) :: evt
  integer :: i_phs, i_term
  call msg_debug (D_TRANSFORMS, "evt_nlo_compute_real")
  i_phs = evt%get_i_phs ()
  i_term = evt%i_evaluation_to_i_term (evt%i_evaluation)
  select type (pcm => evt%process_instance%pcm)
  class is (pcm_instance_nlo_t)
     associate (event_deps => evt%event_deps)
        call evt%process_instance%compute_sqme_rad (i_term, i_phs, .false.)
        evt%sqme_rad = evt%process_instance%get_sqme (i_term)
     end associate
  end select
end subroutine evt_nlo_compute_real
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: boost_to_cms => evt_nlo_boost_to_cms
```

⟨*Evt Nlo: procedures*⟩+≡

```
function evt_nlo_boost_to_cms (evt, p_lab) result (p_cms)
  type(phs_point_t), intent(in) :: p_lab
  class(evt_nlo_t), intent(in) :: evt
  type(phs_point_t) :: p_cms
  type(lorentz_transformation_t) :: lt_lab_to_cms
  integer :: i_boost
  if (evt%event_deps%cm_frame) then
     lt_lab_to_cms = identity
  else
     if (evt%mode == EVT_NLO_COMBINED) then
        i_boost = 1
     else
        i_boost = evt%process_instance%select_i_term ()
     end if
     lt_lab_to_cms = evt%process_instance%get_boost_to_cms (i_boost)
  end if
  p_cms = lt_lab_to_cms * p_lab
end function evt_nlo_boost_to_cms
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: boost_to_lab => evt_nlo_boost_to_lab
```

⟨*Evt Nlo: procedures*⟩+≡

```
function evt_nlo_boost_to_lab (evt, p_cms) result (p_lab)
  type(phs_point_t) :: p_lab
  class(evt_nlo_t), intent(in) :: evt
  type(phs_point_t), intent(in) :: p_cms
  type(lorentz_transformation_t) :: lt_cms_to_lab
  integer :: i_boost
  if (.not. evt%event_deps%cm_frame) then
     lt_cms_to_lab = identity
  else
     if (evt%mode == EVT_NLO_COMBINED) then
        i_boost = 1
     else
        i_boost = evt%process_instance%select_i_term ()
     end if
     lt_cms_to_lab = evt%process_instance%get_boost_to_lab (i_boost)
  end if
  p_lab = lt_cms_to_lab * p_cms
end function evt_nlo_boost_to_lab
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: setup_general_event_kinematics => evt_nlo_setup_general_event_kinematics
```

⟨*Evt Nlo: procedures*⟩+≡

```
subroutine evt_nlo_setup_general_event_kinematics (evt, process_instance)
  class(evt_nlo_t), intent(inout) :: evt
  type(process_instance_t), intent(in) :: process_instance
  integer :: n_born
  associate (event_deps => evt%event_deps)
```

```
      event_deps%cm_frame = process_instance%is_cm_frame (1)
      select type (pcm => process_instance%pcm)
      type is (pcm_instance_nlo_t)
         n_born = pcm%get_n_born ()
      end select
      call event_deps%p_born_cms%init (n_born, 1)
      call event_deps%p_born_lab%init (n_born, 1)
   end associate
end subroutine evt_nlo_setup_general_event_kinematics
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: setup_real_event_kinematics => evt_nlo_setup_real_event_kinematics
```

⟨*Evt Nlo: procedures*⟩+≡

```
subroutine evt_nlo_setup_real_event_kinematics (evt, process_instance)
   class(evt_nlo_t), intent(inout) :: evt
   type(process_instance_t), intent(in) :: process_instance
   integer :: n_real, n_phs
   integer :: i_real
   associate (event_deps => evt%event_deps)
      select type (pcm => process_instance%pcm)
      class is (pcm_instance_nlo_t)
         n_real = pcm%get_n_real ()
      end select
      i_real = process_instance%get_associated_real ()
      select type (phs => process_instance%term(i_real)%k_term%phs)
      type is (phs_fks_t)
         !!! !!! !!! Workaround for ifort 16.0 standard-semantics bug
         allocate (event_deps%phs_identifiers (size (phs%phs_identifiers)))
         event_deps%phs_identifiers = phs%phs_identifiers
      end select
      n_phs = size (event_deps%phs_identifiers)
      call event_deps%p_real_cms%init (n_real, n_phs)
      call event_deps%p_real_lab%init (n_real, n_phs)
      select type (pcm => process_instance%pcm)
      type is (pcm_instance_nlo_t)
         select type (config => pcm%config)
         type is (pcm_nlo_t)
            if (allocated (config%region_data%alr_contributors)) then
               allocate (event_deps%contributors (size (config%region_data%alr_contributors)))
               event_deps%contributors = config%region_data%alr_contributors
            end if
            if (allocated (config%region_data%alr_to_i_contributor)) then
               allocate (event_deps%alr_to_i_con &
                  (size (config%region_data%alr_to_i_contributor)))
               event_deps%alr_to_i_con = config%region_data%alr_to_i_contributor
            end if
         end select
      end select
   end associate
end subroutine evt_nlo_setup_real_event_kinematics
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: set_mode => evt_nlo_set_mode
```

⟨*Evt Nlo: procedures*⟩+≡

```
subroutine evt_nlo_set_mode (evt, process_instance)
  class(evt_nlo_t), intent(inout) :: evt
  type(process_instance_t), intent(in) :: process_instance
  integer :: i_real
  select type (pcm => process_instance%pcm)
  type is (pcm_instance_nlo_t)
     select type (config => pcm%config)
     type is (pcm_nlo_t)
        if (config%settings%combined_integration) then
           evt%mode = EVT_NLO_COMBINED
        else
           i_real = evt%process_instance%get_associated_real ()
           if (i_real == evt%process%extract_fixed_mci ()) then
              evt%mode = EVT_NLO_SEPARATE_REAL
           else
              evt%mode = EVT_NLO_SEPARATE_BORNLIKE
           end if
        end if
     end select
  end select
end subroutine evt_nlo_set_mode
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: is_valid_event => evt_nlo_is_valid_event
```

⟨*Evt Nlo: procedures*⟩+≡

```
function evt_nlo_is_valid_event (evt, i_term) result (valid)
  logical :: valid
  class(evt_nlo_t), intent(in) :: evt
  integer, intent(in) :: i_term
  valid = evt%process_instance%term(i_term)%passed
end function evt_nlo_is_valid_event
```

⟨*Evt Nlo: evt nlo: TBP*⟩+≡

```
procedure :: prepare_new_event => evt_nlo_prepare_new_event
```

⟨*Evt Nlo: procedures*⟩+≡

```
subroutine evt_nlo_prepare_new_event (evt, i_mci, i_term)
  class(evt_nlo_t), intent(inout) :: evt
  integer, intent(in) :: i_mci, i_term
  real(default) :: s, x
  real(default) :: sqme_total
  real(default), dimension(:), allocatable :: sqme_flv
  integer :: i
  call evt%reset ()
  if (evt%i_evaluation > 0) return
  call evt%rng%generate (x)
  sqme_total = zero
  allocate (sqme_flv (evt%process_instance%term(1)%config%data%n_flv))
  sqme_flv = zero
  do i = 1, size (evt%process_instance%term)
     associate (term => evt%process_instance%term(i))
        sqme_total = sqme_total + real (sum ( &
```

```
            term%connected%matrix%get_matrix_element ()))
         sqme_flv = sqme_flv + real (term%connected%matrix%get_matrix_element ())
      end associate
   end do
   !!! Need absolute values to take into account negative weights
   x = x * abs (sqme_total)
   s = zero
   do i = 1, size (sqme_flv)
      s = s + abs (sqme_flv (i))
      if (s > x) then
         evt%selected_i_flv = i
         exit
      end if
   end do
   if (debug2_active (D_TRANSFORMS)) then
      call msg_print_color ("Selected i_flv: ", COL_GREEN)
      print *, evt%selected_i_flv
   end if
end subroutine evt_nlo_prepare_new_event
```

32.7 Complete Events

This module combines hard processes with decay chains, shower, and hadronization (not implemented yet) to complete events. It also manages the input and output of event records in various formats.

⟨events.f90⟩≡
⟨*File header*⟩

```
module events

⟨Use kinds⟩
⟨Use strings⟩
  use constants, only: one
  use io_units
  use format_utils, only: pac_fmt, write_separator
  use format_defs, only: FMT_12, FMT_19
  use numeric_utils
  use diagnostics
  use variables
  use expr_base
  use model_data
  use state_matrices, only: FM_IGNORE_HELICITY, &
       FM_SELECT_HELICITY, FM_FACTOR_HELICITY, FM_CORRELATED_HELICITY
  use particles
  use subevt_expr
  use rng_base
  use process, only: process_t
  use instances, only: process_instance_t
  use pcm, only: pcm_instance_nlo_t
  use process_stacks
  use event_base
```

```
  use event_transforms
  use decays
  use evt_nlo
```

⟨*Standard module head*⟩

⟨*Events: public*⟩

⟨*Events: types*⟩

⟨*Events: interfaces*⟩

```
contains
```

⟨*Events: procedures*⟩

```
end module events
```

32.7.1 Event configuration

The parameters govern the transformation of an event to a particle set.

The `safety_factor` reduces the acceptance probability for unweighting. If greater than one, excess events become less likely, but the reweighting efficiency also drops.

The `sigma` and `n` values, if nontrivial, allow for reweighting the events according to the requested `norm_mode`.

Various `parse_node_t` objects are taken from the SINDARIN input. They encode expressions that apply to the current event. The workspaces for evaluating those expressions are set up in the `event_expr_t` objects. Note that these are really pointers, so the actual nodes are not stored inside the event object.

⟨*Events: types*⟩≡

```
type :: event_config_t
   logical :: unweighted = .false.
   integer :: norm_mode = NORM_UNDEFINED
   integer :: factorization_mode = FM_IGNORE_HELICITY
   logical :: keep_correlations = .false.
   real(default) :: sigma = 1
   integer :: n = 1
   real(default) :: safety_factor = 1
   class(expr_factory_t), allocatable :: ef_selection
   class(expr_factory_t), allocatable :: ef_reweight
   class(expr_factory_t), allocatable :: ef_analysis
 contains
 ⟨Events: event config: TBP⟩
end type event_config_t
```

Output.

⟨*Events: event config: TBP*⟩≡

```
procedure :: write => event_config_write
```

⟨*Events: procedures*⟩≡

```
subroutine event_config_write (object, unit, show_expressions)
  class(event_config_t), intent(in) :: object
```

```
    integer, intent(in), optional :: unit
    logical, intent(in), optional :: show_expressions
    integer :: u
    u = given_output_unit (unit)
    write (u, "(3x,A,L1)")  "Unweighted         = ", object%unweighted
    write (u, "(3x,A,A)")   "Normalization      = ", &
         char (event_normalization_string (object%norm_mode))
    write (u, "(3x,A)", advance="no")  "Helicity handling  = "
    select case (object%factorization_mode)
    case (FM_IGNORE_HELICITY)
       write (u, "(A)")  "drop"
    case (FM_SELECT_HELICITY)
       write (u, "(A)")  "select"
    case (FM_FACTOR_HELICITY)
       write (u, "(A)")  "factorize"
    end select
    write (u, "(3x,A,L1)")  "Keep correlations  = ", object%keep_correlations
    if (.not. nearly_equal (object%safety_factor, one)) then
       write (u, "(3x,A," // FMT_12 // ")")  &
            "Safety factor      = ", object%safety_factor
    end if
    if (present (show_expressions)) then
       if (show_expressions) then
          if (allocated (object%ef_selection)) then
             call write_separator (u)
             write (u, "(3x,A)") "Event selection expression:"
             call object%ef_selection%write (u)
          end if
          if (allocated (object%ef_reweight)) then
             call write_separator (u)
             write (u, "(3x,A)") "Event reweighting expression:"
             call object%ef_reweight%write (u)
          end if
          if (allocated (object%ef_analysis)) then
             call write_separator (u)
             write (u, "(3x,A)") "Analysis expression:"
             call object%ef_analysis%write (u)
          end if
       end if
    end if
  end subroutine event_config_write
```

32.7.2 The event type

This is the concrete implementation of the `generic_event_t` core that is defined above in the `event_base` module. The core manages the main (dressed) particle set pointer and the current values for weights and sqme. The implementation adds configuration data, expressions, process references, and event transforms.

Each event refers to a single elementary process. This process may be dressed by a shower, a decay chain etc. We maintain pointers to a process instance.

A list of event transforms (class `evt_t`) transform the connected interactions of the process instance into the final particle set. In this list, the first transform

is always the trivial one, which just factorizes the process instance. Subsequent transforms may apply decays, etc. The `particle_set` pointer identifies the particle set that we want to be analyzed and returned by the event, usually the last one.

Squared matrix element and weight values: when reading events from file, the `ref` value is the number in the file, while the `prc` value is the number that we calculate from the momenta in the file, possibly with different parameters. When generating events the first time, or if we do not recalculate, the numbers should coincide. Furthermore, the array of `alt` values is copied from an array of alternative event records. These values should represent calculated values.

The `sqme` and `weight` values mirror corresponding values in the `expr` subobject. The idea is that when generating or reading events, the event record is filled first, then the `expr` object acquires copies. These copies are used for writing events and as targets for pointer variables in the analysis expression.

All data that involve user-provided expressions (selection, reweighting, analysis) are handled by the `expr` subobject. In particular, evaluating the event-selection expression sets the `passed` flag. Furthermore, the `expr` subobject collects data that can be used in the analysis and should be written to file, including copies of `sqme` and `weight`.

⟨*Events: public*⟩≡
```
public :: event_t
```

⟨*Events: types*⟩+≡
```
type, extends (generic_event_t) :: event_t
   type(event_config_t) :: config
   type(process_t), pointer :: process => null ()
   type(process_instance_t), pointer :: instance => null ()
   class(rng_t), allocatable :: rng
   integer :: selected_i_mci = 0
   integer :: selected_i_term = 0
   integer :: selected_channel = 0
   logical :: is_complete = .false.
   class(evt_t), pointer :: transform_first => null ()
   class(evt_t), pointer :: transform_last => null ()
   type(event_expr_t) :: expr
   logical :: selection_evaluated = .false.
   logical :: passed = .false.
   real(default), allocatable :: alpha_qcd_forced
   real(default), allocatable :: scale_forced
   real(default) :: reweight = 1
   logical :: analysis_flag = .false.
   integer :: i_event = 0
 contains
 ⟨Events: event: TBP⟩
end type event_t
```

⟨*Events: event: TBP*⟩≡
```
procedure :: clone => event_clone
```

⟨*Events: procedures*⟩+≡
```
subroutine event_clone (event, event_new)
  class(event_t), intent(in), target :: event
  class(event_t), intent(out), target:: event_new
```

```
    type(string_t) :: id
    integer :: num_id
    event_new%config = event%config
    event_new%process => event%process
    event_new%instance => event%instance
    if (allocated (event%rng)) &
         allocate(event_new%rng, source=event%rng)
    event_new%selected_i_mci = event%selected_i_mci
    event_new%selected_i_term = event%selected_i_term
    event_new%selected_channel = event%selected_channel
    event_new%is_complete = event%is_complete
    event_new%transform_first => event%transform_first
    event_new%transform_last => event%transform_last
    event_new%selection_evaluated = event%selection_evaluated
    event_new%passed = event%passed
    if (allocated (event%alpha_qcd_forced)) &
         allocate(event_new%alpha_qcd_forced, source=event%alpha_qcd_forced)
    if (allocated (event%scale_forced)) &
         allocate(event_new%scale_forced, source=event%scale_forced)
    event_new%reweight = event%reweight
    event_new%analysis_flag = event%analysis_flag
    event_new%i_event = event%i_event
    id = event_new%process%get_id ()
    if (id /= "")  call event_new%expr%set_process_id (id)
    num_id = event_new%process%get_num_id ()
    if (num_id /= 0)  call event_new%expr%set_process_num_id (num_id)
    call event_new%expr%setup_vars (event_new%process%get_sqrts ())
    call event_new%expr%link_var_list (event_new%process%get_var_list_ptr ())
  end subroutine event_clone
```

Finalizer: the list of event transforms is deleted iteratively.

⟨*Events: event: TBP*⟩+≡

```
  procedure :: final => event_final
```

⟨*Events: procedures*⟩+≡

```
  subroutine event_final (object)
    class(event_t), intent(inout) :: object
    class(evt_t), pointer :: evt
    if (allocated (object%rng))  call object%rng%final ()
    call object%expr%final ()
    do while (associated (object%transform_first))
       evt => object%transform_first
       object%transform_first => evt%next
       call evt%final ()
       deallocate (evt)
    end do
  end subroutine event_final
```

Output.

Particle set: this is a pointer to one of the event transforms, so it should suffice to print the latter.

⟨*Events: event: TBP*⟩+≡

```
  procedure :: write => event_write
```

⟨*Events: procedures*⟩+≡

```
subroutine event_write (object, unit, show_process, show_transforms, &
     show_decay, verbose, testflag)
  class(event_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: show_process, show_transforms, show_decay
  logical, intent(in), optional :: verbose
  logical, intent(in), optional :: testflag
  logical :: prc, trans, dec, verb
  class(evt_t), pointer :: evt
  character(len=7) :: fmt
  integer :: u, i
  call pac_fmt (fmt, FMT_19, FMT_12, testflag)
  u = given_output_unit (unit)
  prc = .true.;  if (present (show_process))  prc = show_process
  trans = .true.;  if (present (show_transforms))  trans = show_transforms
  dec = .true.;  if (present (show_decay))  dec = show_decay
  verb = .false.;  if (present (verbose))  verb = verbose
  call write_separator (u, 2)
  if (object%is_complete) then
     write (u, "(1x,A)")  "Event"
  else
     write (u, "(1x,A)")  "Event [incomplete]"
  end if
  call write_separator (u)
  call object%config%write (u)
  if (object%sqme_ref_is_known () .or. object%weight_ref_is_known ()) then
     call write_separator (u)
  end if
  if (object%sqme_ref_is_known ()) then
     write (u, "(3x,A," // fmt // ")") &
          "Squared matrix el. (ref) = ", object%get_sqme_ref ()
     if (object%sqme_alt_is_known ()) then
        do i = 1, object%get_n_alt ()
           write (u, "(5x,A," // fmt // ",1x,I0)")  &
                "alternate sqme   = ", object%get_sqme_alt(i), i
        end do
     end if
  end if
  if (object%sqme_prc_is_known ()) &
     write (u, "(3x,A," // fmt // ")") &
          "Squared matrix el. (prc) = ", object%get_sqme_prc ()

  if (object%weight_ref_is_known ()) then
     write (u, "(3x,A," // fmt // ")") &
          "Event weight (ref)       = ", object%get_weight_ref ()
     if (object%weight_alt_is_known ()) then
        do i = 1, object%get_n_alt ()
           write (u, "(5x,A," // fmt // ",1x,I0)")  &
                "alternate weight      = ", object%get_weight_alt(i), i
        end do
     end if
  end if
  if (object%weight_prc_is_known ()) &
```

```
      write (u, "(3x,A," // fmt // ")") &
           "Event weight (prc)       = ", object%get_weight_prc ()

   if (object%selected_i_mci /= 0) then
      call write_separator (u)
      write (u, "(3x,A,I0)")  "Selected MCI group = ", object%selected_i_mci
      write (u, "(3x,A,I0)")  "Selected term      = ", object%selected_i_term
      write (u, "(3x,A,I0)")  "Selected channel   = ", object%selected_channel
   end if
   if (object%selection_evaluated) then
      call write_separator (u)
      write (u, "(3x,A,L1)")  "Passed selection   = ", object%passed
      if (object%passed) then
         write (u, "(3x,A," // fmt // ")") &
              "Reweighting factor = ", object%reweight
         write (u, "(3x,A,L1)") &
              "Analysis flag      = ", object%analysis_flag
      end if
   end if
   if (associated (object%instance)) then
      if (prc) then
         if (verb) then
            call object%instance%write (u, testflag)
         else
            call object%instance%write_header (u)
         end if
      end if
      if (trans) then
         evt => object%transform_first
         do while (associated (evt))
            select type (evt)
            type is (evt_decay_t)
               call evt%write (u, verbose = dec, more_verbose = verb, &
                    testflag = testflag)
            class default
               call evt%write (u, verbose = verb, testflag = testflag)
            end select
            call write_separator (u, 2)
            evt => evt%next
         end do
      else
         call write_separator (u, 2)
      end if
      if (object%expr%subevt_filled) then
         call object%expr%write (u, pacified = testflag)
         call write_separator (u, 2)
      end if
   else
      call write_separator (u, 2)
      write (u, "(1x,A)")  "Process instance: [undefined]"
      call write_separator (u, 2)
   end if
end subroutine event_write
```

32.7.3 Initialization

Initialize: set configuration parameters, using a variable list. We do not call this `init`, because this method name will be used by a type extension.

The default normalization is `NORM_SIGMA`, since the default generation mode is weighted.

For unweighted events, we may want to a apply a safety factor to event rejection. (By default, this factor is unity and can be ignored.)

We also allocate the trivial event transform, which is always the first one.

⟨*Events: event: TBP*⟩+≡

```
procedure :: basic_init => event_init
```

⟨*Events: procedures*⟩+≡

```
subroutine event_init (event, var_list, n_alt)
  class(event_t), intent(out) :: event
  type(var_list_t), intent(in), optional :: var_list
  integer, intent(in), optional :: n_alt
  type(string_t) :: norm_string, mode_string
  logical :: polarized_events
  if (present (n_alt)) then
     call event%base_init (n_alt)
     call event%expr%init (n_alt)
  else
     call event%base_init (0)
  end if
  if (present (var_list)) then
     event%config%unweighted = var_list%get_lval (&
          var_str ("?unweighted"))
     norm_string = var_list%get_sval (&
          var_str ("$sample_normalization"))
     event%config%norm_mode = &
          event_normalization_mode (norm_string, event%config%unweighted)
     polarized_events = &
          var_list%get_lval (var_str ("?polarized_events"))
     if (polarized_events) then
        mode_string = &
             var_list%get_sval (var_str ("$polarization_mode"))
        select case (char (mode_string))
        case ("ignore")
           event%config%factorization_mode = FM_IGNORE_HELICITY
        case ("helicity")
           event%config%factorization_mode = FM_SELECT_HELICITY
        case ("factorized")
           event%config%factorization_mode = FM_FACTOR_HELICITY
        case ("correlated")
           event%config%factorization_mode = FM_CORRELATED_HELICITY
        case default
           call msg_fatal ("Polarization mode " &
                // char (mode_string) // " is undefined")
        end select
     else
        event%config%factorization_mode = FM_IGNORE_HELICITY
     end if
     if (event%config%unweighted) then
```

```
        event%config%safety_factor = var_list%get_rval (&
             var_str ("safety_factor"))
      end if
    else
      event%config%norm_mode = NORM_SIGMA
    end if
    allocate (evt_trivial_t :: event%transform_first)
    event%transform_last => event%transform_first
  end subroutine event_init
```

Set the `sigma` and `n` values in the configuration record that determine non-standard event normalizations. If these numbers are not set explicitly, the default value for both is unity, and event renormalization has no effect.

⟨*Events: event: TBP*⟩+≡

```
  procedure :: set_sigma => event_set_sigma
  procedure :: set_n => event_set_n
```

⟨*Events: procedures*⟩+≡

```
  elemental subroutine event_set_sigma (event, sigma)
    class(event_t), intent(inout) :: event
    real(default), intent(in) :: sigma
    event%config%sigma = sigma
  end subroutine event_set_sigma

  elemental subroutine event_set_n (event, n)
    class(event_t), intent(inout) :: event
    integer, intent(in) :: n
    event%config%n = n
  end subroutine event_set_n
```

Append an event transform (decays, etc.). The transform is not yet connected to a process. The transform is then considered to belong to the event object, and will be finalized together with it. The original pointer is removed.

We can assume that the trivial transform is already present in the event object, at least.

⟨*Events: event: TBP*⟩+≡

```
  procedure :: import_transform => event_import_transform
```

⟨*Events: procedures*⟩+≡

```
  subroutine event_import_transform (event, evt)
    class(event_t), intent(inout) :: event
    class(evt_t), intent(inout), pointer :: evt
    event%transform_last%next => evt
    evt%previous => event%transform_last
    event%transform_last => evt
    evt => null ()
  end subroutine event_import_transform
```

We link the event to an existing process instance. This includes the variable list, which is linked to the process variable list. Note that this is not necessarily identical to the variable list used for event initialization.

The variable list will contain pointers to `event` subobjects, therefore the `target` attribute.

Once we have a process connected, we can use it to obtain an event generator instance.

The model and process stack may be needed by event transforms. The current model setting may be different from the model in the process (regarding unstable particles, etc.). The process stack can be used for assigning extra processes that we need for the event transforms.

⟨*Events: event: TBP*⟩+≡

```
procedure :: connect => event_connect
```

⟨*Events: procedures*⟩+≡

```
subroutine event_connect (event, process_instance, model, process_stack)
  class(event_t), intent(inout), target :: event
  type(process_instance_t), intent(in), target :: process_instance
  class(model_data_t), intent(in), target :: model
  type(process_stack_t), intent(in), optional :: process_stack
  type(string_t) :: id
  integer :: num_id
  class(evt_t), pointer :: evt
  event%process => process_instance%process
  event%instance => process_instance
  id = event%process%get_id ()
  if (id /= "")  call event%expr%set_process_id (id)
  num_id = event%process%get_num_id ()
  if (num_id /= 0)  call event%expr%set_process_num_id (num_id)
  call event%expr%setup_vars (event%process%get_sqrts ())
  call event%expr%link_var_list (event%process%get_var_list_ptr ())
  call event%process%make_rng (event%rng)
  evt => event%transform_first
  do while (associated (evt))
     call evt%connect (process_instance, model, process_stack)
     evt => evt%next
  end do
end subroutine event_connect
```

Set the parse nodes for the associated expressions, individually. The parse-node pointers may be null.

⟨*Events: event: TBP*⟩+≡

```
procedure :: set_selection => event_set_selection
procedure :: set_reweight => event_set_reweight
procedure :: set_analysis => event_set_analysis
```

⟨*Events: procedures*⟩+≡

```
subroutine event_set_selection (event, ef_selection)
  class(event_t), intent(inout) :: event
  class(expr_factory_t), intent(in) :: ef_selection
  allocate (event%config%ef_selection, source = ef_selection)
end subroutine event_set_selection

subroutine event_set_reweight (event, ef_reweight)
  class(event_t), intent(inout) :: event
  class(expr_factory_t), intent(in) :: ef_reweight
  allocate (event%config%ef_reweight, source = ef_reweight)
end subroutine event_set_reweight
```

```
subroutine event_set_analysis (event, ef_analysis)
  class(event_t), intent(inout) :: event
  class(expr_factory_t), intent(in) :: ef_analysis
  allocate (event%config%ef_analysis, source = ef_analysis)
end subroutine event_set_analysis
```

Create evaluation trees from the parse trees. The `target` attribute is required because the expressions contain pointers to event subobjects.

⟨*Events: event: TBP*⟩+≡

```
procedure :: setup_expressions => event_setup_expressions
```

⟨*Events: procedures*⟩+≡

```
subroutine event_setup_expressions (event)
  class(event_t), intent(inout), target :: event
  call event%expr%setup_selection (event%config%ef_selection)
  call event%expr%setup_analysis (event%config%ef_analysis)
  call event%expr%setup_reweight (event%config%ef_reweight)
end subroutine event_setup_expressions
```

32.7.4 Evaluation

To fill the `particle_set`, i.e., the event record proper, we have to apply all event transforms in order. The last transform should fill its associated particle set, factorizing the state matrix according to the current settings. There are several parameters in the event configuration that control this.

We always fill the particle set for the first transform (the hard process) and the last transform, if different from the first (the fully dressed process).

Each event transform is an event generator of its own. We choose to generate an *unweighted* event for each of them, even if the master event is assumed to be weighted. Thus, the overall event weight is the one of the hard process only. (There may be more options in future extensions.)

We can generate the two random numbers that the factorization needs. For testing purpose, we allow for providing them explicitly, as an option.

⟨*Events: event: TBP*⟩+≡

```
procedure :: evaluate_transforms => event_evaluate_transforms
```

⟨*Events: procedures*⟩+≡

```
subroutine event_evaluate_transforms (event, r)
  class(event_t), intent(inout) :: event
  real(default), dimension(:), intent(in), optional :: r
  class(evt_t), pointer :: evt
  real(default) :: sigma_over_sqme
  integer :: i_term
  logical :: failed_but_keep = .false.
  call msg_debug (D_TRANSFORMS, "event_evaluate_transforms")
  call event%discard_particle_set ()
  call event%check ()
  if (event%instance%is_complete_event ()) then
     i_term = event%instance%select_i_term ()
     event%selected_i_term = i_term
     evt => event%transform_first
```

```
do while (associated (evt))
   call evt%prepare_new_event &
        (event%selected_i_mci, event%selected_i_term)
   evt => evt%next
end do
evt => event%transform_first
call msg_debug (D_TRANSFORMS, "Before event transformations")
call msg_debug (D_TRANSFORMS, "event%weight_prc", event%weight_prc)
call msg_debug (D_TRANSFORMS, "event%sqme_prc", event%sqme_prc)
do while (associated (evt))
   call print_transform_name_if_debug ()
   if (evt%only_weighted_events) then
      select type (evt)
      type is (evt_nlo_t)
         failed_but_keep = .not. evt%is_valid_event (i_term) .and. evt%keep_failed_events
         if (.not. evt%is_valid_event (i_term) .and. .not. failed_but_keep) &
            return
      end select
      if (abs (event%weight_prc) > 0._default) then
         sigma_over_sqme = event%weight_prc / event%sqme_prc
         call evt%generate_weighted (event%sqme_prc)
         event%weight_prc = sigma_over_sqme * event%sqme_prc
      else
         if (.not. failed_but_keep) exit
      end if
   else
      call evt%generate_unweighted ()
   end if
   if (signal_is_pending ())  return
   call evt%make_particle_set (event%config%factorization_mode, &
        event%config%keep_correlations)
   if (signal_is_pending ())  return
   if (.not. evt%particle_set_exists) exit
   evt => evt%next
end do
evt => event%transform_last
if ((associated (evt) .and. evt%particle_set_exists) .or. failed_but_keep) then
   if (event%is_nlo ()) then
      select type (evt)
      type is (evt_nlo_t)
         if (evt%i_evaluation > 0) then
            evt%particle_set_radiated (event%i_event + 1) = evt%particle_set
         else
            call evt%keep_and_boost_born_particle_set (event%i_event + 1)
         end if
         evt%i_evaluation = evt%i_evaluation + 1
         call event%link_particle_set &
            (evt%particle_set_radiated(event%i_event + 1))
      end select
   else
      call event%link_particle_set (evt%particle_set)
   end if
end if
call msg_debug (D_TRANSFORMS, "After event transformations")
```

```
      call msg_debug (D_TRANSFORMS, "event%weight_prc", event%weight_prc)
      call msg_debug (D_TRANSFORMS, "event%sqme_prc", event%sqme_prc)
      call msg_debug (D_TRANSFORMS, "evt%particle_set_exists", evt%particle_set_exists)
   end if
contains
   subroutine print_transform_name_if_debug ()
      if (debug_active (D_TRANSFORMS)) then
         print *, 'Current event transform: '
         call evt%write_name ()
      end if
   end subroutine print_transform_name_if_debug
end subroutine event_evaluate_transforms
```

Evaluate the event-related expressions, given a valid `particle_set`. If `update_sqme` is set, we use the process instance for the `sqme_prc` value. The `sqme_ref` value is always taken from the event record.

Note: without the explicit `particle_set` pointer, some gfortran 4.8 version corrupts its memory.

⟨*Events: event: TBP*⟩+≡

```
procedure :: evaluate_expressions => event_evaluate_expressions
```

⟨*Events: procedures*⟩+≡

```
subroutine event_evaluate_expressions (event)
   class(event_t), intent(inout) :: event
   type(particle_set_t), pointer :: particle_set
   if (event%has_valid_particle_set ()) then
      particle_set => event%get_particle_set_ptr ()
      call event%expr%fill_subevt (particle_set)
   end if
   if (event%weight_ref_is_known ()) then
      call event%expr%set (weight_ref = event%get_weight_ref ())
   end if
   if (event%weight_prc_is_known ()) then
      call event%expr%set (weight_prc = event%get_weight_prc ())
   end if
   if (event%excess_prc_is_known ()) then
      call event%expr%set (excess_prc = event%get_excess_prc ())
   end if
   if (event%sqme_ref_is_known ()) then
      call event%expr%set (sqme_ref = event%get_sqme_ref ())
   end if
   if (event%sqme_prc_is_known ()) then
      call event%expr%set (sqme_prc = event%get_sqme_prc ())
   end if
   if (event%has_valid_particle_set ()) then
      call event%expr%evaluate &
           (event%passed, event%reweight, event%analysis_flag)
      event%selection_evaluated = .true.
   end if
end subroutine event_evaluate_expressions
```

Report the result of the `selection` evaluation.

⟨*Events: event: TBP*⟩+≡

```
procedure :: passed_selection => event_passed_selection
```

⟨*Events: procedures*⟩+≡

```
function event_passed_selection (event) result (flag)
  class(event_t), intent(in) :: event
  logical :: flag
  flag = event%passed
end function event_passed_selection
```

Set alternate sqme and weight arrays. This should be merged with the previous routine, if the expressions are allowed to refer to these values.

⟨*Events: event: TBP*⟩+≡

```
procedure :: store_alt_values => event_store_alt_values
```

⟨*Events: procedures*⟩+≡

```
subroutine event_store_alt_values (event)
  class(event_t), intent(inout) :: event
  if (event%weight_alt_is_known ()) then
     call event%expr%set (weight_alt = event%get_weight_alt ())
  end if
  if (event%sqme_alt_is_known ()) then
     call event%expr%set (sqme_alt = event%get_sqme_alt ())
  end if
end subroutine event_store_alt_values
```

⟨*Events: event: TBP*⟩+≡

```
procedure :: is_nlo => event_is_nlo
```

⟨*Events: procedures*⟩+≡

```
function event_is_nlo (event) result (is_nlo)
  logical :: is_nlo
  class(event_t), intent(in) :: event
  if (associated (event%instance)) then
     select type (pcm => event%instance%pcm)
     type is (pcm_instance_nlo_t)
        is_nlo = pcm%is_fixed_order_nlo_events ()
     class default
        is_nlo = .false.
     end select
  else
     is_nlo = .false.
  end if
end function event_is_nlo
```

32.7.5 Reset to empty state

Applying this, current event contents are marked as incomplete but are not deleted. In particular, the initialization is kept.

⟨*Events: event: TBP*⟩+≡

```
procedure :: reset => event_reset
```

⟨*Events: procedures*⟩+≡

```
subroutine event_reset (event)
  class(event_t), intent(inout) :: event
  class(evt_t), pointer :: evt
  call event%base_reset ()
  event%selected_i_mci = 0
  event%selected_i_term = 0
  event%selected_channel = 0
  event%is_complete = .false.
  call event%expr%reset ()
  event%selection_evaluated = .false.
  event%passed = .false.
  event%analysis_flag = .false.
  if (associated (event%instance)) then
     call event%instance%reset (reset_mci = .true.)
  end if
  if (allocated (event%alpha_qcd_forced))  deallocate (event%alpha_qcd_forced)
  if (allocated (event%scale_forced))  deallocate (event%scale_forced)
  evt => event%transform_first
  do while (associated (evt))
     call evt%reset ()
     evt => evt%next
  end do
end subroutine event_reset
```

32.7.6 Squared Matrix Element and Weight

Transfer the result of the process instance calculation to the event record header.

⟨*Events: event: TBP*⟩+≡

```
procedure :: import_instance_results => event_import_instance_results
```

⟨*Events: procedures*⟩+≡

```
subroutine event_import_instance_results (event)
  class(event_t), intent(inout) :: event
  if (associated (event%instance)) then
     if (event%instance%has_evaluated_trace ()) then
        call event%set ( &
             sqme_prc = event%instance%get_sqme (), &
             weight_prc = event%instance%get_weight (), &
             excess_prc = event%instance%get_excess () &
             )
     end if
  end if
end subroutine event_import_instance_results
```

Duplicate the instance result / the reference result in the event record.

⟨*Events: event: TBP*⟩+≡

```
procedure :: accept_sqme_ref => event_accept_sqme_ref
procedure :: accept_sqme_prc => event_accept_sqme_prc
procedure :: accept_weight_ref => event_accept_weight_ref
procedure :: accept_weight_prc => event_accept_weight_prc
```

⟨*Events: procedures*⟩+≡

```
subroutine event_accept_sqme_ref (event)
  class(event_t), intent(inout) :: event
  if (event%sqme_ref_is_known ()) then
     call event%set (sqme_prc = event%get_sqme_ref ())
  end if
end subroutine event_accept_sqme_ref

subroutine event_accept_sqme_prc (event)
  class(event_t), intent(inout) :: event
  if (event%sqme_prc_is_known ()) then
     call event%set (sqme_ref = event%get_sqme_prc ())
  end if
end subroutine event_accept_sqme_prc

subroutine event_accept_weight_ref (event)
  class(event_t), intent(inout) :: event
  if (event%weight_ref_is_known ()) then
     call event%set (weight_prc = event%get_weight_ref ())
  end if
end subroutine event_accept_weight_ref

subroutine event_accept_weight_prc (event)
  class(event_t), intent(inout) :: event
  if (event%weight_prc_is_known ()) then
     call event%set (weight_ref = event%get_weight_prc ())
  end if
end subroutine event_accept_weight_prc
```

Update the weight normalization, just after generation. Unweighted and weighted events are generated with a different default normalization. The intended normalization is stored in the configuration record.

⟨*Events: event: TBP*⟩+≡

```
procedure :: update_normalization => event_update_normalization
```

⟨*Events: procedures*⟩+≡

```
subroutine event_update_normalization (event, mode_ref)
  class(event_t), intent(inout) :: event
  integer, intent(in), optional :: mode_ref
  integer :: mode_old
  real(default) :: weight, excess
  if (present (mode_ref)) then
     mode_old = mode_ref
  else if (event%config%unweighted) then
     mode_old = NORM_UNIT
  else
     mode_old = NORM_SIGMA
  end if
  weight = event%get_weight_prc ()
  call event_normalization_update (weight, &
       event%config%sigma, event%config%n, &
       mode_new = event%config%norm_mode, &
       mode_old = mode_old)
  call event%set_weight_prc (weight)
```

```
    excess = event%get_excess_prc ()
    call event_normalization_update (excess, &
         event%config%sigma, event%config%n, &
         mode_new = event%config%norm_mode, &
         mode_old = mode_old)
    call event%set_excess_prc (excess)
  end subroutine event_update_normalization
```

The event is complete if it has a particle set plus valid entries for the sqme and weight values.

⟨*Events: event: TBP*⟩+≡

```
  procedure :: check => event_check
```

⟨*Events: procedures*⟩+≡

```
  subroutine event_check (event)
    class(event_t), intent(inout) :: event
    event%is_complete = event%has_valid_particle_set () &
         .and. event%sqme_ref_is_known () &
         .and. event%sqme_prc_is_known () &
         .and. event%weight_ref_is_known () &
         .and. event%weight_prc_is_known ()
    if (event%get_n_alt () /= 0) then
       event%is_complete = event%is_complete &
            .and. event%sqme_alt_is_known () &
            .and. event%weight_alt_is_known ()
    end if
  end subroutine event_check
```

32.7.7 Generation

Assuming that we have a valid process associated to the event, we generate an event. We complete the event data, then factorize the spin density matrix and transfer it to the particle set.

When done, we retrieve squared matrix element and weight. In case of explicit generation, the reference values coincide with the process values, so we `accept` the latter.

The explicit random number argument `r` should be generated by a random-number generator. It is taken for the factorization algorithm, bypassing the event-specific random-number generator. This is useful for deterministic testing.

⟨*Events: event: TBP*⟩+≡

```
  procedure :: generate => event_generate
```

⟨*Events: procedures*⟩+≡

```
  subroutine event_generate (event, i_mci, r, i_nlo)
    class(event_t), intent(inout) :: event
    integer, intent(in) :: i_mci
    real(default), dimension(:), intent(in), optional :: r
    integer, intent(in), optional :: i_nlo
    logical :: generate_new = .true.
    if (present (i_nlo)) generate_new = (i_nlo == 1)
    if (generate_new) call event%reset ()
    event%selected_i_mci = i_mci
```

```
    if (event%config%unweighted) then
       call event%instance%generate_unweighted_event (i_mci)
       if (signal_is_pending ()) return
       call event%instance%evaluate_event_data ()
       call event%instance%normalize_weight ()
    else
       if (generate_new) &
          call event%instance%generate_weighted_event (i_mci)
       if (signal_is_pending ()) return
       call event%instance%evaluate_event_data ()
    end if
    event%selected_channel = event%instance%get_channel ()
    call event%import_instance_results ()
    call event%accept_sqme_prc ()
    call event%update_normalization ()
    call event%accept_weight_prc ()
    call event%evaluate_transforms (r)
    if (signal_is_pending ())  return
    call event%check ()
  end subroutine event_generate
```

Get a copy of the particle set belonging to the hard process.

⟨*Events: event: TBP*⟩+≡

```
  procedure :: get_hard_particle_set => event_get_hard_particle_set
```

⟨*Events: procedures*⟩+≡

```
  subroutine event_get_hard_particle_set (event, pset)
    class(event_t), intent(in) :: event
    type(particle_set_t), intent(out) :: pset
    class(evt_t), pointer :: evt
    evt => event%transform_first
    pset = evt%particle_set
  end subroutine event_get_hard_particle_set
```

32.7.8 Recovering an event

Select MC group, term, and integration channel.

⟨*Events: event: TBP*⟩+≡

```
  procedure :: select => event_select
```

⟨*Events: procedures*⟩+≡

```
  subroutine event_select (event, i_mci, i_term, channel)
    class(event_t), intent(inout) :: event
    integer, intent(in) :: i_mci, i_term, channel
    if (associated (event%instance)) then
       event%selected_i_mci = i_mci
       event%selected_i_term = i_term
       event%selected_channel = channel
    else
       event%selected_i_mci = 0
       event%selected_i_term = 0
       event%selected_channel = 0
    end if
```

```
end subroutine event_select
```

Copy a particle set into the event record.

We deliberately use the first (the trivial) transform for this, i.e., the hard process. The event reader may either read in the transformed event separately, or apply all event transforms to the hard particle set to (re)generate a fully dressed event.

Since this makes all subsequent event transforms invalid, we call `reset` on them.

⟨*Events: event: TBP*⟩+≡

```
procedure :: set_hard_particle_set => event_set_hard_particle_set
```

⟨*Events: procedures*⟩+≡

```
subroutine event_set_hard_particle_set (event, particle_set)
  class(event_t), intent(inout) :: event
  type(particle_set_t), intent(in) :: particle_set
  class(evt_t), pointer :: evt
  evt => event%transform_first
  call evt%set_particle_set (particle_set, &
       event%selected_i_mci, event%selected_i_term)
  call event%link_particle_set (evt%particle_set)
  evt => evt%next
  do while (associated (evt))
     call evt%reset ()
     evt => evt%next
  end do
end subroutine event_set_hard_particle_set
```

Set the α_s value that should be used in a recalculation. This should be called only if we explicitly want to override the QCD setting of the process core.

⟨*Events: event: TBP*⟩+≡

```
procedure :: set_alpha_qcd_forced => event_set_alpha_qcd_forced
```

⟨*Events: procedures*⟩+≡

```
subroutine event_set_alpha_qcd_forced (event, alpha_qcd)
  class(event_t), intent(inout) :: event
  real(default), intent(in) :: alpha_qcd
  if (allocated (event%alpha_qcd_forced)) then
     event%alpha_qcd_forced = alpha_qcd
  else
     allocate (event%alpha_qcd_forced, source = alpha_qcd)
  end if
end subroutine event_set_alpha_qcd_forced
```

Analogously, for the common scale. This forces also renormalization and factorization scale.

⟨*Events: event: TBP*⟩+≡

```
procedure :: set_scale_forced => event_set_scale_forced
```

⟨*Events: procedures*⟩+≡

```
subroutine event_set_scale_forced (event, scale)
  class(event_t), intent(inout) :: event
  real(default), intent(in) :: scale
```

```
    if (allocated (event%scale_forced)) then
       event%scale_forced = scale
    else
       allocate (event%scale_forced, source = scale)
    end if
  end subroutine event_set_scale_forced
```

Here we try to recover an event from the `particle_set` subobject and recalculate the structure functions and matrix elements. We have the appropriate `process` object and an initialized `process_instance` at hand, so beam and configuration data are known. From the `particle_set`, we get the momenta.

The quantum-number information may be incomplete, e.g., helicity information may be partial or absent. We recover the event just from the momentum configuration.

We do not transfer the matrix element from the process instance to the event record, as we do when generating an event. The event record may contain the matrix element as read from file, and the current calculation may use different parameters. We thus can compare old and new values.

The event `weight` may also be known already. If yes, we pass it to the `evaluate_event_data` procedure. It should already be normalized. If we have an `weight_factor` value, we obtain the event weight by multiplying the computed `sqme` by this factor. Otherwise, we make use of the MCI setup (which should be valid then) to compute the event weight, and we should normalize the result just as when generating events.

Evaluating event expressions must also be done separately.

⟨*Events: event: TBP*⟩+≡

```
  procedure :: recalculate => event_recalculate
```

⟨*Events: procedures*⟩+≡

```
  subroutine event_recalculate &
       (event, update_sqme, weight_factor, recover_beams)
    class(event_t), intent(inout) :: event
    logical, intent(in) :: update_sqme
    real(default), intent(in), optional :: weight_factor
    logical, intent(in), optional :: recover_beams
    type(particle_set_t), pointer :: particle_set
    integer :: i_mci, i_term, channel
    if (event%has_valid_particle_set ()) then
       particle_set => event%get_particle_set_ptr ()
       i_mci = event%selected_i_mci
       i_term = event%selected_i_term
       channel = event%selected_channel
       if (i_mci == 0 .or. i_term == 0 .or. channel == 0) then
          call msg_bug ("Event: recalculate: undefined selection parameters")
       end if
       call event%instance%choose_mci (i_mci)
       call event%instance%set_trace (particle_set, i_term, recover_beams)
       if (allocated (event%alpha_qcd_forced)) then
          call event%instance%set_alpha_qcd_forced &
               (i_term, event%alpha_qcd_forced)
       end if
       call event%instance%recover (channel, i_term, update_sqme, &
```

```
            event%scale_forced)
      if (signal_is_pending ())  return
      if (update_sqme .and. present (weight_factor)) then
         call event%instance%evaluate_event_data &
              (weight = event%instance%get_sqme () * weight_factor)
      else if (event%weight_ref_is_known ()) then
         call event%instance%evaluate_event_data &
              (weight = event%get_weight_ref ())
      else
         call event%instance%recover_event ()
         if (signal_is_pending ())  return
         call event%instance%evaluate_event_data ()
         if (event%config%unweighted) then
            call event%instance%normalize_weight ()
         end if
      end if
      if (signal_is_pending ())  return
      if (update_sqme) then
         call event%import_instance_results ()
      else
         call event%accept_sqme_ref ()
         call event%accept_weight_ref ()
      end if
   else
      call msg_bug ("Event: can't recalculate, particle set is undefined")
   end if
end subroutine event_recalculate
```

32.7.9 Access content

Pointer to the associated process object (the associated model).

⟨*Events: event: TBP*⟩+≡

```
procedure :: get_process_ptr => event_get_process_ptr
procedure :: get_process_instance_ptr => event_get_process_instance_ptr
procedure :: get_model_ptr => event_get_model_ptr
```

⟨*Events: procedures*⟩+≡

```
function event_get_process_ptr (event) result (ptr)
  class(event_t), intent(in) :: event
  type(process_t), pointer :: ptr
  ptr => event%process
end function event_get_process_ptr

function event_get_process_instance_ptr (event) result (ptr)
  class(event_t), intent(in) :: event
  type(process_instance_t), pointer :: ptr
  ptr => event%instance
end function event_get_process_instance_ptr

function event_get_model_ptr (event) result (model)
  class(event_t), intent(in) :: event
  class(model_data_t), pointer :: model
  if (associated (event%process)) then
```

```
      model => event%process%get_model_ptr ()
    else
      model => null ()
    end if
  end function event_get_model_ptr
```

Return the current values of indices: the MCI group of components, the term index (different terms corresponding, potentially, to different effective kinematics), and the MC integration channel. The `i_mci` call is delegated to the current process instance.

⟨*Events: event: TBP*⟩+≡

```
  procedure :: get_i_mci => event_get_i_mci
  procedure :: get_i_term => event_get_i_term
  procedure :: get_channel => event_get_channel
```

⟨*Events: procedures*⟩+≡

```
  function event_get_i_mci (event) result (i_mci)
    class(event_t), intent(in) :: event
    integer :: i_mci
    i_mci = event%selected_i_mci
  end function event_get_i_mci

  function event_get_i_term (event) result (i_term)
    class(event_t), intent(in) :: event
    integer :: i_term
    i_term = event%selected_i_term
  end function event_get_i_term

  function event_get_channel (event) result (channel)
    class(event_t), intent(in) :: event
    integer :: channel
    channel = event%selected_channel
  end function event_get_channel
```

This flag tells us whether the event consists just of a hard process (i.e., holds at most the first, trivial transform), or is a dressed events with additional transforms.

⟨*Events: event: TBP*⟩+≡

```
  procedure :: has_transform => event_has_transform
```

⟨*Events: procedures*⟩+≡

```
  function event_has_transform (event) result (flag)
    class(event_t), intent(in) :: event
    logical :: flag
    if (associated (event%transform_first)) then
      flag = associated (event%transform_first%next)
    else
      flag = .false.
    end if
  end function event_has_transform
```

Return the currently selected normalization mode, or alternate normalization mode.

⟨*Events: event: TBP*⟩+≡
```
procedure :: get_norm_mode => event_get_norm_mode
```

⟨*Events: procedures*⟩+≡
```
elemental function event_get_norm_mode (event) result (norm_mode)
  class(event_t), intent(in) :: event
  integer :: norm_mode
  norm_mode = event%config%norm_mode
end function event_get_norm_mode
```

Return the kinematical weight, defined as the ratio of event weight and squared matrix element.

⟨*Events: event: TBP*⟩+≡
```
procedure :: get_kinematical_weight => event_get_kinematical_weight
```

⟨*Events: procedures*⟩+≡
```
function event_get_kinematical_weight (event) result (f)
  class(event_t), intent(in) :: event
  real(default) :: f
  if (event%sqme_ref_is_known () .and. event%weight_ref_is_known () &
       .and. abs (event%get_sqme_ref ()) > 0) then
     f = event%get_weight_ref () / event%get_sqme_ref ()
  else
     f = 0
  end if
end function event_get_kinematical_weight
```

Return data used by external event formats.

⟨*Events: event: TBP*⟩+≡
```
procedure :: get_index => event_get_index
procedure :: get_fac_scale => event_get_fac_scale
procedure :: get_alpha_s => event_get_alpha_s
procedure :: get_sqrts => event_get_sqrts
procedure :: get_polarization => event_get_polarization
procedure :: get_beam_file => event_get_beam_file
procedure :: get_process_name => event_get_process_name
```

⟨*Events: procedures*⟩+≡
```
function event_get_index (event) result (index)
  class(event_t), intent(in) :: event
  integer :: index
  index = event%expr%index
end function event_get_index

function event_get_fac_scale (event) result (fac_scale)
  class(event_t), intent(in) :: event
  real(default) :: fac_scale
  fac_scale = event%instance%get_fac_scale (event%selected_i_term)
end function event_get_fac_scale

function event_get_alpha_s (event) result (alpha_s)
  class(event_t), intent(in) :: event
```

```
  real(default) :: alpha_s
  alpha_s = event%instance%get_alpha_s (event%selected_i_term)
end function event_get_alpha_s

function event_get_sqrts (event) result (sqrts)
  class(event_t), intent(in) :: event
  real(default) :: sqrts
  sqrts = event%instance%get_sqrts ()
end function event_get_sqrts

function event_get_polarization (event) result (pol)
  class(event_t), intent(in) :: event
  real(default), dimension(2) :: pol
  pol = event%instance%get_polarization ()
end function event_get_polarization

function event_get_beam_file (event) result (file)
  class(event_t), intent(in) :: event
  type(string_t) :: file
  file = event%instance%get_beam_file ()
end function event_get_beam_file

function event_get_process_name (event) result (name)
  class(event_t), intent(in) :: event
  type(string_t) :: name
  name = event%instance%get_process_name ()
end function event_get_process_name
```

Return the actual number of calls, as stored in the process instance.

⟨*Events: event: TBP*⟩+≡

```
procedure :: get_actual_calls_total => event_get_actual_calls_total
```

⟨*Events: procedures*⟩+≡

```
elemental function event_get_actual_calls_total (event) result (n)
  class(event_t), intent(in) :: event
  integer :: n
  if (associated (event%instance)) then
     n = event%instance%get_actual_calls_total ()
  else
     n = 0
  end if
end function event_get_actual_calls_total
```

Eliminate numerical noise in the `subevt` expression and in the event transforms (which includes associated process instances).

⟨*Events: public*⟩+≡

```
public :: pacify
```

⟨*Events: interfaces*⟩≡

```
interface pacify
   module procedure pacify_event
end interface pacify
```

⟨*Events: procedures*⟩+≡

```
    subroutine pacify_event (event)
      class(event_t), intent(inout) :: event
      class(evt_t), pointer :: evt
      call event%pacify_particle_set ()
      if (event%expr%subevt_filled)  call pacify (event%expr)
      evt => event%transform_first
      do while (associated (evt))
         select type (evt)
         type is (evt_decay_t);  call pacify (evt)
         end select
         evt => evt%next
      end do
    end subroutine pacify_event
```

32.7.10 Unit tests

Test module, followed by the corresponding implementation module.

⟨`events_ut.f90`⟩≡

```
  ⟨File header⟩

  module events_ut
    use unit_tests
    use events_uti

  ⟨Standard module head⟩

  ⟨Events: public test⟩

  contains

  ⟨Events: test driver⟩

  end module events_ut
```

⟨`events_uti.f90`⟩≡

```
  ⟨File header⟩

  module events_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use os_interface
    use model_data
    use particles
    use process_libraries
    use process_stacks
    use event_transforms
    use decays
    use decays_ut, only: prepare_testbed

    use process, only: process_t
    use instances, only: process_instance_t
```

```
  use events

⟨Standard module head⟩

⟨Events: test declarations⟩

contains

⟨Events: tests⟩

end module events_uti
```

API: driver for the unit tests below.

⟨*Events: public test*⟩≡
```
  public :: events_test
```

⟨*Events: test driver*⟩≡
```
  subroutine events_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨Events: execute tests⟩
  end subroutine events_test
```

Empty event record

⟨*Events: execute tests*⟩≡
```
  call test (events_1, "events_1", &
       "empty event record", &
       u, results)
```

⟨*Events: test declarations*⟩≡
```
  public :: events_1
```

⟨*Events: tests*⟩≡
```
  subroutine events_1 (u)
    integer, intent(in) :: u
    type(event_t), target :: event

    write (u, "(A)")  "* Test output: events_1"
    write (u, "(A)")  "*   Purpose: display an empty event object"
    write (u, "(A)")

    call event%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: events_1"

  end subroutine events_1
```

Simple event

⟨*Events: execute tests*⟩+≡

```
    call test (events_2, "events_2", &
         "generate event", &
         u, results)
```

⟨*Events: test declarations*⟩+≡

```
    public :: events_2
```

⟨*Events: tests*⟩+≡

```
    subroutine events_2 (u)
      use processes_ut, only: prepare_test_process, cleanup_test_process
      integer, intent(in) :: u
      type(event_t), allocatable, target :: event
      type(process_t), allocatable, target :: process
      type(process_instance_t), allocatable, target :: process_instance
      type(model_data_t), target :: model

      write (u, "(A)")  "* Test output: events_2"
      write (u, "(A)")  "*   Purpose: generate and display an event"
      write (u, "(A)")

      call model%init_test ()

      write (u, "(A)")  "* Generate test process event"

      allocate (process)
      allocate (process_instance)
      call prepare_test_process (process, process_instance, model)
      call process_instance%setup_event_data ()

      write (u, "(A)")
      write (u, "(A)")  "* Initialize event object"

      allocate (event)
      call event%basic_init ()
      call event%connect (process_instance, process%get_model_ptr ())

      write (u, "(A)")
      write (u, "(A)")  "* Generate test process event"

      call process_instance%generate_weighted_event (1)

      write (u, "(A)")
      write (u, "(A)")  "* Fill event object"
      write (u, "(A)")

      call event%generate (1, [0.4_default, 0.4_default])
      call event%evaluate_expressions ()
      call event%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call event%final ()
      deallocate (event)
```

```
    call cleanup_test_process (process, process_instance)
    deallocate (process_instance)
    deallocate (process)

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: events_2"

  end subroutine events_2
```

Recovering an event

Generate an event and store the particle set. Then reset the event record, recall the particle set, and recover the event from that.

Note: The extra `particle_set_ptr` auxiliary is a workaround for memory corruption in gfortran 4.7.

⟨*Events: execute tests*⟩+≡

```
  call test (events_4, "events_4", &
      "recover event", &
      u, results)
```

⟨*Events: test declarations*⟩+≡

```
  public :: events_4
```

⟨*Events: tests*⟩+≡

```
  subroutine events_4 (u)
    use processes_ut, only: prepare_test_process, cleanup_test_process
    integer, intent(in) :: u
    type(event_t), allocatable, target :: event
    type(process_t), allocatable, target :: process
    type(process_instance_t), allocatable, target :: process_instance
    type(particle_set_t) :: particle_set
    type(particle_set_t), pointer :: particle_set_ptr
    type(model_data_t), target :: model

    write (u, "(A)")  "* Test output: events_4"
    write (u, "(A)")  "*   Purpose: generate and recover an event"
    write (u, "(A)")

    call model%init_test ()

    write (u, "(A)")  "* Generate test process event and save particle set"
    write (u, "(A)")

    allocate (process)
    allocate (process_instance)
    call prepare_test_process (process, process_instance, model)
    call process_instance%setup_event_data ()

    allocate (event)
    call event%basic_init ()
    call event%connect (process_instance, process%get_model_ptr ())
```

```
call event%generate (1, [0.4_default, 0.4_default])
call event%evaluate_expressions ()
call event%write (u)

particle_set_ptr => event%get_particle_set_ptr ()
particle_set = particle_set_ptr

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

write (u, "(A)")
write (u, "(A)")  "* Recover event from particle set"
write (u, "(A)")

allocate (process)
allocate (process_instance)
call prepare_test_process (process, process_instance, model)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call event%select (1, 1, 1)
call event%set_hard_particle_set (particle_set)
call event%recalculate (update_sqme = .true.)
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Transfer sqme and evaluate expressions"
write (u, "(A)")

call event%accept_sqme_prc ()
call event%accept_weight_prc ()
call event%check ()
call event%evaluate_expressions ()
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Reset contents"
write (u, "(A)")

call event%reset ()
event%transform_first%particle_set_exists = .false.
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"
```

```
    call particle_set%final ()

    call event%final ()
    deallocate (event)

    call cleanup_test_process (process, process_instance)
    deallocate (process_instance)
    deallocate (process)

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: events_4"

  end subroutine events_4
```

Partially Recovering an event

Generate an event and store the particle set. Then reset the event record, recall the particle set, and recover the event as far as possible without recomputing the squared matrix element.

⟨*Events: execute tests*⟩+≡

```
  call test (events_5, "events_5", &
       "partially recover event", &
       u, results)
```

⟨*Events: test declarations*⟩+≡

```
  public :: events_5
```

⟨*Events: tests*⟩+≡

```
  subroutine events_5 (u)
    use processes_ut, only: prepare_test_process, cleanup_test_process
    integer, intent(in) :: u
    type(event_t), allocatable, target :: event
    type(process_t), allocatable, target :: process
    type(process_instance_t), allocatable, target :: process_instance
    type(particle_set_t) :: particle_set
    type(particle_set_t), pointer :: particle_set_ptr
    real(default) :: sqme, weight
    type(model_data_t), target :: model

    write (u, "(A)")  "* Test output: events_5"
    write (u, "(A)")  "*   Purpose: generate and recover an event"
    write (u, "(A)")

    call model%init_test ()

    write (u, "(A)")  "* Generate test process event and save particle set"
    write (u, "(A)")

    allocate (process)
    allocate (process_instance)
    call prepare_test_process (process, process_instance, model)
```

```
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call event%generate (1, [0.4_default, 0.4_default])
call event%evaluate_expressions ()
call event%write (u)

particle_set_ptr => event%get_particle_set_ptr ()
particle_set = particle_set_ptr
sqme = event%get_sqme_ref ()
weight = event%get_weight_ref ()

call event%final ()
deallocate (event)

call cleanup_test_process (process, process_instance)
deallocate (process_instance)
deallocate (process)

write (u, "(A)")
write (u, "(A)")  "* Recover event from particle set"
write (u, "(A)")

allocate (process)
allocate (process_instance)
call prepare_test_process (process, process_instance, model)
call process_instance%setup_event_data ()

allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call event%select (1, 1, 1)
call event%set_hard_particle_set (particle_set)
call event%recalculate (update_sqme = .false.)
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Manually set sqme and evaluate expressions"
write (u, "(A)")

call event%set (sqme_ref = sqme, weight_ref = weight)
call event%accept_sqme_ref ()
call event%accept_weight_ref ()
call event%evaluate_expressions ()
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call particle_set%final ()
```

```
    call event%final ()
    deallocate (event)

    call cleanup_test_process (process, process_instance)
    deallocate (process_instance)
    deallocate (process)

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: events_5"

  end subroutine events_5
```

Decays

Generate an event with subsequent decays.

⟨*Events: execute tests*⟩+≡

```
  call test (events_6, "events_6", &
       "decays", &
       u, results)
```

⟨*Events: test declarations*⟩+≡

```
  public :: events_6
```

⟨*Events: tests*⟩+≡

```
  subroutine events_6 (u)
    integer, intent(in) :: u
    type(os_data_t) :: os_data
    class(model_data_t), pointer :: model
    type(string_t) :: prefix, procname1, procname2
    type(process_library_t), target :: lib
    type(process_stack_t) :: process_stack
    class(evt_t), pointer :: evt_decay
    type(event_t), allocatable, target :: event
    type(process_t), pointer :: process
    type(process_instance_t), allocatable, target :: process_instance

    write (u, "(A)")  "* Test output: events_6"
    write (u, "(A)")  "*   Purpose: generate an event with subsequent decays"
    write (u, "(A)")

    write (u, "(A)")  "* Generate test process and decay"
    write (u, "(A)")

    call os_data_init (os_data)

    prefix = "events_6"
    procname1 = prefix // "_p"
    procname2 = prefix // "_d"
    call prepare_testbed &
         (lib, process_stack, prefix, os_data, &
         scattering=.true., decay=.true.)
```

```
  write (u, "(A)")  "* Initialize decay process"

  process => process_stack%get_process_ptr (procname1)
  model => process%get_model_ptr ()
  call model%set_unstable (25, [procname2])

  allocate (process_instance)
  call process_instance%init (process)
  call process_instance%setup_event_data ()
  call process_instance%init_simulation (1)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize event transform: decay"

  allocate (evt_decay_t :: evt_decay)
  call evt_decay%connect (process_instance, model, process_stack)

  write (u, "(A)")
  write (u, "(A)")  "* Initialize event object"
  write (u, "(A)")

  allocate (event)
  call event%basic_init ()
  call event%connect (process_instance, model)
  call event%import_transform (evt_decay)

  call event%write (u, show_decay = .true.)

  write (u, "(A)")
  write (u, "(A)")  "* Generate event"
  write (u, "(A)")

  call event%generate (1, [0.4_default, 0.4_default])
  call event%evaluate_expressions ()
  call event%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call event%final ()
  deallocate (event)

  call process_instance%final ()
  deallocate (process_instance)

  call process_stack%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: events_6"

end subroutine events_6
```

Decays

Generate a decay event with varying options.

⟨*Events: execute tests*⟩+≡

```
call test (events_7, "events_7", &
     "decay options", &
     u, results)
```

⟨*Events: test declarations*⟩+≡

```
public :: events_7
```

⟨*Events: tests*⟩+≡

```
subroutine events_7 (u)
  integer, intent(in) :: u
  type(os_data_t) :: os_data
  class(model_data_t), pointer :: model
  type(string_t) :: prefix, procname2
  type(process_library_t), target :: lib
  type(process_stack_t) :: process_stack
  type(process_t), pointer :: process
  type(process_instance_t), allocatable, target :: process_instance

  write (u, "(A)")  "* Test output: events_7"
  write (u, "(A)")  "*   Purpose: check decay options"
  write (u, "(A)")

  write (u, "(A)")  "* Prepare test process"
  write (u, "(A)")

  call os_data_init (os_data)

  prefix = "events_7"
  procname2 = prefix // "_d"
  call prepare_testbed &
       (lib, process_stack, prefix, os_data, &
       scattering=.false., decay=.true.)

  write (u, "(A)")  "* Generate decay event, default options"
  write (u, "(A)")

  process => process_stack%get_process_ptr (procname2)
  model => process%get_model_ptr ()
  call model%set_unstable (25, [procname2])

  allocate (process_instance)
  call process_instance%init (process)
  call process_instance%setup_event_data (model)
  call process_instance%init_simulation (1)

  call process_instance%generate_weighted_event (1)
  call process_instance%write (u)

  call process_instance%final ()
  deallocate (process_instance)
```

```
  write (u, "(A)")
  write (u, "(A)")  "* Generate decay event, helicity-diagonal decay"
  write (u, "(A)")

  process => process_stack%get_process_ptr (procname2)
  model => process%get_model_ptr ()
  call model%set_unstable (25, [procname2], diagonal = .true.)

  allocate (process_instance)
  call process_instance%init (process)
  call process_instance%setup_event_data (model)
  call process_instance%init_simulation (1)

  call process_instance%generate_weighted_event (1)
  call process_instance%write (u)

  call process_instance%final ()
  deallocate (process_instance)

  write (u, "(A)")
  write (u, "(A)")  "* Generate decay event, isotropic decay, &
       &polarized final state"
  write (u, "(A)")

  process => process_stack%get_process_ptr (procname2)
  model => process%get_model_ptr ()
  call model%set_unstable (25, [procname2], isotropic = .true.)
  call model%set_polarized (6)
  call model%set_polarized (-6)

  allocate (process_instance)
  call process_instance%init (process)
  call process_instance%setup_event_data (model)
  call process_instance%init_simulation (1)

  call process_instance%generate_weighted_event (1)
  call process_instance%write (u)

  call process_instance%final ()
  deallocate (process_instance)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call process_stack%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: events_7"

end subroutine events_7
```

32.8 Raw Event I/O

The raw format is for internal use only. All data are stored unformatted, so they can be efficiently be re-read on the same machine, but not necessarily on another machine.

This module explicitly depends on the `events` module which provides the concrete implementation of `event_base`. The other I/O formats access only the methods that are defined in `event_base`.

⟨`eio_raw.f90`⟩≡

```
⟨File header⟩

module eio_raw

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use diagnostics
  use model_data
  use particles
  use event_base
  use eio_data
  use eio_base
  use events

⟨Standard module head⟩

⟨EIO raw: public⟩

⟨EIO raw: parameters⟩

⟨EIO raw: types⟩

contains

⟨EIO raw: procedures⟩

end module eio_raw
```

32.8.1 File Format Version

This is the current default file version.

⟨*EIO raw: parameters*⟩≡

```
integer, parameter :: CURRENT_FILE_VERSION = 2
```

The user may change this number; this should force some compatibility mode for reading and writing. In any case, the file version stored in a event file that we read has to match the expected file version.

History of version numbers:

1. Format for WHIZARD 2.2.0 to 2.2.3. No version number stored in the raw file.

2. Format from 2.2.4 on. File contains version number. The file contains the transformed particle set (if applicable) after the hard-process particle set.

32.8.2 Type

Note the file version number. The default may be reset during initialization, which should enforce some compatibility mode.

⟨*EIO raw: public*⟩≡
```
public :: eio_raw_t
```

⟨*EIO raw: types*⟩≡
```
type, extends (eio_t) :: eio_raw_t
   logical :: reading = .false.
   logical :: writing = .false.
   integer :: unit = 0
   integer :: norm_mode = NORM_UNDEFINED
   real(default) :: sigma = 1
   integer :: n = 1
   integer :: n_alt = 0
   logical :: check = .false.
   integer :: file_version = CURRENT_FILE_VERSION
 contains
 ⟨EIO raw: eio raw: TBP⟩
end type eio_raw_t
```

Output. This is not the actual event format, but a readable account of the current object status.

⟨*EIO raw: eio raw: TBP*⟩≡
```
procedure :: write => eio_raw_write
```

⟨*EIO raw: procedures*⟩≡
```
subroutine eio_raw_write (object, unit)
  class(eio_raw_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Raw event stream:"
  write (u, "(3x,A,L1)")  "Check MD5 sum     = ", object%check
  if (object%n_alt > 0) then
     write (u, "(3x,A,I0)")  "Alternate weights = ", object%n_alt
  end if
  if (object%reading) then
     write (u, "(3x,A,A)")  "Reading from file = ", char (object%filename)
  else if (object%writing) then
     write (u, "(3x,A,A)")  "Writing to file   = ", char (object%filename)
  else
     write (u, "(3x,A)")  "[closed]"
  end if
end subroutine eio_raw_write
```

Finalizer: close any open file.

⟨*EIO raw: eio raw: TBP*⟩+≡
```
procedure :: final => eio_raw_final
```

⟨*EIO raw: procedures*⟩+≡

```
subroutine eio_raw_final (object)
  class(eio_raw_t), intent(inout) :: object
  if (object%reading .or. object%writing) then
     write (msg_buffer, "(A,A,A)")  "Events: closing raw file '", &
          char (object%filename), "'"
     call msg_message ()
     close (object%unit)
     object%reading = .false.
     object%writing = .false.
  end if
end subroutine eio_raw_final
```

Set the `check` flag which determines whether we compare checksums on input.

⟨*EIO raw: eio raw: TBP*⟩+≡

```
procedure :: set_parameters => eio_raw_set_parameters
```

⟨*EIO raw: procedures*⟩+≡

```
subroutine eio_raw_set_parameters (eio, check, version_string, extension)
  class(eio_raw_t), intent(inout) :: eio
  logical, intent(in), optional :: check
  type(string_t), intent(in), optional :: version_string
  type(string_t), intent(in), optional :: extension
  if (present (check))  eio%check = check
  if (present (version_string)) then
     select case (char (version_string))
     case ("", "2.2.4")
        eio%file_version = CURRENT_FILE_VERSION
     case ("2.2")
        eio%file_version = 1
     case default
        call msg_fatal ("Raw event I/O: unsupported version '" &
             // char (version_string) // "'")
        eio%file_version = 0
     end select
  end if
  if (present (extension)) then
     eio%extension = extension
  else
     eio%extension = "evx"
  end if
end subroutine eio_raw_set_parameters
```

Initialize event writing.

⟨*EIO raw: eio raw: TBP*⟩+≡

```
procedure :: init_out => eio_raw_init_out
```

⟨*EIO raw: procedures*⟩+≡

```
subroutine eio_raw_init_out (eio, sample, data, success, extension)
  class(eio_raw_t), intent(inout) :: eio
  type(string_t), intent(in) :: sample
  type(event_sample_data_t), intent(in), optional :: data
  logical, intent(out), optional :: success
  type(string_t), intent(in), optional :: extension
```

```
    character(32) :: md5sum_prc, md5sum_cfg
    character(32), dimension(:), allocatable :: md5sum_alt
    integer :: i
    if (present (extension)) then
       eio%extension  = extension
    else
       eio%extension = "evx"
    end if
    eio%filename = sample // "." // eio%extension
    eio%unit = free_unit ()
    write (msg_buffer, "(A,A,A)")  "Events: writing to raw file '", &
         char (eio%filename), "'"
    call msg_message ()
    eio%writing = .true.
    if (present (data)) then
       md5sum_prc = data%md5sum_prc
       md5sum_cfg = data%md5sum_cfg
       eio%norm_mode = data%norm_mode
       eio%sigma = data%total_cross_section
       eio%n = data%n_evt
       eio%n_alt = data%n_alt
       if (eio%n_alt > 0) then
          !!! !!! !!! Workaround for gfortran 5.0 ICE
          allocate (md5sum_alt (data%n_alt))
          md5sum_alt = data%md5sum_alt
          !!! allocate (md5sum_alt (data%n_alt), source = data%md5sum_alt)
       end if
    else
       md5sum_prc = ""
       md5sum_cfg = ""
    end if
    open (eio%unit, file = char (eio%filename), form = "unformatted", &
         action = "write", status = "replace")
    select case (eio%file_version)
    case (2:);  write (eio%unit)  eio%file_version
    end select
    write (eio%unit)  md5sum_prc
    write (eio%unit)  md5sum_cfg
    write (eio%unit)  eio%norm_mode
    write (eio%unit)  eio%n_alt
    if (allocated (md5sum_alt)) then
       do i = 1, eio%n_alt
          write (eio%unit)  md5sum_alt(i)
       end do
    end if
    if (present (success))  success = .true.
  end subroutine eio_raw_init_out
```

Initialize event reading.

⟨*EIO raw: eio raw: TBP*⟩+≡

```
  procedure :: init_in => eio_raw_init_in
```

⟨*EIO raw: procedures*⟩+≡

```
  subroutine eio_raw_init_in (eio, sample, data, success, extension)
```

```
class(eio_raw_t), intent(inout) :: eio
type(string_t), intent(in) :: sample
type(event_sample_data_t), intent(inout), optional :: data
logical, intent(out), optional :: success
type(string_t), intent(in), optional :: extension
character(32) :: md5sum_prc, md5sum_cfg
character(32), dimension(:), allocatable :: md5sum_alt
integer :: i, file_version
if (present (success))  success = .true.
if (present (extension)) then
   eio%extension = extension
else
   eio%extension = "evx"
end if
eio%filename = sample // "." // eio%extension
eio%unit = free_unit ()
if (present (data)) then
   eio%sigma = data%total_cross_section
   eio%n = data%n_evt
end if
write (msg_buffer, "(A,A,A)")  "Events: reading from raw file '", &
     char (eio%filename), "'"
call msg_message ()
eio%reading = .true.
open (eio%unit, file = char (eio%filename), form = "unformatted", &
     action = "read", status = "old")
select case (eio%file_version)
case (2:);  read (eio%unit)  file_version
case default;  file_version = 1
end select
if (file_version /= eio%file_version) then
   call msg_error ("Reading event file: raw-file version mismatch.")
   if (present (success))  success = .false.
   return
else if (file_version /= CURRENT_FILE_VERSION) then
   call msg_warning ("Reading event file: compatibility mode.")
end if
read (eio%unit)  md5sum_prc
read (eio%unit)  md5sum_cfg
read (eio%unit)  eio%norm_mode
read (eio%unit)  eio%n_alt
if (present (data)) then
   if (eio%n_alt /= data%n_alt) then
      if (present (success))  success = .false.
      return
   end if
end if
allocate (md5sum_alt (eio%n_alt))
do i = 1, eio%n_alt
   read (eio%unit)  md5sum_alt(i)
end do
if (present (success)) then
   if (present (data)) then
      if (eio%check) then
```

```
            if (data%md5sum_prc /= "") then
               success = success .and. md5sum_prc == data%md5sum_prc
            end if
            if (data%md5sum_cfg /= "") then
               success = success .and. md5sum_cfg == data%md5sum_cfg
            end if
            do i = 1, eio%n_alt
               if (data%md5sum_alt(i) /= "") then
                  success = success .and. md5sum_alt(i) == data%md5sum_alt(i)
               end if
            end do
         else
            call msg_warning ("Reading event file: MD5 sum check disabled")
         end if
      end if
   end if
end subroutine eio_raw_init_in
```

Switch from input to output: reopen the file for reading.

⟨*EIO raw: eio raw: TBP*⟩+≡

```
procedure :: switch_inout => eio_raw_switch_inout
```

⟨*EIO raw: procedures*⟩+≡

```
subroutine eio_raw_switch_inout (eio, success)
  class(eio_raw_t), intent(inout) :: eio
  logical, intent(out), optional :: success
  write (msg_buffer, "(A,A,A)")  "Events: appending to raw file '", &
       char (eio%filename), "'"
  call msg_message ()
  close (eio%unit, status = "keep")
  eio%reading = .false.
  open (eio%unit, file = char (eio%filename), form = "unformatted", &
       action = "write", position = "append", status = "old")
  eio%writing = .true.
  if (present (success))  success = .true.
end subroutine eio_raw_switch_inout
```

Output an event. Write first the event indices, then weight and squared matrix element, then the particle set.

We always write the particle set of the hard process. (Note: this should be reconsidered.) We do make a physical copy.

On output, we write the `prc` values for weight and sqme, since these are the values just computed. On input, we store the values as `ref` values. The caller can then decide whether to recompute values and thus obtain distinct `prc` values, or just accept them.

The `passed` flag is not written. This allow us to apply different selection criteria upon rereading.

⟨*EIO raw: eio raw: TBP*⟩+≡

```
procedure :: output => eio_raw_output
```

⟨*EIO raw: procedures*⟩+≡

```
subroutine eio_raw_output (eio, event, i_prc, reading, passed, pacify)
  class(eio_raw_t), intent(inout) :: eio
```

```
    class(generic_event_t), intent(in), target :: event
    logical, intent(in), optional :: reading, passed, pacify
    integer, intent(in) :: i_prc
    type(particle_set_t), pointer :: pset
    integer :: i
    if (eio%writing) then
       if (event%has_valid_particle_set ()) then
          select type (event)
          type is (event_t)
             write (eio%unit)  i_prc
             write (eio%unit)  event%get_i_mci ()
             write (eio%unit)  event%get_i_term ()
             write (eio%unit)  event%get_channel ()
             write (eio%unit)  event%expr%weight_prc
             write (eio%unit)  event%expr%excess_prc
             write (eio%unit)  event%expr%sqme_prc
             do i = 1, eio%n_alt
                write (eio%unit)  event%expr%weight_alt(i)
                write (eio%unit)  event%expr%sqme_alt(i)
             end do
             allocate (pset)
             call event%get_hard_particle_set (pset)
             call pset%write_raw (eio%unit)
             call pset%final ()
             deallocate (pset)
             select case (eio%file_version)
             case (2:)
                if (event%has_transform ()) then
                   write (eio%unit)  .true.
                   pset => event%get_particle_set_ptr ()
                   call pset%write_raw (eio%unit)
                else
                   write (eio%unit)  .false.
                end if
             end select
          class default
             call msg_bug ("Event: write raw: defined only for full event_t")
          end select
       else
          call msg_bug ("Event: write raw: particle set is undefined")
       end if
    else
       call eio%write ()
       call msg_fatal ("Raw event file is not open for writing")
    end if
  end subroutine eio_raw_output
```

Input an event.

Note: the particle set is physically copied. If there is a performance issue, we might choose to pointer-assign it instead, with a different version of `event%set_hard_particle_set`.

⟨*EIO raw: eio raw: TBP*⟩+≡

```
  procedure :: input_i_prc => eio_raw_input_i_prc
```

```
procedure :: input_event => eio_raw_input_event
```

⟨*EIO raw: procedures*⟩+≡

```
subroutine eio_raw_input_i_prc (eio, i_prc, iostat)
  class(eio_raw_t), intent(inout) :: eio
  integer, intent(out) :: i_prc
  integer, intent(out) :: iostat
  if (eio%reading) then
     read (eio%unit, iostat = iostat)  i_prc
  else
     call eio%write ()
     call msg_fatal ("Raw event file is not open for reading")
  end if
end subroutine eio_raw_input_i_prc

subroutine eio_raw_input_event (eio, event, iostat)
  class(eio_raw_t), intent(inout) :: eio
  class(generic_event_t), intent(inout), target :: event
  integer, intent(out) :: iostat
  integer :: i_mci, i_term, channel, i
  real(default) :: weight, excess, sqme
  real(default), dimension(:), allocatable :: weight_alt, sqme_alt
  logical :: has_transform
  type(particle_set_t), pointer :: pset
  class(model_data_t), pointer :: model
  if (eio%reading) then
     select type (event)
     type is (event_t)
        read (eio%unit, iostat = iostat)  i_mci
        if (iostat /= 0)  return
        read (eio%unit, iostat = iostat)  i_term
        if (iostat /= 0)  return
        read (eio%unit, iostat = iostat)  channel
        if (iostat /= 0)  return
        read (eio%unit, iostat = iostat)  weight
        if (iostat /= 0)  return
        read (eio%unit, iostat = iostat)  excess
        if (iostat /= 0)  return
        read (eio%unit, iostat = iostat)  sqme
        if (iostat /= 0)  return
        call event%reset ()
        call event%select (i_mci, i_term, channel)
        if (eio%norm_mode /= NORM_UNDEFINED) then
           call event_normalization_update (weight, &
                eio%sigma, eio%n, event%get_norm_mode (), eio%norm_mode)
           call event_normalization_update (excess, &
                eio%sigma, eio%n, event%get_norm_mode (), eio%norm_mode)
        end if
        call event%set (sqme_ref = sqme, weight_ref = weight, &
             excess_prc = excess)
        if (eio%n_alt /= 0) then
           allocate (sqme_alt (eio%n_alt), weight_alt (eio%n_alt))
           do i = 1, eio%n_alt
              read (eio%unit, iostat = iostat)  weight_alt(i)
              if (iostat /= 0)  return
```

```
            read (eio%unit, iostat = iostat)  sqme_alt(i)
            if (iostat /= 0)  return
         end do
         call event%set (sqme_alt = sqme_alt, weight_alt = weight_alt)
      end if
      model => null ()
      if (associated (event%process)) then
         model => event%process%get_model_ptr ()
      end if
      allocate (pset)
      call pset%read_raw (eio%unit, iostat)
      if (iostat /= 0)  return
      if (associated (model))  call pset%set_model (model)
      call event%set_hard_particle_set (pset)
      call pset%final ()
      deallocate (pset)
      select case (eio%file_version)
      case (2:)
         read (eio%unit, iostat = iostat)  has_transform
         if (iostat /= 0)  return
         if (has_transform) then
            allocate (pset)
            call pset%read_raw (eio%unit, iostat)
            if (iostat /= 0)  return
            if (associated (model)) &
                 call pset%set_model (model)
            call event%link_particle_set (pset)
         end if
      end select
   class default
      call msg_bug ("Event: read raw: defined only for full event_t")
   end select
else
   call eio%write ()
   call msg_fatal ("Raw event file is not open for reading")
end if
end subroutine eio_raw_input_event
```

⟨*EIO raw: eio raw: TBP*⟩+≡

```
procedure :: skip => eio_raw_skip
```

⟨*EIO raw: procedures*⟩+≡

```
subroutine eio_raw_skip (eio, iostat)
  class(eio_raw_t), intent(inout) :: eio
  integer, intent(out) :: iostat
  if (eio%reading) then
     read (eio%unit, iostat = iostat)
  else
     call eio%write ()
     call msg_fatal ("Raw event file is not open for reading")
  end if
end subroutine eio_raw_skip
```

32.8.3 Unit tests

Test module, followed by the corresponding implementation module.

⟨eio_raw_ut.f90⟩≡
```
⟨File header⟩

module eio_raw_ut
  use unit_tests
  use eio_raw_uti

⟨Standard module head⟩

⟨EIO raw: public test⟩

contains

⟨EIO raw: test driver⟩

end module eio_raw_ut
```

⟨eio_raw_uti.f90⟩≡
```
⟨File header⟩

module eio_raw_uti

⟨Use kinds⟩
⟨Use strings⟩
  use model_data
  use variables
  use events
  use eio_data
  use eio_base

  use eio_raw

  use process, only: process_t
  use instances, only: process_instance_t

⟨Standard module head⟩

⟨EIO raw: test declarations⟩

contains

⟨EIO raw: tests⟩

end module eio_raw_uti
```

API: driver for the unit tests below.

⟨*EIO raw: public test*⟩≡
```
  public :: eio_raw_test
```

⟨*EIO raw: test driver*⟩≡
```
  subroutine eio_raw_test (u, results)
    integer, intent(in) :: u
```

```
    type(test_results_t), intent(inout) :: results
  ⟨EIO raw: execute tests⟩
  end subroutine eio_raw_test
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO raw: execute tests*⟩≡

```
  call test (eio_raw_1, "eio_raw_1", &
       "read and write event contents", &
       u, results)
```

⟨*EIO raw: test declarations*⟩≡

```
  public :: eio_raw_1
```

⟨*EIO raw: tests*⟩≡

```
  subroutine eio_raw_1 (u)
    use processes_ut, only: prepare_test_process, cleanup_test_process
    integer, intent(in) :: u
    type(model_data_t), target :: model
    type(event_t), allocatable, target :: event
    type(process_t), allocatable, target :: process
    type(process_instance_t), allocatable, target :: process_instance
    class(eio_t), allocatable :: eio
    integer :: i_prc, iostat
    type(string_t) :: sample

    write (u, "(A)")  "* Test output: eio_raw_1"
    write (u, "(A)")  "*   Purpose: generate and read/write an event"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize test process"

    call model%init_test ()

    allocate (process)
    allocate (process_instance)
    call prepare_test_process (process, process_instance, model)
    call process_instance%setup_event_data ()

    allocate (event)
    call event%basic_init ()
    call event%connect (process_instance, process%get_model_ptr ())

    write (u, "(A)")
    write (u, "(A)")  "* Generate and write an event"
    write (u, "(A)")

    sample = "eio_raw_1"

    allocate (eio_raw_t :: eio)

    call eio%init_out (sample)
    call event%generate (1, [0._default, 0._default])
```

```
call event%evaluate_expressions ()
call event%write (u)
write (u, "(A)")

call eio%output (event, i_prc = 42)
call eio%write (u)
call eio%final ()

call event%final ()
deallocate (event)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event"
write (u, "(A)")

call eio%init_in (sample)

allocate (process_instance)
call process_instance%init (process)
call process_instance%setup_event_data ()
allocate (event)
call event%basic_init ()
call event%connect (process_instance, process%get_model_ptr ())

call eio%input_i_prc (i_prc, iostat)
if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (i_prc):", iostat
call eio%input_event (event, iostat)
if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (event):", iostat
call eio%write (u)

write (u, "(A)")
write (u, "(1x,A,I0)")  "i_prc = ", i_prc
write (u, "(A)")
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Generate and append another event"
write (u, "(A)")

call eio%switch_inout ()
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()
call event%write (u)
write (u, "(A)")

call eio%output (event, i_prc = 5)
call eio%write (u)
call eio%final ()

call event%final ()
deallocate (event)
call process_instance%final ()
```

```
    deallocate (process_instance)

    write (u, "(A)")
    write (u, "(A)")  "* Re-read both events"
    write (u, "(A)")

    call eio%init_in (sample)

    allocate (process_instance)
    call process_instance%init (process)
    call process_instance%setup_event_data ()
    allocate (event)
    call event%basic_init ()
    call event%connect (process_instance, process%get_model_ptr ())

    call eio%input_i_prc (i_prc, iostat)
    if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (i_prc/1):", iostat
    call eio%input_event (event, iostat)
    if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (event/1):", iostat
    call eio%input_i_prc (i_prc, iostat)
    if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (i_prc/2):", iostat
    call eio%input_event (event, iostat)
    if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (event/2):", iostat
    call eio%write (u)

    write (u, "(A)")
    write (u, "(1x,A,I0)")  "i_prc = ", i_prc
    write (u, "(A)")
    call event%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio%final ()
    deallocate (eio)

    call event%final ()
    deallocate (event)

    call cleanup_test_process (process, process_instance)
    deallocate (process_instance)
    deallocate (process)

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_raw_1"

  end subroutine eio_raw_1
```

Test I/O methods

We test the implementation of all I/O methods.

⟨*EIO raw: execute tests*⟩+≡

```
call test (eio_raw_2, "eio_raw_2", &
     "handle multiple weights", &
     u, results)
```

⟨*EIO raw: test declarations*⟩+≡

```
public :: eio_raw_2
```

⟨*EIO raw: tests*⟩+≡

```
subroutine eio_raw_2 (u)
  use processes_ut, only: prepare_test_process, cleanup_test_process
  integer, intent(in) :: u
  type(model_data_t), target :: model
  type(var_list_t) :: var_list
  type(event_t), allocatable, target :: event
  type(process_t), allocatable, target :: process
  type(process_instance_t), allocatable, target :: process_instance
  type(event_sample_data_t) :: data
  class(eio_t), allocatable :: eio
  integer :: i_prc, iostat
  type(string_t) :: sample

  write (u, "(A)")  "* Test output: eio_raw_2"
  write (u, "(A)")  "*   Purpose: generate and read/write an event"
  write (u, "(A)")  "*            with multiple weights"
  write (u, "(A)")

  call model%init_test ()

  write (u, "(A)")  "* Initialize test process"

  allocate (process)
  allocate (process_instance)
  call prepare_test_process (process, process_instance, model)
  call process_instance%setup_event_data ()

  call data%init (n_proc = 1, n_alt = 2)

  call var_list_append_log (var_list, var_str ("?unweighted"), .false., &
       intrinsic = .true.)
  call var_list_append_string (var_list, var_str ("$sample_normalization"), &
       var_str ("auto"), intrinsic = .true.)
  call var_list_append_real (var_list, var_str ("safety_factor"), &
       1._default, intrinsic = .true.)

  allocate (event)
  call event%basic_init (var_list, n_alt = 2)
  call event%connect (process_instance, process%get_model_ptr ())

  write (u, "(A)")
  write (u, "(A)")  "* Generate and write an event"
  write (u, "(A)")

  sample = "eio_raw_2"
```

```
allocate (eio_raw_t :: eio)

call eio%init_out (sample, data)
call event%generate (1, [0._default, 0._default])
call event%evaluate_expressions ()
call event%set (sqme_alt = [2._default, 3._default])
call event%set (weight_alt = &
     [2 * event%get_weight_ref (), 3 * event%get_weight_ref ()])
call event%store_alt_values ()
call event%check ()

call event%write (u)
write (u, "(A)")

call eio%output (event, i_prc = 42)
call eio%write (u)
call eio%final ()

call event%final ()
deallocate (event)
call process_instance%final ()
deallocate (process_instance)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event"
write (u, "(A)")

call eio%init_in (sample, data)

allocate (process_instance)
call process_instance%init (process)
call process_instance%setup_event_data ()
allocate (event)
call event%basic_init (var_list, n_alt = 2)
call event%connect (process_instance, process%get_model_ptr ())

call eio%input_i_prc (i_prc, iostat)
if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (i_prc):", iostat
call eio%input_event (event, iostat)
if (iostat /= 0)  write (u, "(A,I0)")  "I/O error (event):", iostat
call eio%write (u)

write (u, "(A)")
write (u, "(1x,A,I0)")  "i_prc = ", i_prc
write (u, "(A)")
call event%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call eio%final ()
deallocate (eio)

call event%final ()
```

```
    deallocate (event)

    call cleanup_test_process (process, process_instance)
    deallocate (process_instance)
    deallocate (process)

    call model%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: eio_raw_2"

  end subroutine eio_raw_2
```

32.9 Dispatch

An event transform is responsible for dressing a partonic event. Since event transforms are not mutually exclusive but are concatenated, we provide individual dispatchers for each of them.

⟨dispatch_transforms.f90⟩≡

```
⟨File header⟩

module dispatch_transforms

⟨Use strings⟩
  use process
  use variables
  use system_defs, only: LF
  use system_dependencies, only: LHAPDF6_AVAILABLE
  use sf_lhapdf, only: lhapdf_initialize
  use diagnostics
  use models
  use os_interface
  use beam_structures

  use event_base, only: event_callback_t, event_callback_nop_t
  use eio_base
  use eio_raw
  use eio_checkpoints
  use eio_callback
  use eio_lhef
  use eio_hepmc
  use eio_lcio
  use eio_stdhep
  use eio_ascii
  use eio_weights
  use eio_dump

  use event_transforms
  use decays
  use shower_base
  use shower_core
```

```
  use shower
  use shower_pythia6
  use hadrons
  use mlm_matching
  use powheg_matching
  use ckkw_matching
  use tauola_interface !NODEP!
  use evt_nlo
```

⟨*Standard module head*⟩

⟨*Dispatch transforms: public*⟩

```
contains
```

⟨*Dispatch transforms: procedures*⟩

```
end module dispatch_transforms
```

⟨*Dispatch transforms: public*⟩≡

```
  public :: dispatch_evt_nlo
```

⟨*Dispatch transforms: procedures*⟩≡

```
  subroutine dispatch_evt_nlo (evt, keep_failed_events)
    class(evt_t), intent(out), pointer :: evt
    logical, intent(in) :: keep_failed_events
    call msg_message ("Simuate: activating fixed-order NLO events")
    allocate (evt_nlo_t :: evt)
    evt%only_weighted_events = .true.
    select type (evt)
    type is (evt_nlo_t)
       evt%i_evaluation = 0
       evt%keep_failed_events = keep_failed_events
    end select
  end subroutine dispatch_evt_nlo
```

⟨*Dispatch transforms: public*⟩+≡

```
  public :: dispatch_evt_decay
```

⟨*Dispatch transforms: procedures*⟩+≡

```
  subroutine dispatch_evt_decay (evt, var_list)
    class(evt_t), intent(out), pointer :: evt
    type(var_list_t), intent(in) :: var_list
    logical :: allow_decays
    allow_decays = var_list%get_lval (var_str ("?allow_decays"))
    if (allow_decays) then
       allocate (evt_decay_t :: evt)
       call msg_message ("Simulate: activating decays")
    else
       evt => null ()
    end if
  end subroutine dispatch_evt_decay
```

⟨*Dispatch transforms: public*⟩+≡

```
  public :: dispatch_evt_shower
```

⟨*Dispatch transforms: procedures*⟩+≡

```
subroutine dispatch_evt_shower (evt, var_list, model, fallback_model, &
     os_data, beam_structure, process)
  class(evt_t), intent(out), pointer :: evt
  type(var_list_t), intent(in) :: var_list
  type(model_t), pointer, intent(in) :: model, fallback_model
  type(os_data_t), intent(in) :: os_data
  type(beam_structure_t), intent(in) :: beam_structure
  type(process_t), intent(in), optional :: process
  type(string_t) :: lhapdf_file, lhapdf_dir, process_name
  integer :: lhapdf_member
  type(shower_settings_t) :: settings
  type(taudec_settings_t) :: taudec_settings
  call msg_message ("Simulate: activating parton shower")
  allocate (evt_shower_t :: evt)
  call settings%init (var_list)
  if (associated (model)) then
     call taudec_settings%init (var_list, model)
  else
     call taudec_settings%init (var_list, fallback_model)
  end if
  if (present (process)) then
     process_name = process%get_id ()
  else
     process_name = 'dispatch_testing'
  end if
  select type (evt)
  type is (evt_shower_t)
     call evt%init (fallback_model, os_data)
     lhapdf_member = &
          var_list%get_ival (var_str ("lhapdf_member"))
     if (LHAPDF6_AVAILABLE) then
        lhapdf_dir = &
             var_list%get_sval (var_str ("$lhapdf_dir"))
        lhapdf_file = &
             var_list%get_sval (var_str ("$lhapdf_file"))
        call lhapdf_initialize &
             (1, lhapdf_dir, lhapdf_file, lhapdf_member, evt%pdf_data%pdf)
     end if
     if (present (process))  call evt%pdf_data%setup ("Shower", &
          beam_structure, lhapdf_member, process%get_pdf_set ())
     select case (settings%method)
     case (PS_WHIZARD)
        allocate (shower_t :: evt%shower)
     case (PS_PYTHIA6)
        allocate (shower_pythia6_t :: evt%shower)
     case default
        call msg_fatal ('Shower: Method ' // &
          char (var_list%get_sval (var_str ("$shower_method"))) // &
          'not implemented!')
     end select
     call evt%shower%init (settings, taudec_settings, evt%pdf_data)
  end select
  call dispatch_matching (evt, settings, var_list, process_name)
```

```
end subroutine dispatch_evt_shower
```

⟨*Dispatch transforms: public*⟩+≡

```
public :: dispatch_matching
```

⟨*Dispatch transforms: procedures*⟩+≡

```
subroutine dispatch_matching (evt, settings, var_list, process_name)
  class(evt_t), intent(inout) :: evt
  type(var_list_t), intent(in) :: var_list
  type(string_t), intent(in) :: process_name
  type(shower_settings_t), intent(in) :: settings
  select type (evt)
  type is (evt_shower_t)
     if (settings%mlm_matching .and. settings%ckkw_matching) then
        call msg_fatal ("Both MLM and CKKW matching activated," // &
             LF // "     aborting simulation")
     end if
     if (settings%powheg_matching) then
        call msg_message ("Simulate: applying POWHEG matching")
        allocate (powheg_matching_t :: evt%matching)
     end if
     if (settings%mlm_matching) then
        call msg_message ("Simulate: applying MLM matching")
        allocate (mlm_matching_t :: evt%matching)
     end if
     if (settings%ckkw_matching) then
        call msg_warning ("Simulate: CKKW(-L) matching not yet supported")
        allocate (ckkw_matching_t :: evt%matching)
     end if
     if (allocated (evt%matching)) &
          call evt%matching%init (var_list, process_name)
  end select
end subroutine dispatch_matching
```

⟨*Dispatch transforms: public*⟩+≡

```
public :: dispatch_evt_hadrons
```

⟨*Dispatch transforms: procedures*⟩+≡

```
subroutine dispatch_evt_hadrons (evt, var_list, fallback_model)
  class(evt_t), intent(out), pointer :: evt
  type(var_list_t), intent(in) :: var_list
  type(model_t), pointer, intent(in) :: fallback_model
  type(shower_settings_t) :: shower_settings
  type(hadron_settings_t) :: hadron_settings
  allocate (evt_hadrons_t :: evt)
  call msg_message ("Simulate: activating hadronization")
  call shower_settings%init (var_list)
  call hadron_settings%init (var_list)
  select type (evt)
  type is (evt_hadrons_t)
     call evt%init (fallback_model)
     select case (hadron_settings%method)
     case (HADRONS_WHIZARD)
        allocate (hadrons_hadrons_t :: evt%hadrons)
```

```
      case (HADRONS_PYTHIA6)
         allocate (hadrons_pythia6_t :: evt%hadrons)
      case (HADRONS_PYTHIA8)
         allocate (hadrons_pythia8_t :: evt%hadrons)
      case default
         call msg_fatal ('Hadronization: Method ' // &
           char (var_list%get_sval (var_str ("hadronization_method"))) // &
           'not implemented!')
      end select
      call evt%hadrons%init &
           (shower_settings, hadron_settings, fallback_model)
    end select
  end subroutine dispatch_evt_hadrons
```

We cannot put this in the `events` subdir due to `eio_raw_t`, which is defined here.

⟨*Dispatch transforms: public*⟩+≡

```
  public :: dispatch_eio
```

⟨*Dispatch transforms: procedures*⟩+≡

```
  subroutine dispatch_eio (eio, method, var_list, fallback_model, &
         event_callback)
    class(eio_t), allocatable, intent(inout) :: eio
    type(string_t), intent(in) :: method
    type(var_list_t), intent(in) :: var_list
    type(model_t), target, intent(in) :: fallback_model
    class(event_callback_t), allocatable, intent(in) :: event_callback
    logical :: check, keep_beams, keep_remnants, recover_beams
    logical :: use_alpha_s_from_file, use_scale_from_file
    logical :: write_sqme_prc, write_sqme_ref, write_sqme_alt
    logical :: output_cross_section, ensure_order
    type(string_t) :: lhef_version, lhef_extension, raw_version
    type(string_t) :: extension_default, debug_extension, dump_extension, &
         extension_hepmc, &
         extension_lha, extension_hepevt, extension_ascii_short, &
         extension_ascii_long, extension_athena, extension_mokka, &
         extension_stdhep, extension_stdhep_up, extension_stdhep_ev4, &
         extension_raw, extension_hepevt_verb, extension_lha_verb, &
         extension_lcio
    integer :: checkpoint
    logical :: show_process, show_transforms, show_decay, verbose, pacified
    logical :: dump_weights, dump_compressed, dump_summary, dump_screen
    keep_beams = &
         var_list%get_lval (var_str ("?keep_beams"))
    keep_remnants = &
         var_list%get_lval (var_str ("?keep_remnants"))
    ensure_order = &
         var_list%get_lval (var_str ("?hepevt_ensure_order"))
    recover_beams = &
         var_list%get_lval (var_str ("?recover_beams"))
    use_alpha_s_from_file = &
         var_list%get_lval (var_str ("?use_alpha_s_from_file"))
    use_scale_from_file = &
         var_list%get_lval (var_str ("?use_scale_from_file"))
```

```
select case (char (method))
case ("raw")
   allocate (eio_raw_t :: eio)
   select type (eio)
   type is (eio_raw_t)
      check = &
           var_list%get_lval (var_str ("?check_event_file"))
      raw_version = &
           var_list%get_sval (var_str ("$event_file_version"))
      extension_raw = &
           var_list%get_sval (var_str ("$extension_raw"))
      call eio%set_parameters (check, raw_version, extension_raw)
   end select
case ("checkpoint")
   allocate (eio_checkpoints_t :: eio)
   select type (eio)
   type is (eio_checkpoints_t)
      checkpoint = &
           var_list%get_ival (var_str ("checkpoint"))
      pacified = &
           var_list%get_lval (var_str ("?pacify"))
      call eio%set_parameters (checkpoint, blank = pacified)
   end select
case ("callback")
   allocate (eio_callback_t :: eio)
   select type (eio)
   type is (eio_callback_t)
      checkpoint = &
           var_list%get_ival (var_str ("event_callback_interval"))
      if (allocated (event_callback)) then
         call eio%set_parameters (event_callback, checkpoint)
      else
         call eio%set_parameters (event_callback_nop_t (), 0)
      end if
   end select
case ("lhef")
   allocate (eio_lhef_t :: eio)
   select type (eio)
   type is (eio_lhef_t)
      lhef_version = &
           var_list%get_sval (var_str ("$lhef_version"))
      lhef_extension = &
           var_list%get_sval (var_str ("$lhef_extension"))
      write_sqme_prc = &
           var_list%get_lval (var_str ("?lhef_write_sqme_prc"))
      write_sqme_ref = &
           var_list%get_lval (var_str ("?lhef_write_sqme_ref"))
      write_sqme_alt = &
           var_list%get_lval (var_str ("?lhef_write_sqme_alt"))
      call eio%set_parameters ( &
           keep_beams, keep_remnants, recover_beams, &
           use_alpha_s_from_file, use_scale_from_file, &
           char (lhef_version), lhef_extension, &
           write_sqme_ref, write_sqme_prc, write_sqme_alt)
```

```
   end select
case ("hepmc")
   allocate (eio_hepmc_t :: eio)
   select type (eio)
   type is (eio_hepmc_t)
      output_cross_section = &
           var_list%get_lval (var_str ("?hepmc_output_cross_section"))
      extension_hepmc = &
           var_list%get_sval (var_str ("$extension_hepmc"))
      call eio%set_parameters (recover_beams, &
           use_alpha_s_from_file, use_scale_from_file, &
           extension_hepmc, output_cross_section)
   end select
case ("lcio")
   allocate (eio_lcio_t :: eio)
   select type (eio)
   type is (eio_lcio_t)
      extension_lcio = &
           var_list%get_sval (var_str ("$extension_lcio"))
      call eio%set_parameters (recover_beams, &
           use_alpha_s_from_file, use_scale_from_file, &
           extension_lcio)
   end select
case ("stdhep")
   allocate (eio_stdhep_hepevt_t :: eio)
   select type (eio)
   type is (eio_stdhep_hepevt_t)
      extension_stdhep = &
           var_list%get_sval (var_str ("$extension_stdhep"))
      call eio%set_parameters &
           (keep_beams, keep_remnants, ensure_order, recover_beams, &
            use_alpha_s_from_file, use_scale_from_file, extension_stdhep)
   end select
case ("stdhep_up")
   allocate (eio_stdhep_hepeup_t :: eio)
   select type (eio)
   type is (eio_stdhep_hepeup_t)
      extension_stdhep_up = &
           var_list%get_sval (var_str ("$extension_stdhep_up"))
      call eio%set_parameters (keep_beams, keep_remnants, ensure_order, &
           recover_beams, use_alpha_s_from_file, &
           use_scale_from_file, extension_stdhep_up)
   end select
case ("stdhep_ev4")
   allocate (eio_stdhep_hepev4_t :: eio)
   select type (eio)
   type is (eio_stdhep_hepev4_t)
      extension_stdhep_ev4 = &
           var_list%get_sval (var_str ("$extension_stdhep_ev4"))
      call eio%set_parameters &
           (keep_beams, keep_remnants, ensure_order, recover_beams, &
            use_alpha_s_from_file, use_scale_from_file, extension_stdhep_ev4)
   end select
case ("ascii")
```

```
   allocate (eio_ascii_ascii_t :: eio)
   select type (eio)
   type is (eio_ascii_ascii_t)
      extension_default = &
           var_list%get_sval (var_str ("$extension_default"))
      call eio%set_parameters &
           (keep_beams, keep_remnants, ensure_order, extension_default)
   end select
case ("athena")
   allocate (eio_ascii_athena_t :: eio)
   select type (eio)
   type is (eio_ascii_athena_t)
      extension_athena = &
           var_list%get_sval (var_str ("$extension_athena"))
      call eio%set_parameters &
           (keep_beams, keep_remnants, ensure_order, extension_athena)
   end select
case ("debug")
   allocate (eio_ascii_debug_t :: eio)
   select type (eio)
   type is (eio_ascii_debug_t)
      debug_extension = &
           var_list%get_sval (var_str ("$debug_extension"))
      show_process = &
           var_list%get_lval (var_str ("?debug_process"))
      show_transforms = &
           var_list%get_lval (var_str ("?debug_transforms"))
      show_decay = &
           var_list%get_lval (var_str ("?debug_decay"))
      verbose = &
           var_list%get_lval (var_str ("?debug_verbose"))
      call eio%set_parameters ( &
           extension = debug_extension, &
           show_process = show_process, &
           show_transforms = show_transforms, &
           show_decay = show_decay, &
           verbose = verbose)
   end select
case ("dump")
   allocate (eio_dump_t :: eio)
   select type (eio)
   type is (eio_dump_t)
      dump_extension = &
           var_list%get_sval (var_str ("$dump_extension"))
      pacified = &
           var_list%get_lval (var_str ("?pacify"))
      dump_weights = &
           var_list%get_lval (var_str ("?dump_weights"))
      dump_compressed = &
           var_list%get_lval (var_str ("?dump_compressed"))
      dump_summary = &
           var_list%get_lval (var_str ("?dump_summary"))
      dump_screen = &
           var_list%get_lval (var_str ("?dump_screen"))
```

```
      call eio%set_parameters ( &
           extension = dump_extension, &
           pacify = pacified, &
           weights = dump_weights, &
           compressed = dump_compressed, &
           summary = dump_summary, &
           screen = dump_screen)
   end select
case ("hepevt")
   allocate (eio_ascii_hepevt_t :: eio)
   select type (eio)
   type is (eio_ascii_hepevt_t)
      extension_hepevt = &
           var_list%get_sval (var_str ("$extension_hepevt"))
      call eio%set_parameters &
           (keep_beams, keep_remnants, ensure_order, extension_hepevt)
   end select
case ("hepevt_verb")
   allocate (eio_ascii_hepevt_verb_t :: eio)
   select type (eio)
   type is (eio_ascii_hepevt_verb_t)
      extension_hepevt_verb = &
           var_list%get_sval (var_str ("$extension_hepevt_verb"))
      call eio%set_parameters &
           (keep_beams, keep_remnants, ensure_order, extension_hepevt_verb)
   end select
case ("lha")
   allocate (eio_ascii_lha_t :: eio)
   select type (eio)
   type is (eio_ascii_lha_t)
      extension_lha = &
           var_list%get_sval (var_str ("$extension_lha"))
      call eio%set_parameters &
           (keep_beams, keep_remnants, ensure_order, extension_lha)
   end select
case ("lha_verb")
   allocate (eio_ascii_lha_verb_t :: eio)
   select type (eio)
   type is (eio_ascii_lha_verb_t)
      extension_lha_verb = var_list%get_sval ( &
           var_str ("$extension_lha_verb"))
      call eio%set_parameters &
           (keep_beams, keep_remnants, ensure_order, extension_lha_verb)
   end select
case ("long")
   allocate (eio_ascii_long_t :: eio)
   select type (eio)
   type is (eio_ascii_long_t)
      extension_ascii_long = &
           var_list%get_sval (var_str ("$extension_ascii_long"))
      call eio%set_parameters &
           (keep_beams, keep_remnants, ensure_order, extension_ascii_long)
   end select
case ("mokka")
```

```
         allocate (eio_ascii_mokka_t :: eio)
         select type (eio)
         type is (eio_ascii_mokka_t)
            extension_mokka = &
                 var_list%get_sval (var_str ("$extension_mokka"))
            call eio%set_parameters &
                 (keep_beams, keep_remnants, ensure_order, extension_mokka)
         end select
      case ("short")
         allocate (eio_ascii_short_t :: eio)
         select type (eio)
         type is (eio_ascii_short_t)
            extension_ascii_short = &
                 var_list%get_sval (var_str ("$extension_ascii_short"))
            call eio%set_parameters &
                 (keep_beams, keep_remnants, ensure_order, extension_ascii_short)
         end select
      case ("weight_stream")
         allocate (eio_weights_t :: eio)
         select type (eio)
         type is (eio_weights_t)
            pacified = &
                 var_list%get_lval (var_str ("?pacify"))
            call eio%set_parameters (pacify = pacified)
         end select
      case default
         call msg_fatal ("Event I/O method '" // char (method) &
              // "' not implemented")
      end select
      call eio%set_fallback_model (fallback_model)
    end subroutine dispatch_eio
```

32.9.1 Unit tests

Test module, followed by the corresponding implementation module.

⟨dispatch_transforms_ut.f90⟩≡

```
⟨File header⟩

module dispatch_transforms_ut
  use unit_tests
  use dispatch_transforms_uti

⟨Standard module head⟩

⟨Dispatch transforms: public test⟩

contains

⟨Dispatch transforms: test driver⟩

end module dispatch_transforms_ut
```

⟨dispatch_transforms_uti.f90⟩≡

```
⟨File header⟩

module dispatch_transforms_uti

⟨Use strings⟩
  use format_utils, only: write_separator
  use variables
  use event_base, only: event_callback_t
  use models, only: model_t, model_list_t
  use models, only: syntax_model_file_init, syntax_model_file_final
  use beam_structures, only: beam_structure_t
  use eio_base, only: eio_t
  use os_interface, only: os_data_t, os_data_init
  use event_transforms, only: evt_t
  use dispatch_transforms

⟨Standard module head⟩

⟨Dispatch transforms: test declarations⟩

contains

⟨Dispatch transforms: tests⟩

end module dispatch_transforms_uti
```

API: driver for the unit tests below.

⟨*Dispatch transforms: public test*⟩≡

```
  public ::dispatch_transforms_test
```

⟨*Dispatch transforms: test driver*⟩≡

```
  subroutine dispatch_transforms_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨Dispatch transforms: execute tests⟩
  end subroutine dispatch_transforms_test
```

Event I/O

⟨*Dispatch transforms: execute tests*⟩≡

```
  call test (dispatch_transforms_1, "dispatch_transforms_1", &
       "event I/O", &
       u, results)
```

⟨*Dispatch transforms: test declarations*⟩≡

```
  public :: dispatch_transforms_1
```

⟨*Dispatch transforms: tests*⟩≡

```
  subroutine dispatch_transforms_1 (u)
    integer, intent(in) :: u
    type(var_list_t) :: var_list
    type(model_list_t) :: model_list
    type(model_t), pointer :: model
```

```
type(os_data_t) :: os_data
class(event_callback_t), allocatable :: event_callback
class(eio_t), allocatable :: eio

write (u, "(A)")  "* Test output: dispatch_transforms_1"
write (u, "(A)")  "*   Purpose: allocate an event I/O (eio) stream"
write (u, "(A)")

call var_list%init_defaults (0)
call os_data_init (os_data)
call syntax_model_file_init ()
call model_list%read_model (var_str ("SM_hadrons"), &
     var_str ("SM_hadrons.mdl"), os_data, model)

write (u, "(A)")  "* Allocate as raw"
write (u, "(A)")

call dispatch_eio (eio, var_str ("raw"), var_list, &
     model, event_callback)

call eio%write (u)

call eio%final ()
deallocate (eio)

write (u, "(A)")
write (u, "(A)")  "* Allocate as checkpoints:"
write (u, "(A)")

call dispatch_eio (eio, var_str ("checkpoint"), var_list, &
     model, event_callback)

call eio%write (u)

call eio%final ()
deallocate (eio)

write (u, "(A)")
write (u, "(A)")  "* Allocate as LHEF:"
write (u, "(A)")

call var_list%set_string (var_str ("$lhef_extension"), &
     var_str ("lhe_custom"), is_known = .true.)
call dispatch_eio (eio, var_str ("lhef"), var_list, &
     model, event_callback)

call eio%write (u)

call eio%final ()
deallocate (eio)

write (u, "(A)")
write (u, "(A)")  "* Allocate as HepMC:"
write (u, "(A)")
```

```
    call dispatch_eio (eio, var_str ("hepmc"), var_list, &
         model, event_callback)

    call eio%write (u)

    call eio%final ()
    deallocate (eio)

    write (u, "(A)")
    write (u, "(A)")  "* Allocate as weight_stream"
    write (u, "(A)")

    call dispatch_eio (eio, var_str ("weight_stream"), var_list, &
         model, event_callback)

    call eio%write (u)

    call eio%final ()
    deallocate (eio)

    write (u, "(A)")
    write (u, "(A)")  "* Allocate as debug format"
    write (u, "(A)")

    call var_list%set_log (var_str ("?debug_verbose"), &
         .false., is_known = .true.)
    call dispatch_eio (eio, var_str ("debug"), var_list, &
         model, event_callback)

    call eio%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio%final ()
    call var_list%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: dispatch_transforms_1"

  end subroutine dispatch_transforms_1
```

Event transforms

This test dispatches an `evt` (event transform) object. Currently, the only non-trivial transform is the partonic decay chain..

⟨*Dispatch transforms: execute tests*⟩+≡

```
  call test (dispatch_transforms_2, "dispatch_transforms_2", &
       "event transforms", &
       u, results)
```

⟨*Dispatch transforms: test declarations*⟩+≡

```
public :: dispatch_transforms_2
```

⟨*Dispatch transforms: tests*⟩+≡

```
subroutine dispatch_transforms_2 (u)
  integer, intent(in) :: u
  type(var_list_t) :: var_list
  type(model_list_t) :: model_list
  type(model_t), pointer :: model
  type(os_data_t) :: os_data
  type(beam_structure_t) :: beam_structure
  class(evt_t), pointer :: evt

  write (u, "(A)")  "* Test output: dispatch_transforms_2"
  write (u, "(A)")  "*   Purpose: configure event transform"
  write (u, "(A)")

  call syntax_model_file_init ()
  call var_list%init_defaults (0)
  call os_data_init (os_data)
  call model_list%read_model (var_str ("SM_hadrons"), &
       var_str ("SM_hadrons.mdl"), os_data, model)

  write (u, "(A)")  "* Partonic decays"
  write (u, "(A)")

  call dispatch_evt_decay (evt, var_list)
  call evt%write (u, verbose = .true., more_verbose = .true.)

  call evt%final ()
  deallocate (evt)

  write (u, "(A)")
  write (u, "(A)")  "* Shower"
  write (u, "(A)")

  call var_list%set_log (var_str ("?allow_shower"), .true., &
       is_known = .true.)
  call var_list%set_string (var_str ("$shower_method"), &
       var_str ("WHIZARD"), is_known = .true.)
  call dispatch_evt_shower (evt, var_list, model, &
       model, os_data, beam_structure)
  call evt%write (u)
  call write_separator (u, 2)

  call evt%final ()
  deallocate (evt)

  call var_list%final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: dispatch_transforms_2"

end subroutine dispatch_transforms_2
```

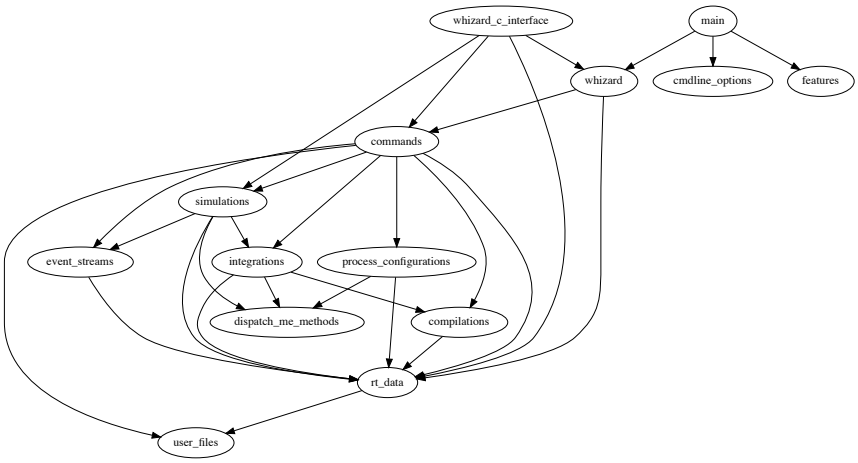

Figure 32.2: Module dependencies in `src/whizard-core`.

Chapter 33

Integration and Simulation

33.1 User-controlled File I/O

The SINDARIN language includes commands that write output to file (input may be added later). We identify files by their name, and manage the unit internally. We need procedures for opening, closing, and printing files.

⟨`user_files.f90`⟩≡

```
⟨File header⟩

module user_files

⟨Use strings⟩
  use io_units
  use diagnostics
  use ifiles
  use analysis

⟨Standard module head⟩

⟨User files: public⟩

⟨User files: types⟩

⟨User files: interfaces⟩

contains

⟨User files: procedures⟩

end module user_files
```

33.1.1 The file type

This is a type that describes an open user file and its properties. The entry is part of a doubly-linked list.

⟨*User files: types*⟩≡

```
type :: file_t
   private
   type(string_t) :: name
   integer :: unit = -1
   logical :: reading = .false.
   logical :: writing = .false.
   type(file_t), pointer :: prev => null ()
   type(file_t), pointer :: next => null ()
end type file_t
```

The initializer opens the file.

⟨*User files: procedures*⟩≡

```
subroutine file_init (file, name, action, status, position)
  type(file_t), intent(out) :: file
  type(string_t), intent(in) :: name
  character(len=*), intent(in) :: action, status, position
  file%unit = free_unit ()
  file%name = name
  open (unit = file%unit, file = char (file%name), &
        action = action, status = status, position = position)
  select case (action)
  case ("read")
     file%reading = .true.
  case ("write")
     file%writing = .true.
  case ("readwrite")
     file%reading = .true.
     file%writing = .true.
  end select
end subroutine file_init
```

The finalizer closes it.

⟨*User files: procedures*⟩+≡

```
subroutine file_final (file)
  type(file_t), intent(inout) :: file
  close (unit = file%unit)
  file%unit = -1
end subroutine file_final
```

Check if a file is open with correct status.

⟨*User files: procedures*⟩+≡

```
function file_is_open (file, action) result (flag)
  logical :: flag
  type(file_t), intent(in) :: file
  character(*), intent(in) :: action
  select case (action)
  case ("read")
     flag = file%reading
  case ("write")
     flag = file%writing
  case ("readwrite")
     flag = file%reading .and. file%writing
```

```
      case default
         call msg_bug ("Checking file '" // char (file%name) &
              // "': illegal action specifier")
      end select
   end function file_is_open
```

Write to the file. Error if in wrong mode. If there is no string, just write an empty record. If there is a string, respect the `advancing` option.

⟨*User files: procedures*⟩+≡

```
   subroutine file_write_string (file, string, advancing)
      type(file_t), intent(in) :: file
      type(string_t), intent(in), optional :: string
      logical, intent(in), optional :: advancing
      if (file%writing) then
         if (present (string)) then
            if (present (advancing)) then
               if (advancing) then
                  write (file%unit, "(A)")  char (string)
               else
                  write (file%unit, "(A)", advance="no")  char (string)
               end if
            else
               write (file%unit, "(A)")  char (string)
            end if
         else
            write (file%unit, *)
         end if
      else
         call msg_error ("Writing to file: File '" // char (file%name) &
              // "' is not open for writing.")
      end if
   end subroutine file_write_string
```

Write a whole ifile, line by line.

⟨*User files: procedures*⟩+≡

```
   subroutine file_write_ifile (file, ifile)
      type(file_t), intent(in) :: file
      type(ifile_t), intent(in) :: ifile
      type(line_p) :: line
      call line_init (line, ifile)
      do while (line_is_associated (line))
         call file_write_string (file, line_get_string_advance (line))
      end do
   end subroutine file_write_ifile
```

Write an analysis object (or all objects) to an open file.

⟨*User files: procedures*⟩+≡

```
   subroutine file_write_analysis (file, tag)
      type(file_t), intent(in) :: file
      type(string_t), intent(in), optional :: tag
      if (file%writing) then
         if (present (tag)) then
```

```
         call analysis_write (tag, unit = file%unit)
      else
         call analysis_write (unit = file%unit)
      end if
   else
      call msg_error ("Writing analysis to file: File '" // char (file%name) &
           // "' is not open for writing.")
   end if
end subroutine file_write_analysis
```

33.1.2 The file list

We maintain a list of all open files and their attributes. The list must be doubly-linked because we may delete entries.

⟨*User files: public*⟩≡

```
public :: file_list_t
```

⟨*User files: types*⟩+≡

```
type :: file_list_t
   type(file_t), pointer :: first => null ()
   type(file_t), pointer :: last => null ()
end type file_list_t
```

There is no initialization routine, but a finalizer which deletes all:

⟨*User files: public*⟩+≡

```
public :: file_list_final
```

⟨*User files: procedures*⟩+≡

```
subroutine file_list_final (file_list)
  type(file_list_t), intent(inout) :: file_list
  type(file_t), pointer :: current
  do while (associated (file_list%first))
     current => file_list%first
     file_list%first => current%next
     call file_final (current)
     deallocate (current)
  end do
  file_list%last => null ()
end subroutine file_list_final
```

Find an entry in the list. Return null pointer on failure.

⟨*User files: procedures*⟩+≡

```
function file_list_get_file_ptr (file_list, name) result (current)
  type(file_t), pointer :: current
  type(file_list_t), intent(in) :: file_list
  type(string_t), intent(in) :: name
  current => file_list%first
  do while (associated (current))
     if (current%name == name)  return
     current => current%next
  end do
end function file_list_get_file_ptr
```

Check if a file is open, public version:

⟨*User files: public*⟩+≡

```
public :: file_list_is_open
```

⟨*User files: procedures*⟩+≡

```
function file_list_is_open (file_list, name, action) result (flag)
  logical :: flag
  type(file_list_t), intent(in) :: file_list
  type(string_t), intent(in) :: name
  character(len=*), intent(in) :: action
  type(file_t), pointer :: current
  current => file_list_get_file_ptr (file_list, name)
  if (associated (current)) then
     flag = file_is_open (current, action)
  else
     flag = .false.
  end if
end function file_list_is_open
```

Append a new file entry, i.e., open this file. Error if it is already open.

⟨*User files: public*⟩+≡

```
public :: file_list_open
```

⟨*User files: procedures*⟩+≡

```
subroutine file_list_open (file_list, name, action, status, position)
  type(file_list_t), intent(inout) :: file_list
  type(string_t), intent(in) :: name
  character(len=*), intent(in) :: action, status, position
  type(file_t), pointer :: current
  if (.not. associated (file_list_get_file_ptr (file_list, name))) then
     allocate (current)
     call msg_message ("Opening file '" // char (name) // "' for output")
     call file_init (current, name, action, status, position)
     if (associated (file_list%last)) then
        file_list%last%next => current
        current%prev => file_list%last
     else
        file_list%first => current
     end if
     file_list%last => current
  else
     call msg_error ("Opening file: File '" // char (name) &
          // "' is already open.")
  end if
end subroutine file_list_open
```

Delete a file entry, i.e., close this file. Error if it is not open.

⟨*User files: public*⟩+≡

```
public :: file_list_close
```

⟨*User files: procedures*⟩+≡

```
subroutine file_list_close (file_list, name)
  type(file_list_t), intent(inout) :: file_list
  type(string_t), intent(in) :: name
```

```
    type(file_t), pointer :: current
    current => file_list_get_file_ptr (file_list, name)
    if (associated (current)) then
       if (associated (current%prev)) then
          current%prev%next => current%next
       else
          file_list%first => current%next
       end if
       if (associated (current%next)) then
          current%next%prev => current%prev
       else
          file_list%last => current%prev
       end if
       call msg_message ("Closing file '" // char (name) // "' for output")
       call file_final (current)
       deallocate (current)
    else
       call msg_error ("Closing file: File '" // char (name) &
            // "' is not open.")
    end if
  end subroutine file_list_close
```

Write a string to file. Error if it is not open.

⟨*User files: public*⟩+≡

```
  public :: file_list_write
```

⟨*User files: interfaces*⟩≡

```
  interface file_list_write
     module procedure file_list_write_string
     module procedure file_list_write_ifile
  end interface
```

⟨*User files: procedures*⟩+≡

```
  subroutine file_list_write_string (file_list, name, string, advancing)
    type(file_list_t), intent(in) :: file_list
    type(string_t), intent(in) :: name
    type(string_t), intent(in), optional :: string
    logical, intent(in), optional :: advancing
    type(file_t), pointer :: current
    current => file_list_get_file_ptr (file_list, name)
    if (associated (current)) then
       call file_write_string (current, string, advancing)
    else
       call msg_error ("Writing to file: File '" // char (name) &
            // "'is not open.")
    end if
  end subroutine file_list_write_string

  subroutine file_list_write_ifile (file_list, name, ifile)
    type(file_list_t), intent(in) :: file_list
    type(string_t), intent(in) :: name
    type(ifile_t), intent(in) :: ifile
    type(file_t), pointer :: current
    current => file_list_get_file_ptr (file_list, name)
    if (associated (current)) then
```

```
      call file_write_ifile (current, ifile)
   else
      call msg_error ("Writing to file: File '" // char (name) &
           // "'is not open.")
   end if
end subroutine file_list_write_ifile
```

Write an analysis object or all objects to data file. Error if it is not open. If the file name is empty, write to standard output.

⟨*User files: public*⟩+≡

```
public :: file_list_write_analysis
```

⟨*User files: procedures*⟩+≡

```
subroutine file_list_write_analysis (file_list, name, tag)
   type(file_list_t), intent(in) :: file_list
   type(string_t), intent(in) :: name
   type(string_t), intent(in), optional :: tag
   type(file_t), pointer :: current
   if (name == "") then
      if (present (tag)) then
         call analysis_write (tag)
      else
         call analysis_write
      end if
   else
      current => file_list_get_file_ptr (file_list, name)
      if (associated (current)) then
         call file_write_analysis (current, tag)
      else
         call msg_error ("Writing analysis to file: File '" // char (name) &
              // "' is not open.")
      end if
   end if
end subroutine file_list_write_analysis
```

33.2 Runtime data

⟨rt_data.f90⟩≡
⟨*File header*⟩

```
module rt_data
```

⟨*Use kinds*⟩
⟨*Use strings*⟩

```
  use io_units
  use format_utils, only: write_separator
  use format_defs, only: FMT_19, FMT_12
  use system_dependencies
  use diagnostics
  use os_interface
  use lexers
  use parser
  use models
  use subevents
  use pdg_arrays
  use variables, only: var_list_t
  use process_libraries
  use prclib_stacks
  use prc_core, only: helicity_selection_t
  use beam_structures
  use event_base, only: event_callback_t
  use user_files
  use process_stacks
  use iterations
```

⟨*Standard module head*⟩

⟨*RT data: public*⟩

⟨*RT data: types*⟩

```
contains
```

⟨*RT data: procedures*⟩

```
end module rt_data
```

33.2.1 Strategy for models and variables

The program manages its data via a main `rt_data_t` object. During program flow, various commands create and use local `rt_data_t` objects. Those transient blocks contain either pointers to global object or local copies which are deleted after use.

Each `rt_data_t` object contains a variable list component. This lists holds (local copies of) all kinds of intrinsic or user-defined variables. The variable list is linked to the variable list contained in the local process library. This, in turn, is linked to the variable list of the `rt_data_t` context, and so on.

A variable lookup will thus be recursively delegated to the linked variable lists, until a match is found. When modifying a variable which is not yet local, the program creates a local copy and uses this afterwards. Thus, when the local `rt_data_t` object is deleted, the context value is recovered.

Models are kept in a model list which is separate from the variable list. Otherwise, they are treated in a similar manner: the local list is linked to the context model list. Model lookup is thus recursively delegated. When a model or any part of it is modified, the model is copied to the local `rt_data_t` object, so the context model is not modified. Commands such as `integrate` will create their own copy of the current model (and of the current variable list) at the point where they are executed.

When a model is encountered for the first time, it is read from file. The reading is automatically delegated to the global context. Thus, this master copy survives until the main `rt_data_t` object is deleted, at program completion.

If there is a currently active model, its variable list is linked to the main variable list. Variable lookups will then start from the model variable list. When the current model is switched, the new active model will get this link instead. Consequently, a change to the current model is kept as long as this model has a local copy; it survives local model switches. On the other hand, a parameter change in the current model doesn't affect any other model, even if the parameter name is identical.

33.2.2 Container for parse nodes

The runtime data set contains a bunch of parse nodes (chunks of code that have not been compiled into evaluation trees but saved for later use). We collect them here.

This implementation has the useful effect that an assignment between two objects of this type will establish a pointer-target relationship for all components.

⟨*RT data: types*⟩≡

```
type :: rt_parse_nodes_t
   type(parse_node_t), pointer :: cuts_lexpr => null ()
   type(parse_node_t), pointer :: scale_expr => null ()
   type(parse_node_t), pointer :: fac_scale_expr => null ()
   type(parse_node_t), pointer :: ren_scale_expr => null ()
   type(parse_node_t), pointer :: weight_expr => null ()
   type(parse_node_t), pointer :: selection_lexpr => null ()
   type(parse_node_t), pointer :: reweight_expr => null ()
   type(parse_node_t), pointer :: analysis_lexpr => null ()
   type(parse_node_p), dimension(:), allocatable :: alt_setup
 contains
 ⟨RT data: rt parse nodes: TBP⟩
end type rt_parse_nodes_t
```

Clear individual components. The parse nodes are nullified. No finalization needed since the pointer targets are part of the global parse tree.

⟨*RT data: rt parse nodes: TBP*⟩≡

```
procedure :: clear => rt_parse_nodes_clear
```

⟨*RT data: procedures*⟩≡

```
subroutine rt_parse_nodes_clear (rt_pn, name)
  class(rt_parse_nodes_t), intent(inout) :: rt_pn
  type(string_t), intent(in) :: name
  select case (char (name))
  case ("cuts")
     rt_pn%cuts_lexpr => null ()
  case ("scale")
     rt_pn%scale_expr => null ()
  case ("factorization_scale")
     rt_pn%fac_scale_expr => null ()
  case ("renormalization_scale")
     rt_pn%ren_scale_expr => null ()
  case ("weight")
     rt_pn%weight_expr => null ()
  case ("selection")
     rt_pn%selection_lexpr => null ()
  case ("reweight")
     rt_pn%reweight_expr => null ()
  case ("analysis")
     rt_pn%analysis_lexpr => null ()
  end select
end subroutine rt_parse_nodes_clear
```

Output for the parse nodes.

⟨*RT data: rt parse nodes: TBP*⟩+≡

```
procedure :: write => rt_parse_nodes_write
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_parse_nodes_write (object, unit)
  class(rt_parse_nodes_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  call wrt ("Cuts", object%cuts_lexpr)
  call write_separator (u)
  call wrt ("Scale", object%scale_expr)
  call write_separator (u)
  call wrt ("Factorization scale", object%fac_scale_expr)
  call write_separator (u)
  call wrt ("Renormalization scale", object%ren_scale_expr)
  call write_separator (u)
  call wrt ("Weight", object%weight_expr)
  call write_separator (u, 2)
  call wrt ("Event selection", object%selection_lexpr)
  call write_separator (u)
  call wrt ("Event reweighting factor", object%reweight_expr)
  call write_separator (u)
  call wrt ("Event analysis", object%analysis_lexpr)
  if (allocated (object%alt_setup)) then
     call write_separator (u, 2)
     write (u, "(1x,A,':')")  "Alternative setups"
     do i = 1, size (object%alt_setup)
        call write_separator (u)
```

```
        call wrt ("Commands", object%alt_setup(i)%ptr)
      end do
    end if
  contains
    subroutine wrt (title, pn)
      character(*), intent(in) :: title
      type(parse_node_t), intent(in), pointer :: pn
      if (associated (pn)) then
         write (u, "(1x,A,':')")  title
         call write_separator (u)
         call parse_node_write_rec (pn, u)
      else
         write (u, "(1x,A,':',1x,A)")  title, "[undefined]"
      end if
    end subroutine wrt
  end subroutine rt_parse_nodes_write
```

Screen output for individual components. (This should eventually be more condensed, currently we print the internal representation tree.)

⟨*RT data: rt parse nodes: TBP*⟩+≡

```
  procedure :: show => rt_parse_nodes_show
```

⟨*RT data: procedures*⟩+≡

```
  subroutine rt_parse_nodes_show (rt_pn, name, unit)
    class(rt_parse_nodes_t), intent(in) :: rt_pn
    type(string_t), intent(in) :: name
    integer, intent(in), optional :: unit
    type(parse_node_t), pointer :: pn
    integer :: u
    u = given_output_unit (unit)
    select case (char (name))
    case ("cuts")
       pn => rt_pn%cuts_lexpr
    case ("scale")
       pn => rt_pn%scale_expr
    case ("factorization_scale")
       pn => rt_pn%fac_scale_expr
    case ("renormalization_scale")
       pn => rt_pn%ren_scale_expr
    case ("weight")
       pn => rt_pn%weight_expr
    case ("selection")
       pn => rt_pn%selection_lexpr
    case ("reweight")
       pn => rt_pn%reweight_expr
    case ("analysis")
       pn => rt_pn%analysis_lexpr
    end select
    if (associated (pn)) then
       write (u, "(A,1x,A,1x,A)")  "Expression:", char (name), "(parse tree):"
       call parse_node_write_rec (pn, u)
    else
       write (u, "(A,1x,A,A)")  "Expression:", char (name), ": [undefined]"
    end if
```

```
end subroutine rt_parse_nodes_show
```

33.2.3 The data type

This is a big data container which contains everything that is used and modified during the command flow. A local copy of this can be used to temporarily override defaults. The data set is transparent.

⟨*RT data: public*⟩≡

```
public :: rt_data_t
```

⟨*RT data: types*⟩+≡

```
type :: rt_data_t
   type(lexer_t), pointer :: lexer => null ()
   type(rt_data_t), pointer :: context => null ()
   type(var_list_t) :: var_list
   type(iterations_list_t) :: it_list
   type(os_data_t) :: os_data
   type(model_list_t) :: model_list
   type(model_t), pointer :: model => null ()
   logical :: model_is_copy = .false.
   type(model_t), pointer :: preload_model => null ()
   type(model_t), pointer :: fallback_model => null ()
   type(prclib_stack_t) :: prclib_stack
   type(process_library_t), pointer :: prclib => null ()
   type(beam_structure_t) :: beam_structure
   type(rt_parse_nodes_t) :: pn
   type(process_stack_t) :: process_stack
   type(string_t), dimension(:), allocatable :: sample_fmt
   class(event_callback_t), allocatable :: event_callback
   type(file_list_t), pointer :: out_files => null ()
   logical :: quit = .false.
   integer :: quit_code = 0
   type(string_t) :: logfile
   logical :: nlo_fixed_order = .false.
   logical, dimension(0:5) :: selected_nlo_parts = .false.
 contains
 ⟨RT data: rt data: TBP⟩
end type rt_data_t
```

33.2.4 Output

⟨*RT data: rt data: TBP*⟩≡

```
procedure :: write => rt_data_write
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_write (object, unit, vars, pacify)
  class(rt_data_t), intent(in) :: object
  integer, intent(in), optional :: unit
  type(string_t), dimension(:), intent(in), optional :: vars
  logical, intent(in), optional :: pacify
  integer :: u, i
  u = given_output_unit (unit)
```

```
  call write_separator (u, 2)
  write (u, "(1x,A)")  "Runtime data:"
  if (present (vars)) then
     if (size (vars) /= 0) then
        call write_separator (u, 2)
        write (u, "(1x,A)")  "Selected variables:"
        call write_separator (u)
        call object%write_vars (u, vars)
     end if
  else
     call write_separator (u, 2)
     if (associated (object%model)) then
        call object%model%write_var_list (u, follow_link=.true.)
     else
        call object%var_list%write (u, follow_link=.true.)
     end if
  end if
  if (object%it_list%get_n_pass () > 0) then
     call write_separator (u, 2)
     write (u, "(1x)", advance="no")
     call object%it_list%write (u)
  end if
  if (associated (object%model)) then
     call write_separator (u, 2)
     call object%model%write (u)
  end if
  call object%prclib_stack%write (u)
  call object%beam_structure%write (u)
  call write_separator (u, 2)
  call object%pn%write (u)
  if (allocated (object%sample_fmt)) then
     call write_separator (u)
     write (u, "(1x,A)", advance="no")  "Event sample formats = "
     do i = 1, size (object%sample_fmt)
        if (i > 1)  write (u, "(A,1x)", advance="no")  ","
        write (u, "(A)", advance="no")  char (object%sample_fmt(i))
     end do
     write (u, "(A)")
  end if
  call write_separator (u)
  write (u, "(1x,A)", advance="no")  "Event callback:"
  if (allocated (object%event_callback)) then
     call object%event_callback%write (u)
  else
     write (u, "(1x,A)")  "[undefined]"
  end if
  call object%process_stack%write (u, pacify)
  write (u, "(1x,A,1x,L1)")  "quit     :", object%quit
  write (u, "(1x,A,1x,I0)")  "quit_code:", object%quit_code
  call write_separator (u, 2)
  write (u, "(1x,A,1x,A)")   "Logfile  :", "'" // trim (char (object%logfile)) // "'"
  call write_separator (u, 2)
end subroutine rt_data_write
```

Write only selected variables.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: write_vars => rt_data_write_vars
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_write_vars (object, unit, vars)
  class(rt_data_t), intent(in), target :: object
  integer, intent(in), optional :: unit
  type(string_t), dimension(:), intent(in), optional :: vars
  type(var_list_t), pointer :: var_list
  integer :: u, i
  u = given_output_unit (unit)
  if (present (vars)) then
     var_list => object%get_var_list_ptr ()
     do i = 1, size (vars)
        associate (var => vars(i))
          if (var_list%contains (var, follow_link=.true.)) then
             call var_list%write_var (var, unit = u, &
                  follow_link = .true.)
          end if
        end associate
     end do
  end if
end subroutine rt_data_write_vars
```

Write only the model list.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: write_model_list => rt_data_write_model_list
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_write_model_list (object, unit)
  class(rt_data_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  call object%model_list%write (u)
end subroutine rt_data_write_model_list
```

Write only the library stack.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: write_libraries => rt_data_write_libraries
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_write_libraries (object, unit, libpath)
  class(rt_data_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: libpath
  integer :: u
  u = given_output_unit (unit)
  call object%prclib_stack%write (u, libpath)
end subroutine rt_data_write_libraries
```

Write only the beam data.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: write_beams => rt_data_write_beams
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_write_beams (object, unit)
  class(rt_data_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  call write_separator (u, 2)
  call object%beam_structure%write (u)
  call write_separator (u, 2)
end subroutine rt_data_write_beams
```

Write only the process and event expressions.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: write_expr => rt_data_write_expr
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_write_expr (object, unit)
  class(rt_data_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u
  u = given_output_unit (unit)
  call write_separator (u, 2)
  call object%pn%write (u)
  call write_separator (u, 2)
end subroutine rt_data_write_expr
```

Write only the process stack.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: write_process_stack => rt_data_write_process_stack
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_write_process_stack (object, unit)
  class(rt_data_t), intent(in) :: object
  integer, intent(in), optional :: unit
  call object%process_stack%write (unit)
end subroutine rt_data_write_process_stack
```

33.2.5 Clear

The `clear` command can remove the contents of various subobjects. The objects themselves should stay.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: clear_beams => rt_data_clear_beams
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_clear_beams (global)
  class(rt_data_t), intent(inout) :: global
  call global%beam_structure%final_sf ()
  call global%beam_structure%final_pol ()
```

```
    call global%beam_structure%final_mom ()
  end subroutine rt_data_clear_beams
```

33.2.6 Initialization

Initialize runtime data. This defines special variables such as `sqrts`, and should be done only for the instance that is actually global. Local copies will inherit the special variables.

We link the global variable list to the process stack variable list, so the latter is always available (and kept global).

⟨*RT data: rt data: TBP*⟩+≡

```
  procedure :: global_init => rt_data_global_init
```

⟨*RT data: procedures*⟩+≡

```
  subroutine rt_data_global_init (global, paths, logfile)
    class(rt_data_t), intent(out), target :: global
    type(paths_t), intent(in), optional :: paths
    type(string_t), intent(in), optional :: logfile
    integer :: seed
    call os_data_init (global%os_data, paths)
    if (present (logfile)) then
       global%logfile = logfile
    else
       global%logfile = ""
    end if
    allocate (global%out_files)
    call system_clock (seed)
    call global%var_list%init_defaults (seed, paths)
    call global%init_pointer_variables ()
    call global%process_stack%init_var_list (global%var_list)
  end subroutine rt_data_global_init
```

33.2.7 Local copies

This is done at compile time when a local copy of runtime data is needed: Link the variable list and initialize all derived parameters. This allows for synchronizing them with local variable changes without affecting global data.

Also re-initialize pointer variables, so they point to local copies of their targets.

⟨*RT data: rt data: TBP*⟩+≡

```
  procedure :: local_init => rt_data_local_init
```

⟨*RT data: procedures*⟩+≡

```
  subroutine rt_data_local_init (local, global, env)
    class(rt_data_t), intent(inout), target :: local
    type(rt_data_t), intent(in), target :: global
    integer, intent(in), optional :: env
    local%context => global
    call local%process_stack%link (global%process_stack)
    call local%process_stack%init_var_list (local%var_list)
    call local%process_stack%link_var_list (global%var_list)
```

```
    call local%var_list%append_string (var_str ("$model_name"), &
         var_str (""), intrinsic=.true.)
    call local%init_pointer_variables ()
    local%fallback_model => global%fallback_model
    local%os_data = global%os_data
    local%logfile = global%logfile
    call local%model_list%link (global%model_list)
    local%model => global%model
    if (associated (local%model)) then
       call local%model%link_var_list (local%var_list)
    end if
    if (allocated (global%event_callback)) then
       allocate (local%event_callback, source = global%event_callback)
    end if
  end subroutine rt_data_local_init
```

These variables point to objects which get local copies:

⟨*RT data: rt data: TBP*⟩+≡

```
  procedure :: init_pointer_variables => rt_data_init_pointer_variables
```

⟨*RT data: procedures*⟩+≡

```
  subroutine rt_data_init_pointer_variables (local)
    class(rt_data_t), intent(inout), target :: local
    logical, target, save :: known = .true.
    call local%var_list%append_string_ptr (var_str ("$fc"), &
         local%os_data%fc, known, intrinsic=.true.)
    call local%var_list%append_string_ptr (var_str ("$fcflags"), &
         local%os_data%fcflags, known, intrinsic=.true.)
  end subroutine rt_data_init_pointer_variables
```

This is done at execution time: Copy data, transfer pointers. `local` has intent(inout) because its local variable list has already been prepared by the previous routine.

To be pedantic, the local pointers to model and library should point to the entries in the local copies. (However, as long as these are just shallow copies with identical content, this is actually irrelevant.)

The process library and process stacks behave as global objects. The copies of the process library and process stacks should be shallow copies, so the contents stay identical. Since objects may be pushed on the stack in the local environment, upon restoring the global environment, we should reverse the assignment. Then the added stack elements will end up on the global stack. (This should be reconsidered in a parallel environment.)

⟨*RT data: rt data: TBP*⟩+≡

```
  procedure :: activate => rt_data_activate
```

⟨*RT data: procedures*⟩+≡

```
  subroutine rt_data_activate (local)
    class(rt_data_t), intent(inout), target :: local
    class(rt_data_t), pointer :: global
    global => local%context
    if (associated (global)) then
       local%lexer => global%lexer
       call global%copy_globals (local)
```

```
      local%os_data = global%os_data
      local%logfile = global%logfile
      if (associated (global%prclib)) then
         local%prclib => &
              local%prclib_stack%get_library_ptr (global%prclib%get_name ())
      end if
      call local%import_values ()
      call local%process_stack%link (global%process_stack)
      local%it_list = global%it_list
      local%beam_structure = global%beam_structure
      local%pn = global%pn
      if (allocated (local%sample_fmt))  deallocate (local%sample_fmt)
      if (allocated (global%sample_fmt)) then
         allocate (local%sample_fmt (size (global%sample_fmt)), &
              source = global%sample_fmt)
      end if
      local%out_files => global%out_files
      local%model => global%model
      local%model_is_copy = .false.
   else if (.not. associated (local%model)) then
      local%model => local%preload_model
      local%model_is_copy = .false.
   end if
   if (associated (local%model)) then
      call local%model%link_var_list (local%var_list)
      call local%var_list%set_string (var_str ("$model_name"), &
           local%model%get_name (), is_known = .true.)
   else
      call local%var_list%set_string (var_str ("$model_name"), &
           var_str (""), is_known = .false.)
   end if
end subroutine rt_data_activate
```

Restore the previous state of data, without actually finalizing the local environment. We also clear the local process stack. Some local modifications (model list and process library stack) are communicated to the global context, if there is any.

If the `keep_local` flag is set, we want to retain current settings in the local environment. In particular, we create an instance of the currently selected model (which thus becomes separated from the model library!). The local variables are also kept.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: deactivate => rt_data_deactivate
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_deactivate (local, global, keep_local)
  class(rt_data_t), intent(inout), target :: local
  class(rt_data_t), intent(inout), optional, target :: global
  logical, intent(in), optional :: keep_local
  type(string_t) :: local_model, local_scheme
  logical :: same_model, delete
  delete = .true.;  if (present (keep_local))  delete = .not. keep_local
  if (present (global)) then
```

```
      if (associated (global%model) .and. associated (local%model)) then
         local_model = local%model%get_name ()
         if (global%model%has_schemes ()) then
            local_scheme = local%model%get_scheme ()
            same_model = &
                 global%model%matches (local_model, local_scheme)
         else
            same_model = global%model%matches (local_model)
         end if
      else
         same_model = .false.
      end if
      if (delete) then
         call local%process_stack%clear ()
         call local%unselect_model ()
         call local%unset_values ()
      else if (associated (local%model)) then
         call local%ensure_model_copy ()
      end if
      if (.not. same_model .and. associated (global%model)) then
         if (global%model%has_schemes ()) then
            call msg_message ("Restoring model '" // &
                 char (global%model%get_name ()) // "', scheme '" // &
                 char (global%model%get_scheme ()) // "'")
         else
            call msg_message ("Restoring model '" // &
                 char (global%model%get_name ()) // "'")
         end if
      end if
      if (associated (global%model)) then
         call global%model%link_var_list (global%var_list)
      end if
      call global%restore_globals (local)
   else
      call local%unselect_model ()
   end if
end subroutine rt_data_deactivate
```

This imports the global objects for which local modifications should be kept. Currently, this is only the process library stack.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: copy_globals => rt_data_copy_globals
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_copy_globals (global, local)
  class(rt_data_t), intent(in) :: global
  class(rt_data_t), intent(inout) :: local
  local%prclib_stack = global%prclib_stack
end subroutine rt_data_copy_globals
```

This restores global objects, for which local modifications should be kept.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: restore_globals => rt_data_restore_globals
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_restore_globals (global, local)
  class(rt_data_t), intent(inout) :: global
  class(rt_data_t), intent(in) :: local
  global%prclib_stack = local%prclib_stack
end subroutine rt_data_restore_globals
```

33.2.8 Finalization

Finalizer for the variable list and the structure-function list. This is done only for the global RT dataset; local copies contain pointers to this and do not need a finalizer.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: final => rt_data_global_final
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_global_final (global)
  class(rt_data_t), intent(inout) :: global
  call global%process_stack%final ()
  call global%prclib_stack%final ()
  call global%model_list%final ()
  call global%var_list%final (follow_link=.false.)
  if (associated (global%out_files)) then
     call file_list_final (global%out_files)
     deallocate (global%out_files)
  end if
end subroutine rt_data_global_final
```

The local copy needs a finalizer for the variable list, which consists of local copies. This finalizer is called only when the local environment is finally discarded. (Note that the process stack should already have been cleared after execution, which can occur many times for the same local environment.)

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: local_final => rt_data_local_final
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_local_final (local)
  class(rt_data_t), intent(inout) :: local
  call local%process_stack%clear ()
  call local%model_list%final ()
  call local%var_list%final (follow_link=.false.)
end subroutine rt_data_local_final
```

33.2.9 Model Management

Read a model, so it becomes available for activation. No variables or model copies, this is just initialization.

If this is a local environment, the model will be automatically read into the global context.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: read_model => rt_data_read_model
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_read_model (global, name, model, scheme)
  class(rt_data_t), intent(inout) :: global
  type(string_t), intent(in) :: name
  type(string_t), intent(in), optional :: scheme
  type(model_t), pointer, intent(out) :: model
  type(string_t) :: filename
  filename = name // ".mdl"
  call global%model_list%read_model &
       (name, filename, global%os_data, model, scheme)
end subroutine rt_data_read_model
```

Initialize the fallback model. This model is used whenever the current model does not describe all physical particles (hadrons, mainly). It is not supposed to be modified, and the pointer should remain linked to this model.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: init_fallback_model => rt_data_init_fallback_model
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_init_fallback_model (global, name, filename)
  class(rt_data_t), intent(inout) :: global
  type(string_t), intent(in) :: name, filename
  call global%model_list%read_model &
       (name, filename, global%os_data, global%fallback_model)
end subroutine rt_data_init_fallback_model
```

Activate a model: assign the current-model pointer and set the model name in the variable list. If necessary, read the model from file. Link the global variable list to the model variable list.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: select_model => rt_data_select_model
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_select_model (global, name, scheme)
  class(rt_data_t), intent(inout), target :: global
  type(string_t), intent(in) :: name
  type(string_t), intent(in), optional :: scheme
  logical :: same_model
  if (associated (global%model)) then
     same_model = global%model%matches (name, scheme)
  else
     same_model = .false.
  end if
  if (.not. same_model) then
     global%model => global%model_list%get_model_ptr (name, scheme)
     if (.not. associated (global%model)) then
        call global%read_model (name, global%model)
        global%model_is_copy = .false.
     else if (associated (global%context)) then
        global%model_is_copy = &
             global%model_list%model_exists (name, scheme, &
             follow_link=.false.)
     else
```

```
         global%model_is_copy = .false.
      end if
   end if
   if (associated (global%model)) then
      call global%model%link_var_list (global%var_list)
      call global%var_list%set_string (var_str ("$model_name"), &
           name, is_known = .true.)
      if (global%model%has_schemes ()) then
         call msg_message ("Switching to model '" // char (name) // "', " &
              // "scheme '" // char (global%model%get_scheme ()) // "'")
      else
         call msg_message ("Switching to model '" // char (name) // "'")
      end if
   else
      call global%var_list%set_string (var_str ("$model_name"), &
           var_str (""), is_known = .false.)
   end if
end subroutine rt_data_select_model
```

Remove the model link and unset the model name variable.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: unselect_model => rt_data_unselect_model
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_unselect_model (global)
  class(rt_data_t), intent(inout), target :: global
  if (associated (global%model)) then
     global%model => null ()
     global%model_is_copy = .false.
     call global%var_list%set_string (var_str ("$model_name"), &
          var_str (""), is_known = .false.)
  end if
end subroutine rt_data_unselect_model
```

Create a copy of the currently selected model and append it to the local model list. The model pointer is redirected to the copy. (Not applicable for the global model list, those models will be modified in-place.)

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: ensure_model_copy => rt_data_ensure_model_copy
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_ensure_model_copy (global)
  class(rt_data_t), intent(inout), target :: global
  if (associated (global%context)) then
     if (.not. global%model_is_copy) then
        call global%model_list%append_copy (global%model, global%model)
        global%model_is_copy = .true.
        call global%model%link_var_list (global%var_list)
     end if
  end if
end subroutine rt_data_ensure_model_copy
```

Modify a model variable. The update mechanism will ensure that the model parameter set remains consistent. This has to take place in a local copy of the current model. If there is none yet, create one.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: model_set_real => rt_data_model_set_real
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_model_set_real (global, name, rval, verbose, pacified)
  class(rt_data_t), intent(inout), target :: global
  type(string_t), intent(in) :: name
  real(default), intent(in) :: rval
  logical, intent(in), optional :: verbose, pacified
  call global%ensure_model_copy ()
  call global%model%set_real (name, rval, verbose, pacified)
end subroutine rt_data_model_set_real
```

Modify particle properties. This has to take place in a local copy of the current model. If there is none yet, create one.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: modify_particle => rt_data_modify_particle
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_modify_particle &
     (global, pdg, polarized, stable, decay, &
     isotropic_decay, diagonal_decay, decay_helicity)
  class(rt_data_t), intent(inout), target :: global
  integer, intent(in) :: pdg
  logical, intent(in), optional :: polarized, stable
  logical, intent(in), optional :: isotropic_decay, diagonal_decay
  integer, intent(in), optional :: decay_helicity
  type(string_t), dimension(:), intent(in), optional :: decay
  call global%ensure_model_copy ()
  if (present (polarized)) then
     if (polarized) then
        call global%model%set_polarized (pdg)
     else
        call global%model%set_unpolarized (pdg)
     end if
  end if
  if (present (stable)) then
     if (stable) then
        call global%model%set_stable (pdg)
     else if (present (decay)) then
        call global%model%set_unstable &
             (pdg, decay, isotropic_decay, diagonal_decay, decay_helicity)
     else
        call msg_bug ("Setting particle unstable: missing decay processes")
     end if
  end if
end subroutine rt_data_modify_particle
```

33.2.10 Managing Variables

Return a pointer to the currently active variable list. If there is no model, this is the global variable list. If there is one, it is the model variable list, which should be linked to the former.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: get_var_list_ptr => rt_data_get_var_list_ptr
```

⟨*RT data: procedures*⟩+≡
```
function rt_data_get_var_list_ptr (global) result (var_list)
  class(rt_data_t), intent(in), target :: global
  type(var_list_t), pointer :: var_list
  if (associated (global%model)) then
     var_list => global%model%get_var_list_ptr ()
  else
     var_list => global%var_list
  end if
end function rt_data_get_var_list_ptr
```

Initialize a local variable: append it to the current variable list. No initial value, yet.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: append_log => rt_data_append_log
procedure :: append_int => rt_data_append_int
procedure :: append_real => rt_data_append_real
procedure :: append_cmplx => rt_data_append_cmplx
procedure :: append_subevt => rt_data_append_subevt
procedure :: append_pdg_array => rt_data_append_pdg_array
procedure :: append_string => rt_data_append_string
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_append_log (local, name, lval, intrinsic, user)
  class(rt_data_t), intent(inout) :: local
  type(string_t), intent(in) :: name
  logical, intent(in), optional :: lval
  logical, intent(in), optional :: intrinsic, user
  call local%var_list%append_log (name, lval, &
       intrinsic = intrinsic, user = user)
end subroutine rt_data_append_log

subroutine rt_data_append_int (local, name, ival, intrinsic, user)
  class(rt_data_t), intent(inout) :: local
  type(string_t), intent(in) :: name
  integer, intent(in), optional :: ival
  logical, intent(in), optional :: intrinsic, user
  call local%var_list%append_int (name, ival, &
       intrinsic = intrinsic, user = user)
end subroutine rt_data_append_int

subroutine rt_data_append_real (local, name, rval, intrinsic, user)
  class(rt_data_t), intent(inout) :: local
  type(string_t), intent(in) :: name
  real(default), intent(in), optional :: rval
  logical, intent(in), optional :: intrinsic, user
```

```
    call local%var_list%append_real (name, rval, &
         intrinsic = intrinsic, user = user)
  end subroutine rt_data_append_real

  subroutine rt_data_append_cmplx (local, name, cval, intrinsic, user)
    class(rt_data_t), intent(inout) :: local
    type(string_t), intent(in) :: name
    complex(default), intent(in), optional :: cval
    logical, intent(in), optional :: intrinsic, user
    call local%var_list%append_cmplx (name, cval, &
         intrinsic = intrinsic, user = user)
  end subroutine rt_data_append_cmplx

  subroutine rt_data_append_subevt (local, name, pval, intrinsic, user)
    class(rt_data_t), intent(inout) :: local
    type(string_t), intent(in) :: name
    type(subevt_t), intent(in), optional :: pval
    logical, intent(in) :: intrinsic, user
    call local%var_list%append_subevt (name, &
         intrinsic = intrinsic, user = user)
  end subroutine rt_data_append_subevt

  subroutine rt_data_append_pdg_array (local, name, aval, intrinsic, user)
    class(rt_data_t), intent(inout) :: local
    type(string_t), intent(in) :: name
    type(pdg_array_t), intent(in), optional :: aval
    logical, intent(in), optional :: intrinsic, user
    call local%var_list%append_pdg_array (name, aval, &
         intrinsic = intrinsic, user = user)
  end subroutine rt_data_append_pdg_array

  subroutine rt_data_append_string (local, name, sval, intrinsic, user)
    class(rt_data_t), intent(inout) :: local
    type(string_t), intent(in) :: name
    type(string_t), intent(in), optional :: sval
    logical, intent(in), optional :: intrinsic, user
    call local%var_list%append_string (name, sval, &
         intrinsic = intrinsic, user = user)
  end subroutine rt_data_append_string
```

Import values for all local variables, given a global context environment where these variables are defined.

⟨*RT data: rt data: TBP*⟩+≡

```
  procedure :: import_values => rt_data_import_values
```

⟨*RT data: procedures*⟩+≡

```
  subroutine rt_data_import_values (local)
    class(rt_data_t), intent(inout) :: local
    type(rt_data_t), pointer :: global
    global => local%context
    if (associated (global)) then
       call local%var_list%import (global%var_list)
    end if
  end subroutine rt_data_import_values
```

Unset all variable values.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: unset_values => rt_data_unset_values
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_unset_values (global)
  class(rt_data_t), intent(inout) :: global
  call global%var_list%undefine (follow_link=.false.)
end subroutine rt_data_unset_values
```

Set a variable. (Not a model variable, these are handled separately.) We can assume that the variable has been initialized.

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: set_log => rt_data_set_log
procedure :: set_int => rt_data_set_int
procedure :: set_real => rt_data_set_real
procedure :: set_cmplx => rt_data_set_cmplx
procedure :: set_subevt => rt_data_set_subevt
procedure :: set_pdg_array => rt_data_set_pdg_array
procedure :: set_string => rt_data_set_string
```

⟨*RT data: procedures*⟩+≡
```
subroutine rt_data_set_log (global, name, lval, is_known, verbose)
  class(rt_data_t), intent(inout) :: global
  type(string_t), intent(in) :: name
  logical, intent(in) :: lval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: verbose
  call global%var_list%set_log (name, lval, is_known, &
       verbose=verbose)
end subroutine rt_data_set_log

subroutine rt_data_set_int (global, name, ival, is_known, verbose)
  class(rt_data_t), intent(inout) :: global
  type(string_t), intent(in) :: name
  integer, intent(in) :: ival
  logical, intent(in) :: is_known
  logical, intent(in), optional :: verbose
  call global%var_list%set_int (name, ival, is_known, &
       verbose=verbose)
end subroutine rt_data_set_int

subroutine rt_data_set_real (global, name, rval, is_known, verbose, pacified)
  class(rt_data_t), intent(inout) :: global
  type(string_t), intent(in) :: name
  real(default), intent(in) :: rval
  logical, intent(in) :: is_known
  logical, intent(in), optional :: verbose, pacified
  call global%var_list%set_real (name, rval, is_known, &
       verbose=verbose, pacified=pacified)
end subroutine rt_data_set_real

subroutine rt_data_set_cmplx (global, name, cval, is_known, verbose, pacified)
```

```
    class(rt_data_t), intent(inout) :: global
    type(string_t), intent(in) :: name
    complex(default), intent(in) :: cval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: verbose, pacified
    call global%var_list%set_cmplx (name, cval, is_known, &
         verbose=verbose, pacified=pacified)
  end subroutine rt_data_set_cmplx

  subroutine rt_data_set_subevt (global, name, pval, is_known, verbose)
    class(rt_data_t), intent(inout) :: global
    type(string_t), intent(in) :: name
    type(subevt_t), intent(in) :: pval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: verbose
    call global%var_list%set_subevt (name, pval, is_known, &
         verbose=verbose)
  end subroutine rt_data_set_subevt

  subroutine rt_data_set_pdg_array (global, name, aval, is_known, verbose)
    class(rt_data_t), intent(inout) :: global
    type(string_t), intent(in) :: name
    type(pdg_array_t), intent(in) :: aval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: verbose
    call global%var_list%set_pdg_array (name, aval, is_known, &
         verbose=verbose)
  end subroutine rt_data_set_pdg_array

  subroutine rt_data_set_string (global, name, sval, is_known, verbose)
    class(rt_data_t), intent(inout) :: global
    type(string_t), intent(in) :: name
    type(string_t), intent(in) :: sval
    logical, intent(in) :: is_known
    logical, intent(in), optional :: verbose
    call global%var_list%set_string (name, sval, is_known, &
         verbose=verbose)
  end subroutine rt_data_set_string
```

Return the value of a variable, assuming that the type is correct.

⟨*RT data: rt data: TBP*⟩+≡

```
  procedure :: get_lval => rt_data_get_lval
  procedure :: get_ival => rt_data_get_ival
  procedure :: get_rval => rt_data_get_rval
  procedure :: get_cval => rt_data_get_cval
  procedure :: get_pval => rt_data_get_pval
  procedure :: get_aval => rt_data_get_aval
  procedure :: get_sval => rt_data_get_sval
```

⟨*RT data: procedures*⟩+≡

```
  function rt_data_get_lval (global, name) result (lval)
    logical :: lval
    class(rt_data_t), intent(in), target :: global
    type(string_t), intent(in) :: name
```

```
  type(var_list_t), pointer :: var_list
  var_list => global%get_var_list_ptr ()
  lval = var_list%get_lval (name)
end function rt_data_get_lval

function rt_data_get_ival (global, name) result (ival)
  integer :: ival
  class(rt_data_t), intent(in), target :: global
  type(string_t), intent(in) :: name
  type(var_list_t), pointer :: var_list
  var_list => global%get_var_list_ptr ()
  ival = var_list%get_ival (name)
end function rt_data_get_ival

function rt_data_get_rval (global, name) result (rval)
  real(default) :: rval
  class(rt_data_t), intent(in), target :: global
  type(string_t), intent(in) :: name
  type(var_list_t), pointer :: var_list
  var_list => global%get_var_list_ptr ()
  rval = var_list%get_rval (name)
end function rt_data_get_rval

function rt_data_get_cval (global, name) result (cval)
  complex(default) :: cval
  class(rt_data_t), intent(in), target :: global
  type(string_t), intent(in) :: name
  type(var_list_t), pointer :: var_list
  var_list => global%get_var_list_ptr ()
  cval = var_list%get_cval (name)
end function rt_data_get_cval

function rt_data_get_aval (global, name) result (aval)
  type(pdg_array_t) :: aval
  class(rt_data_t), intent(in), target :: global
  type(string_t), intent(in) :: name
  type(var_list_t), pointer :: var_list
  var_list => global%get_var_list_ptr ()
  aval = var_list%get_aval (name)
end function rt_data_get_aval

function rt_data_get_pval (global, name) result (pval)
  type(subevt_t) :: pval
  class(rt_data_t), intent(in), target :: global
  type(string_t), intent(in) :: name
  type(var_list_t), pointer :: var_list
  var_list => global%get_var_list_ptr ()
  pval = var_list%get_pval (name)
end function rt_data_get_pval

function rt_data_get_sval (global, name) result (sval)
  type(string_t) :: sval
  class(rt_data_t), intent(in), target :: global
  type(string_t), intent(in) :: name
```

```
    type(var_list_t), pointer :: var_list
    var_list => global%get_var_list_ptr ()
    sval = var_list%get_sval (name)
  end function rt_data_get_sval
```

Return true if the variable exists in the global list.

⟨*RT data: rt data: TBP*⟩+≡

```
  procedure :: contains => rt_data_contains
```

⟨*RT data: procedures*⟩+≡

```
  function rt_data_contains (global, name) result (lval)
    logical :: lval
    class(rt_data_t), intent(in) :: global
    type(string_t), intent(in) :: name
    type(var_list_t), pointer :: var_list
    var_list => global%get_var_list_ptr ()
    lval = var_list%contains (name)
  end function rt_data_contains
```

33.2.11 Further Content

Add a library (available via a pointer of type `prclib_entry_t`) to the stack and update the pointer and variable list to the current library. The pointer association of `prclib_entry` will be discarded.

⟨*RT data: rt data: TBP*⟩+≡

```
  procedure :: add_prclib => rt_data_add_prclib
```

⟨*RT data: procedures*⟩+≡

```
  subroutine rt_data_add_prclib (global, prclib_entry)
    class(rt_data_t), intent(inout) :: global
    type(prclib_entry_t), intent(inout), pointer :: prclib_entry
    call global%prclib_stack%push (prclib_entry)
    call global%update_prclib (global%prclib_stack%get_first_ptr ())
  end subroutine rt_data_add_prclib
```

Given a pointer to a process library, make this the currently active library.

⟨*RT data: rt data: TBP*⟩+≡

```
  procedure :: update_prclib => rt_data_update_prclib
```

⟨*RT data: procedures*⟩+≡

```
  subroutine rt_data_update_prclib (global, lib)
    class(rt_data_t), intent(inout) :: global
    type(process_library_t), intent(in), target :: lib
    global%prclib => lib
    if (global%var_list%contains (&
         var_str ("$library_name"), follow_link = .false.)) then
       call global%var_list%set_string (var_str ("$library_name"), &
            global%prclib%get_name (), is_known=.true.)
    else
       call global%var_list%append_string ( &
            var_str ("$library_name"), global%prclib%get_name (), &
            intrinsic = .true.)
```

```
    end if
  end subroutine rt_data_update_prclib
```

33.2.12 Miscellaneous

The helicity selection data are distributed among several parameters. Here, we collect them in a single record.

⟨*RT data: rt data: TBP*⟩+≡

```
  procedure :: get_helicity_selection => rt_data_get_helicity_selection
```

⟨*RT data: procedures*⟩+≡

```
  function rt_data_get_helicity_selection (rt_data) result (helicity_selection)
    class(rt_data_t), intent(in) :: rt_data
    type(helicity_selection_t) :: helicity_selection
    associate (var_list => rt_data%var_list)
      helicity_selection%active = var_list%get_lval (&
           var_str ("?helicity_selection_active"))
      if (helicity_selection%active) then
         helicity_selection%threshold = var_list%get_rval (&
              var_str ("helicity_selection_threshold"))
         helicity_selection%cutoff = var_list%get_ival (&
              var_str ("helicity_selection_cutoff"))
      end if
    end associate
  end function rt_data_get_helicity_selection
```

Show the beam setup: beam structure and relevant global variables.

⟨*RT data: rt data: TBP*⟩+≡

```
  procedure :: show_beams => rt_data_show_beams
```

⟨*RT data: procedures*⟩+≡

```
  subroutine rt_data_show_beams (rt_data, unit)
    class(rt_data_t), intent(in) :: rt_data
    integer, intent(in), optional :: unit
    type(string_t) :: s
    integer :: u
    u = given_output_unit (unit)
    associate (beams => rt_data%beam_structure, var_list => rt_data%var_list)
      call beams%write (u)
      if (.not. beams%asymmetric () .and. beams%get_n_beam () == 2) then
         write (u, "(2x,A," // FMT_19 // ",1x,'GeV')") "sqrts =", &
              var_list%get_rval (var_str ("sqrts"))
      end if
      if (beams%contains ("pdf_builtin")) then
         s = var_list%get_sval (var_str ("$pdf_builtin_set"))
         if (s /= "") then
            write (u, "(2x,A,1x,3A)")  "PDF set =", '"', char (s), '"'
         else
            write (u, "(2x,A,1x,A)")  "PDF set =", "[undefined]"
         end if
      end if
      if (beams%contains ("lhapdf")) then
         s = var_list%get_sval (var_str ("$lhapdf_dir"))
```

```
   if (s /= "") then
      write (u, "(2x,A,1x,3A)")  "LHAPDF dir    =", '"', char (s), '"'
   end if
   s = var_list%get_sval (var_str ("$lhapdf_file"))
   if (s /= "") then
      write (u, "(2x,A,1x,3A)")  "LHAPDF file   =", '"', char (s), '"'
      write (u, "(2x,A,1x,I0)") "LHAPDF member =", &
           var_list%get_ival (var_str ("lhapdf_member"))
   else
      write (u, "(2x,A,1x,A)")  "LHAPDF file   =", "[undefined]"
   end if
end if
if (beams%contains ("lhapdf_photon")) then
   s = var_list%get_sval (var_str ("$lhapdf_dir"))
   if (s /= "") then
      write (u, "(2x,A,1x,3A)")  "LHAPDF dir    =", '"', char (s), '"'
   end if
   s = var_list%get_sval (var_str ("$lhapdf_photon_file"))
   if (s /= "") then
      write (u, "(2x,A,1x,3A)")  "LHAPDF file   =", '"', char (s), '"'
      write (u, "(2x,A,1x,I0)") "LHAPDF member =", &
           var_list%get_ival (var_str ("lhapdf_member"))
      write (u, "(2x,A,1x,I0)") "LHAPDF scheme =", &
           var_list%get_ival (&
           var_str ("lhapdf_photon_scheme"))
   else
      write (u, "(2x,A,1x,A)")  "LHAPDF file   =", "[undefined]"
   end if
end if
if (beams%contains ("isr")) then
   write (u, "(2x,A," // FMT_19 // ")") "ISR alpha        =", &
        var_list%get_rval (var_str ("isr_alpha"))
   write (u, "(2x,A," // FMT_19 // ")") "ISR Q max        =", &
        var_list%get_rval (var_str ("isr_q_max"))
   write (u, "(2x,A," // FMT_19 // ")") "ISR mass         =", &
        var_list%get_rval (var_str ("isr_mass"))
   write (u, "(2x,A,1x,I0)") "ISR order        =", &
        var_list%get_ival (var_str ("isr_order"))
   write (u, "(2x,A,1x,L1)") "ISR recoil       =", &
        var_list%get_lval (var_str ("?isr_recoil"))
   write (u, "(2x,A,1x,L1)") "ISR energy cons. =", &
        var_list%get_lval (var_str ("?isr_keep_energy"))
end if
if (beams%contains ("epa")) then
   write (u, "(2x,A," // FMT_19 // ")") "EPA alpha         =", &
        var_list%get_rval (var_str ("epa_alpha"))
   write (u, "(2x,A," // FMT_19 // ")") "EPA x min         =", &
        var_list%get_rval (var_str ("epa_x_min"))
   write (u, "(2x,A," // FMT_19 // ")") "EPA Q min         =", &
        var_list%get_rval (var_str ("epa_q_min"))
   write (u, "(2x,A," // FMT_19 // ")") "EPA E max         =", &
        var_list%get_rval (var_str ("epa_e_max"))
   write (u, "(2x,A," // FMT_19 // ")") "EPA mass          =", &
        var_list%get_rval (var_str ("epa_mass"))
```

```
   write (u, "(2x,A,1x,L1)") "EPA recoil        =", &
        var_list%get_lval (var_str ("?epa_recoil"))
   write (u, "(2x,A,1x,L1)") "EPA  energy cons. =", &
        var_list%get_lval (var_str ("?epa_keep_energy"))
end if
if (beams%contains ("ewa")) then
   write (u, "(2x,A," // FMT_19 // ")") "EWA x min       =", &
        var_list%get_rval (var_str ("ewa_x_min"))
   write (u, "(2x,A," // FMT_19 // ")") "EWA Pt max      =", &
        var_list%get_rval (var_str ("ewa_pt_max"))
   write (u, "(2x,A," // FMT_19 // ")") "EWA mass        =", &
        var_list%get_rval (var_str ("ewa_mass"))
   write (u, "(2x,A,1x,L1)") "EWA recoil       =", &
        var_list%get_lval (var_str ("?ewa_recoil"))
   write (u, "(2x,A,1x,L1)") "EWA energy cons. =", &
        var_list%get_lval (var_str ("ewa_keep_energy"))
end if
if (beams%contains ("circe1")) then
   write (u, "(2x,A,1x,I0)") "CIRCE1 version    =", &
        var_list%get_ival (var_str ("circe1_ver"))
   write (u, "(2x,A,1x,I0)") "CIRCE1 revision   =", &
        var_list%get_ival (var_str ("circe1_rev"))
   s = var_list%get_sval (var_str ("$circe1_acc"))
   write (u, "(2x,A,1x,A)") "CIRCE1 acceler.   =", char (s)
   write (u, "(2x,A,1x,I0)") "CIRCE1 chattin.   =", &
        var_list%get_ival (var_str ("circe1_chat"))
   write (u, "(2x,A," // FMT_19 // ")") "CIRCE1 sqrts      =", &
        var_list%get_rval (var_str ("circe1_sqrts"))
   write (u, "(2x,A," // FMT_19 // ")") "CIRCE1 epsil.     =", &
        var_list%get_rval (var_str ("circe1_eps"))
   write (u, "(2x,A,1x,L1)") "CIRCE1 phot. 1  =", &
        var_list%get_lval (var_str ("?circe1_photon1"))
   write (u, "(2x,A,1x,L1)") "CIRCE1 phot. 2  =", &
        var_list%get_lval (var_str ("?circe1_photon2"))
   write (u, "(2x,A,1x,L1)") "CIRCE1 generat. =", &
        var_list%get_lval (var_str ("?circe1_generate"))
   write (u, "(2x,A,1x,L1)") "CIRCE1 mapping  =", &
        var_list%get_lval (var_str ("?circe1_map"))
   write (u, "(2x,A," // FMT_19 // ")") "CIRCE1 map. slope =", &
        var_list%get_rval (var_str ("circe1_mapping_slope"))
   write (u, "(2x,A,1x,L1)") "CIRCE recoil photon =", &
        var_list%get_lval (var_str ("?circe1_with_radiation"))
end if
if (beams%contains ("circe2")) then
   s = var_list%get_sval (var_str ("$circe2_design"))
   write (u, "(2x,A,1x,A)") "CIRCE2 design   =", char (s)
   s = var_list%get_sval (var_str ("$circe2_file"))
   write (u, "(2x,A,1x,A)") "CIRCE2 file     =", char (s)
   write (u, "(2x,A,1x,L1)") "CIRCE2 polarized =", &
        var_list%get_lval (var_str ("?circe2_polarized"))
end if
if (beams%contains ("gaussian")) then
   write (u, "(2x,A,1x," // FMT_12 // ")") "Gaussian spread 1    =", &
        var_list%get_rval (var_str ("gaussian_spread1"))
```

```
      write (u, "(2x,A,1x," // FMT_12 // ")") "Gaussian spread 2    =", &
           var_list%get_rval (var_str ("gaussian_spread2"))
    end if
    if (beams%contains ("beam_events")) then
      s = var_list%get_sval (var_str ("$beam_events_file"))
      write (u, "(2x,A,1x,A)") "Beam events file     =", char (s)
      write (u, "(2x,A,1x,L1)") "Beam events EOF warn =", &
           var_list%get_lval (var_str ("?beam_events_warn_eof"))
    end if
  end associate
end subroutine rt_data_show_beams
```

Return the collision energy as determined by the current beam settings. Without beam setup, this is the `sqrts` variable.

If the value is meaningless for a setup, the function returns zero.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: get_sqrts => rt_data_get_sqrts
```

⟨*RT data: procedures*⟩+≡

```
function rt_data_get_sqrts (rt_data) result (sqrts)
  class(rt_data_t), intent(in) :: rt_data
  real(default) :: sqrts
  sqrts = rt_data%var_list%get_rval (var_str ("sqrts"))
end function rt_data_get_sqrts
```

For testing purposes, the `rt_data_t` contents can be pacified to suppress numerical fluctuations in (constant) test matrix elements.

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: pacify => rt_data_pacify
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_pacify (rt_data, efficiency_reset, error_reset)
  class(rt_data_t), intent(inout) :: rt_data
  logical, intent(in), optional :: efficiency_reset, error_reset
  type(process_entry_t), pointer :: process
  process => rt_data%process_stack%first
  do while (associated (process))
     call process%pacify (efficiency_reset, error_reset)
     process => process%next
  end do
end subroutine rt_data_pacify
```

⟨*RT data: rt data: TBP*⟩+≡

```
procedure :: set_event_callback => rt_data_set_event_callback
```

⟨*RT data: procedures*⟩+≡

```
subroutine rt_data_set_event_callback (global, callback)
  class(rt_data_t), intent(inout) :: global
  class(event_callback_t), intent(in) :: callback
  if (allocated (global%event_callback))  deallocate (global%event_callback)
  allocate (global%event_callback, source = callback)
end subroutine rt_data_set_event_callback
```

⟨*RT data: rt data: TBP*⟩+≡
```
procedure :: has_event_callback => rt_data_has_event_callback
procedure :: get_event_callback => rt_data_get_event_callback
```

⟨*RT data: procedures*⟩+≡
```
function rt_data_has_event_callback (global) result (flag)
  class(rt_data_t), intent(in) :: global
  logical :: flag
  flag = allocated (global%event_callback)
end function rt_data_has_event_callback

function rt_data_get_event_callback (global) result (callback)
  class(rt_data_t), intent(in) :: global
  class(event_callback_t), allocatable :: callback
  if (allocated (global%event_callback)) then
     allocate (callback, source = global%event_callback)
  end if
end function rt_data_get_event_callback
```

33.2.13 Unit Tests

Test module, followed by the corresponding implementation module.

⟨rt_data_ut.f90⟩≡
```
⟨File header⟩

module rt_data_ut
  use unit_tests
  use rt_data_uti

⟨Standard module head⟩

⟨RT data: public test⟩

contains

⟨RT data: test driver⟩

end module rt_data_ut
```

⟨rt_data_uti.f90⟩≡
```
⟨File header⟩

module rt_data_uti

⟨Use kinds⟩
⟨Use strings⟩
  use format_defs, only: FMT_19
  use ifiles
  use lexers
  use parser
  use flavors
  use variables, only: var_list_t
  use eval_trees
```

```
  use models
  use prclib_stacks

  use rt_data
```

⟨*Standard module head*⟩

⟨*RT data: test declarations*⟩

```
contains
```

⟨*RT data: test auxiliary*⟩

⟨*RT data: tests*⟩

```
end module rt_data_uti
```

API: driver for the unit tests below.

⟨*RT data: public test*⟩≡

```
  public :: rt_data_test
```

⟨*RT data: test driver*⟩≡

```
  subroutine rt_data_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
```
⟨*RT data: execute tests*⟩
```
  end subroutine rt_data_test
```

Initial content

Force system-dependent objects to well-defined values. Some of the variables are locked and therefore must be addressed directly.

This is, of course, only required for testing purposes. In principle, the `real_specimen` variables could be set to their values in `rt_data_t`, but this depends on the precision again, so we set them to some dummy values.

⟨*RT data: test auxiliary*⟩≡

```
  subroutine fix_system_dependencies (global)
    class(rt_data_t), intent(inout), target :: global
    type(var_list_t), pointer :: var_list

    var_list => global%get_var_list_ptr ()
    call var_list%set_log (var_str ("?omega_openmp"), &
         .false., is_known = .true., force=.true.)
    call var_list%set_log (var_str ("?openmp_is_active"), &
         .false., is_known = .true., force=.true.)
    call var_list%set_int (var_str ("openmp_num_threads_default"), &
         1, is_known = .true., force=.true.)
    call var_list%set_int (var_str ("openmp_num_threads"), &
         1, is_known = .true., force=.true.)
    call var_list%set_int (var_str ("real_range"), &
         307, is_known = .true., force=.true.)
    call var_list%set_int (var_str ("real_precision"), &
         15, is_known = .true., force=.true.)
```

```
    call var_list%set_real (var_str ("real_epsilon"), &
         1.e-16_default, is_known = .true., force=.true.)
    call var_list%set_real (var_str ("real_tiny"), &
         1.e-300_default, is_known = .true., force=.true.)

    global%os_data%fc = "Fortran-compiler"
    global%os_data%fcflags = "Fortran-flags"

  end subroutine fix_system_dependencies
```

Display the RT data in the state just after (global) initialization.

⟨*RT data: execute tests*⟩≡

```
  call test (rt_data_1, "rt_data_1", &
       "initialize", &
       u, results)
```

⟨*RT data: test declarations*⟩≡

```
  public :: rt_data_1
```

⟨*RT data: tests*⟩≡

```
  subroutine rt_data_1 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global

    write (u, "(A)")  "* Test output: rt_data_1"
    write (u, "(A)")  "*   Purpose: initialize global runtime data"
    write (u, "(A)")

    call global%global_init (logfile = var_str ("rt_data.log"))
    call fix_system_dependencies (global)

    call global%set_int (var_str ("seed"), 0, is_known=.true.)

    call global%it_list%init ([2, 3], [5000, 20000])

    call global%write (u)

    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rt_data_1"

  end subroutine rt_data_1
```

Fill values

Fill in empty slots in the runtime data block.

⟨*RT data: execute tests*⟩+≡

```
  call test (rt_data_2, "rt_data_2", &
       "fill", &
       u, results)
```

⟨*RT data: test declarations*⟩+≡

```
  public :: rt_data_2
```

⟨*RT data: tests*⟩+≡

```
subroutine rt_data_2 (u)
  integer, intent(in) :: u
  type(rt_data_t), target :: global
  type(flavor_t), dimension(2) :: flv
  type(string_t) :: cut_expr_text
  type(ifile_t) :: ifile
  type(stream_t) :: stream
  type(parse_tree_t) :: parse_tree

  write (u, "(A)")  "* Test output: rt_data_2"
  write (u, "(A)")  "*   Purpose: initialize global runtime data &
       &and fill contents"
  write (u, "(A)")

  call syntax_model_file_init ()

  call global%global_init ()
  call fix_system_dependencies (global)

  call global%select_model (var_str ("Test"))

  call global%set_real (var_str ("sqrts"), &
       1000._default, is_known = .true.)
  call global%set_int (var_str ("seed"), &
       0, is_known=.true.)
  call flv%init ([25,25], global%model)

  call global%set_string (var_str ("$run_id"), &
       var_str ("run1"), is_known = .true.)
  call global%set_real (var_str ("luminosity"), &
       33._default, is_known = .true.)

  call syntax_pexpr_init ()
  cut_expr_text = "all Pt > 100 [s]"
  call ifile_append (ifile, cut_expr_text)
  call stream_init (stream, ifile)
  call parse_tree_init_lexpr (parse_tree, stream, .true.)
  global%pn%cuts_lexpr => parse_tree%get_root_ptr ()

  allocate (global%sample_fmt (2))
  global%sample_fmt(1) = "foo_fmt"
  global%sample_fmt(2) = "bar_fmt"

  call global%write (u)

  call parse_tree_final (parse_tree)
  call stream_final (stream)
  call ifile_final (ifile)
  call syntax_pexpr_final ()

  call global%final ()
  call syntax_model_file_final ()
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rt_data_2"

  end subroutine rt_data_2
```

Save and restore

Set up a local runtime data block, change some contents, restore the global block.

⟨*RT data: execute tests*⟩+≡

```
  call test (rt_data_3, "rt_data_3", &
       "save/restore", &
       u, results)
```

⟨*RT data: test declarations*⟩+≡

```
  public :: rt_data_3
```

⟨*RT data: tests*⟩+≡

```
  subroutine rt_data_3 (u)
    use event_base, only: event_callback_nop_t
    integer, intent(in) :: u
    type(rt_data_t), target :: global, local
    type(flavor_t), dimension(2) :: flv
    type(string_t) :: cut_expr_text
    type(ifile_t) :: ifile
    type(stream_t) :: stream
    type(parse_tree_t) :: parse_tree
    type(prclib_entry_t), pointer :: lib
    type(event_callback_nop_t) :: event_callback_nop

    write (u, "(A)")  "* Test output: rt_data_3"
    write (u, "(A)")  "*   Purpose: initialize global runtime data &
         &and fill contents;"
    write (u, "(A)")  "*            copy to local block and back"
    write (u, "(A)")

    write (u, "(A)")  "* Init global data"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()
    call fix_system_dependencies (global)

    call global%set_int (var_str ("seed"), &
         0, is_known=.true.)

    call global%select_model (var_str ("Test"))

    call global%set_real (var_str ("sqrts"),&
         1000._default, is_known = .true.)
    call flv%init ([25,25], global%model)
```

```
call global%beam_structure%init_sf (flv%get_name (), [1])
call global%beam_structure%set_sf (1, 1, var_str ("pdf_builtin"))

call global%set_string (var_str ("$run_id"), &
     var_str ("run1"), is_known = .true.)
call global%set_real (var_str ("luminosity"), &
     33._default, is_known = .true.)

call syntax_pexpr_init ()
cut_expr_text = "all Pt > 100 [s]"
call ifile_append (ifile, cut_expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (parse_tree, stream, .true.)
global%pn%cuts_lexpr => parse_tree%get_root_ptr ()

allocate (global%sample_fmt (2))
global%sample_fmt(1) = "foo_fmt"
global%sample_fmt(2) = "bar_fmt"

allocate (lib)
call lib%init (var_str ("library_1"))
call global%add_prclib (lib)

write (u, "(A)")  "* Init and modify local data"
write (u, "(A)")

call local%local_init (global)
call local%append_string (var_str ("$integration_method"), intrinsic=.true.)
call local%append_string (var_str ("$phs_method"), intrinsic=.true.)

call local%activate ()

write (u, "(1x,A,L1)")  "model associated   = ", associated (local%model)
write (u, "(1x,A,L1)")  "library associated = ", associated (local%prclib)
write (u, *)

call local%model_set_real (var_str ("ms"), 150._default)
call local%set_string (var_str ("$integration_method"), &
     var_str ("midpoint"), is_known = .true.)
call local%set_string (var_str ("$phs_method"), &
     var_str ("single"), is_known = .true.)

local%os_data%fc = "Local compiler"

allocate (lib)
call lib%init (var_str ("library_2"))
call local%add_prclib (lib)

call local%set_event_callback (event_callback_nop)

call local%write (u)

write (u, "(A)")
write (u, "(A)")  "* Restore global data"
```

```
    write (u, "(A)")

    call local%deactivate (global)

    write (u, "(1x,A,L1)")  "model associated   = ", associated (global%model)
    write (u, "(1x,A,L1)")  "library associated = ", associated (global%prclib)
    write (u, *)

    call global%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call parse_tree_final (parse_tree)
    call stream_final (stream)
    call ifile_final (ifile)
    call syntax_pexpr_final ()

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rt_data_3"

  end subroutine rt_data_3
```

Show variables

Display selected variables in the global record.

⟨*RT data: execute tests*⟩+≡

```
  call test (rt_data_4, "rt_data_4", &
       "show variables", &
       u, results)
```

⟨*RT data: test declarations*⟩+≡

```
  public :: rt_data_4
```

⟨*RT data: tests*⟩+≡

```
  subroutine rt_data_4 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global

    type(string_t), dimension(0) :: empty_string_array

    write (u, "(A)")  "* Test output: rt_data_4"
    write (u, "(A)")  "*   Purpose: display selected variables"
    write (u, "(A)")

    call global%global_init ()

    write (u, "(A)")  "* No variables:"
    write (u, "(A)")

    call global%write_vars (u, empty_string_array)
```

```
    write (u, "(A)")  "* Two variables:"
    write (u, "(A)")

    call global%write_vars (u, &
         [var_str ("?unweighted"), var_str ("$phs_method")])

    write (u, "(A)")
    write (u, "(A)")  "* Display whole record with selected variables"
    write (u, "(A)")

    call global%write (u, &
         vars = [var_str ("?unweighted"), var_str ("$phs_method")])

    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rt_data_4"

  end subroutine rt_data_4
```

Show parts

Display only selected parts in the state just after (global) initialization.

⟨*RT data: execute tests*⟩+≡

```
  call test (rt_data_5, "rt_data_5", &
       "show parts", &
       u, results)
```

⟨*RT data: test declarations*⟩+≡

```
  public :: rt_data_5
```

⟨*RT data: tests*⟩+≡

```
  subroutine rt_data_5 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global

    write (u, "(A)")  "* Test output: rt_data_5"
    write (u, "(A)")  "*   Purpose: display parts of rt data"
    write (u, "(A)")

    call global%global_init ()
    call global%write_libraries (u)

    write (u, "(A)")

    call global%write_beams (u)

    write (u, "(A)")

    call global%write_process_stack (u)

    call global%final ()
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rt_data_5"

  end subroutine rt_data_5
```

Local Model

Locally modify a model and restore the global one. We need an auxiliary function to determine the status of a model particle:

⟨*RT data: test auxiliary*⟩+≡

```
  function is_stable (pdg, global) result (flag)
    integer, intent(in) :: pdg
    type(rt_data_t), intent(in) :: global
    logical :: flag
    type(flavor_t) :: flv
    call flv%init (pdg, global%model)
    flag = flv%is_stable ()
  end function is_stable

  function is_polarized (pdg, global) result (flag)
    integer, intent(in) :: pdg
    type(rt_data_t), intent(in) :: global
    logical :: flag
    type(flavor_t) :: flv
    call flv%init (pdg, global%model)
    flag = flv%is_polarized ()
  end function is_polarized
```

⟨*RT data: execute tests*⟩+≡

```
  call test (rt_data_6, "rt_data_6", &
       "local model", &
       u, results)
```

⟨*RT data: test declarations*⟩+≡

```
  public :: rt_data_6
```

⟨*RT data: tests*⟩+≡

```
  subroutine rt_data_6 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global, local
    type(var_list_t), pointer :: model_vars
    type(string_t) :: var_name

    write (u, "(A)")  "* Test output: rt_data_6"
    write (u, "(A)")  "*   Purpose: apply and keep local modifications to model"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()
    call global%select_model (var_str ("Test"))

    write (u, "(A)")  "* Original model"
```

```
write (u, "(A)")

call global%write_model_list (u)
write (u, *)
write (u, "(A,L1)")  "s is stable    = ", is_stable (25, global)
write (u, "(A,L1)")  "f is polarized = ", is_polarized (6, global)

write (u, *)

var_name = "ff"

write (u, "(A)", advance="no")  "Global model variable: "
model_vars => global%model%get_var_list_ptr ()
call model_vars%write_var (var_name, u)

write (u, "(A)")
write (u, "(A)")  "* Apply local modifications: unstable"
write (u, "(A)")

call local%local_init (global)
call local%activate ()

call local%model_set_real (var_name, 0.4_default)
call local%modify_particle (25, stable = .false., decay = [var_str ("d1")])
call local%modify_particle (6, stable = .false., &
     decay = [var_str ("f1")], isotropic_decay = .true.)
call local%modify_particle (-6, stable = .false., &
     decay = [var_str ("f2"), var_str ("f3")], diagonal_decay = .true.)

call local%model%write (u)

write (u, "(A)")
write (u, "(A)")  "* Further modifications"
write (u, "(A)")

call local%modify_particle (6, stable = .false., &
     decay = [var_str ("f1")], &
     diagonal_decay = .true., isotropic_decay = .false.)
call local%modify_particle (-6, stable = .false., &
     decay = [var_str ("f2"), var_str ("f3")], &
     diagonal_decay = .false., isotropic_decay = .true.)
call local%model%write (u)

write (u, "(A)")
write (u, "(A)")  "* Further modifications: f stable but polarized"
write (u, "(A)")

call local%modify_particle (6, stable = .true., polarized = .true.)
call local%modify_particle (-6, stable = .true.)
call local%model%write (u)

write (u, "(A)")
write (u, "(A)")  "* Global model"
write (u, "(A)")
```

```
call global%model%write (u)
write (u, *)
write (u, "(A,L1)")  "s is stable    = ", is_stable (25, global)
write (u, "(A,L1)")  "f is polarized = ", is_polarized (6, global)

write (u, "(A)")
write (u, "(A)")  "* Local model"
write (u, "(A)")

call local%model%write (u)
write (u, *)
write (u, "(A,L1)")  "s is stable    = ", is_stable (25, local)
write (u, "(A,L1)")  "f is polarized = ", is_polarized (6, local)

write (u, *)

write (u, "(A)", advance="no")  "Global model variable: "
model_vars => global%model%get_var_list_ptr ()
call model_vars%write_var (var_name, u)

write (u, "(A)", advance="no")  "Local model variable: "
associate (model_var_list_ptr => local%model%get_var_list_ptr())
   call model_var_list_ptr%write_var (var_name, u)
end associate

write (u, "(A)")
write (u, "(A)")  "* Restore global"

call local%deactivate (global, keep_local = .true.)

write (u, "(A)")
write (u, "(A)")  "* Global model"
write (u, "(A)")

call global%model%write (u)
write (u, *)
write (u, "(A,L1)")  "s is stable    = ", is_stable (25, global)
write (u, "(A,L1)")  "f is polarized = ", is_polarized (6, global)

write (u, "(A)")
write (u, "(A)")  "* Local model"
write (u, "(A)")

call local%model%write (u)
write (u, *)
write (u, "(A,L1)")  "s is stable    = ", is_stable (25, local)
write (u, "(A,L1)")  "f is polarized = ", is_polarized (6, local)

write (u, *)

write (u, "(A)", advance="no")  "Global model variable: "
model_vars => global%model%get_var_list_ptr ()
call model_vars%write_var (var_name, u)
```

```
    write (u, "(A)", advance="no")  "Local model variable: "
    associate (model_var_list_ptr => local%model%get_var_list_ptr())
       call model_var_list_ptr%write_var (var_name, u)
    end associate

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call local%model%final ()
    deallocate (local%model)

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rt_data_6"

  end subroutine rt_data_6
```

Result variables

Initialize result variables and check that they are accessible via the global variable list.

⟨*RT data: execute tests*⟩+≡

```
  call test (rt_data_7, "rt_data_7", &
       "result variables", &
       u, results)
```

⟨*RT data: test declarations*⟩+≡

```
  public :: rt_data_7
```

⟨*RT data: tests*⟩+≡

```
  subroutine rt_data_7 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global

    write (u, "(A)")  "* Test output: rt_data_7"
    write (u, "(A)")  "*   Purpose: set and access result variables"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize process variables"
    write (u, "(A)")

    call global%global_init ()
    call global%process_stack%init_result_vars (var_str ("testproc"))

    call global%var_list%write_var (&
         var_str ("integral(testproc)"), u)
    call global%var_list%write_var (&
         var_str ("error(testproc)"), u)

    write (u, "(A)")
```

```
    write (u, "(A)")  "* Cleanup"

    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rt_data_7"

  end subroutine rt_data_7
```

Beam energy

If beam parameters are set, the variable `sqrts` is not necessarily the collision energy. The method `get_sqrts` fetches the correct value.

⟨*RT data: execute tests*⟩+≡

```
  call test (rt_data_8, "rt_data_8", &
       "beam energy", &
       u, results)
```

⟨*RT data: test declarations*⟩+≡

```
  public :: rt_data_8
```

⟨*RT data: tests*⟩+≡

```
  subroutine rt_data_8 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global

    write (u, "(A)")  "* Test output: rt_data_8"
    write (u, "(A)")  "*   Purpose: get correct collision energy"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize"
    write (u, "(A)")

    call global%global_init ()

    write (u, "(A)")  "* Set sqrts"
    write (u, "(A)")

    call global%set_real (var_str ("sqrts"), &
         1000._default, is_known = .true.)
    write (u, "(1x,A," // FMT_19 // ")")  "sqrts =", global%get_sqrts ()

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: rt_data_8"

  end subroutine rt_data_8
```

Local variable modifications

⟨*RT data: execute tests*⟩+≡

```
call test (rt_data_9, "rt_data_9", &
     "local variables", &
     u, results)
```

⟨*RT data: test declarations*⟩+≡

```
public :: rt_data_9
```

⟨*RT data: tests*⟩+≡

```
subroutine rt_data_9 (u)
  integer, intent(in) :: u
  type(rt_data_t), target :: global, local
  type(var_list_t), pointer :: var_list

  write (u, "(A)")  "* Test output: rt_data_9"
  write (u, "(A)")  "*   Purpose: handle local variables"
  write (u, "(A)")

  call syntax_model_file_init ()

  write (u, "(A)")  "* Initialize global record and set some variables"
  write (u, "(A)")

  call global%global_init ()
  call global%select_model (var_str ("Test"))

  call global%set_real (var_str ("sqrts"), 17._default, is_known = .true.)
  call global%set_real (var_str ("luminosity"), 2._default, is_known = .true.)
  call global%model_set_real (var_str ("ff"), 0.5_default)
  call global%model_set_real (var_str ("gy"), 1.2_default)

  var_list => global%get_var_list_ptr ()

  call var_list%write_var (var_str ("sqrts"), u)
  call var_list%write_var (var_str ("luminosity"), u)
  call var_list%write_var (var_str ("ff"), u)
  call var_list%write_var (var_str ("gy"), u)
  call var_list%write_var (var_str ("mf"), u)
  call var_list%write_var (var_str ("x"), u)

  write (u, "(A)")

  write (u, "(1x,A,1x,F5.2)")  "sqrts      = ", &
       global%get_rval (var_str ("sqrts"))
  write (u, "(1x,A,1x,F5.2)")  "luminosity = ", &
       global%get_rval (var_str ("luminosity"))
  write (u, "(1x,A,1x,F5.2)")  "ff         = ", &
       global%get_rval (var_str ("ff"))
  write (u, "(1x,A,1x,F5.2)")  "gy         = ", &
       global%get_rval (var_str ("gy"))
  write (u, "(1x,A,1x,F5.2)")  "mf         = ", &
       global%get_rval (var_str ("mf"))
  write (u, "(1x,A,1x,F5.2)")  "x          = ", &
       global%get_rval (var_str ("x"))
```

```
write (u, "(A)")
write (u, "(A)")  "* Create local record with local variables"
write (u, "(A)")

call local%local_init (global)

call local%append_real (var_str ("luminosity"), intrinsic = .true.)
call local%append_real (var_str ("x"), user = .true.)

call local%activate ()

var_list => local%get_var_list_ptr ()

call var_list%write_var (var_str ("sqrts"), u)
call var_list%write_var (var_str ("luminosity"), u)
call var_list%write_var (var_str ("ff"), u)
call var_list%write_var (var_str ("gy"), u)
call var_list%write_var (var_str ("mf"), u)
call var_list%write_var (var_str ("x"), u)

write (u, "(A)")

write (u, "(1x,A,1x,F5.2)")  "sqrts      = ", &
     local%get_rval (var_str ("sqrts"))
write (u, "(1x,A,1x,F5.2)")  "luminosity = ", &
     local%get_rval (var_str ("luminosity"))
write (u, "(1x,A,1x,F5.2)")  "ff         = ", &
     local%get_rval (var_str ("ff"))
write (u, "(1x,A,1x,F5.2)")  "gy         = ", &
     local%get_rval (var_str ("gy"))
write (u, "(1x,A,1x,F5.2)")  "mf         = ", &
     local%get_rval (var_str ("mf"))
write (u, "(1x,A,1x,F5.2)")  "x          = ", &
     local%get_rval (var_str ("x"))

write (u, "(A)")
write (u, "(A)")  "* Modify some local variables"
write (u, "(A)")

call local%set_real (var_str ("luminosity"), 42._default, is_known=.true.)
call local%set_real (var_str ("x"), 6.66_default, is_known=.true.)
call local%model_set_real (var_str ("ff"), 0.7_default)

var_list => local%get_var_list_ptr ()

call var_list%write_var (var_str ("sqrts"), u)
call var_list%write_var (var_str ("luminosity"), u)
call var_list%write_var (var_str ("ff"), u)
call var_list%write_var (var_str ("gy"), u)
call var_list%write_var (var_str ("mf"), u)
call var_list%write_var (var_str ("x"), u)

write (u, "(A)")
```

```
write (u, "(1x,A,1x,F5.2)")  "sqrts      = ", &
     local%get_rval (var_str ("sqrts"))
write (u, "(1x,A,1x,F5.2)")  "luminosity = ", &
     local%get_rval (var_str ("luminosity"))
write (u, "(1x,A,1x,F5.2)")  "ff         = ", &
     local%get_rval (var_str ("ff"))
write (u, "(1x,A,1x,F5.2)")  "gy         = ", &
     local%get_rval (var_str ("gy"))
write (u, "(1x,A,1x,F5.2)")  "mf         = ", &
     local%get_rval (var_str ("mf"))
write (u, "(1x,A,1x,F5.2)")  "x          = ", &
     local%get_rval (var_str ("x"))

write (u, "(A)")
write (u, "(A)")  "* Restore globals"
write (u, "(A)")

call local%deactivate (global)

var_list => global%get_var_list_ptr ()

call var_list%write_var (var_str ("sqrts"), u)
call var_list%write_var (var_str ("luminosity"), u)
call var_list%write_var (var_str ("ff"), u)
call var_list%write_var (var_str ("gy"), u)
call var_list%write_var (var_str ("mf"), u)
call var_list%write_var (var_str ("x"), u)

write (u, "(A)")

write (u, "(1x,A,1x,F5.2)")  "sqrts      = ", &
     global%get_rval (var_str ("sqrts"))
write (u, "(1x,A,1x,F5.2)")  "luminosity = ", &
     global%get_rval (var_str ("luminosity"))
write (u, "(1x,A,1x,F5.2)")  "ff         = ", &
     global%get_rval (var_str ("ff"))
write (u, "(1x,A,1x,F5.2)")  "gy         = ", &
     global%get_rval (var_str ("gy"))
write (u, "(1x,A,1x,F5.2)")  "mf         = ", &
     global%get_rval (var_str ("mf"))
write (u, "(1x,A,1x,F5.2)")  "x          = ", &
     global%get_rval (var_str ("x"))

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call local%local_final ()

call global%final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: rt_data_9"
```

```
end subroutine rt_data_9
```

33.3 Select implementations

For abstract types (process core, integrator, phase space, etc.), we need a way to dynamically select a concrete type, using either data given by the user or a previous selection of a concrete type. This is done by subroutines in the current module.

We would like to put this in the `me_methods` folder but it also depends on `gosam` and `openloops`, so it is unclear where to put it.

⟨`dispatch_me_methods.f90`⟩≡

```
⟨File header⟩

module dispatch_me_methods

⟨Use strings⟩
  use physics_defs, only: BORN
  use diagnostics
  use sm_qcd
  use variables, only: var_list_t
  use models
  use model_data

  use prc_core_def
  use prc_core
  use prc_test_core
  use prc_template_me
  use prc_test
  use prc_omega
  use prc_user_defined
  use prc_gosam
  use prc_openloops
  use prc_recola
  use prc_threshold

⟨Standard module head⟩

⟨Dispatch me methods: public⟩

contains

⟨Dispatch me methods: procedures⟩

end module dispatch_me_methods
```

33.3.1 Process Core Definition

The `prc_core_def_t` abstract type can be instantiated by providing a `$method` string variable.

Note: `core_def` has intent(inout) because gfortran 4.7.1 crashes for intent(out).

⟨*Dispatch me methods: public*⟩≡

```
  public :: dispatch_core_def
```

⟨*Dispatch me methods: procedures*⟩≡

```
subroutine dispatch_core_def (core_def, prt_in, prt_out, &
     model, var_list, id, nlo_type)
  class(prc_core_def_t), allocatable, intent(inout) :: core_def
  type(string_t), dimension(:), intent(in) :: prt_in
  type(string_t), dimension(:), intent(in) :: prt_out
  type(model_t), pointer, intent(in) :: model
  type(var_list_t), intent(in) :: var_list
  type(string_t), intent(in), optional :: id
  integer, intent(in), optional :: nlo_type
  type(string_t) :: method
  type(string_t) :: model_name
  logical :: cms_scheme
  type(string_t) :: restrictions
  logical :: openmp_support
  logical :: report_progress
  logical :: diags, diags_color
  type(string_t) :: extra_options
  integer :: nlo
  nlo = BORN;  if (present (nlo_type))  nlo = nlo_type
  method = var_list%get_sval (var_str ("$method"))
  if (associated (model)) then
     model_name = model%get_name ()
     cms_scheme = model%get_scheme () == "Complex_Mass_Scheme"
  else
     model_name = ""
     cms_scheme = .false.
  end if
  restrictions = var_list%get_sval (&
       var_str ("$restrictions"))
  diags = var_list%get_lval (&
       var_str ("?vis_diags"))
  diags_color = var_list%get_lval (&
       var_str ("?vis_diags_color"))
  openmp_support = var_list%get_lval (&
       var_str ("?omega_openmp"))
  report_progress = var_list%get_lval (&
       var_str ("?report_progress"))
  extra_options = var_list%get_sval (&
       var_str ("$omega_flags"))
  select case (char (method))
  case ("unit_test")
     allocate (prc_test_def_t :: core_def)
     select type (core_def)
     type is (prc_test_def_t)
        call core_def%init (model_name, prt_in, prt_out)
     end select
  case ("template")
     allocate (template_me_def_t :: core_def)
     select type (core_def)
     type is (template_me_def_t)
        call core_def%init (model, prt_in, prt_out, unity = .false.)
     end select
  case ("template_unity")
```

```
   allocate (template_me_def_t :: core_def)
   select type (core_def)
   type is (template_me_def_t)
      call core_def%init (model, prt_in, prt_out, unity = .true.)
   end select
case ("omega")
   allocate (omega_omega_def_t :: core_def)
   select type (core_def)
   type is (omega_omega_def_t)
      call core_def%init (model_name, prt_in, prt_out, &
           restrictions, cms_scheme, openmp_support, &
           report_progress, extra_options, diags, diags_color)
   end select
case ("ovm")
   allocate (omega_ovm_def_t :: core_def)
   select type (core_def)
   type is (omega_ovm_def_t)
      call core_def%init (model_name, prt_in, prt_out, &
           restrictions, cms_scheme, openmp_support, &
           report_progress, extra_options, diags, diags_color)
   end select
case ("gosam")
  allocate (gosam_def_t :: core_def)
  select type (core_def)
  type is (gosam_def_t)
    if (present (id)) then
       call core_def%init (id, model_name, prt_in, &
          prt_out, nlo, var_list)
    else
       call msg_fatal ("Dispatch GoSam def: No id!")
    end if
  end select
case ("openloops")
   allocate (openloops_def_t :: core_def)
   select type (core_def)
   type is (openloops_def_t)
      if (present (id)) then
         call core_def%init (id, model_name, prt_in, &
            prt_out, nlo, var_list)
      else
         call msg_fatal ("Dispatch OpenLoops def: No id!")
      end if
   end select
case ("recola")
   call abort_if_recola_not_active ()
   allocate (recola_def_t :: core_def)
   select type (core_def)
   type is (recola_def_t)
      if (present (id)) then
         call core_def%init (id, model_name, prt_in, &
            prt_out, nlo)
      else
         call msg_fatal ("Dispatch RECOLA def: No id!")
      end if
```

```
      end select
   case ("dummy")
      allocate (user_defined_test_def_t :: core_def)
      select type (core_def)
      type is (user_defined_test_def_t)
         if (present (id)) then
            call core_def%init (id, model_name, prt_in, prt_out)
         else
            call msg_fatal ("Dispatch User-Defined Test def: No id!")
         end if
      end select
   case ("threshold")
      allocate (threshold_def_t :: core_def)
      select type (core_def)
      type is (threshold_def_t)
         if (present (id)) then
            call core_def%init (id, model_name, prt_in, prt_out, &
                 nlo, restrictions)
         else
            call msg_fatal ("Dispatch Threshold def: No id!")
         end if
      end select
   case default
      call msg_fatal ("Process configuration: method '" &
           // char (method) // "' not implemented")
   end select
end subroutine dispatch_core_def
```

33.3.2 Process core allocation

Here we allocate an object of abstract type `prc_core_t` with a concrete type that matches a process definition. The `prc_omega_t` extension will require the current parameter set, so we take the opportunity to grab it from the model.

⟨*Dispatch me methods: public*⟩+≡

```
public :: dispatch_core
```

⟨*Dispatch me methods: procedures*⟩+≡

```
subroutine dispatch_core (core, core_def, model, &
     helicity_selection, qcd, use_color_factors)
   class(prc_core_t), allocatable, intent(inout) :: core
   class(prc_core_def_t), intent(in) :: core_def
   class(model_data_t), intent(in), target, optional :: model
   type(helicity_selection_t), intent(in), optional :: helicity_selection
   type(qcd_t), intent(in), optional :: qcd
   logical, intent(in), optional :: use_color_factors
   select type (core_def)
   type is (prc_test_def_t)
      allocate (test_t :: core)
   type is (template_me_def_t)
      allocate (prc_template_me_t :: core)
      select type (core)
      type is (prc_template_me_t)
         call core%set_parameters (model)
```

```
      end select
   class is (omega_def_t)
      if (.not. allocated (core)) allocate (prc_omega_t :: core)
      select type (core)
      type is (prc_omega_t)
         call core%set_parameters (model, &
              helicity_selection, qcd, use_color_factors)
      end select
   type is (gosam_def_t)
     if (.not. allocated (core)) allocate (prc_gosam_t :: core)
     select type (core)
     type is (prc_gosam_t)
       call core%set_parameters (qcd)
     end select
   type is (openloops_def_t)
     if (.not. allocated (core)) allocate (prc_openloops_t :: core)
     select type (core)
     type is (prc_openloops_t)
        call core%set_parameters (qcd)
     end select
   type is (recola_def_t)
     if (.not. allocated (core)) allocate (prc_recola_t :: core)
     select type (core)
     type is (prc_recola_t)
        call core%set_parameters (qcd, model)
     end select
   type is (user_defined_test_def_t)
     if (.not. allocated (core)) allocate (prc_user_defined_test_t :: core)
     select type (core)
     type is (prc_user_defined_test_t)
        call core%set_parameters (qcd, model)
     end select
   type is (threshold_def_t)
     if (.not. allocated (core)) allocate (prc_threshold_t :: core)
     select type (core)
     type is (prc_threshold_t)
        call core%set_parameters (qcd, model)
     end select
   class default
      call msg_bug ("Process core: unexpected process definition type")
   end select
 end subroutine dispatch_core
```

33.3.3 Process core update and restoration

Here we take an existing object of abstract type `prc_core_t` and update the parameters as given by the current state of `model`. Optionally, we can save the previous state as `saved_core`. The second routine restores the original from the save.

(In the test case, there is no possible update.)

⟨*Dispatch me methods: public*⟩+≡

```
public :: dispatch_core_update
```

```
public :: dispatch_core_restore
```

⟨*Dispatch me methods: procedures*⟩+≡

```
subroutine dispatch_core_update &
     (core, model, helicity_selection, qcd, saved_core)

  class(prc_core_t), allocatable, intent(inout) :: core
  class(model_data_t), intent(in), optional, target :: model
  type(helicity_selection_t), intent(in), optional :: helicity_selection
  type(qcd_t), intent(in), optional :: qcd
  class(prc_core_t), allocatable, intent(inout), optional :: saved_core

  if (present (saved_core)) then
     allocate (saved_core, source = core)
  end if
  select type (core)
  type is (test_t)
  type is (prc_omega_t)
     call core%set_parameters (model, helicity_selection, qcd)
     call core%activate_parameters ()
  class is (prc_user_defined_base_t)
    call msg_message ("Updating user defined cores is not implemented yet.")
  class default
     call msg_bug ("Process core update: unexpected process definition type")
  end select
end subroutine dispatch_core_update

subroutine dispatch_core_restore (core, saved_core)

  class(prc_core_t), allocatable, intent(inout) :: core
  class(prc_core_t), allocatable, intent(inout) :: saved_core

  call move_alloc (from = saved_core, to = core)
  select type (core)
  type is (test_t)
  type is (prc_omega_t)
     call core%activate_parameters ()
  class default
     call msg_bug ("Process core restore: unexpected process definition type")
  end select
end subroutine dispatch_core_restore
```

33.3.4 Unit Tests

Test module, followed by the corresponding implementation module.

⟨`dispatch_ut.f90`⟩≡

```
⟨File header⟩

module dispatch_ut
  use unit_tests
  use dispatch_uti

⟨Standard module head⟩
```

⟨*Dispatch: public test*⟩

⟨*Dispatch: public test auxiliary*⟩

```
contains
```

⟨*Dispatch: test driver*⟩

```
end module dispatch_ut
```

⟨dispatch_uti.f90⟩≡

⟨*File header*⟩

```
module dispatch_uti
```
⟨*Use kinds*⟩
⟨*Use strings*⟩
```
  use os_interface, only: os_data_t, os_data_init
  use physics_defs, only: ELECTRON, PROTON
  use sm_qcd, only: qcd_t
  use flavors, only: flavor_t
  use interactions, only: reset_interaction_counter
  use pdg_arrays, only: pdg_array_t, assignment(=)
  use prc_core_def, only: prc_core_def_t
  use prc_test_core, only: test_t
  use prc_core, only: prc_core_t
  use prc_test, only: prc_test_def_t
  use prc_omega, only: omega_omega_def_t, prc_omega_t
  use sf_mappings, only: sf_channel_t
  use sf_base, only: sf_data_t, sf_config_t
  use phs_base, only: phs_channel_collection_t
  use variables, only: var_list_t
  use model_data, only: model_data_t
  use models, only: syntax_model_file_init, syntax_model_file_final
  use rt_data, only: rt_data_t

  use dispatch_phase_space, only: dispatch_sf_channels
  use dispatch_beams, only: sf_prop_t, dispatch_qcd
  use dispatch_beams, only: dispatch_sf_config, dispatch_sf_data
  use dispatch_me_methods, only: dispatch_core_def, dispatch_core
  use dispatch_me_methods, only: dispatch_core_update, dispatch_core_restore

  use sf_base_ut, only: sf_test_data_t
```

⟨*Standard module head*⟩

⟨*Dispatch: public test auxiliary*⟩

⟨*Dispatch: test declarations*⟩

```
contains
```

⟨*Dispatch: tests*⟩

⟨*Dispatch: test auxiliary*⟩

```
end module dispatch_uti
```

API: driver for the unit tests below.

⟨*Dispatch: public test*⟩≡
```
public :: dispatch_test
```

⟨*Dispatch: test driver*⟩≡
```
subroutine dispatch_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨Dispatch: execute tests⟩
end subroutine dispatch_test
```

Select type: process definition

⟨*Dispatch: execute tests*⟩≡
```
call test (dispatch_1, "dispatch_1", &
     "process configuration method", &
     u, results)
```

⟨*Dispatch: test declarations*⟩≡
```
public :: dispatch_1
```

⟨*Dispatch: tests*⟩≡
```
subroutine dispatch_1 (u)
  integer, intent(in) :: u
  type(string_t), dimension(2) :: prt_in, prt_out
  type(rt_data_t), target :: global
  class(prc_core_def_t), allocatable :: core_def

  write (u, "(A)")  "* Test output: dispatch_1"
  write (u, "(A)")  "*   Purpose: select process configuration method"
  write (u, "(A)")

  call global%global_init ()

  call global%set_log (var_str ("?omega_openmp"), &
       .false., is_known = .true.)

  prt_in = [var_str ("a"), var_str ("b")]
  prt_out = [var_str ("c"), var_str ("d")]

  write (u, "(A)")  "* Allocate core_def as prc_test_def"

  call global%set_string (var_str ("$method"), &
       var_str ("unit_test"), is_known = .true.)
  call dispatch_core_def (core_def, prt_in, prt_out, global%model, global%var_list)
  select type (core_def)
  type is (prc_test_def_t)
     call core_def%write (u)
  end select

  deallocate (core_def)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Allocate core_def as omega_def"
    write (u, "(A)")

    call global%set_string (var_str ("$method"), &
         var_str ("omega"), is_known = .true.)
    call dispatch_core_def (core_def, prt_in, prt_out, global%model, global%var_list)
    select type (core_def)
    type is (omega_omega_def_t)
       call core_def%write (u)
    end select

    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: dispatch_1"

  end subroutine dispatch_1
```

Select type: process core

⟨*Dispatch: execute tests*⟩+≡

```
  call test (dispatch_2, "dispatch_2", &
       "process core", &
       u, results)
```

⟨*Dispatch: test declarations*⟩+≡

```
  public :: dispatch_2
```

⟨*Dispatch: tests*⟩+≡

```
  subroutine dispatch_2 (u)
    integer, intent(in) :: u
    type(string_t), dimension(2) :: prt_in, prt_out
    type(rt_data_t), target :: global
    class(prc_core_def_t), allocatable :: core_def
    class(prc_core_t), allocatable :: core

    write (u, "(A)")  "* Test output: dispatch_2"
    write (u, "(A)")  "*   Purpose: select process configuration method"
    write (u, "(A)")  "             and allocate process core"
    write (u, "(A)")

    call syntax_model_file_init ()
    call global%global_init ()

    prt_in = [var_str ("a"), var_str ("b")]
    prt_out = [var_str ("c"), var_str ("d")]

    write (u, "(A)")  "* Allocate core as test_t"
    write (u, "(A)")

    call global%set_string (var_str ("$method"), &
         var_str ("unit_test"), is_known = .true.)
```

```
    call dispatch_core_def (core_def, prt_in, prt_out, global%model, global%var_list)
    call dispatch_core (core, core_def)
    select type (core)
    type is (test_t)
       call core%write (u)
    end select

    deallocate (core)
    deallocate (core_def)

    write (u, "(A)")
    write (u, "(A)")  "* Allocate core as prc_omega_t"
    write (u, "(A)")

    call global%set_string (var_str ("$method"), &
         var_str ("omega"), is_known = .true.)
    call dispatch_core_def (core_def, prt_in, prt_out, global%model, global%var_list)

    call global%select_model (var_str ("Test"))

    call global%set_log (&
         var_str ("?helicity_selection_active"), &
         .true., is_known = .true.)
    call global%set_real (&
         var_str ("helicity_selection_threshold"), &
         1e9_default, is_known = .true.)
    call global%set_int (&
         var_str ("helicity_selection_cutoff"), &
         10, is_known = .true.)

    call dispatch_core (core, core_def, &
         global%model, &
         global%get_helicity_selection ())
    call core_def%allocate_driver (core%driver, var_str (""))

    select type (core)
    type is (prc_omega_t)
       call core%write (u)
    end select

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: dispatch_2"

  end subroutine dispatch_2
```

Select type: structure-function data

This is an extra dispatcher that enables the test structure functions. This procedure should be assigned to the `dispatch_sf_data_extra` hook before any

tests are executed.

⟨*Dispatch: public test auxiliary*⟩≡

```
public :: dispatch_sf_data_test
```

⟨*Dispatch: test auxiliary*⟩≡

```
subroutine dispatch_sf_data_test (data, sf_method, i_beam, sf_prop, &
     var_list, var_list_global, model, os_data, sqrts, pdg_in, pdg_prc, polarized)
  class(sf_data_t), allocatable, intent(inout) :: data
  type(string_t), intent(in) :: sf_method
  integer, dimension(:), intent(in) :: i_beam
  type(var_list_t), intent(in) :: var_list
  type(var_list_t), intent(inout) :: var_list_global
  class(model_data_t), target, intent(in) :: model
  type(os_data_t), intent(in) :: os_data
  real(default), intent(in) :: sqrts
  type(pdg_array_t), dimension(:), intent(inout) :: pdg_in
  type(pdg_array_t), dimension(:,:), intent(in) :: pdg_prc
  type(sf_prop_t), intent(inout) :: sf_prop
  logical, intent(in) :: polarized
  select case (char (sf_method))
  case ("sf_test_0", "sf_test_1")
     allocate (sf_test_data_t :: data)
     select type (data)
     type is (sf_test_data_t)
        select case (char (sf_method))
        case ("sf_test_0");  call data%init (model, pdg_in(i_beam(1)))
        case ("sf_test_1");  call data%init (model, pdg_in(i_beam(1)),&
             mode = 1)
        end select
     end select
  end select
end subroutine dispatch_sf_data_test
```

The actual test. We can't move this to `beams` as it depends on `model_features` for the `model_list_t`.

⟨*Dispatch: execute tests*⟩+≡

```
call test (dispatch_7, "dispatch_7", &
     "structure-function data", &
     u, results)
```

⟨*Dispatch: test declarations*⟩+≡

```
public :: dispatch_7
```

⟨*Dispatch: tests*⟩+≡

```
subroutine dispatch_7 (u)
  integer, intent(in) :: u
  type(rt_data_t), target :: global
  type(os_data_t) :: os_data
  type(string_t) :: prt, sf_method
  type(sf_prop_t) :: sf_prop
  class(sf_data_t), allocatable :: data
  type(pdg_array_t), dimension(1) :: pdg_in
  type(pdg_array_t), dimension(1,1) :: pdg_prc
  type(pdg_array_t), dimension(1) :: pdg_out
  integer, dimension(:), allocatable :: pdg1
```

```
write (u, "(A)")  "* Test output: dispatch_7"
write (u, "(A)")  "*   Purpose: select and configure &
     &structure function data"
write (u, "(A)")

call global%global_init ()

call os_data_init (os_data)
call syntax_model_file_init ()
call global%select_model (var_str ("QCD"))

call reset_interaction_counter ()
call global%set_real (var_str ("sqrts"), &
     14000._default, is_known = .true.)
prt = "p"
call global%beam_structure%init_sf ([prt, prt], [1])
pdg_in = 2212

write (u, "(A)")  "* Allocate data as sf_pdf_builtin_t"
write (u, "(A)")

sf_method = "pdf_builtin"
call dispatch_sf_data (data, sf_method, [1], sf_prop, &
     global%get_var_list_ptr (), global%var_list, &
     global%model, global%os_data, global%get_sqrts (), &
     pdg_in, pdg_prc, .false.)
call data%write (u)

call data%get_pdg_out (pdg_out)
pdg1 = pdg_out(1)
write (u, "(A)")
write (u, "(1x,A,99(1x,I0))")  "PDG(out) = ", pdg1

deallocate (data)

write (u, "(A)")
write (u, "(A)")  "* Allocate data for different PDF set"
write (u, "(A)")

pdg_in = 2212

call global%set_string (var_str ("$pdf_builtin_set"), &
     var_str ("CTEQ6M"), is_known = .true.)
sf_method = "pdf_builtin"
call dispatch_sf_data (data, sf_method, [1], sf_prop, &
     global%get_var_list_ptr (), global%var_list, &
     global%model, global%os_data, global%get_sqrts (), &
     pdg_in, pdg_prc, .false.)
call data%write (u)

call data%get_pdg_out (pdg_out)
pdg1 = pdg_out(1)
write (u, "(A)")
```

```
    write (u, "(1x,A,99(1x,I0))")  "PDG(out) = ", pdg1

    deallocate (data)

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: dispatch_7"

  end subroutine dispatch_7
```

Beam structure

The actual test. We can't move this to `beams` as it depends on `model_features` for the `model_list_t`.

⟨*Dispatch: execute tests*⟩+≡

```
  call test (dispatch_8, "dispatch_8", &
       "beam structure", &
       u, results)
```

⟨*Dispatch: test declarations*⟩+≡

```
  public :: dispatch_8
```

⟨*Dispatch: tests*⟩+≡

```
  subroutine dispatch_8 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global
    type(os_data_t) :: os_data
    type(flavor_t), dimension(2) :: flv
    type(sf_config_t), dimension(:), allocatable :: sf_config
    type(sf_prop_t) :: sf_prop
    type(sf_channel_t), dimension(:), allocatable :: sf_channel
    type(phs_channel_collection_t) :: coll
    type(string_t) :: sf_string
    integer :: i
    type(pdg_array_t), dimension (2,1) :: pdg_prc

    write (u, "(A)")  "* Test output: dispatch_8"
    write (u, "(A)")  "*   Purpose: configure a structure-function chain"
    write (u, "(A)")

    call global%global_init ()

    call os_data_init (os_data)
    call syntax_model_file_init ()
    call global%select_model (var_str ("QCD"))

    write (u, "(A)")  "* Allocate LHC beams with PDF builtin"
    write (u, "(A)")

    call flv(1)%init (PROTON, global%model)
    call flv(2)%init (PROTON, global%model)
```

```
call reset_interaction_counter ()
call global%set_real (var_str ("sqrts"), &
     14000._default, is_known = .true.)

call global%beam_structure%init_sf (flv%get_name (), [1])
call global%beam_structure%set_sf (1, 1, var_str ("pdf_builtin"))

call dispatch_sf_config (sf_config, sf_prop, global%beam_structure, &
     global%get_var_list_ptr (), global%var_list, &
     global%model, global%os_data, global%get_sqrts (), pdg_prc)
do i = 1, size (sf_config)
   call sf_config(i)%write (u)
end do

call dispatch_sf_channels (sf_channel, sf_string, sf_prop, coll, &
     global%var_list, global%get_sqrts(), global%beam_structure)
write (u, "(1x,A)")  "Mapping configuration:"
do i = 1, size (sf_channel)
   write (u, "(2x)", advance = "no")
   call sf_channel(i)%write (u)
end do

write (u, "(A)")
write (u, "(A)")  "* Allocate ILC beams with CIRCE1"
write (u, "(A)")

call global%select_model (var_str ("QED"))
call flv(1)%init ( ELECTRON, global%model)
call flv(2)%init (-ELECTRON, global%model)

call reset_interaction_counter ()
call global%set_real (var_str ("sqrts"), &
     500._default, is_known = .true.)
call global%set_log (var_str ("?circe1_generate"), &
     .false., is_known = .true.)

call global%beam_structure%init_sf (flv%get_name (), [1])
call global%beam_structure%set_sf (1, 1, var_str ("circe1"))

call dispatch_sf_config (sf_config, sf_prop, global%beam_structure, &
     global%get_var_list_ptr (), global%var_list, &
     global%model, global%os_data, global%get_sqrts (), pdg_prc)
do i = 1, size (sf_config)
   call sf_config(i)%write (u)
end do

call dispatch_sf_channels (sf_channel, sf_string, sf_prop, coll, &
     global%var_list, global%get_sqrts(), global%beam_structure)
write (u, "(1x,A)")  "Mapping configuration:"
do i = 1, size (sf_channel)
   write (u, "(2x)", advance = "no")
   call sf_channel(i)%write (u)
end do
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: dispatch_8"

  end subroutine dispatch_8
```

Update process core parameters

This test dispatches a process core, temporarily modifies parameters, then restores the original.

⟨*Dispatch: execute tests*⟩+≡

```
  call test (dispatch_10, "dispatch_10", &
       "process core update", &
       u, results)
```

⟨*Dispatch: test declarations*⟩+≡

```
  public :: dispatch_10
```

⟨*Dispatch: tests*⟩+≡

```
  subroutine dispatch_10 (u)
    integer, intent(in) :: u
    type(string_t), dimension(2) :: prt_in, prt_out
    type(rt_data_t), target :: global
    class(prc_core_def_t), allocatable :: core_def
    class(prc_core_t), allocatable :: core, saved_core
    type(var_list_t), pointer :: model_vars

    write (u, "(A)")  "* Test output: dispatch_10"
    write (u, "(A)")  "*   Purpose: select process configuration method,"
    write (u, "(A)")  "             allocate process core,"
    write (u, "(A)")  "             temporarily reset parameters"
    write (u, "(A)")

    call syntax_model_file_init ()
    call global%global_init ()

    prt_in = [var_str ("a"), var_str ("b")]
    prt_out = [var_str ("c"), var_str ("d")]

    write (u, "(A)")  "* Allocate core as prc_omega_t"
    write (u, "(A)")

    call global%set_string (var_str ("$method"), &
         var_str ("omega"), is_known = .true.)
    call dispatch_core_def (core_def, prt_in, prt_out, global%model, global%var_list)

    call global%select_model (var_str ("Test"))
```

```
call dispatch_core (core, core_def, global%model)
call core_def%allocate_driver (core%driver, var_str (""))

select type (core)
type is (prc_omega_t)
   call core%write (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Update core with modified model and helicity selection"
write (u, "(A)")

model_vars => global%model%get_var_list_ptr ()

call model_vars%set_real (var_str ("gy"), 2._default, &
     is_known = .true.)
call global%model%update_parameters ()

call global%set_log (&
     var_str ("?helicity_selection_active"), &
     .true., is_known = .true.)
call global%set_real (&
     var_str ("helicity_selection_threshold"), &
     2e10_default, is_known = .true.)
call global%set_int (&
     var_str ("helicity_selection_cutoff"), &
     5, is_known = .true.)

call dispatch_core_update (core, &
     global%model, &
     global%get_helicity_selection (), &
     saved_core = saved_core)
select type (core)
type is (prc_omega_t)
   call core%write (u)
end select

write (u, "(A)")
write (u, "(A)")  "* Restore core from save"
write (u, "(A)")

call dispatch_core_restore (core, saved_core)
select type (core)
type is (prc_omega_t)
   call core%write (u)
end select

call global%final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: dispatch_10"
```

```
end subroutine dispatch_10
```

QCD Coupling

This test dispatches an `qcd` object, which is used to compute the (running) coupling by one of several possible methods.

We can't move this to `beams` as it depends on `model_features` for the `model_list_t`.

⟨*Dispatch: execute tests*⟩+≡

```
call test (dispatch_11, "dispatch_11", &
     "QCD coupling", &
     u, results)
```

⟨*Dispatch: test declarations*⟩+≡

```
public :: dispatch_11
```

⟨*Dispatch: tests*⟩+≡

```
subroutine dispatch_11 (u)
  integer, intent(in) :: u
  type(rt_data_t), target :: global
  type(qcd_t) :: qcd
  type(var_list_t), pointer :: model_vars

  write (u, "(A)")  "* Test output: dispatch_11"
  write (u, "(A)")  "*   Purpose: select QCD coupling formula"
  write (u, "(A)")

  call syntax_model_file_init ()
  call global%global_init ()
  call global%select_model (var_str ("SM"))
  model_vars => global%get_var_list_ptr ()

  write (u, "(A)")  "* Allocate alpha_s as fixed"
  write (u, "(A)")

  call global%set_log (var_str ("?alpha_s_is_fixed"), &
       .true., is_known = .true.)
  call dispatch_qcd (qcd, global%get_var_list_ptr (), global%os_data)
  call qcd%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Allocate alpha_s as running (built-in)"
  write (u, "(A)")

  call global%set_log (var_str ("?alpha_s_is_fixed"), &
       .false., is_known = .true.)
  call global%set_log (var_str ("?alpha_s_from_mz"), &
       .true., is_known = .true.)
  call global%set_int &
       (var_str ("alpha_s_order"), 1, is_known = .true.)
  call model_vars%set_real (var_str ("alphas"), 0.1234_default, &
        is_known=.true.)
  call model_vars%set_real (var_str ("mZ"), 91.234_default, &
```

```
      is_known=.true.)
  call dispatch_qcd (qcd, global%get_var_list_ptr (), global%os_data)
  call qcd%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Allocate alpha_s as running (built-in, Lambda defined)"
  write (u, "(A)")

  call global%set_log (var_str ("?alpha_s_from_mz"), &
       .false., is_known = .true.)
  call global%set_log (&
       var_str ("?alpha_s_from_lambda_qcd"), &
       .true., is_known = .true.)
  call global%set_real &
       (var_str ("lambda_qcd"), 250.e-3_default, &
        is_known=.true.)
  call global%set_int &
       (var_str ("alpha_s_order"), 2, is_known = .true.)
  call global%set_int &
       (var_str ("alpha_s_nf"), 4, is_known = .true.)
  call dispatch_qcd (qcd, global%get_var_list_ptr (), global%os_data)
  call qcd%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Allocate alpha_s as running (using builtin PDF set)"
  write (u, "(A)")

  call global%set_log (&
       var_str ("?alpha_s_from_lambda_qcd"), &
       .false., is_known = .true.)
  call global%set_log &
       (var_str ("?alpha_s_from_pdf_builtin"), &
       .true., is_known = .true.)
  call dispatch_qcd (qcd, global%get_var_list_ptr (), global%os_data)
  call qcd%write (u)

  call global%final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: dispatch_11"

end subroutine dispatch_11
```

33.4 Process Configuration

This module communicates between the toplevel command structure with its runtime data set and the process-library handling modules which collect the definition of individual processes. Its primary purpose is to select from the available matrix-element generating methods and configure the entry in the process library accordingly.

⟨process_configurations.f90⟩≡

```
⟨File header⟩

module process_configurations

⟨Use strings⟩
  use diagnostics
  use models
  use prc_core_def
  use particle_specifiers
  use process_libraries
  use rt_data
  use variables, only: var_list_t

  use dispatch_me_methods, only: dispatch_core_def
  use prc_user_defined, only: user_defined_def_t

⟨Standard module head⟩

⟨Process configurations: public⟩

⟨Process configurations: types⟩

contains

⟨Process configurations: procedures⟩

end module process_configurations
```

33.4.1 Data Type

⟨Process configurations: public⟩≡

```
  public :: process_configuration_t
```

⟨Process configurations: types⟩≡

```
  type :: process_configuration_t
     type(process_def_entry_t), pointer :: entry => null ()
     type(string_t) :: id
     integer :: num_id = 0
   contains
   ⟨Process configurations: process configuration: TBP⟩
  end type process_configuration_t
```

Initialize a process. We only need the name, the number of incoming particles, and the number of components.

⟨Process configurations: process configuration: TBP⟩≡

```
procedure :: init => process_configuration_init
```

⟨*Process configurations: procedures*⟩≡

```
subroutine process_configuration_init &
     (config, prc_name, n_in, n_components, global, nlo_proc)
  class(process_configuration_t), intent(out) :: config
  type(string_t), intent(in) :: prc_name
  integer, intent(in) :: n_in
  integer, intent(in) :: n_components
  type(rt_data_t), intent(in) :: global
  logical, intent(in), optional :: nlo_proc
  type(model_t), pointer :: model
  logical :: nlo_process
  model => global%model
  config%id = prc_name
  if (present (nlo_proc)) then
     nlo_process = nlo_proc
  else
     nlo_process = global%nlo_fixed_order
  end if
  allocate (config%entry)
  if (global%var_list%is_known (var_str ("process_num_id"))) then
     config%num_id = &
          global%var_list%get_ival (var_str ("process_num_id"))
     call config%entry%init (prc_name, &
          model = model, n_in = n_in, n_components = n_components, &
          num_id = config%num_id, nlo_process = nlo_process)
  else
     call config%entry%init (prc_name, &
          model = model, n_in = n_in, n_components = n_components, &
          nlo_process = nlo_process)
  end if
end subroutine process_configuration_init
```

Initialize a process component. The details depend on the process method, which determines the type of the process component core. We set the incoming and outgoing particles (as strings, to be interpreted by the process driver). All other information is taken from the variable list.

The dispatcher gets only the names of the particles. The process component definition gets the complete specifiers which contains a polarization flag and names of decay processes, where applicable.

⟨*Process configurations: process configuration: TBP*⟩+≡

```
procedure :: setup_component => process_configuration_setup_component
```

⟨*Process configurations: procedures*⟩+≡

```
subroutine process_configuration_setup_component &
     (config, i_component, prt_in, prt_out, model, var_list, &
      nlo_type, can_be_integrated)
  class(process_configuration_t), intent(inout) :: config
  integer, intent(in) :: i_component
  type(prt_spec_t), dimension(:), intent(in) :: prt_in
  type(prt_spec_t), dimension(:), intent(in) :: prt_out
  type(model_t), pointer, intent(in) :: model
  type(var_list_t), intent(in) :: var_list
```

```
    integer, intent(in), optional :: nlo_type
    logical, intent(in), optional :: can_be_integrated
    type(string_t), dimension(:), allocatable :: prt_str_in
    type(string_t), dimension(:), allocatable :: prt_str_out
    class(prc_core_def_t), allocatable :: core_def
    type(string_t) :: method
    integer :: i

    allocate (prt_str_in  (size (prt_in)))
    allocate (prt_str_out (size (prt_out)))
    forall (i = 1:size (prt_in))  prt_str_in(i)  = prt_in(i)% get_name ()
    forall (i = 1:size (prt_out)) prt_str_out(i) = prt_out(i)%get_name ()

    call dispatch_core_def (core_def, prt_str_in, prt_str_out, &
         model, var_list, config%id, nlo_type)
    select type (core_def)
    class is (user_defined_def_t)
       if (present (can_be_integrated)) then
          call core_def%set_active_writer (can_be_integrated)
       else
          call msg_fatal ("Cannot decide if user-defined core is integrated!")
       end if
    end select
    method = var_list%get_sval (var_str ("$method"))
    call config%entry%import_component (i_component, &
         n_out = size (prt_out), &
         prt_in = prt_in, &
         prt_out = prt_out, &
         method = method, &
         variant = core_def, &
         nlo_type = nlo_type, &
         can_be_integrated = can_be_integrated)
end subroutine process_configuration_setup_component
```

⟨*Process configurations: process configuration: TBP*⟩+≡

```
procedure :: set_fixed_emitter => process_configuration_set_fixed_emitter
```

⟨*Process configurations: procedures*⟩+≡

```
subroutine process_configuration_set_fixed_emitter (config, i, emitter)
   class(process_configuration_t), intent(inout) :: config
   integer, intent(in) :: i, emitter
   call config%entry%set_fixed_emitter (i, emitter)
end subroutine process_configuration_set_fixed_emitter
```

⟨*Process configurations: process configuration: TBP*⟩+≡

```
procedure :: set_coupling_powers => process_configuration_set_coupling_powers
```

⟨*Process configurations: procedures*⟩+≡

```
subroutine process_configuration_set_coupling_powers (config, alpha_power, alphas_power)
  class(process_configuration_t), intent(inout) :: config
  integer, intent(in) :: alpha_power, alphas_power
  call config%entry%set_coupling_powers (alpha_power, alphas_power)
end subroutine process_configuration_set_coupling_powers
```

⟨*Process configurations: process configuration: TBP*⟩+≡

```
procedure :: set_component_associations => &
     process_configuration_set_component_associations
```

⟨*Process configurations: procedures*⟩+≡

```
subroutine process_configuration_set_component_associations &
       (config, i_list, pdf, use_real_finite, mismatch)
  class(process_configuration_t), intent(inout) :: config
  integer, intent(in), dimension(:) :: i_list
  logical, intent(in) :: pdf, use_real_finite, mismatch
  integer :: i_component
  do i_component = 1, config%entry%get_n_components ()
     if (any (i_list == i_component)) then
        if (pdf) then
           call config%entry%set_associated_components (i_component, &
                  i_list(1), i_list(2), i_list(3), i_list(4), i_pdf = i_list(5))
        else if (mismatch) then
           call config%entry%set_associated_components (i_component, &
                  i_list(1), i_list(2), i_list(3), i_list(4), i_pdf = i_list(5))
        else if (use_real_finite) then
           call config%entry%set_associated_components (i_component, &
                  i_list(1), i_list(2), i_list(3), i_list(4), i_rfin = i_list(5))
        else
           call config%entry%set_associated_components (i_component, &
                  i_list(1), i_list(2), i_list(3), i_list(4))
        end if
     end if
  end do
end subroutine process_configuration_set_component_associations
```

Record a process configuration: append it to the currently selected process definition library.

⟨*Process configurations: process configuration: TBP*⟩+≡

```
procedure :: record => process_configuration_record
```

⟨*Process configurations: procedures*⟩+≡

```
subroutine process_configuration_record (config, global)
  class(process_configuration_t), intent(inout) :: config
  type(rt_data_t), intent(inout) :: global
  if (associated (global%prclib)) then
     call global%prclib%open ()
     call global%prclib%append (config%entry)
     if (config%num_id /= 0) then
        write (msg_buffer, "(5A,I0,A)") "Process library '", &
              char (global%prclib%get_name ()), &
              "': recorded process '", char (config%id), "' (", &
              config%num_id, ")"
     else
        write (msg_buffer, "(5A)") "Process library '", &
              char (global%prclib%get_name ()), &
              "': recorded process '", char (config%id), "'"
     end if
     call msg_message ()
  else
     call msg_fatal ("Recording process '" // char (config%id) &
```

```
            // "': active process library undefined")
      end if
    end subroutine process_configuration_record
```

33.4.2 Unit Tests

Test module, followed by the corresponding implementation module.

⟨process_configurations_ut.f90⟩≡

```
  ⟨File header⟩

  module process_configurations_ut
    use unit_tests
    use process_configurations_uti

  ⟨Standard module head⟩

  ⟨Process configurations: public test⟩

  ⟨Process configurations: public test auxiliary⟩

  contains

  ⟨Process configurations: test driver⟩

  end module process_configurations_ut
```

⟨process_configurations_uti.f90⟩≡

```
  ⟨File header⟩

  module process_configurations_uti

  ⟨Use strings⟩
    use particle_specifiers, only: new_prt_spec
    use prclib_stacks
    use models
    use rt_data

    use process_configurations

  ⟨Standard module head⟩

  ⟨Process configurations: test declarations⟩

  ⟨Process configurations: public test auxiliary⟩

  contains

  ⟨Process configurations: test auxiliary⟩

  ⟨Process configurations: tests⟩

  end module process_configurations_uti
```

API: driver for the unit tests below.

⟨*Process configurations: public test*⟩≡
```
public :: process_configurations_test
```

⟨*Process configurations: test driver*⟩≡
```
subroutine process_configurations_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨Process configurations: execute tests⟩
end subroutine process_configurations_test
```

Minimal setup

The workflow for setting up a minimal process configuration with the test matrix element method.

We wrap this in a public procedure, so we can reuse it in later modules. The procedure prepares a process definition list for two processes (one `prc_test` and one `omega` type) and appends this to the process library stack in the global data set.

The `mode` argument determines which processes to build.

The `procname` argument replaces the predefined procname(s).

This is re-exported by the UT module.

⟨*Process configurations: public test auxiliary*⟩≡
```
public :: prepare_test_library
```

⟨*Process configurations: test auxiliary*⟩≡
```
subroutine prepare_test_library (global, libname, mode, procname)
  type(rt_data_t), intent(inout), target :: global
  type(string_t), intent(in) :: libname
  integer, intent(in) :: mode
  type(string_t), intent(in), dimension(:), optional :: procname
  type(prclib_entry_t), pointer :: lib
  type(string_t) :: prc_name
  type(string_t), dimension(:), allocatable :: prt_in, prt_out
  integer :: n_components
  type(process_configuration_t) :: prc_config

  if (.not. associated (global%prclib_stack%get_first_ptr ())) then
     allocate (lib)
     call lib%init (libname)
     call global%add_prclib (lib)
  end if

  if (btest (mode, 0)) then

     call global%select_model (var_str ("Test"))

     if (present (procname)) then
        prc_name = procname(1)
     else
        prc_name = "prc_config_a"
     end if
     n_components = 1
```

```
   allocate (prt_in (2), prt_out (2))
   prt_in = [var_str ("s"), var_str ("s")]
   prt_out = [var_str ("s"), var_str ("s")]

   call global%set_string (var_str ("$method"),&
        var_str ("unit_test"), is_known = .true.)

   call prc_config%init (prc_name, size (prt_in), n_components, global)
   call prc_config%setup_component (1, &
        new_prt_spec (prt_in), new_prt_spec (prt_out), global%model, global%var_list)
   call prc_config%record (global)

   deallocate (prt_in, prt_out)

end if

if (btest (mode, 1)) then

   call global%select_model (var_str ("QED"))

   if (present (procname)) then
      prc_name = procname(2)
   else
      prc_name = "prc_config_b"
   end if
   n_components = 1
   allocate (prt_in (2), prt_out (2))
   prt_in = [var_str ("e+"), var_str ("e-")]
   prt_out = [var_str ("m+"), var_str ("m-")]

   call global%set_string (var_str ("$method"),&
        var_str ("omega"), is_known = .true.)

   call prc_config%init (prc_name, size (prt_in), n_components, global)
   call prc_config%setup_component (1, &
        new_prt_spec (prt_in), new_prt_spec (prt_out), global%model, global%var_list)
   call prc_config%record (global)

   deallocate (prt_in, prt_out)

end if

if (btest (mode, 2)) then

   call global%select_model (var_str ("Test"))

   if (present (procname)) then
      prc_name = procname(1)
   else
      prc_name = "prc_config_a"
   end if
   n_components = 1
   allocate (prt_in (1), prt_out (2))
   prt_in = [var_str ("s")]
```

```
      prt_out = [var_str ("f"), var_str ("fbar")]

      call global%set_string (var_str ("$method"),&
           var_str ("unit_test"), is_known = .true.)

      call prc_config%init (prc_name, size (prt_in), n_components, global)
      call prc_config%setup_component (1, &
           new_prt_spec (prt_in), new_prt_spec (prt_out), global%model, global%var_list)
      call prc_config%record (global)

      deallocate (prt_in, prt_out)

    end if

  end subroutine prepare_test_library
```

The actual test: the previous procedure with some prelude and postlude. In the global variable list, just before printing we reset the variables where the value may depend on the system and run environment.

⟨*Process configurations: execute tests*⟩≡

```
  call test (process_configurations_1, "process_configurations_1", &
      "test processes", &
      u, results)
```

⟨*Process configurations: test declarations*⟩≡

```
  public :: process_configurations_1
```

⟨*Process configurations: tests*⟩≡

```
  subroutine process_configurations_1 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global

    write (u, "(A)")  "* Test output: process_configurations_1"
    write (u, "(A)")  "*   Purpose: configure test processes"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()
    call global%set_log (var_str ("?omega_openmp"), &
         .false., is_known = .true.)

    write (u, "(A)")  "* Configure processes as prc_test, model Test"
    write (u, "(A)")  "*                      and omega, model QED"
    write (u, *)

    call global%set_int (var_str ("process_num_id"), &
         42, is_known = .true.)
    call prepare_test_library (global, var_str ("prc_config_lib_1"), 3)

    global%os_data%fc = "Fortran-compiler"
    global%os_data%fcflags = "Fortran-flags"

    call global%write_libraries (u)
```

```
    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: process_configurations_1"

  end subroutine process_configurations_1
```

O'MEGA options

Slightly extended example where we pass O'MEGA options to the library. The prepare_test_library contents are spelled out.

⟨*Process configurations: execute tests*⟩+≡

```
  call test (process_configurations_2, "process_configurations_2", &
       "omega options", &
       u, results)
```

⟨*Process configurations: test declarations*⟩+≡

```
  public :: process_configurations_2
```

⟨*Process configurations: tests*⟩+≡

```
  subroutine process_configurations_2 (u)
    integer, intent(in) :: u
    type(rt_data_t), target :: global

    type(string_t) :: libname
    type(prclib_entry_t), pointer :: lib
    type(string_t) :: prc_name
    type(string_t), dimension(:), allocatable :: prt_in, prt_out
    integer :: n_components
    type(process_configuration_t) :: prc_config

    write (u, "(A)")  "* Test output: process_configurations_2"
    write (u, "(A)")  "*   Purpose: configure test processes with options"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()

    write (u, "(A)")  "* Configure processes as omega, model QED"
    write (u, *)

    libname = "prc_config_lib_2"

    allocate (lib)
    call lib%init (libname)
    call global%add_prclib (lib)

    call global%select_model (var_str ("QED"))

    prc_name = "prc_config_c"
```

```
  n_components = 2
  allocate (prt_in (2), prt_out (2))
  prt_in = [var_str ("e+"), var_str ("e-")]
  prt_out = [var_str ("m+"), var_str ("m-")]

  call global%set_string (var_str ("$method"),&
       var_str ("omega"), is_known = .true.)
  call global%set_log (var_str ("?omega_openmp"), &
       .false., is_known = .true.)

  call prc_config%init (prc_name, size (prt_in), n_components, global)

  call global%set_log (var_str ("?report_progress"), &
       .true., is_known = .true.)
  call prc_config%setup_component (1, &
       new_prt_spec (prt_in), new_prt_spec (prt_out), global%model, global%var_list)

  call global%set_log (var_str ("?report_progress"), &
       .false., is_known = .true.)
  call global%set_log (var_str ("?omega_openmp"), &
       .true., is_known = .true.)
  call global%set_string (var_str ("$restrictions"),&
       var_str ("3+4~A"), is_known = .true.)
  call global%set_string (var_str ("$omega_flags"), &
       var_str ("-fusion:progress_file omega_prc_config.log"), &
       is_known = .true.)
  call prc_config%setup_component (2, &
       new_prt_spec (prt_in), new_prt_spec (prt_out), global%model, global%var_list)

  call prc_config%record (global)

  deallocate (prt_in, prt_out)

  global%os_data%fc = "Fortran-compiler"
  global%os_data%fcflags = "Fortran-flags"

  call global%write_vars (u, [ &
       var_str ("$model_name"), &
       var_str ("$method"), &
       var_str ("?report_progress"), &
       var_str ("$restrictions"), &
       var_str ("$omega_flags")])
  write (u, "(A)")
  call global%write_libraries (u)

  call global%final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: process_configurations_2"

end subroutine process_configurations_2
```

33.5 Compilation

This module manages compilation and loading of of process libraries. It is needed as a separate module because integration depends on it.

⟨compilations.f90⟩≡

```
⟨File header⟩

module compilations

⟨Use strings⟩
  use io_units
  use system_defs, only: TAB
  use diagnostics
  use os_interface
  use variables, only: var_list_t
  use model_data
  use process_libraries
  use prclib_stacks
  use rt_data

⟨Standard module head⟩

⟨Compilations: public⟩

⟨Compilations: types⟩

contains

⟨Compilations: procedures⟩

end module compilations
```

33.5.1 The data type

The compilation item handles the compilation and loading of a single process library.

⟨*Compilations: public*⟩≡

```
public :: compilation_item_t
```

⟨*Compilations: types*⟩≡

```
type :: compilation_item_t
   private
   type(string_t) :: libname
   type(string_t) :: static_external_tag
   type(process_library_t), pointer :: lib => null ()
   logical :: recompile_library = .false.
 contains
 ⟨Compilations: compilation item: TBP⟩
end type compilation_item_t
```

Initialize:

⟨*Compilations: compilation item: TBP*⟩≡

```
procedure :: init => compilation_item_init
```

⟨*Compilations: procedures*⟩≡

```
subroutine compilation_item_init (comp, libname, stack, var_list)
  class(compilation_item_t), intent(out) :: comp
  type(string_t), intent(in) :: libname
  type(prclib_stack_t), intent(inout) :: stack
  type(var_list_t), intent(in) :: var_list
  comp%libname = libname
  comp%lib => stack%get_library_ptr (comp%libname)
  if (.not. associated (comp%lib)) then
     call msg_fatal ("Process library '" // char (comp%libname) &
          // "' has not been declared.")
  end if
  comp%recompile_library = &
       var_list%get_lval (var_str ("?recompile_library"))
end subroutine compilation_item_init
```

Compile the current library. The `force` flag has the effect that we first delete any previous files, as far as accessible by the current makefile. It also guarantees that previous files not accessible by a makefile will be overwritten.

⟨*Compilations: compilation item: TBP*⟩+≡

```
procedure :: compile => compilation_item_compile
```

⟨*Compilations: procedures*⟩+≡

```
subroutine compilation_item_compile (comp, model, os_data, force, recompile)
  class(compilation_item_t), intent(inout) :: comp
  class(model_data_t), intent(in), target :: model
  type(os_data_t), intent(in) :: os_data
  logical, intent(in) :: force, recompile
  if (associated (comp%lib)) then
     call msg_message ("Process library '" &
          // char (comp%libname) // "': compiling ...")
     call comp%lib%configure (os_data)
     if (signal_is_pending ())  return
     call comp%lib%compute_md5sum (model)
     call comp%lib%write_makefile (os_data, force)
     if (signal_is_pending ())  return
     if (force) then
        call comp%lib%clean (os_data, distclean = .false.)
        if (signal_is_pending ())  return
     end if
     call comp%lib%write_driver (force)
     if (signal_is_pending ())  return
     if (recompile) then
        call comp%lib%load (os_data, keep_old_source = .true.)
        if (signal_is_pending ())  return
     end if
     call comp%lib%update_status (os_data)
  end if
end subroutine compilation_item_compile
```

Load the current library, just after compiling it.

⟨*Compilations: compilation item: TBP*⟩+≡

```
procedure :: load => compilation_item_load
```

⟨*Compilations: procedures*⟩+≡

```
subroutine compilation_item_load (comp, os_data)
  class(compilation_item_t), intent(inout) :: comp
  type(os_data_t), intent(in) :: os_data
  if (associated (comp%lib)) then
     call comp%lib%load (os_data)
  end if
end subroutine compilation_item_load
```

Message as a separate call:

⟨*Compilations: compilation item: TBP*⟩+≡

```
procedure :: success => compilation_item_success
```

⟨*Compilations: procedures*⟩+≡

```
subroutine compilation_item_success (comp)
  class(compilation_item_t), intent(in) :: comp
  if (associated (comp%lib)) then
     call msg_message ("Process library '" // char (comp%libname) &
          // "': ... success.")
  else
     call msg_fatal ("Process library '" // char (comp%libname) &
          // "': ... failure.")
  end if
end subroutine compilation_item_success
```

33.5.2 API for library compilation and loading

This is a shorthand for compiling and loading a single library. The `compilation_item` object is used only internally.

The `global` data set may actually be local to the caller. The compilation affects the library specified by its name if it is on the stack, but it does not reset the currently selected library.

⟨*Compilations: public*⟩+≡

```
public :: compile_library
```

⟨*Compilations: procedures*⟩+≡

```
subroutine compile_library (libname, global)
  type(string_t), intent(in) :: libname
  type(rt_data_t), intent(inout), target :: global
  type(compilation_item_t) :: comp
  logical :: force, recompile
  force = &
       global%var_list%get_lval (var_str ("?rebuild_library"))
  recompile = &
       global%var_list%get_lval (var_str ("?recompile_library"))
  if (associated (global%model)) then
     call comp%init (libname, global%prclib_stack, global%var_list)
     call comp%compile (global%model, global%os_data, force, recompile)
     if (signal_is_pending ())  return
     call comp%load (global%os_data)
     if (signal_is_pending ())  return
  else
```

```
      call msg_fatal ("Process library compilation: " &
           // " model is undefined.")
  end if
  call comp%success ()
end subroutine compile_library
```

33.5.3 Compiling static executable

This object handles the creation of a static executable which should contain a set of static process libraries.

⟨*Compilations: public*⟩+≡

```
public :: compilation_t
```

⟨*Compilations: types*⟩+≡

```
type :: compilation_t
   private
   type(string_t) :: exe_name
   type(string_t), dimension(:), allocatable :: lib_name
 contains
 ⟨Compilations: compilation: TBP⟩
end type compilation_t
```

Output.

⟨*Compilations: compilation: TBP*⟩≡

```
procedure :: write => compilation_write
```

⟨*Compilations: procedures*⟩+≡

```
subroutine compilation_write (object, unit)
  class(compilation_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Compilation object:"
  write (u, "(3x,3A)")  "executable        = '", &
       char (object%exe_name), "'"
  write (u, "(3x,A)", advance="no")  "process libraries ="
  do i = 1, size (object%lib_name)
     write (u, "(1x,3A)", advance="no")  "'", char (object%lib_name(i)), "'"
  end do
  write (u, *)
end subroutine compilation_write
```

Initialize: we know the names of the executable and of the libraries.

⟨*Compilations: compilation: TBP*⟩+≡

```
procedure :: init => compilation_init
```

⟨*Compilations: procedures*⟩+≡

```
subroutine compilation_init (compilation, exe_name, lib_name)
  class(compilation_t), intent(out) :: compilation
  type(string_t), intent(in) :: exe_name
  type(string_t), dimension(:), intent(in) :: lib_name
  compilation%exe_name = exe_name
```

```
    allocate (compilation%lib_name (size (lib_name)))
    compilation%lib_name = lib_name
  end subroutine compilation_init
```

Write the dispatcher subroutine for the compiled libraries. Also write a subroutine which returns the names of the compiled libraries.

⟨*Compilations: compilation: TBP*⟩+≡

```
  procedure :: write_dispatcher => compilation_write_dispatcher
```

⟨*Compilations: procedures*⟩+≡

```
  subroutine compilation_write_dispatcher (compilation)
    class(compilation_t), intent(in) :: compilation
    type(string_t) :: file
    integer :: u, i
    file = compilation%exe_name // "_prclib_dispatcher.f90"
    call msg_message ("Static executable '" // char (compilation%exe_name) &
         // "': writing library dispatcher")
    u = free_unit ()
    open (u, file = char (file), status="replace", action="write")
    write (u, "(3A)")  "! Whizard: process libraries for executable '", &
         char (compilation%exe_name), "'"
    write (u, "(A)")  "! Automatically generated file, do not edit"
    write (u, "(A)")  "subroutine dispatch_prclib_static " // &
         "(driver, basename, modellibs_ldflags)"
    write (u, "(A)")  "  use iso_varying_string, string_t => varying_string"
    write (u, "(A)")  "  use prclib_interfaces"
    do i = 1, size (compilation%lib_name)
       associate (lib_name => compilation%lib_name(i))
         write (u, "(A)")  "  use " // char (lib_name) // "_driver"
       end associate
    end do
    write (u, "(A)")  "  implicit none"
    write (u, "(A)")  "  class(prclib_driver_t), intent(inout), allocatable &
         &:: driver"
    write (u, "(A)")  "  type(string_t), intent(in) :: basename"
    write (u, "(A)")  "  logical, intent(in), optional :: " // &
         "modellibs_ldflags"
    write (u, "(A)")  "  select case (char (basename))"
    do i = 1, size (compilation%lib_name)
       associate (lib_name => compilation%lib_name(i))
         write (u, "(3A)")  "  case ('", char (lib_name), "')"
         write (u, "(3A)")  "     allocate (", char (lib_name), "_driver_t &
              &:: driver)"
       end associate
    end do
    write (u, "(A)")  "  end select"
    write (u, "(A)")  "end subroutine dispatch_prclib_static"
    write (u, *)
    write (u, "(A)")  "subroutine get_prclib_static (libname)"
    write (u, "(A)")  "  use iso_varying_string, string_t => varying_string"
    write (u, "(A)")  "  implicit none"
    write (u, "(A)")  "  type(string_t), dimension(:), intent(inout), &
         &allocatable :: libname"
    write (u, "(A,I0,A)")  "  allocate (libname (", &
```

```
      size (compilation%lib_name), "))"
  do i = 1, size (compilation%lib_name)
     associate (lib_name => compilation%lib_name(i))
       write (u, "(A,I0,A,A,A)")  "  libname(", i, ") = '", &
            char (lib_name), "'"
     end associate
  end do
  write (u, "(A)")  "end subroutine get_prclib_static"
  close (u)
end subroutine compilation_write_dispatcher
```

Write the Makefile subroutine for the compiled libraries.

⟨*Compilations: compilation: TBP*⟩+≡

```
procedure :: write_makefile => compilation_write_makefile
```

⟨*Compilations: procedures*⟩+≡

```
subroutine compilation_write_makefile (compilation, os_data, ext_libtag)
  class(compilation_t), intent(in) :: compilation
  type(os_data_t), intent(in) :: os_data
  type(string_t), intent(in), optional :: ext_libtag
  type(string_t) :: file, ext_tag
  integer :: u, i
  if (present (ext_libtag)) then
     ext_tag = ext_libtag
  else
     ext_tag = ""
  end if
  file = compilation%exe_name // ".makefile"
  call msg_message ("Static executable '" // char (compilation%exe_name) &
       // "': writing makefile")
  u = free_unit ()
  open (u, file = char (file), status="replace", action="write")
  write (u, "(3A)")  "# WHIZARD: Makefile for executable '", &
       char (compilation%exe_name), "'"
  write (u, "(A)")  "# Automatically generated file, do not edit"
  write (u, "(A)") ""
  write (u, "(A)") "# Executable name"
  write (u, "(A)") "EXE = " // char (compilation%exe_name)
  write (u, "(A)") ""
  write (u, "(A)") "# Compiler"
  write (u, "(A)") "FC = " // char (os_data%fc)
  write (u, "(A)") ""
  write (u, "(A)") "# Included libraries"
  write (u, "(A)") "FCINCL = " // char (os_data%whizard_includes)
  write (u, "(A)") ""
  write (u, "(A)") "# Compiler flags"
  write (u, "(A)") "FCFLAGS = " // char (os_data%fcflags)
  write (u, "(A)") "LDFLAGS = " // char (os_data%ldflags)
  write (u, "(A)") "LDFLAGS_STATIC = " // char (os_data%ldflags_static)
  write (u, "(A)") "LDFLAGS_HEPMC = " // char (os_data%ldflags_hepmc)
  write (u, "(A)") "LDFLAGS_LCIO = " // char (os_data%ldflags_lcio)
  write (u, "(A)") "LDFLAGS_HOPPET = " // char (os_data%ldflags_hoppet)
  write (u, "(A)") "LDFLAGS_LOOPTOOLS = " // char (os_data%ldflags_looptools)
  write (u, "(A)") "LDWHIZARD = " // char (os_data%whizard_ldflags)
```

```
  write (u, "(A)") ""
  write (u, "(A)") "# Libtool"
  write (u, "(A)") "LIBTOOL = " // char (os_data%whizard_libtool)
  write (u, "(A)") "FCOMPILE = $(LIBTOOL) --tag=FC --mode=compile"
  write (u, "(A)") "LINK = $(LIBTOOL) --tag=FC --mode=link"
  write (u, "(A)") ""
  write (u, "(A)") "# Compile commands (default)"
  write (u, "(A)") "LTFCOMPILE = $(FCOMPILE) $(FC) -c $(FCINCL) $(FCFLAGS)"
  write (u, "(A)") ""
  write (u, "(A)") "# Default target"
  write (u, "(A)") "all: link"
  write (u, "(A)") ""
  write (u, "(A)") "# Libraries"
  do i = 1, size (compilation%lib_name)
     associate (lib_name => compilation%lib_name(i))
       write (u, "(A)") "LIBRARIES += " // char (lib_name) // ".la"
       write (u, "(A)") char (lib_name) // ".la:"
       write (u, "(A)") TAB // "$(MAKE) -f " // char (lib_name) // ".makefile"
     end associate
  end do
  write (u, "(A)") ""
  write (u, "(A)") "# Library dispatcher"
  write (u, "(A)") "DISP = $(EXE)_prclib_dispatcher"
  write (u, "(A)") "$(DISP).lo: $(DISP).f90 $(LIBRARIES)"
  write (u, "(A)") TAB // "$(LTFCOMPILE) $<"
  write (u, "(A)") ""
  write (u, "(A)") "# Executable"
  write (u, "(A)") "$(EXE): $(DISP).lo $(LIBRARIES)"
  write (u, "(A)") TAB // "$(LINK) $(FC) -static-libtool-libs $(FCFLAGS) \"
  write (u, "(A)") TAB // "   $(LDWHIZARD) $(LDFLAGS) \"
  write (u, "(A)") TAB // "   -o $(EXE) $^ \"
  write (u, "(A)") TAB // "   $(LDFLAGS_HEPMC) $(LDFLAGS_LCIO) $(LDFLAGS_HOPPET) \"
  write (u, "(A)") TAB // "   $(LDFLAGS_LOOPTOOLS) $(LDFLAGS_STATIC)" // char (ext_tag)
  write (u, "(A)") ""
  write (u, "(A)") "# Main targets"
  write (u, "(A)") "link: compile $(EXE)"
  write (u, "(A)") "compile: $(LIBRARIES) $(DISP).lo"
  write (u, "(A)") ".PHONY: link compile"
  write (u, "(A)") ""
  write (u, "(A)") "# Cleanup targets"
  write (u, "(A)") "clean-exe:"
  write (u, "(A)") TAB // "rm -f $(EXE)"
  write (u, "(A)") "clean-objects:"
  write (u, "(A)") TAB // "rm -f $(DISP).lo"
  write (u, "(A)") "clean-source:"
  write (u, "(A)") TAB // "rm -f $(DISP).f90"
  write (u, "(A)") "clean-makefile:"
  write (u, "(A)") TAB // "rm -f $(EXE).makefile"
  write (u, "(A)") ""
  write (u, "(A)") "clean: clean-exe clean-objects clean-source"
  write (u, "(A)") "distclean: clean clean-makefile"
  write (u, "(A)") ".PHONY: clean distclean"
  close (u)
end subroutine compilation_write_makefile
```

Compile the dispatcher source code.

⟨*Compilations: compilation: TBP*⟩+≡
```
procedure :: make_compile => compilation_make_compile
```

⟨*Compilations: procedures*⟩+≡
```
subroutine compilation_make_compile (compilation, os_data)
  class(compilation_t), intent(in) :: compilation
  type(os_data_t), intent(in) :: os_data
  call os_system_call ("make compile " // os_data%makeflags &
       // " -f " // compilation%exe_name // ".makefile")
end subroutine compilation_make_compile
```

Link the dispatcher together with all matrix-element code and the `WHIZARD` and `O'MEGA` main libraries, to generate a static executable.

⟨*Compilations: compilation: TBP*⟩+≡
```
procedure :: make_link => compilation_make_link
```

⟨*Compilations: procedures*⟩+≡
```
subroutine compilation_make_link (compilation, os_data)
  class(compilation_t), intent(in) :: compilation
  type(os_data_t), intent(in) :: os_data
  call os_system_call ("make link " // os_data%makeflags &
       // " -f " // compilation%exe_name // ".makefile")
end subroutine compilation_make_link
```

Cleanup.

⟨*Compilations: compilation: TBP*⟩+≡
```
procedure :: make_clean_exe => compilation_make_clean_exe
```

⟨*Compilations: procedures*⟩+≡
```
subroutine compilation_make_clean_exe (compilation, os_data)
  class(compilation_t), intent(in) :: compilation
  type(os_data_t), intent(in) :: os_data
  call os_system_call ("make clean-exe " // os_data%makeflags &
       // " -f " // compilation%exe_name // ".makefile")
end subroutine compilation_make_clean_exe
```

33.5.4 API for executable compilation

This is a shorthand for compiling and loading an executable, including the enclosed libraries. The `compilation` object is used only internally.

The `global` data set may actually be local to the caller. The compilation affects the library specified by its name if it is on the stack, but it does not reset the currently selected library.

⟨*Compilations: public*⟩+≡
```
public :: compile_executable
```

⟨*Compilations: procedures*⟩+≡

```
subroutine compile_executable (exename, libname, global)
  type(string_t), intent(in) :: exename
  type(string_t), dimension(:), intent(in) :: libname
  type(rt_data_t), intent(inout), target :: global
  type(compilation_t) :: compilation
  type(compilation_item_t) :: item
  type(string_t) :: ext_libtag
  logical :: force, recompile
  integer :: i
  ext_libtag = ""
  force = &
       global%var_list%get_lval (var_str ("?rebuild_library"))
  recompile = &
       global%var_list%get_lval (var_str ("?recompile_library"))
  call compilation%init (exename, [libname])
  if (signal_is_pending ())  return
  call compilation%write_dispatcher ()
  if (signal_is_pending ())  return
  do i = 1, size (libname)
     call item%init (libname(i), global%prclib_stack, global%var_list)
     call item%compile (global%model, global%os_data, &
          force=force, recompile=recompile)
     ext_libtag = "" // item%lib%get_static_modelname (global%os_data)
     if (signal_is_pending ())  return
     call item%success ()
  end do
  call compilation%write_makefile (global%os_data, ext_libtag)
  if (signal_is_pending ())  return
  call compilation%make_compile (global%os_data)
  if (signal_is_pending ())  return
  call compilation%make_link (global%os_data)
end subroutine compile_executable
```

33.5.5 Unit Tests

Test module, followed by the stand-alone unit-test procedures.

⟨`compilations_ut.f90`⟩≡

```
⟨File header⟩

module compilations_ut
  use unit_tests
  use compilations_uti

⟨Standard module head⟩

⟨Compilations: public test⟩

contains

⟨Compilations: test driver⟩
```

```
  end module compilations_ut
```

⟨compilations_uti.f90⟩≡

```
  ⟨File header⟩

  module compilations_uti

  ⟨Use strings⟩
    use io_units
    use models
    use rt_data
    use process_configurations_ut, only: prepare_test_library

    use compilations

  ⟨Standard module head⟩

  ⟨Compilations: test declarations⟩

  contains

  ⟨Compilations: tests⟩

  end module compilations_uti
```

API: driver for the unit tests below.

⟨*Compilations: public test*⟩≡

```
    public :: compilations_test
```

⟨*Compilations: test driver*⟩≡

```
    subroutine compilations_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Compilations: execute tests⟩
  end subroutine compilations_test
```

Intrinsic Matrix Element

Compile an intrinsic test matrix element (`prc_test` type).

Note: In this and the following test, we reset the Fortran compiler and flag variables immediately before they are printed, so the test is portable.

⟨*Compilations: execute tests*⟩≡

```
    call test (compilations_1, "compilations_1", &
         "intrinsic test processes", &
         u, results)
```

⟨*Compilations: test declarations*⟩≡

```
    public :: compilations_1
```

⟨*Compilations: tests*⟩≡

```
    subroutine compilations_1 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname
      type(rt_data_t), target :: global
```

```
    write (u, "(A)")  "* Test output: compilations_1"
    write (u, "(A)")  "*   Purpose: configure and compile test process"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()

    libname = "compilation_1"
    procname = "prc_comp_1"
    call prepare_test_library (global, libname, 1, [procname])

    call compile_library (libname, global)

    call global%write_libraries (u)

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: compilations_1"

  end subroutine compilations_1
```

External Matrix Element

Compile an external test matrix element (omega type)

⟨*Compilations: execute tests*⟩+≡

```
  call test (compilations_2, "compilations_2", &
      "external process (omega)", &
      u, results)
```

⟨*Compilations: test declarations*⟩+≡

```
  public :: compilations_2
```

⟨*Compilations: tests*⟩+≡

```
  subroutine compilations_2 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname
    type(rt_data_t), target :: global

    write (u, "(A)")  "* Test output: compilations_2"
    write (u, "(A)")  "*   Purpose: configure and compile test process"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()
    call global%set_log (var_str ("?omega_openmp"), &
        .false., is_known = .true.)

    libname = "compilation_2"
    procname = "prc_comp_2"
```

```
    call prepare_test_library (global, libname, 2, [procname,procname])

    call compile_library (libname, global)

    call global%write_libraries (u, libpath = .false.)

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: compilations_2"

  end subroutine compilations_2
```

External Matrix Element

Compile an external test matrix element (omega type) and create driver files for a static executable.

⟨*Compilations: execute tests*⟩+≡

```
  call test (compilations_3, "compilations_3", &
       "static executable: driver", &
       u, results)
```

⟨*Compilations: test declarations*⟩+≡

```
  public :: compilations_3
```

⟨*Compilations: tests*⟩+≡

```
  subroutine compilations_3 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname, exename
    type(rt_data_t), target :: global
    type(compilation_t) :: compilation
    integer :: u_file
    character(80) :: buffer

    write (u, "(A)")  "* Test output: compilations_3"
    write (u, "(A)")  "*   Purpose: make static executable"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize library"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()
    call global%set_log (var_str ("?omega_openmp"), &
         .false., is_known = .true.)

    libname = "compilations_3_lib"
    procname = "prc_comp_3"
    exename = "compilations_3"

    call prepare_test_library (global, libname, 2, [procname,procname])
```

```
    call compilation%init (exename, [libname])
    call compilation%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Write dispatcher"
    write (u, "(A)")

    call compilation%write_dispatcher ()

    u_file = free_unit ()
    open (u_file, file = char (exename) // "_prclib_dispatcher.f90", &
         status = "old", action = "read")
    do
       read (u_file, "(A)", end = 1)  buffer
       write (u, "(A)")  trim (buffer)
    end do
1   close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Write Makefile"
    write (u, "(A)")

    associate (os_data => global%os_data)
      os_data%fc = "fortran-compiler"
      os_data%whizard_includes = "my-includes"
      os_data%fcflags = "my-fcflags"
      os_data%ldflags = "my-ldflags"
      os_data%ldflags_static = "my-ldflags-static"
      os_data%ldflags_hepmc = "my-ldflags-hepmc"
      os_data%ldflags_lcio = "my-ldflags-lcio"
      os_data%ldflags_hoppet = "my-ldflags-hoppet"
      os_data%ldflags_looptools = "my-ldflags-looptools"
      os_data%whizard_ldflags = "my-ldwhizard"
      os_data%whizard_libtool = "my-libtool"
    end associate

    call compilation%write_makefile (global%os_data)

    open (u_file, file = char (exename) // ".makefile", &
         status = "old", action = "read")
    do
       read (u_file, "(A)", end = 2)  buffer
       write (u, "(A)")  trim (buffer)
    end do
2   close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
```

```
    write (u, "(A)")  "* Test output end: compilations_3"

  end subroutine compilations_3
```

33.5.6 Test static build

The tests for building a static executable are separate, since they should be skipped if the WHIZARD build itself has static libraries disabled.

⟨*Compilations: public test*⟩+≡

```
  public :: compilations_static_test
```

⟨*Compilations: test driver*⟩+≡

```
  subroutine compilations_static_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨Compilations: static tests⟩
 end subroutine compilations_static_test
```

External Matrix Element

Compile an external test matrix element (omega type) and incorporate this in a new static WHIZARD executable.

⟨*Compilations: static tests*⟩≡

```
  call test (compilations_static_1, "compilations_static_1", &
       "static executable: compilation", &
       u, results)
```

⟨*Compilations: test declarations*⟩+≡

```
  public :: compilations_static_1
```

⟨*Compilations: tests*⟩+≡

```
  subroutine compilations_static_1 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname, exename
    type(rt_data_t), target :: global
    type(compilation_item_t) :: item
    type(compilation_t) :: compilation
    logical :: exist

    write (u, "(A)")  "* Test output: compilations_static_1"
    write (u, "(A)")  "*   Purpose: make static executable"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize library"

    call syntax_model_file_init ()

    call global%global_init ()
    call global%set_log (var_str ("?omega_openmp"), &
         .false., is_known = .true.)

    libname = "compilations_static_1_lib"
```

```
procname = "prc_comp_stat_1"
exename = "compilations_static_1"

call prepare_test_library (global, libname, 2, [procname,procname])

call compilation%init (exename, [libname])

write (u, "(A)")
write (u, "(A)")  "* Write dispatcher"

call compilation%write_dispatcher ()

write (u, "(A)")
write (u, "(A)")  "* Write Makefile"

call compilation%write_makefile (global%os_data)

write (u, "(A)")
write (u, "(A)")  "* Build libraries"

call item%init (libname, global%prclib_stack, global%var_list)
call item%compile &
     (global%model, global%os_data, force=.true., recompile=.false.)
call item%success ()

write (u, "(A)")
write (u, "(A)")  "* Check executable (should be absent)"
write (u, "(A)")

call compilation%make_clean_exe (global%os_data)
inquire (file = char (exename), exist = exist)
write (u, "(A,A,L1)")  char (exename), " exists = ", exist

write (u, "(A)")
write (u, "(A)")  "* Build executable"
write (u, "(A)")

call compilation%make_compile (global%os_data)
call compilation%make_link (global%os_data)

write (u, "(A)")  "* Check executable (should be present)"
write (u, "(A)")

inquire (file = char (exename), exist = exist)
write (u, "(A,A,L1)")  char (exename), " exists = ", exist

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call compilation%make_clean_exe (global%os_data)

call global%final ()
call syntax_model_file_final ()
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: compilations_static_1"

  end subroutine compilations_static_1
```

External Matrix Element

Compile an external test matrix element (omega type) and incorporate this in a new static WHIZARD executable. In this version, we use the wrapper compile_executable procedure.

⟨*Compilations: static tests*⟩+≡

```
  call test (compilations_static_2, "compilations_static_2", &
       "static executable: shortcut", &
       u, results)
```

⟨*Compilations: test declarations*⟩+≡

```
  public :: compilations_static_2
```

⟨*Compilations: tests*⟩+≡

```
  subroutine compilations_static_2 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname, exename
    type(rt_data_t), target :: global
    logical :: exist
    integer :: u_file

    write (u, "(A)")  "* Test output: compilations_static_2"
    write (u, "(A)")  "*   Purpose: make static executable"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize library and compile"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()
    call global%set_log (var_str ("?omega_openmp"), &
         .false., is_known = .true.)

    libname = "compilations_static_2_lib"
    procname = "prc_comp_stat_2"
    exename = "compilations_static_2"

    call prepare_test_library (global, libname, 2, [procname,procname])

    call compile_executable (exename, [libname], global)

    write (u, "(A)")  "* Check executable (should be present)"
    write (u, "(A)")

    inquire (file = char (exename), exist = exist)
    write (u, "(A,A,L1)")  char (exename), " exists = ", exist
```

```
  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  u_file = free_unit ()
  open (u_file, file = char (exename), status = "old", action = "write")
  close (u_file, status = "delete")

  call global%final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: compilations_static_2"

end subroutine compilations_static_2
```

33.6 Integration

This module manages phase space setup, matrix-element evaluation and integration, as far as it is not done by lower-level routines, in particular in the `processes` module.

⟨`integrations.f90`⟩≡

```
⟨File header⟩

module integrations

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use diagnostics
  use os_interface
  use cputime
  use sm_qcd
  use physics_defs
  use model_data
  use pdg_arrays
  use variables, only: var_list_t
  use eval_trees
  use sf_mappings
  use sf_base
  use phs_base
  use mappings
  use phs_forests, only: phs_parameters_t
  use rng_base
  use mci_base
  use process_libraries
  use prc_core
  use process_config, only: COMP_MASTER, COMP_REAL_FIN, &
       COMP_MISMATCH, COMP_PDF, COMP_REAL, COMP_SUB, COMP_VIRT, &
       COMP_REAL_SING
  use process
  use instances
  use process_stacks
  use models
  use iterations
  use rt_data

  use dispatch_rng, only: dispatch_rng_factory
  use dispatch_me_methods, only: dispatch_core
  use dispatch_beams, only: dispatch_qcd, sf_prop_t, dispatch_sf_config
  use dispatch_phase_space, only: dispatch_sf_channels
  use dispatch_phase_space, only: dispatch_phs
  use dispatch_mci, only: dispatch_mci_s

  use compilations, only: compile_library

  use dispatch_fks, only: dispatch_fks_s
  use blha_olp_interfaces
  use nlo_data
```

⟨*Standard module head*⟩

⟨*Integrations: public*⟩

⟨*Integrations: types*⟩

```
contains
```

⟨*Integrations: procedures*⟩

```
end module integrations
```

33.6.1 The integration type

This type holds all relevant data, the integration methods operates on this. In contrast to the `simulation_t` introduced later, the `integration_t` applies to a single process.

⟨*Integrations: public*⟩≡

```
public :: integration_t
```

⟨*Integrations: types*⟩≡

```
type :: integration_t
  private
  type(string_t) :: process_id
  type(string_t) :: run_id
  type(process_t), pointer :: process => null ()
  type(var_list_t), pointer :: model_vars => null ()
  type(qcd_t) :: qcd
  logical :: rebuild_phs = .false.
  logical :: ignore_phs_mismatch = .false.
  logical :: phs_only = .false.
  logical :: process_has_me = .true.
  integer :: n_calls_test = 0
  logical :: vis_history = .true.
  type(string_t) :: history_filename
  type(string_t) :: log_filename
  logical :: combined_integration = .false.
  type(iteration_multipliers_t) :: iteration_multipliers
  type(nlo_settings_t) :: nlo_settings
 contains
 ⟨Integrations: integration: TBP⟩
end type integration_t
```

33.6.2 Initialization

Initialization, first part: Create a process entry. Push it on the stack if the `global` environment is supplied.

⟨*Integrations: integration: TBP*⟩≡

```
procedure :: create_process => integration_create_process
```

⟨*Integrations: procedures*⟩≡

```
subroutine integration_create_process (intg, process_id, global)
  class(integration_t), intent(out) :: intg
  type(rt_data_t), intent(inout), optional, target :: global
  type(string_t), intent(in) :: process_id
  type(process_entry_t), pointer :: process_entry
  call msg_debug (D_CORE, "integration_create_process")
  intg%process_id = process_id
  if (present (global)) then
     allocate (process_entry)
     intg%process => process_entry%process_t
     call global%process_stack%push (process_entry)
  else
     allocate (process_t :: intg%process)
  end if
  intg%model_vars => null ()
end subroutine integration_create_process
```

Initialization, second part: Initialize the process object, using the local environment. We allocate a RNG factory and a QCD object. We also fetch a pointer to the model that the process uses. The process initializer will create a snapshot of that model.

This procedure does not modify the `local` stack directly. The intent(inout) attribute for the `local` data set is due to the random generator seed which may be incremented during initialization.

NOTE: Changes to model parameters within the current context are respected only if the process model coincides with the current model. This is the usual case. If not, we read the model from the global model library, which has default parameters. To become more flexible, we should implement a local model library which records local changes to currently inactive models.

⟨*Integrations: integration: TBP*⟩+≡

```
procedure :: init_process => integration_init_process
```

⟨*Integrations: procedures*⟩+≡

```
subroutine integration_init_process (intg, local)
  class(integration_t), intent(inout) :: intg
  type(rt_data_t), intent(inout), target :: local
  type(string_t) :: model_name
  type(model_t), pointer :: model
  class(model_data_t), pointer :: model_instance
  class(rng_factory_t), allocatable :: rng_factory
  call msg_debug (D_CORE, "integration_init_process")
  if (.not. local%prclib%contains (intg%process_id)) then
     call msg_fatal ("Process '" // char (intg%process_id) // "' not found" &
          // " in library '" // char (local%prclib%get_name ()) // "'")
     return
  end if
  intg%run_id = &
       local%var_list%get_sval (var_str ("$run_id"))
  call dispatch_qcd (intg%qcd, local%get_var_list_ptr (), local%os_data)
  call dispatch_rng_factory (rng_factory, local%var_list)
  model_name = local%prclib%get_model_name (intg%process_id)
  if (local%get_sval (var_str ("$model_name")) == model_name) then
```

```
      model => local%model
    else
      model => local%model_list%get_model_ptr (model_name)
    end if
    allocate (model_t :: model_instance)
    select type (model_instance)
    type is (model_t)
      call model_instance%init_instance (model)
      intg%model_vars => model_instance%get_var_list_ptr ()
    end select
    call intg%process%init (intg%process_id, intg%run_id, &
        local%prclib, &
        local%os_data, intg%qcd, rng_factory, model_instance)
  end subroutine integration_init_process
```

Initialization, third part: complete process configuration.

⟨*Integrations: integration: TBP*⟩+≡

```
  procedure :: setup_process => integration_setup_process
```

⟨*Integrations: procedures*⟩+≡

```
  subroutine integration_setup_process (intg, local, verbose)
    class(integration_t), intent(inout) :: intg
    type(rt_data_t), intent(inout), target :: local
    logical, intent(in), optional :: verbose
    type(var_list_t), pointer :: var_list
    class(prc_core_t), allocatable :: core_template
    class(prc_core_t), pointer :: core => null ()
    class(phs_config_t), allocatable :: phs_config_template
    type(phs_parameters_t) :: phs_par
    type(mapping_defaults_t) :: mapping_defs
    class(mci_t), allocatable :: mci_template
    integer :: n_components, i_component
    type(process_component_def_t), pointer :: config
    type(helicity_selection_t) :: helicity_selection
    logical :: use_color_factors
    type(sf_config_t), dimension(:), allocatable :: sf_config
    type(sf_prop_t) :: sf_prop
    type(sf_channel_t), dimension(:), allocatable :: sf_channel
    type(phs_channel_collection_t) :: phs_channel_collection
    logical :: sf_trace
    logical :: verb
    type(blha_template_t) :: blha_template
    type(fks_template_t) :: fks_template
    type(string_t) :: sf_string
    class(phs_config_t), allocatable :: phs_config_template_other
    integer :: i_real = 0
    integer :: i_core
    integer :: i_core_born, i_core_real
    logical :: first_real_component, use_real_partition
    real(default) :: real_partition_scale
    verb = .true.; if (present (verbose))  verb = verbose
    call intg%process%set_var_list (local%get_var_list_ptr ())
    var_list => intg%process%get_var_list_ptr ()
```

```
call setup_phase_space ()

intg%n_calls_test = &
     var_list%get_ival (var_str ("n_calls_test"))

call setup_log_and_history ()

call dispatch_mci_s (mci_template, local%get_var_list_ptr (), intg%process_id, &
     intg%process%is_nlo_calculation ())

call display_init_message (verb)

n_components = intg%process%get_n_components ()

intg%combined_integration = var_list%get_lval &
   (var_str ('?combined_nlo_integration')) .and. &
    intg%process%is_nlo_calculation ()

helicity_selection = local%get_helicity_selection ()
use_color_factors = var_list%get_lval &
     (var_str ("?read_color_factors"))
do i_component = 1, n_components
   config => intg%process%get_component_def_ptr (i_component)
   call intg%process%core_manager_register &
      (config%get_nlo_type (), i_component, config%get_def_type_string ())
end do
call intg%process%allocate_cm_arrays (n_components)
do i_core = 1, intg%process%get_n_cores ()
   i_component = intg%process%get_core_manager_index (i_core)
   config => intg%process%get_component_def_ptr (i_component)
   call dispatch_core (core_template, config%get_core_def_ptr (), &
        intg%process%get_model_ptr (), &
        helicity_selection, intg%qcd, &
        use_color_factors)
   call intg%process%allocate_core (i_core, core_template)
   deallocate (core_template)
end do

call intg%process%init_cores ()
first_real_component = .true.
do i_component = 1, n_components
   config => intg%process%get_component_def_ptr (i_component)
   core => intg%process%get_core_nlo_type ( &
        intg%process%get_md5sum_constants (i_component, &
        config%get_def_type_string (), config%get_nlo_type ()))
   select case (config%get_nlo_type ())
   case (NLO_VIRTUAL)
      call setup_virtual_component ()
   case (NLO_REAL)
      call setup_real_component ()
      if (intg%process%get_component_type (i_component) /= COMP_REAL_FIN) &
           i_real = i_component
   case (NLO_MISMATCH)
      call setup_mismatch_component ()
```

```
   case (NLO_DGLAP)
      call setup_dglap_component ()
   case (BORN)
      call setup_born_component ()
   case (NLO_SUBTRACTION)
      call setup_subtraction_component ()
   case (GKS)
      call intg%process%init_component (i_component, &
           core%has_matrix_element (), mci_template, &
           phs_config_template)
   case default
      call msg_fatal ("setup_process: NLO type not implemented!")
   end select
   if (allocated (phs_config_template_other)) &
        deallocate (phs_config_template_other)
end do

intg%process_has_me = intg%process%has_matrix_element ()
if (.not. intg%process_has_me) then
   call msg_warning ("Process '" &
        // char (intg%process_id) // "': matrix element vanishes")
end if

call setup_beams ()
call setup_structure_functions ()

call intg%process%configure_phs &
     (intg%rebuild_phs, intg%ignore_phs_mismatch, & !verbose=verbose, &
      combined_integration = intg%combined_integration)

if (intg%process%is_nlo_calculation ()) then
   call dispatch_fks_s (fks_template, local%var_list)
   call intg%process%init_nlo_settings (var_list, fks_template)
   call intg%process%check_if_threshold_method ()
   i_core_real = intg%process%get_i_core_nlo_type (NLO_REAL)
   i_core_born = intg%process%get_i_core_nlo_type (BORN)
   call intg%process%setup_region_data (i_real, &
        intg%process%get_constants(i_core_born), &
        intg%process%get_constants(i_core_real))
   if (var_list%get_lval (var_str ("?nlo_use_real_partition"))) then
      call intg%process%setup_real_partition &
           (var_list%get_rval (var_str ("real_partition_scale")))
   end if
end if

call intg%process%setup_terms (with_beams = local%beam_structure%is_set ())

if (intg%process%needs_extra_code ()) then
   call blha_template%init (local%beam_structure%has_polarized_beams(), &
        var_list%get_lval (var_str ("?openloops_switch_off_muon_yukawa")), &
        var_list%get_rval (var_str ("blha_use_top_yukawa")), &
        var_list%get_sval (var_str ("$blha_ew_scheme")))
   call intg%process%init_blha_cores(blha_template, var_list)
   call intg%process%create_and_load_extra_libraries &
```

```
          (local%beam_structure, var_list, local%os_data)
   end if

   if (verb) then
      call intg%process%write (screen = .true.)
      call intg%process%print_phs_startup_message ()
   end if

   if (intg%process_has_me) then
      if (size (sf_config) > 0) then
         call intg%process%collect_channels (phs_channel_collection)
      else if (intg%process%contains_trivial_component ()) then
         call msg_fatal ("Integrate: 2 -> 1 process can't be handled &
              &with fixed-energy beams")
      end if
      call dispatch_sf_channels &
           (sf_channel, sf_string, sf_prop, phs_channel_collection, &
           local%var_list, local%get_sqrts(), local%beam_structure)
      if (allocated (sf_channel)) then
         if (size (sf_channel) > 0) then
            call intg%process%set_sf_channel (sf_channel)
         end if
      end if
      call phs_channel_collection%final ()
      if (verb)  call intg%process%sf_startup_message (sf_string)
   end if

   call intg%setup_process_mci ()

   call setup_expressions ()

   call intg%process%compute_md5sum ()
 contains
   subroutine setup_phase_space ()
     intg%rebuild_phs = &
          var_list%get_lval (var_str ("?rebuild_phase_space"))
     intg%ignore_phs_mismatch = &
          .not. var_list%get_lval (var_str ("?check_phs_file"))
     intg%phs_only = &
          var_list%get_lval (var_str ("?phs_only"))
     phs_par%m_threshold_s = &
          var_list%get_rval (var_str ("phs_threshold_s"))
     phs_par%m_threshold_t = &
          var_list%get_rval (var_str ("phs_threshold_t"))
     phs_par%off_shell = &
          var_list%get_ival (var_str ("phs_off_shell"))
     phs_par%keep_nonresonant = &
          var_list%get_lval (var_str ("?phs_keep_nonresonant"))
     phs_par%t_channel = &
          var_list%get_ival (var_str ("phs_t_channel"))
     mapping_defs%energy_scale = &
          var_list%get_rval (var_str ("phs_e_scale"))
     mapping_defs%invariant_mass_scale = &
          var_list%get_rval (var_str ("phs_m_scale"))
```

```
  mapping_defs%momentum_transfer_scale = &
       var_list%get_rval (var_str ("phs_q_scale"))
  mapping_defs%step_mapping = &
       var_list%get_lval (var_str ("?phs_step_mapping"))
  mapping_defs%step_mapping_exp = &
       var_list%get_lval (var_str ("?phs_step_mapping_exp"))
  mapping_defs%enable_s_mapping = &
       var_list%get_lval (var_str ("?phs_s_mapping"))

  call dispatch_phs (phs_config_template, local%var_list, &
       local%os_data, intg%process_id, mapping_defs, phs_par)
end subroutine setup_phase_space

subroutine setup_log_and_history ()
   !!! We avoid two dots in the filename due to a bug in certain MetaPost versions.
   if (intg%run_id /= "") then
      intg%history_filename = intg%process_id // "." // intg%run_id &
           // "-history"
      intg%log_filename = intg%process_id // "." // intg%run_id // ".log"
   else
      intg%history_filename = intg%process_id // "-history"
      intg%log_filename = intg%process_id // ".log"
   end if
   intg%vis_history = &
      var_list%get_lval (var_str ("?vis_history"))
end subroutine setup_log_and_history

subroutine display_init_message (verb)
  logical, intent(in) :: verb
  if (verb) then
     call msg_message ("Initializing integration for process " &
          // char (intg%process_id) // ":")
     if (intg%run_id /= "") &
          call msg_message ("Run ID = " // '"' // char (intg%run_id) // '"')
  end if
end subroutine display_init_message

function get_me_method (nlo_type) result (me_method)
  type(string_t) :: me_method
  integer, intent(in) :: nlo_type
  select case (nlo_type)
  case (BORN)
     me_method = var_list%get_sval (var_str ("$born_me_method"))
  case (NLO_REAL)
     me_method = var_list%get_sval (var_str ("$real_tree_me_method"))
  case (NLO_VIRTUAL)
     me_method = var_list%get_sval (var_str ("$loop_me_method"))
  case (NLO_SUBTRACTION)
     me_method = var_list%get_sval (var_str ("$correlation_me_method"))
  end select
end function get_me_method

subroutine setup_born_component ()
  call intg%process%init_component (i_component, &
```

```
      core%has_matrix_element (), mci_template, phs_config_template)
  call intg%process%set_component_type (i_component, COMP_MASTER)
end subroutine setup_born_component

subroutine setup_virtual_component ()
  call intg%process%init_component (i_component, &
      core%has_matrix_element (), mci_template, phs_config_template)
  call intg%process%set_component_type (i_component, COMP_VIRT)
end subroutine setup_virtual_component

subroutine setup_real_component ()
  logical :: use_finite_real
  use_finite_real = var_list%get_lval (var_str ("?nlo_use_real_partition"))
  if (first_real_component) then
     call dispatch_phs (phs_config_template_other, local%var_list, &
          local%os_data, intg%process_id, mapping_defs, phs_par, &
          var_str ('fks'))
  else
     call dispatch_phs (phs_config_template_other, local%var_list, &
          local%os_data, intg%process_id, mapping_defs, phs_par, &
          var_str ('wood'))
  end if
  call intg%process%init_component (i_component, &
      core%has_matrix_element (), mci_template, phs_config_template_other)
  if (use_finite_real) then
     if (first_real_component) then
        call intg%process%set_component_type (i_component, COMP_REAL_SING)
        first_real_component = .false.
     else
        call intg%process%set_component_type (i_component, COMP_REAL_FIN)
     end if
  else
     call intg%process%set_component_type (i_component, COMP_REAL)
  end if
end subroutine setup_real_component

subroutine setup_mismatch_component ()
  call dispatch_phs (phs_config_template_other, local%var_list, &
      local%os_data, intg%process_id, mapping_defs, phs_par, var_str ('fks'))
  call intg%process%init_component (i_component, &
      core%has_matrix_element (), mci_template, phs_config_template_other)
  if (intg%combined_integration) &
      call intg%process%set_component_type (i_component, COMP_MISMATCH)
end subroutine setup_mismatch_component

subroutine setup_dglap_component ()
  call dispatch_phs (phs_config_template_other, local%var_list, local%os_data, &
      intg%process_id, mapping_defs, phs_par, var_str ('fks'))
  call intg%process%init_component (i_component, &
      core%has_matrix_element (), mci_template, phs_config_template_other)
  if (intg%combined_integration) &
      call intg%process%set_component_type (i_component, COMP_PDF)
end subroutine setup_dglap_component
```

```
subroutine setup_subtraction_component ()
  call intg%process%init_component (i_component, .false., &
       mci_template, phs_config_template)
  if (intg%combined_integration) &
       call intg%process%set_component_type (i_component, COMP_SUB)
end subroutine setup_subtraction_component

subroutine setup_beams ()
  real(default) :: sqrts
  logical :: decay_rest_frame
  sqrts = local%get_sqrts ()
  decay_rest_frame = &
       var_list%get_lval (var_str ("?decay_rest_frame"))
  if (intg%process_has_me) then
     call intg%process%setup_beams_beam_structure &
          (local%beam_structure, sqrts, decay_rest_frame)
  end if
  call intg%process%check_masses ()
  if (verb .and. intg%process_has_me) then
     call intg%process%beams_startup_message &
          (beam_structure = local%beam_structure)
  end if
end subroutine setup_beams

subroutine setup_structure_functions ()
  integer :: n_in
  type(pdg_array_t), dimension(:,:), allocatable :: pdg_prc
  type(string_t) :: sf_trace_file
  if (intg%process_has_me) then
     call intg%process%get_pdg_in (pdg_prc)
  else
     n_in = intg%process%get_n_in ()
     allocate (pdg_prc (n_in, n_components))
     pdg_prc = 0
  end if
  call dispatch_sf_config (sf_config, sf_prop, local%beam_structure, &
     local%get_var_list_ptr (), local%var_list, &
     local%model, local%os_data, local%get_sqrts (), pdg_prc)
  sf_trace = &
       var_list%get_lval (var_str ("?sf_trace"))
  sf_trace_file = &
       var_list%get_sval (var_str ("$sf_trace_file"))
  if (sf_trace) then
     call intg%process%init_sf_chain (sf_config, sf_trace_file)
  else
     call intg%process%init_sf_chain (sf_config)
  end if
end subroutine setup_structure_functions

subroutine setup_expressions ()
  type(eval_tree_factory_t) :: expr_factory
  if (associated (local%pn%cuts_lexpr)) then
     if (verb)  call msg_message ("Applying user-defined cuts.")
     call expr_factory%init (local%pn%cuts_lexpr)
```

```
      call intg%process%set_cuts (expr_factory)
   else
      if (verb)  call msg_warning ("No cuts have been defined.")
   end if
   if (associated (local%pn%scale_expr)) then
      if (verb) call msg_message ("Using user-defined general scale.")
      call expr_factory%init (local%pn%scale_expr)
      call intg%process%set_scale (expr_factory)
   end if
   if (associated (local%pn%fac_scale_expr)) then
      if (verb) call msg_message ("Using user-defined factorization scale.")
      call expr_factory%init (local%pn%fac_scale_expr)
      call intg%process%set_fac_scale (expr_factory)
   end if
   if (associated (local%pn%ren_scale_expr)) then
      if (verb) call msg_message ("Using user-defined renormalization scale.")
      call expr_factory%init (local%pn%ren_scale_expr)
      call intg%process%set_ren_scale (expr_factory)
   end if
   if (associated (local%pn%weight_expr)) then
      if (verb) call msg_message ("Using user-defined reweighting factor.")
      call expr_factory%init (local%pn%weight_expr)
      call intg%process%set_weight (expr_factory)
   end if
 end subroutine setup_expressions
end subroutine integration_setup_process
```

33.6.3 Integration

Integrate: do the final integration. Here, we do a multi-iteration integration. Again, we skip iterations that are already on file. Record the results in the global variable list.

⟨*Integrations: integration: TBP*⟩+≡

```
procedure :: evaluate => integration_evaluate
```

⟨*Integrations: procedures*⟩+≡

```
subroutine integration_evaluate &
     (intg, process_instance, i_mci, pass, it_list, pacify)
  class(integration_t), intent(inout) :: intg
  type(process_instance_t), intent(inout), target :: process_instance
  integer, intent(in) :: i_mci
  integer, intent(in) :: pass
  type(iterations_list_t), intent(in) :: it_list
  logical, intent(in), optional :: pacify
  integer :: n_calls, n_it
  logical :: adapt_grids, adapt_weights, final
  n_it = it_list%get_n_it (pass)
  n_calls = it_list%get_n_calls (pass)
  adapt_grids = it_list%adapt_grids (pass)
  adapt_weights = it_list%adapt_weights (pass)
  final = pass == it_list%get_n_pass ()
  call process_instance%integrate ( &
       i_mci, n_it, n_calls, adapt_grids, adapt_weights, &
```

```
        final, pacify)
end subroutine integration_evaluate
```

In case the user has not provided a list of iterations, make a reasonable default. This can depend on the process. The usual approach is to define two distinct passes, one for adaptation and one for integration.

⟨*Integrations: integration: TBP*⟩+≡

```
procedure :: make_iterations_list => integration_make_iterations_list
```

⟨*Integrations: procedures*⟩+≡

```
subroutine integration_make_iterations_list (intg, it_list)
  class(integration_t), intent(in) :: intg
  type(iterations_list_t), intent(out) :: it_list
  integer :: pass, n_pass
  integer, dimension(:), allocatable :: n_it, n_calls
  logical, dimension(:), allocatable :: adapt_grids, adapt_weights
  n_pass = intg%process%get_n_pass_default ()
  allocate (n_it (n_pass), n_calls (n_pass))
  allocate (adapt_grids (n_pass), adapt_weights (n_pass))
  do pass = 1, n_pass
     n_it(pass)          = intg%process%get_n_it_default (pass)
     n_calls(pass)       = intg%process%get_n_calls_default (pass)
     adapt_grids(pass)   = intg%process%adapt_grids_default (pass)
     adapt_weights(pass) = intg%process%adapt_weights_default (pass)
  end do
  call it_list%init (n_it, n_calls, &
        adapt_grids = adapt_grids, adapt_weights = adapt_weights)
end subroutine integration_make_iterations_list
```

In NLO calculations, the individual components might scale very differently with the number of calls. This especially applies to the real-subtracted component, which usually fluctuates more than the Born and virtual component, making it a bottleneck of the calculation. Thus, the calculation is throttled twice, first by the number of calls for the real component, second by the number of surplus calls of computation-intense virtual matrix elements. Therefore, we want to set a different number of calls for each component, which is done by the subroutine `integration_apply_multipliers`.

⟨*Integrations: integration: TBP*⟩+≡

```
procedure :: init_iteration_multipliers => integration_init_iteration_multipliers
```

⟨*Integrations: procedures*⟩+≡

```
subroutine integration_init_iteration_multipliers (intg, local)
  class(integration_t), intent(inout) :: intg
  type(rt_data_t), intent(in) :: local
  integer :: n_pass, pass
  type(iterations_list_t) :: it_list
  n_pass = local%it_list%get_n_pass ()
  if (n_pass == 0) then
     call intg%make_iterations_list (it_list)
     n_pass = it_list%get_n_pass ()
  end if
  associate (it_multipliers => intg%iteration_multipliers)
     allocate (it_multipliers%n_calls0 (n_pass))
```

```
      do pass = 1, n_pass
         it_multipliers%n_calls0(pass) = local%it_list%get_n_calls (pass)
      end do
      it_multipliers%mult_real = local%var_list%get_rval &
          (var_str ("mult_call_real"))
      it_multipliers%mult_virt = local%var_list%get_rval &
          (var_str ("mult_call_virt"))
      it_multipliers%mult_dglap = local%var_list%get_rval &
          (var_str ("mult_call_dglap"))
    end associate
  end subroutine integration_init_iteration_multipliers
```

⟨*Integrations: integration: TBP*⟩+≡

```
  procedure :: apply_call_multipliers => integration_apply_call_multipliers
```

⟨*Integrations: procedures*⟩+≡

```
  subroutine integration_apply_call_multipliers (intg, n_pass, i_component, it_list)
    class(integration_t), intent(in) :: intg
    integer, intent(in) :: n_pass, i_component
    type(iterations_list_t), intent(inout) :: it_list
    integer :: nlo_type
    integer :: n_calls0, n_calls
    integer :: pass
    real(default) :: multiplier
    nlo_type = intg%process%get_component_nlo_type (i_component)
    do pass = 1, n_pass
       associate (multipliers => intg%iteration_multipliers)
          select case (nlo_type)
          case (NLO_REAL)
             multiplier = multipliers%mult_real
          case (NLO_VIRTUAL)
             multiplier = multipliers%mult_virt
          case (NLO_DGLAP)
             multiplier = multipliers%mult_dglap
          case default
             return
          end select
       end associate
       if (n_pass <= size (intg%iteration_multipliers%n_calls0)) then
          n_calls0 = intg%iteration_multipliers%n_calls0 (pass)
          n_calls = floor (multiplier * n_calls0)
          call it_list%set_n_calls (pass, n_calls)
       end if
    end do
  end subroutine integration_apply_call_multipliers
```

33.6.4 API for integration objects

This initializer does everything except assigning cuts/scale/weight expressions.

⟨*Integrations: integration: TBP*⟩+≡

```
  procedure :: init => integration_init
```

⟨*Integrations: procedures*⟩+≡

```
subroutine integration_init (intg, process_id, local, global, local_stack)
  class(integration_t), intent(out) :: intg
  type(string_t), intent(in) :: process_id
  type(rt_data_t), intent(inout), target :: local
  type(rt_data_t), intent(inout), optional, target :: global
  logical, intent(in), optional :: local_stack
  logical :: use_local
  use_local = .false.;  if (present (local_stack))  use_local = local_stack
  if (present (global)) then
     call intg%create_process (process_id, global)
  else if (use_local) then
     call intg%create_process (process_id, local)
  else
     call intg%create_process (process_id)
  end if
  call intg%init_process (local)
  call intg%setup_process (local)
  call intg%init_iteration_multipliers (local)
end subroutine integration_init
```

Do the integration for a single process, both warmup and final evaluation. The `eff_reset` flag is to suppress numerical noise in the graphical output of the integration history.

⟨*Integrations: integration: TBP*⟩+≡

```
procedure :: integrate => integration_integrate
```

⟨*Integrations: procedures*⟩+≡

```
subroutine integration_integrate (intg, local, eff_reset)
  class(integration_t), intent(inout) :: intg
  type(rt_data_t), intent(in), target :: local
  logical, intent(in), optional :: eff_reset
  type(string_t) :: log_filename
  type(var_list_t), pointer :: var_list
  type(process_instance_t), allocatable, target :: process_instance
  type(iterations_list_t) :: it_list
  logical :: pacify
  integer :: pass, i_mci, n_mci, n_pass
  integer :: i_component
  integer :: nlo_type
  logical :: display_summed
  logical :: nlo_active
  type(string_t) :: component_output

  allocate (process_instance)
  call process_instance%init (intg%process)

  if (intg%process%needs_extra_code ()) &
     call process_instance%setup_blha_helicities ()

  var_list => intg%process%get_var_list_ptr ()
  call openmp_set_num_threads_verbose &
       (var_list%get_ival (var_str ("openmp_num_threads")), &
        var_list%get_lval (var_str ("?openmp_logging")))
```

```
pacify = var_list%get_lval (var_str ("?pacify"))

display_summed = .true.
n_mci = intg%process%get_n_mci ()
if (n_mci == 1) then
   write (msg_buffer, "(A,A,A)") &
        "Starting integration for process '", &
        char (intg%process%get_id ()), "'"
   call msg_message ()
end if

nlo_active = any (intg%process%get_component_nlo_type &
     ([(i_mci, i_mci = 1, n_mci)]) /= BORN)
do i_mci = 1, n_mci
   i_component = intg%process%i_mci_to_i_component (i_mci)
   nlo_type = intg%process%get_component_nlo_type (i_component)
   if (intg%process%component_can_be_integrated (i_component)) then
      if (n_mci > 1) then
         if (nlo_active) then
            if (intg%combined_integration .and. nlo_type == BORN) then
               component_output = var_str ("Combined")
            else
               component_output = component_status (nlo_type)
            end if
            write (msg_buffer, "(A,A,A,A,A)") &
                 "Starting integration for process '", &
                 char (intg%process%get_id ()), "' part '", &
                 char (component_output), "'"
         else
            write (msg_buffer, "(A,A,A,I0)") &
                 "Starting integration for process '", &
                 char (intg%process%get_id ()), "' part ", i_mci
         end if
         call msg_message ()
      end if
      n_pass = local%it_list%get_n_pass ()
      if (n_pass == 0) then
         call msg_message ("Integrate: iterations not specified, &
              &using default")
         call intg%make_iterations_list (it_list)
         n_pass = it_list%get_n_pass ()
      else
         it_list = local%it_list
      end if
      call intg%apply_call_multipliers (n_pass, i_mci, it_list)
      call msg_message ("Integrate: " // char (it_list%to_string ()))
      do pass = 1, n_pass
         call intg%evaluate (process_instance, i_mci, pass, it_list, pacify)
         if (signal_is_pending ())  return
      end do
      call intg%process%final_integration (i_mci)
      if (intg%vis_history) then
         call intg%process%display_integration_history &
              (i_mci, intg%history_filename, local%os_data, eff_reset)
```

```
         end if
         if (local%logfile == intg%log_filename) then
            if (intg%run_id /= "") then
               log_filename = intg%process_id // "." // intg%run_id // &
                    ".var.log"
            else
               log_filename = intg%process_id // ".var.log"
            end if
            call msg_message ("Name clash for global logfile and process log: ", &
                 arr =[var_str ("| Renaming log file from ") // local%logfile, &
                       var_str ("|   to ") // log_filename // var_str (" .")])
         else
            log_filename = intg%log_filename
         end if
         call intg%process%write_logfile (i_mci, log_filename)
      end if
   end do

   if (n_mci > 1 .and. display_summed) then
      call msg_message ("Integrate: sum of all components")
      call intg%process%display_summed_results (pacify)
   end if

   call process_instance%final ()
   deallocate (process_instance)

end subroutine integration_integrate
```

⟨*Integrations: integration: TBP*⟩+≡

```
procedure :: setup_process_mci => integration_setup_process_mci
```

⟨*Integrations: procedures*⟩+≡

```
subroutine integration_setup_process_mci (intg)
  class(integration_t), intent(inout) :: intg
  call intg%process%setup_mci (intg%combined_integration)
end subroutine integration_setup_process_mci
```

Do a dummy integration for a process which could not be initialized (e.g., has no matrix element). The result is zero.

⟨*Integrations: integration: TBP*⟩+≡

```
procedure :: integrate_dummy => integration_integrate_dummy
```

⟨*Integrations: procedures*⟩+≡

```
subroutine integration_integrate_dummy (intg)
  class(integration_t), intent(inout) :: intg
  call intg%process%integrate_dummy ()
end subroutine integration_integrate_dummy
```

Just sample the matrix element under realistic conditions (but no cuts); throw away the results.

⟨*Integrations: integration: TBP*⟩+≡

```
procedure :: sampler_test => integration_sampler_test
```

⟨*Integrations: procedures*⟩+≡

```
subroutine integration_sampler_test (intg)
  class(integration_t), intent(inout) :: intg
  type(process_instance_t), allocatable, target :: process_instance
  integer :: n_mci, i_mci
  type(timer_t) :: timer_mci, timer_tot
  real(default) :: t_mci, t_tot
  allocate (process_instance)
  call process_instance%init (intg%process)
  n_mci = intg%process%get_n_mci ()
  if (n_mci == 1) then
     write (msg_buffer, "(A,A,A)") &
          "Test: probing process '", &
          char (intg%process%get_id ()), "'"
     call msg_message ()
  end if
  call timer_tot%start ()
  do i_mci = 1, n_mci
     if (n_mci > 1) then
        write (msg_buffer, "(A,A,A,I0)") &
             "Test: probing process '", &
             char (intg%process%get_id ()), "' part ", i_mci
        call msg_message ()
     end if
     call timer_mci%start ()
     call process_instance%sampler_test (i_mci, intg%n_calls_test)
     call timer_mci%stop ()
     t_mci = timer_mci
     write (msg_buffer, "(A,ES12.5)")  "Test: " &
          // "time in seconds (wallclock): ", t_mci
     call msg_message ()
  end do
  call timer_tot%stop ()
  t_tot = timer_tot
  if (n_mci > 1) then
     write (msg_buffer, "(A,ES12.5)")  "Test: " &
          // "total time      (wallclock): ", t_tot
     call msg_message ()
  end if
  call process_instance%final ()
end subroutine integration_sampler_test
```

Return the process pointer (needed by simulate):

⟨*Integrations: integration: TBP*⟩+≡

```
procedure :: get_process_ptr => integration_get_process_ptr
```

⟨*Integrations: procedures*⟩+≡

```
function integration_get_process_ptr (intg) result (ptr)
  class(integration_t), intent(in) :: intg
  type(process_t), pointer :: ptr
  ptr => intg%process
end function integration_get_process_ptr
```

Simply integrate, do a dummy integration if necessary. The `integration` object exists only internally.

If the `global` environment is provided, the process object is appended to the global stack. Otherwise, if `local_stack` is set, we append to the local process stack. If this is unset, the `process` object is not recorded permanently.

The `init_only` flag can be used to skip the actual integration part. We will end up with a process object that is completely initialized, including phase space configuration.

The `eff_reset` flag is to suppress numerical noise in the visualization of the integration history.

⟨*Integrations: public*⟩+≡

```
public :: integrate_process
```

⟨*Integrations: procedures*⟩+≡

```
subroutine integrate_process (process_id, local, global, local_stack, init_only, eff_reset)
  type(string_t), intent(in) :: process_id
  type(rt_data_t), intent(inout), target :: local
  type(rt_data_t), intent(inout), optional, target :: global
  logical, intent(in), optional :: local_stack, init_only, eff_reset
  type(string_t) :: prclib_name
  type(integration_t) :: intg
  character(32) :: buffer

  if (.not. associated (local%prclib)) then
     call msg_fatal ("Integrate: current process library is undefined")
     return
  end if

  if (.not. local%prclib%is_active ()) then
     call msg_message ("Integrate: current process library needs compilation")
     prclib_name = local%prclib%get_name ()
     call compile_library (prclib_name, local)
     if (signal_is_pending ())  return
     call msg_message ("Integrate: compilation done")
  end if

  call intg%init (process_id, local, global, local_stack)
  if (signal_is_pending ())  return

  if (present (init_only)) then
     if (init_only) return
  end if

  if (intg%n_calls_test > 0) then
     write (buffer, "(I0)")  intg%n_calls_test
     call msg_message ("Integrate: test (" // trim (buffer) // " calls) ...")
     call intg%sampler_test ()
     call msg_message ("Integrate: ... test complete.")
     if (signal_is_pending ())  return
  end if

  if (intg%phs_only) then
     call msg_message ("Integrate: phase space only, skipping integration")
  else
```

```
      if (intg%process_has_me) then
         call intg%integrate (local, eff_reset)
      else
         call intg%integrate_dummy ()
      end if
    end if

  end subroutine integrate_process
```

33.6.5 Unit Tests

Test module, followed by the stand-alone unit-test procedures.

⟨integrations_ut.f90⟩≡

```
⟨File header⟩

module integrations_ut
  use unit_tests
  use integrations_uti

⟨Standard module head⟩

⟨Integrations: public test⟩

contains

⟨Integrations: test driver⟩

end module integrations_ut
```

⟨integrations_uti.f90⟩≡

```
⟨File header⟩

module integrations_uti

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use ifiles
  use lexers
  use parser
  use io_units
  use flavors
  use interactions, only: reset_interaction_counter
  use phs_forests
  use eval_trees
  use models
  use rt_data
  use process_configurations_ut, only: prepare_test_library
  use compilations, only: compile_library

  use integrations

  use phs_wood_ut, only: write_test_phs_file
```

```
⟨Standard module head⟩

⟨Integrations: test declarations⟩

contains

⟨Integrations: tests⟩

end module integrations_uti
```

API: driver for the unit tests below.

```
⟨Integrations: public test⟩≡
    public :: integrations_test
⟨Integrations: test driver⟩≡
    subroutine integrations_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Integrations: execute tests⟩
    end subroutine integrations_test
```

```
⟨Integrations: public test⟩+≡
    public :: integrations_history_test
⟨Integrations: test driver⟩+≡
    subroutine integrations_history_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Integrations: execute history tests⟩
    end subroutine integrations_history_test
```

Integration of test process

Compile and integrate an intrinsic test matrix element (`prc_test` type). The phase-space implementation is `phs_single` (single-particle phase space), the integrator is `mci_midpoint`.

The cross section for the $2 \to 2$ process $ss \to ss$ with its constant matrix element is given by

$$\sigma = c \times f \times \Phi_2 \times |M|^2. \tag{33.1}$$

c is the conversion constant

$$c = 0.3894 \times 10^{12}\ \mathrm{fb\,GeV}^2. \tag{33.2}$$

f is the flux of the incoming particles with mass $m = 125\,\mathrm{GeV}$ and energy $\sqrt{s} = 1000\,\mathrm{GeV}$

$$f = \frac{(2\pi)^4}{2\lambda^{1/2}(s, m^2, m^2)} = \frac{(2\pi)^4}{2\sqrt{s}\,\sqrt{s - 4m^2}} = 8.048 \times 10^{-4}\ \mathrm{GeV}^{-2} \tag{33.3}$$

Φ_2 is the volume of the two-particle phase space

$$\Phi_2 = \frac{1}{4(2\pi)^5} = 2.5529 \times 10^{-5}. \tag{33.4}$$

The squared matrix element $|M|^2$ is unity. Combining everything, we obtain

$$\sigma = 8000 \text{ fb} \tag{33.5}$$

This number should appear as the final result.

Note: In this and the following test, we reset the Fortran compiler and flag variables immediately before they are printed, so the test is portable.

⟨*Integrations: execute tests*⟩≡
```
call test (integrations_1, "integrations_1", &
     "intrinsic test process", &
     u, results)
```

⟨*Integrations: test declarations*⟩≡
```
public :: integrations_1
```

⟨*Integrations: tests*⟩≡
```
subroutine integrations_1 (u)
  integer, intent(in) :: u
  type(string_t) :: libname, procname
  type(rt_data_t), target :: global

  write (u, "(A)")  "* Test output: integrations_1"
  write (u, "(A)")  "*   Purpose: integrate test process"
  write (u, "(A)")

  call syntax_model_file_init ()

  call global%global_init ()

  libname = "integration_1"
  procname = "prc_config_a"

  call prepare_test_library (global, libname, 1)
  call compile_library (libname, global)

  call global%set_string (var_str ("$run_id"), &
       var_str ("integrations1"), is_known = .true.)
  call global%set_string (var_str ("$method"), &
       var_str ("unit_test"), is_known = .true.)
  call global%set_string (var_str ("$phs_method"), &
       var_str ("single"), is_known = .true.)
  call global%set_string (var_str ("$integration_method"),&
       var_str ("midpoint"), is_known = .true.)
  call global%set_log (var_str ("?vis_history"),&
       .false., is_known = .true.)
  call global%set_log (var_str ("?integration_timer"),&
       .false., is_known = .true.)
  call global%set_int (var_str ("seed"), &
       0, is_known=.true.)

  call global%set_real (var_str ("sqrts"),&
       1000._default, is_known = .true.)

  call global%it_list%init ([1], [1000])
```

```
    call reset_interaction_counter ()
    call integrate_process (procname, global, local_stack=.true.)

    call global%write (u, vars = [ &
         var_str ("$method"), &
         var_str ("sqrts"), &
         var_str ("$integration_method"), &
         var_str ("$phs_method"), &
         var_str ("$run_id")])

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: integrations_1"

  end subroutine integrations_1
```

Integration with cuts

Compile and integrate an intrinsic test matrix element (`prc_test` type) with cuts set.

⟨*Integrations: execute tests*⟩+≡

```
  call test (integrations_2, "integrations_2", &
       "intrinsic test process with cut", &
       u, results)
```

⟨*Integrations: test declarations*⟩+≡

```
  public :: integrations_2
```

⟨*Integrations: tests*⟩+≡

```
  subroutine integrations_2 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname
    type(rt_data_t), target :: global

    type(string_t) :: cut_expr_text
    type(ifile_t) :: ifile
    type(stream_t) :: stream
    type(parse_tree_t) :: parse_tree

    type(string_t), dimension(0) :: empty_string_array

    write (u, "(A)")  "* Test output: integrations_2"
    write (u, "(A)")  "*   Purpose: integrate test process with cut"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()

    write (u, "(A)")  "* Prepare a cut expression"
    write (u, "(A)")
```

```
  call syntax_pexpr_init ()
  cut_expr_text = "all Pt > 100 [s]"
  call ifile_append (ifile, cut_expr_text)
  call stream_init (stream, ifile)
  call parse_tree_init_lexpr (parse_tree, stream, .true.)
  global%pn%cuts_lexpr => parse_tree%get_root_ptr ()

  write (u, "(A)")  "* Build and initialize a test process"
  write (u, "(A)")

  libname = "integration_3"
  procname = "prc_config_a"

  call prepare_test_library (global, libname, 1)
  call compile_library (libname, global)

  call global%set_string (var_str ("$run_id"), &
       var_str ("integrations1"), is_known = .true.)
  call global%set_string (var_str ("$method"), &
       var_str ("unit_test"), is_known = .true.)
  call global%set_string (var_str ("$phs_method"), &
       var_str ("single"), is_known = .true.)
  call global%set_string (var_str ("$integration_method"),&
       var_str ("midpoint"), is_known = .true.)
  call global%set_log (var_str ("?vis_history"),&
       .false., is_known = .true.)
  call global%set_log (var_str ("?integration_timer"),&
       .false., is_known = .true.)
  call global%set_int (var_str ("seed"), &
       0, is_known=.true.)

  call global%set_real (var_str ("sqrts"),&
       1000._default, is_known = .true.)

  call global%it_list%init ([1], [1000])

  call reset_interaction_counter ()
  call integrate_process (procname, global, local_stack=.true.)

  call global%write (u, vars = empty_string_array)

  call global%final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: integrations_2"

end subroutine integrations_2
```

Standard phase space

Compile and integrate an intrinsic test matrix element (prc_test type) using the default (phs_wood) phase-space implementation. We use an explicit phase-space configuration file with a single channel and integrate by mci_midpoint.

⟨*Integrations: execute tests*⟩+≡
```
call test (integrations_3, "integrations_3", &
     "standard phase space", &
     u, results)
```

⟨*Integrations: test declarations*⟩+≡
```
public :: integrations_3
```

⟨*Integrations: tests*⟩+≡
```
subroutine integrations_3 (u)
⟨Use kinds⟩
⟨Use strings⟩
  use interactions, only: reset_interaction_counter
  use models
  use rt_data
  use process_configurations_ut, only: prepare_test_library
  use compilations, only: compile_library
  use integrations

  implicit none

  integer, intent(in) :: u
  type(string_t) :: libname, procname
  type(rt_data_t), target :: global
  integer :: u_phs

  write (u, "(A)")  "* Test output: integrations_3"
  write (u, "(A)")  "*   Purpose: integrate test process"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize process and parameters"
  write (u, "(A)")

  call syntax_model_file_init ()
  call syntax_phs_forest_init ()

  call global%global_init ()

  libname = "integration_3"
  procname = "prc_config_a"

  call prepare_test_library (global, libname, 1)
  call compile_library (libname, global)

  call global%set_string (var_str ("$run_id"), &
       var_str ("integrations1"), is_known = .true.)
  call global%set_string (var_str ("$method"), &
       var_str ("unit_test"), is_known = .true.)
  call global%set_string (var_str ("$phs_method"), &
       var_str ("default"), is_known = .true.)
```

```
  call global%set_string (var_str ("$integration_method"),&
       var_str ("midpoint"), is_known = .true.)
  call global%set_log (var_str ("?vis_history"),&
       .false., is_known = .true.)
  call global%set_log (var_str ("?integration_timer"),&
       .false., is_known = .true.)
  call global%set_log (var_str ("?phs_s_mapping"),&
       .false., is_known = .true.)
  call global%set_int (var_str ("seed"), &
       0, is_known=.true.)

  call global%set_real (var_str ("sqrts"),&
       1000._default, is_known = .true.)

  write (u, "(A)")  "* Create a scratch phase-space file"
  write (u, "(A)")

  u_phs = free_unit ()
  open (u_phs, file = "integrations_3.phs", &
       status = "replace", action = "write")
  call write_test_phs_file (u_phs, var_str ("prc_config_a_i1"))
  close (u_phs)

  call global%set_string (var_str ("$phs_file"),&
       var_str ("integrations_3.phs"), is_known = .true.)

  call global%it_list%init ([1], [1000])

  write (u, "(A)")  "* Integrate"
  write (u, "(A)")

  call reset_interaction_counter ()
  call integrate_process (procname, global, local_stack=.true.)

  call global%write (u, vars = [ &
       var_str ("$phs_method"), &
       var_str ("$phs_file")])

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call global%final ()
  call syntax_phs_forest_final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: integrations_3"

end subroutine integrations_3
```

VAMP integration

Compile and integrate an intrinsic test matrix element (prc_test type) using the single-channel (phs_single) phase-space implementation. The integration method is vamp.

⟨*Integrations: execute tests*⟩+≡
```
    call test (integrations_4, "integrations_4", &
         "VAMP integration (one iteration)", &
         u, results)
```

⟨*Integrations: test declarations*⟩+≡
```
    public :: integrations_4
```

⟨*Integrations: tests*⟩+≡
```
    subroutine integrations_4 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname
      type(rt_data_t), target :: global

      write (u, "(A)")  "* Test output: integrations_4"
      write (u, "(A)")  "*   Purpose: integrate test process using VAMP"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize process and parameters"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()

      libname = "integrations_4_lib"
      procname = "integrations_4"

      call prepare_test_library (global, libname, 1, [procname])
      call compile_library (libname, global)

      call global%append_log (&
           var_str ("?rebuild_grids"), .true., intrinsic = .true.)

      call global%set_string (var_str ("$run_id"), &
           var_str ("r1"), is_known = .true.)
      call global%set_string (var_str ("$method"), &
           var_str ("unit_test"), is_known = .true.)
      call global%set_string (var_str ("$phs_method"), &
           var_str ("single"), is_known = .true.)
      call global%set_string (var_str ("$integration_method"),&
           var_str ("vamp"), is_known = .true.)
      call global%set_log (var_str ("?use_vamp_equivalences"),&
           .false., is_known = .true.)
      call global%set_log (var_str ("?vis_history"),&
           .false., is_known = .true.)
      call global%set_log (var_str ("?integration_timer"),&
           .false., is_known = .true.)
      call global%set_int (var_str ("seed"), &
           0, is_known=.true.)
```

```
    call global%set_real (var_str ("sqrts"),&
         1000._default, is_known = .true.)

    call global%it_list%init ([1], [1000])

    write (u, "(A)")  "* Integrate"
    write (u, "(A)")

    call reset_interaction_counter ()
    call integrate_process (procname, global, local_stack=.true.)

    call global%pacify (efficiency_reset = .true., error_reset = .true.)
    call global%write (u, vars = [var_str ("$integration_method")], &
            pacify = .true.)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: integrations_4"

  end subroutine integrations_4
```

Multiple iterations integration

Compile and integrate an intrinsic test matrix element (`prc_test` type) using the single-channel (`phs_single`) phase-space implementation. The integration method is `vamp`. We launch three iterations.

⟨*Integrations: execute tests*⟩+≡

```
  call test (integrations_5, "integrations_5", &
       "VAMP integration (three iterations)", &
       u, results)
```

⟨*Integrations: test declarations*⟩+≡

```
  public :: integrations_5
```

⟨*Integrations: tests*⟩+≡

```
  subroutine integrations_5 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname
    type(rt_data_t), target :: global

    write (u, "(A)")  "* Test output: integrations_5"
    write (u, "(A)")  "*   Purpose: integrate test process using VAMP"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize process and parameters"
    write (u, "(A)")
```

```
call syntax_model_file_init ()

call global%global_init ()

libname = "integrations_5_lib"
procname = "integrations_5"

call prepare_test_library (global, libname, 1, [procname])
call compile_library (libname, global)

call global%append_log (&
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)

call global%set_string (var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call global%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call global%set_string (var_str ("$phs_method"), &
     var_str ("single"), is_known = .true.)
call global%set_string (var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call global%set_log (var_str ("?use_vamp_equivalences"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?vis_history"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)
call global%set_int (var_str ("seed"), &
     0, is_known=.true.)

call global%set_real (var_str ("sqrts"),&
     1000._default, is_known = .true.)

call global%it_list%init ([3], [1000])

write (u, "(A)")  "* Integrate"
write (u, "(A)")

call reset_interaction_counter ()
call integrate_process (procname, global, local_stack=.true.)

call global%pacify (efficiency_reset = .true., error_reset = .true.)
call global%write (u, vars = [var_str ("$integration_method")], &
        pacify = .true.)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call global%final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: integrations_5"
```

```
end subroutine integrations_5
```

Multiple passes integration

Compile and integrate an intrinsic test matrix element (prc_test type) using the single-channel (phs_single) phase-space implementation. The integration method is vamp. We launch three passes with three iterations each.

⟨*Integrations: execute tests*⟩+≡

```
call test (integrations_6, "integrations_6", &
     "VAMP integration (three passes)", &
     u, results)
```

⟨*Integrations: test declarations*⟩+≡

```
public :: integrations_6
```

⟨*Integrations: tests*⟩+≡

```
subroutine integrations_6 (u)
  integer, intent(in) :: u
  type(string_t) :: libname, procname
  type(rt_data_t), target :: global
  type(string_t), dimension(0) :: no_vars

  write (u, "(A)")  "* Test output: integrations_6"
  write (u, "(A)")  "*   Purpose: integrate test process using VAMP"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize process and parameters"
  write (u, "(A)")

  call syntax_model_file_init ()

  call global%global_init ()

  libname = "integrations_6_lib"
  procname = "integrations_6"

  call prepare_test_library (global, libname, 1, [procname])
  call compile_library (libname, global)

  call global%append_log (&
       var_str ("?rebuild_grids"), .true., intrinsic = .true.)

  call global%set_string (var_str ("$run_id"), &
       var_str ("r1"), is_known = .true.)
  call global%set_string (var_str ("$method"), &
       var_str ("unit_test"), is_known = .true.)
  call global%set_string (var_str ("$phs_method"), &
       var_str ("single"), is_known = .true.)
  call global%set_string (var_str ("$integration_method"),&
       var_str ("vamp"), is_known = .true.)
  call global%set_log (var_str ("?use_vamp_equivalences"),&
       .false., is_known = .true.)
  call global%set_log (var_str ("?vis_history"),&
```

```
      .false., is_known = .true.)
  call global%set_log (var_str ("?integration_timer"),&
      .false., is_known = .true.)
  call global%set_int (var_str ("seed"), &
      0, is_known=.true.)

  call global%set_real (var_str ("sqrts"),&
      1000._default, is_known = .true.)

  call global%it_list%init ([3, 3, 3], [1000, 1000, 1000], &
      adapt = [.true., .true., .false.], &
      adapt_code = [var_str ("wg"), var_str ("g"), var_str ("")])

  write (u, "(A)")  "* Integrate"
  write (u, "(A)")

  call reset_interaction_counter ()
  call integrate_process (procname, global, local_stack=.true.)

  call global%pacify (efficiency_reset = .true., error_reset = .true.)
  call global%write (u, vars = no_vars, pacify = .true.)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call global%final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: integrations_6"

end subroutine integrations_6
```

VAMP and default phase space

Compile and integrate an intrinsic test matrix element (prc_test type) using the default (phs_wood) phase-space implementation. The integration method is vamp. We launch three passes with three iterations each. We enable channel equivalences and groves.

⟨*Integrations: execute tests*⟩+≡

```
call test (integrations_7, "integrations_7", &
    "VAMP integration with wood phase space", &
    u, results)
```

⟨*Integrations: test declarations*⟩+≡

```
public :: integrations_7
```

⟨*Integrations: tests*⟩+≡

```
subroutine integrations_7 (u)
  integer, intent(in) :: u
  type(string_t) :: libname, procname
  type(rt_data_t), target :: global
  type(string_t), dimension(0) :: no_vars
```

```
integer :: iostat, u_phs
character(95) :: buffer
type(string_t) :: phs_file
logical :: exist

write (u, "(A)")  "* Test output: integrations_7"
write (u, "(A)")  "*   Purpose: integrate test process using VAMP"
write (u, "(A)")

write (u, "(A)")  "* Initialize process and parameters"
write (u, "(A)")

call syntax_model_file_init ()
call syntax_phs_forest_init ()

call global%global_init ()

libname = "integrations_7_lib"
procname = "integrations_7"

call prepare_test_library (global, libname, 1, [procname])
call compile_library (libname, global)

call global%append_log (&
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)

call global%set_string (var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call global%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call global%set_string (var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call global%set_string (var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call global%set_log (var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call global%set_log (var_str ("?vis_history"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?phs_s_mapping"),&
     .false., is_known = .true.)
call global%set_int (var_str ("seed"), &
     0, is_known=.true.)

call global%set_real (var_str ("sqrts"),&
     1000._default, is_known = .true.)

call global%it_list%init ([3, 3, 3], [1000, 1000, 1000], &
     adapt = [.true., .true., .false.], &
     adapt_code = [var_str ("wg"), var_str ("g"), var_str ("")])
```

```
    write (u, "(A)")  "* Integrate"
    write (u, "(A)")

    call reset_interaction_counter ()
    call integrate_process (procname, global, local_stack=.true.)

    call global%pacify (efficiency_reset = .true., error_reset = .true.)
    call global%write (u, vars = no_vars, pacify = .true.)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call global%final ()
    call syntax_phs_forest_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Generated phase-space file"
    write (u, "(A)")

    phs_file = procname // "_i1.r1.phs"
    inquire (file = char (phs_file), exist = exist)
    if (exist) then
       u_phs = free_unit ()
       open (u_phs, file = char (phs_file), action = "read", status = "old")
       iostat = 0
       do while (iostat == 0)
          read (u_phs, "(A)", iostat = iostat)  buffer
          if (iostat == 0)  write (u, "(A)")  trim (buffer)
       end do
       close (u_phs)
    else
       write (u, "(A)")  "[file is missing]"
    end if

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: integrations_7"

  end subroutine integrations_7
```

Structure functions

Compile and integrate an intrinsic test matrix element (`prc_test` type) using the default (`phs_wood`) phase-space implementation. The integration method is `vamp`. There is a structure function of type `unit_test`.

We use a test structure function $f(x) = x$ for both beams. Together with the $1/x_1x_2$ factor from the phase-space flux and a unit matrix element, we should get the same result as previously for the process without structure functions. There is a slight correction due to the m_s mass which we set to zero here.

⟨*Integrations: execute tests*⟩+≡

```
  call test (integrations_8, "integrations_8", &
       "integration with structure function", &
```

```
        u, results)
```

⟨*Integrations: test declarations*⟩+≡

```
    public :: integrations_8
```

⟨*Integrations: tests*⟩+≡

```
    subroutine integrations_8 (u)
    ⟨Use kinds⟩
    ⟨Use strings⟩
      use interactions, only: reset_interaction_counter
      use phs_forests
      use models
      use rt_data
      use process_configurations_ut, only: prepare_test_library
      use compilations, only: compile_library
      use integrations

      implicit none

      integer, intent(in) :: u
      type(string_t) :: libname, procname
      type(rt_data_t), target :: global
      type(flavor_t) :: flv
      type(string_t) :: name

      write (u, "(A)")  "* Test output: integrations_8"
      write (u, "(A)")  "*   Purpose: integrate test process using VAMP &
           &with structure function"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize process and parameters"
      write (u, "(A)")

      call syntax_model_file_init ()
      call syntax_phs_forest_init ()

      call global%global_init ()

      libname = "integrations_8_lib"
      procname = "integrations_8"

      call prepare_test_library (global, libname, 1, [procname])
      call compile_library (libname, global)

      call global%append_log (&
           var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
      call global%append_log (&
           var_str ("?rebuild_grids"), .true., intrinsic = .true.)

      call global%set_string (var_str ("$run_id"), &
           var_str ("r1"), is_known = .true.)
      call global%set_string (var_str ("$method"), &
           var_str ("unit_test"), is_known = .true.)
      call global%set_string (var_str ("$phs_method"), &
           var_str ("wood"), is_known = .true.)
```

```
  call global%set_string (var_str ("$integration_method"),&
       var_str ("vamp"), is_known = .true.)
  call global%set_log (var_str ("?use_vamp_equivalences"),&
       .true., is_known = .true.)
  call global%set_log (var_str ("?vis_history"),&
       .false., is_known = .true.)
  call global%set_log (var_str ("?integration_timer"),&
       .false., is_known = .true.)
  call global%set_log (var_str ("?phs_s_mapping"),&
       .false., is_known = .true.)
  call global%set_int (var_str ("seed"), &
       0, is_known=.true.)

  call global%set_real (var_str ("sqrts"),&
       1000._default, is_known = .true.)
  call global%model_set_real (var_str ("ms"), 0._default)

  call reset_interaction_counter ()

  call flv%init (25, global%model)

  name = flv%get_name ()
  call global%beam_structure%init_sf ([name, name], [1])
  call global%beam_structure%set_sf (1, 1, var_str ("sf_test_1"))

  write (u, "(A)")  "* Integrate"
  write (u, "(A)")

  call global%it_list%init ([1], [1000])
  call integrate_process (procname, global, local_stack=.true.)

  call global%write (u, vars = [var_str ("ms")])

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call global%final ()
  call syntax_phs_forest_final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: integrations_8"

end subroutine integrations_8
```

Integration with sign change

Compile and integrate an intrinsic test matrix element (`prc_test` type). The phase-space implementation is `phs_single` (single-particle phase space), the integrator is `mci_midpoint`. The weight that is applied changes the sign in half of phase space. The weight is -3 and 1, respectively, so the total result is equal to the original, but negative sign.

The efficiency should (approximately) become the average of 1 and 1/3, that is 2/3.

⟨*Integrations: execute tests*⟩+≡
```
call test (integrations_9, "integrations_9", &
     "handle sign change", &
     u, results)
```

⟨*Integrations: test declarations*⟩+≡
```
public :: integrations_9
```

⟨*Integrations: tests*⟩+≡
```
subroutine integrations_9 (u)
  integer, intent(in) :: u
  type(string_t) :: libname, procname
  type(rt_data_t), target :: global

  type(string_t) :: wgt_expr_text
  type(ifile_t) :: ifile
  type(stream_t) :: stream
  type(parse_tree_t) :: parse_tree

  write (u, "(A)")  "* Test output: integrations_9"
  write (u, "(A)")  "*   Purpose: integrate test process"
  write (u, "(A)")

  call syntax_model_file_init ()

  call global%global_init ()

  write (u, "(A)")  "* Prepare a weight expression"
  write (u, "(A)")

  call syntax_pexpr_init ()
  wgt_expr_text = "eval 2 * sgn (Pz) - 1 [s]"
  call ifile_append (ifile, wgt_expr_text)
  call stream_init (stream, ifile)
  call parse_tree_init_expr (parse_tree, stream, .true.)
  global%pn%weight_expr => parse_tree%get_root_ptr ()

  write (u, "(A)")  "* Build and evaluate a test process"
  write (u, "(A)")

  libname = "integration_9"
  procname = "prc_config_a"

  call prepare_test_library (global, libname, 1)
  call compile_library (libname, global)

  call global%set_string (var_str ("$run_id"), &
       var_str ("integrations1"), is_known = .true.)
  call global%set_string (var_str ("$method"), &
       var_str ("unit_test"), is_known = .true.)
  call global%set_string (var_str ("$phs_method"), &
       var_str ("single"), is_known = .true.)
  call global%set_string (var_str ("$integration_method"),&
```

```
        var_str ("midpoint"), is_known = .true.)
    call global%set_log (var_str ("?vis_history"),&
        .false., is_known = .true.)
    call global%set_log (var_str ("?integration_timer"),&
        .false., is_known = .true.)
    call global%set_int (var_str ("seed"), &
        0, is_known=.true.)

    call global%set_real (var_str ("sqrts"),&
        1000._default, is_known = .true.)

    call global%it_list%init ([1], [1000])

    call reset_interaction_counter ()
    call integrate_process (procname, global, local_stack=.true.)

    call global%write (u, vars = [ &
        var_str ("$method"), &
        var_str ("sqrts"), &
        var_str ("$integration_method"), &
        var_str ("$phs_method"), &
        var_str ("$run_id")])

    call global%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: integrations_9"

  end subroutine integrations_9
```

Integration history for VAMP integration with default phase space

This test is only run when event analysis can be done.

⟨*Integrations: execute history tests*⟩≡

```
  call test (integrations_history_1, "integrations_history_1", &
      "Test integration history files", &
      u, results)
```

⟨*Integrations: test declarations*⟩+≡

```
  public :: integrations_history_1
```

⟨*Integrations: tests*⟩+≡

```
  subroutine integrations_history_1 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname
    type(rt_data_t), target :: global
    type(string_t), dimension(0) :: no_vars
    integer :: iostat, u_his
    character(91) :: buffer
    type(string_t) :: his_file, ps_file, pdf_file
    logical :: exist, exist_ps, exist_pdf

    write (u, "(A)")  "* Test output: integrations_history_1"
```

```
write (u, "(A)")  "*   Purpose: test integration history files"
write (u, "(A)")

write (u, "(A)")  "* Initialize process and parameters"
write (u, "(A)")

call syntax_model_file_init ()
call syntax_phs_forest_init ()

call global%global_init ()

libname = "integrations_history_1_lib"
procname = "integrations_history_1"

call global%set_log (var_str ("?vis_history"), &
     .true., is_known = .true.)
call global%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?phs_s_mapping"),&
     .false., is_known = .true.)

call prepare_test_library (global, libname, 1, [procname])
call compile_library (libname, global)

call global%append_log (&
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)

call global%set_string (var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call global%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call global%set_string (var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call global%set_string (var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call global%set_log (var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call global%set_real (var_str ("error_threshold"),&
     5E-6_default, is_known = .true.)
call global%set_int (var_str ("seed"), &
     0, is_known=.true.)

call global%set_real (var_str ("sqrts"),&
     1000._default, is_known = .true.)

call global%it_list%init ([2, 2, 2], [1000, 1000, 1000], &
     adapt = [.true., .true., .false.], &
     adapt_code = [var_str ("wg"), var_str ("g"), var_str ("")])

write (u, "(A)")  "* Integrate"
write (u, "(A)")
```

```
  call reset_interaction_counter ()
  call integrate_process (procname, global, local_stack=.true., &
       eff_reset = .true.)

  call global%pacify (efficiency_reset = .true., error_reset = .true.)
  call global%write (u, vars = no_vars, pacify = .true.)

  write (u, "(A)")
  write (u, "(A)")  "* Generated history files"
  write (u, "(A)")

  his_file = procname // ".r1-history.tex"
  ps_file  = procname // ".r1-history.ps"
  pdf_file = procname // ".r1-history.pdf"
  inquire (file = char (his_file), exist = exist)
  if (exist) then
     u_his = free_unit ()
     open (u_his, file = char (his_file), action = "read", status = "old")
     iostat = 0
     do while (iostat == 0)
        read (u_his, "(A)", iostat = iostat)  buffer
        if (iostat == 0)  write (u, "(A)")  trim (buffer)
     end do
     close (u_his)
  else
     write (u, "(A)")  "[History LaTeX file is missing]"
  end if
  inquire (file = char (ps_file), exist = exist_ps)
  if (exist_ps) then
     write (u, "(A)")  "[History Postscript file exists and is nonempty]"
  else
     write (u, "(A)")  "[History Postscript file is missing/non-regular]"
  end if
  inquire (file = char (pdf_file), exist = exist_pdf)
  if (exist_pdf) then
     write (u, "(A)")  "[History PDF file exists and is nonempty]"
  else
     write (u, "(A)")  "[History PDF file is missing/non-regular]"
  end if

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call global%final ()
  call syntax_phs_forest_final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: integrations_history_1"

end subroutine integrations_history_1
```

33.7 Event Streams

This module manages I/O from/to multiple concurrent event streams. Usually, there is at most one input stream, but several output streams. For the latter, we set up an array which can hold `eio_t` (event I/O) objects of different dynamic types simultaneously. One of them may be marked as an input channel.

⟨`event_streams.f90`⟩≡

```
⟨File header⟩

module event_streams

⟨Use strings⟩
  use io_units
  use diagnostics
  use events
  use eio_data
  use eio_base
  use rt_data

  use dispatch_transforms, only: dispatch_eio

⟨Standard module head⟩

⟨Event streams: public⟩

⟨Event streams: types⟩

contains

⟨Event streams: procedures⟩

end module event_streams
```

33.7.1 Event Stream Array

Each entry is an `eio_t` object. Since the type is dynamic, we need a wrapper:

⟨*Event streams: types*⟩≡

```
type :: event_stream_entry_t
   class(eio_t), allocatable :: eio
end type event_stream_entry_t
```

An array of event-stream entry objects. If one of the entries is an input channel, `i_in` is the corresponding index.

⟨*Event streams: public*⟩≡

```
public :: event_stream_array_t
```

⟨*Event streams: types*⟩+≡

```
type :: event_stream_array_t
   type(event_stream_entry_t), dimension(:), allocatable :: entry
   integer :: i_in = 0
 contains
 ⟨Event streams: event stream array: TBP⟩
end type event_stream_array_t
```

Output.

⟨*Event streams: event stream array: TBP*⟩≡

```
procedure :: write => event_stream_array_write
```

⟨*Event streams: procedures*⟩≡

```
subroutine event_stream_array_write (object, unit)
  class(event_stream_array_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(1x,A)")  "Event stream array:"
  if (allocated (object%entry)) then
     select case (size (object%entry))
     case (0)
        write (u, "(3x,A)")  "[empty]"
     case default
        do i = 1, size (object%entry)
           if (i == object%i_in)  write (u, "(1x,A)")  "Input stream:"
           call object%entry(i)%eio%write (u)
        end do
     end select
  else
     write (u, "(3x,A)")  "[undefined]"
  end if
end subroutine event_stream_array_write
```

Finalize all streams.

⟨*Event streams: event stream array: TBP*⟩+≡

```
procedure :: final => event_stream_array_final
```

⟨*Event streams: procedures*⟩+≡

```
subroutine event_stream_array_final (es_array)
  class(event_stream_array_t), intent(inout) :: es_array
  integer :: i
  do i = 1, size (es_array%entry)
     call es_array%entry(i)%eio%final ()
  end do
end subroutine event_stream_array_final
```

Initialization. We use a generic `sample` name, open event I/O objects for all provided stream types (using the `dispatch_eio` routine), and initialize for the given list of process pointers. If there is an `input` argument, this channel is initialized as an input channel and appended to the array.

The `input_data` or, if not present, `data` may be modified. This happens if we open a stream for reading and get new information there.

⟨*Event streams: event stream array: TBP*⟩+≡

```
procedure :: init => event_stream_array_init
```

⟨*Event streams: procedures*⟩+≡

```
subroutine event_stream_array_init &
     (es_array, sample, stream_fmt, global, &
     data, input, input_sample, input_data, allow_switch, &
```

```
   checkpoint, callback, &
   error)
class(event_stream_array_t), intent(out) :: es_array
type(string_t), intent(in) :: sample
type(string_t), dimension(:), intent(in) :: stream_fmt
type(rt_data_t), intent(in) :: global
type(event_sample_data_t), intent(inout), optional :: data
type(string_t), intent(in), optional :: input
type(string_t), intent(in), optional :: input_sample
type(event_sample_data_t), intent(inout), optional :: input_data
logical, intent(in), optional :: allow_switch
integer, intent(in), optional :: checkpoint
integer, intent(in), optional :: callback
logical, intent(out), optional :: error
type(string_t) :: sample_in
integer :: n, i, n_output, i_input, i_checkpoint, i_callback
logical :: success, switch
if (present (input_sample)) then
   sample_in = input_sample
else
   sample_in = sample
end if
if (present (allow_switch)) then
   switch = allow_switch
else
   switch = .true.
end if
if (present (error)) then
   error = .false.
end if
n = size (stream_fmt)
n_output = n
if (present (input)) then
   n = n + 1
   i_input = n
else
   i_input = 0
end if
if (present (checkpoint)) then
   n = n + 1
   i_checkpoint = n
else
   i_checkpoint = 0
end if
if (present (callback)) then
   n = n + 1
   i_callback = n
else
   i_callback = 0
end if
allocate (es_array%entry (n))
if (i_checkpoint > 0) then
   call dispatch_eio &
        (es_array%entry(i_checkpoint)%eio, var_str ("checkpoint"), &
```

```
            global%var_list, global%fallback_model, &
            global%event_callback)
         call es_array%entry(i_checkpoint)%eio%init_out (sample, data)
      end if
      if (i_callback > 0) then
         call dispatch_eio &
            (es_array%entry(i_callback)%eio, var_str ("callback"), &
            global%var_list, global%fallback_model, &
            global%event_callback)
         call es_array%entry(i_callback)%eio%init_out (sample, data)
      end if
      if (i_input > 0) then
         call dispatch_eio (es_array%entry(i_input)%eio, input, &
            global%var_list, global%fallback_model, &
            global%event_callback)
         if (present (input_data)) then
            call es_array%entry(i_input)%eio%init_in &
               (sample_in, input_data, success)
         else
            call es_array%entry(i_input)%eio%init_in &
               (sample_in, data, success)
         end if
         if (success) then
            es_array%i_in = i_input
         else if (present (input_sample)) then
            if (present (error)) then
               error = .true.
            else
               call msg_fatal ("Events: &
                  &parameter mismatch in input, aborting")
            end if
         else
            call msg_message ("Events: &
               &parameter mismatch, discarding old event set")
            call es_array%entry(i_input)%eio%final ()
            if (switch) then
               call msg_message ("Events: generating new events")
               call es_array%entry(i_input)%eio%init_out (sample, data)
            end if
         end if
      end if
      do i = 1, n_output
         call dispatch_eio (es_array%entry(i)%eio, stream_fmt(i), &
            global%var_list, global%fallback_model, &
            global%event_callback)
         call es_array%entry(i)%eio%init_out (sample, data)
      end do
   end subroutine event_stream_array_init
```

Switch the (only) input channel to an output channel, so further events are appended to the respective stream.

⟨*Event streams: event stream array: TBP*⟩+≡

```
   procedure :: switch_inout => event_stream_array_switch_inout
```

⟨*Event streams: procedures*⟩+≡

```
subroutine event_stream_array_switch_inout (es_array)
  class(event_stream_array_t), intent(inout) :: es_array
  integer :: n
  if (es_array%has_input ()) then
     n = es_array%i_in
     call es_array%entry(n)%eio%switch_inout ()
     es_array%i_in = 0
  else
     call msg_bug ("Reading events: switch_inout: no input stream selected")
  end if
end subroutine event_stream_array_switch_inout
```

Output an event (with given process number) to all output streams. If there is no output stream, do nothing.

⟨*Event streams: event stream array: TBP*⟩+≡

```
procedure :: output => event_stream_array_output
```

⟨*Event streams: procedures*⟩+≡

```
subroutine event_stream_array_output (es_array, event, i_prc, &
                                      event_index, passed, pacify)
  class(event_stream_array_t), intent(inout) :: es_array
  type(event_t), intent(in), target :: event
  integer, intent(in) :: i_prc, event_index
  logical, intent(in), optional :: passed, pacify
  logical :: increased
  integer :: i
  do i = 1, size (es_array%entry)
     if (i /= es_array%i_in) then
        associate (eio => es_array%entry(i)%eio)
          if (eio%split) then
             if (eio%split_n_evt > 0 &
                  .and. event_index > 1 &
                  .and. mod (event_index, eio%split_n_evt) == 1) then
                call eio%split_out ()
             else if (eio%split_n_kbytes > 0) then
                call eio%update_split_count (increased)
                if (increased)  call eio%split_out ()
             end if
          end if
          call eio%output (event, i_prc, reading = es_array%i_in /= 0, &
               passed = passed, &
               pacify = pacify)
        end associate
     end if
  end do
end subroutine event_stream_array_output
```

Input the `i_prc` index which selects the process for the current event. This is separated from reading the event, because it determines which event record to read. `iostat` may indicate an error or an EOF condition, as usual.

⟨*Event streams: event stream array: TBP*⟩+≡

```
procedure :: input_i_prc => event_stream_array_input_i_prc
```

⟨*Event streams: procedures*⟩+≡

```
subroutine event_stream_array_input_i_prc (es_array, i_prc, iostat)
  class(event_stream_array_t), intent(inout) :: es_array
  integer, intent(out) :: i_prc
  integer, intent(out) :: iostat
  integer :: n
  if (es_array%has_input ()) then
     n = es_array%i_in
     call es_array%entry(n)%eio%input_i_prc (i_prc, iostat)
  else
     call msg_fatal ("Reading events: no input stream selected")
  end if
end subroutine event_stream_array_input_i_prc
```

Input an event from the selected input stream. `iostat` may indicate an error or an EOF condition, as usual.

⟨*Event streams: event stream array: TBP*⟩+≡

```
procedure :: input_event => event_stream_array_input_event
```

⟨*Event streams: procedures*⟩+≡

```
subroutine event_stream_array_input_event (es_array, event, iostat)
  class(event_stream_array_t), intent(inout) :: es_array
  type(event_t), intent(inout), target :: event
  integer, intent(out) :: iostat
  integer :: n
  if (es_array%has_input ()) then
     n = es_array%i_in
     call es_array%entry(n)%eio%input_event (event, iostat)
  else
     call msg_fatal ("Reading events: no input stream selected")
  end if
end subroutine event_stream_array_input_event
```

Skip an entry of eio_t. Used to synchronize the event read-in for NLO events.

⟨*Event streams: event stream array: TBP*⟩+≡

```
procedure :: skip_eio_entry => event_stream_array_skip_eio_entry
```

⟨*Event streams: procedures*⟩+≡

```
subroutine event_stream_array_skip_eio_entry (es_array, iostat)
  class(event_stream_array_t), intent(inout) :: es_array
  integer, intent(out) :: iostat
  integer :: n
  if (es_array%has_input ()) then
     n = es_array%i_in
     call es_array%entry(n)%eio%skip (iostat)
  else
     call msg_fatal ("Reading events: no input stream selected")
  end if
end subroutine event_stream_array_skip_eio_entry
```

Return true if there is an input channel among the event streams.

⟨*Event streams: event stream array: TBP*⟩+≡

```
procedure :: has_input => event_stream_array_has_input
```

⟨*Event streams: procedures*⟩+≡

```
function event_stream_array_has_input (es_array) result (flag)
  class(event_stream_array_t), intent(in) :: es_array
  logical :: flag
  flag = es_array%i_in /= 0
end function event_stream_array_has_input
```

33.7.2 Unit Tests

Test module, followed by the stand-alone unit-test procedures.

⟨event_streams_ut.f90⟩≡

```
⟨File header⟩

module event_streams_ut
  use unit_tests
  use event_streams_uti

⟨Standard module head⟩

⟨Event streams: public test⟩

contains

⟨Event streams: test driver⟩

end module event_streams_ut
```

⟨event_streams_uti.f90⟩≡

```
⟨File header⟩

module event_streams_uti

⟨Use kinds⟩
⟨Use strings⟩
  use model_data
  use eio_data
  use process, only: process_t
  use instances, only: process_instance_t
  use models
  use rt_data
  use events

  use event_streams

⟨Standard module head⟩

⟨Event streams: test declarations⟩

contains

⟨Event streams: tests⟩

end module event_streams_uti
```

API: driver for the unit tests below.

⟨*Event streams: public test*⟩≡
```
public :: event_streams_test
```

⟨*Event streams: test driver*⟩≡
```
subroutine event_streams_test (u, results)
  integer, intent(in) :: u
  type(test_results_t), intent(inout) :: results
⟨Event streams: execute tests⟩
end subroutine event_streams_test
```

Empty event stream

This should set up an empty event output stream array, including initialization, output, and finalization (which are all no-ops).

⟨*Event streams: execute tests*⟩≡
```
call test (event_streams_1, "event_streams_1", &
     "empty event stream array", &
     u, results)
```

⟨*Event streams: test declarations*⟩≡
```
public :: event_streams_1
```

⟨*Event streams: tests*⟩≡
```
subroutine event_streams_1 (u)
  integer, intent(in) :: u
  type(event_stream_array_t) :: es_array
  type(rt_data_t) :: global
  type(event_t) :: event
  type(string_t) :: sample
  type(string_t), dimension(0) :: empty_string_array

  write (u, "(A)")  "* Test output: event_streams_1"
  write (u, "(A)")  "*   Purpose: handle empty event stream array"
  write (u, "(A)")

  sample = "event_streams_1"

  call es_array%init (sample, empty_string_array, global)
  call es_array%output (event, 42, 1)
  call es_array%write (u)
  call es_array%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: event_streams_1"

end subroutine event_streams_1
```

Nontrivial event stream

Here we generate a trivial event and choose `raw` output as an entry in the stream array.

⟨*Event streams: execute tests*⟩+≡
```
call test (event_streams_2, "event_streams_2", &
     "nontrivial event stream array", &
     u, results)
```

⟨*Event streams: test declarations*⟩+≡
```
public :: event_streams_2
```

⟨*Event streams: tests*⟩+≡
```
subroutine event_streams_2 (u)
  use processes_ut, only: prepare_test_process
  integer, intent(in) :: u
  type(event_stream_array_t) :: es_array
  type(rt_data_t) :: global
  type(model_data_t), target :: model
  type(event_t), allocatable, target :: event
  type(process_t), allocatable, target :: process
  type(process_instance_t), allocatable, target :: process_instance
  type(string_t) :: sample
  type(string_t), dimension(0) :: empty_string_array
  integer :: i_prc, iostat

  write (u, "(A)")  "* Test output: event_streams_2"
  write (u, "(A)")  "*   Purpose: handle empty event stream array"
  write (u, "(A)")

  call syntax_model_file_init ()
  call global%global_init ()
  call global%init_fallback_model &
       (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"))

  call model%init_test ()

  write (u, "(A)")  "* Generate test process event"
  write (u, "(A)")

  allocate (process)
  allocate (process_instance)
  call prepare_test_process (process, process_instance, model)
  call process_instance%setup_event_data ()

  allocate (event)
  call event%basic_init ()
  call event%connect (process_instance, process%get_model_ptr ())
  call event%generate (1, [0.4_default, 0.4_default])
  call event%evaluate_expressions ()
  call event%write (u)

  write (u, "(A)")
  write (u, "(A)") "* Allocate raw eio stream and write event to file"
  write (u, "(A)")
```

```
    sample = "event_streams_2"

    call es_array%init (sample, [var_str ("raw")], global)
    call es_array%output (event, 1, 1)
    call es_array%write (u)
    call es_array%final ()

    write (u, "(A)")
    write (u, "(A)") "* Reallocate raw eio stream for reading"
    write (u, "(A)")

    sample = "foo"
    call es_array%init (sample, empty_string_array, global, &
         input = var_str ("raw"), input_sample = var_str ("event_streams_2"))
    call es_array%write (u)

    write (u, "(A)")
    write (u, "(A)") "* Reread event"
    write (u, "(A)")

    call es_array%input_i_prc (i_prc, iostat)

    write (u, "(1x,A,I0)")  "i_prc = ", i_prc
    write (u, "(A)")
    call es_array%input_event (event, iostat)
    call es_array%final ()

    call event%write (u)

    call global%final ()

    call model%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: event_streams_2"

  end subroutine event_streams_2
```

Switch in/out

Here we generate an event file and test switching from writing to reading when the file is exhausted.

⟨*Event streams: execute tests*⟩+≡

```
  call test (event_streams_3, "event_streams_3", &
       "switch input/output", &
       u, results)
```

⟨*Event streams: test declarations*⟩+≡

```
  public :: event_streams_3
```

⟨*Event streams: tests*⟩+≡

```
subroutine event_streams_3 (u)
  use processes_ut, only: prepare_test_process
  integer, intent(in) :: u
  type(event_stream_array_t) :: es_array
  type(rt_data_t) :: global
  type(model_data_t), target :: model
  type(event_t), allocatable, target :: event
  type(process_t), allocatable, target :: process
  type(process_instance_t), allocatable, target :: process_instance
  type(string_t) :: sample
  type(string_t), dimension(0) :: empty_string_array
  integer :: i_prc, iostat

  write (u, "(A)")  "* Test output: event_streams_3"
  write (u, "(A)")  "*   Purpose: handle in/out switching"
  write (u, "(A)")

  call syntax_model_file_init ()
  call global%global_init ()
  call global%init_fallback_model &
       (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"))

  call model%init_test ()

  write (u, "(A)")  "* Generate test process event"
  write (u, "(A)")

  allocate (process)
  allocate (process_instance)
  call prepare_test_process (process, process_instance, model)
  call process_instance%setup_event_data ()

  allocate (event)
  call event%basic_init ()
  call event%connect (process_instance, process%get_model_ptr ())
  call event%generate (1, [0.4_default, 0.4_default])
  call event%evaluate_expressions ()

  write (u, "(A)") "* Allocate raw eio stream and write event to file"
  write (u, "(A)")

  sample = "event_streams_3"

  call es_array%init (sample, [var_str ("raw")], global)
  call es_array%output (event, 1, 1)
  call es_array%write (u)
  call es_array%final ()

  write (u, "(A)")
  write (u, "(A)") "* Reallocate raw eio stream for reading"
  write (u, "(A)")

  call es_array%init (sample, empty_string_array, global, &
       input = var_str ("raw"))
```

```
call es_array%write (u)

write (u, "(A)")
write (u, "(A)") "* Reread event"
write (u, "(A)")

call es_array%input_i_prc (i_prc, iostat)
call es_array%input_event (event, iostat)

write (u, "(A)") "* Attempt to read another event (fail), then generate"
write (u, "(A)")

call es_array%input_i_prc (i_prc, iostat)
if (iostat < 0) then
   call es_array%switch_inout ()
   call event%generate (1, [0.3_default, 0.3_default])
   call event%evaluate_expressions ()
   call es_array%output (event, 1, 2)
end if
call es_array%write (u)
call es_array%final ()

write (u, "(A)")
call event%write (u)

write (u, "(A)")
write (u, "(A)") "* Reallocate raw eio stream for reading"
write (u, "(A)")

call es_array%init (sample, empty_string_array, global, &
     input = var_str ("raw"))
call es_array%write (u)

write (u, "(A)")
write (u, "(A)") "* Reread two events and display 2nd event"
write (u, "(A)")

call es_array%input_i_prc (i_prc, iostat)
call es_array%input_event (event, iostat)
call es_array%input_i_prc (i_prc, iostat)

call es_array%input_event (event, iostat)
call es_array%final ()

call event%write (u)

call global%final ()

call model%final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: event_streams_3"
```

```
end subroutine event_streams_3
```

Checksum

Here we generate an event file and repeat twice, once with identical parameters and once with modified parameters.

⟨*Event streams: execute tests*⟩+≡

```
call test (event_streams_4, "event_streams_4", &
     "check MD5 sum", &
     u, results)
```

⟨*Event streams: test declarations*⟩+≡

```
public :: event_streams_4
```

⟨*Event streams: tests*⟩+≡

```
subroutine event_streams_4 (u)
  integer, intent(in) :: u
  type(event_stream_array_t) :: es_array
  type(rt_data_t) :: global
  type(process_t), allocatable, target :: process
  type(string_t) :: sample
  type(string_t), dimension(0) :: empty_string_array
  type(event_sample_data_t) :: data

  write (u, "(A)")  "* Test output: event_streams_4"
  write (u, "(A)")  "*   Purpose: handle in/out switching"
  write (u, "(A)")

  write (u, "(A)")  "* Generate test process event"
  write (u, "(A)")

  call syntax_model_file_init ()
  call global%global_init ()
  call global%init_fallback_model &
       (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"))

  call global%set_log (var_str ("?check_event_file"), &
       .true., is_known = .true.)

  allocate (process)

  write (u, "(A)") "* Allocate raw eio stream for writing"
  write (u, "(A)")

  sample = "event_streams_4"
  data%md5sum_cfg = "1234567890abcdef1234567890abcdef"

  call es_array%init (sample, [var_str ("raw")], global, data)
  call es_array%write (u)
  call es_array%final ()

  write (u, "(A)")
  write (u, "(A)") "* Reallocate raw eio stream for reading"
```

```
  write (u, "(A)")

  call es_array%init (sample, empty_string_array, global, &
       data, input = var_str ("raw"))
  call es_array%write (u)
  call es_array%final ()

  write (u, "(A)")
  write (u, "(A)") "* Reallocate modified raw eio stream for reading (fail)"
  write (u, "(A)")

  data%md5sum_cfg = "1234567890______1234567890______"
  call es_array%init (sample, empty_string_array, global, &
       data, input = var_str ("raw"))
  call es_array%write (u)
  call es_array%final ()

  write (u, "(A)")
  write (u, "(A)") "* Repeat ignoring checksum"
  write (u, "(A)")

  call global%set_log (var_str ("?check_event_file"), &
       .false., is_known = .true.)
  call es_array%init (sample, empty_string_array, global, &
       data, input = var_str ("raw"))
  call es_array%write (u)
  call es_array%final ()

  call global%final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: event_streams_4"

end subroutine event_streams_4
```

33.8 Simulation

This module manages simulation: event generation and reading/writing of event files. The `simulation` object is intended to be used (via a pointer) outside of `WHIZARD`, if events are generated individually by an external driver.

⟨`simulations.f90`⟩≡

```
⟨File header⟩

module simulations

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use format_utils, only: write_separator
  use format_defs, only: FMT_19
  use numeric_utils
  use diagnostics
  use sm_qcd
  use md5
  use variables, only: var_list_t
  use eval_trees
  use model_data
  use flavors
  use particles
  use state_matrices, only: FM_IGNORE_HELICITY
  use beam_structures, only: beam_structure_t
  use beams
  use rng_base
  use selectors
  use process_libraries, only: process_library_t
  use prc_core
  !  TODO: (bcn 2016-09-13) should be ideally only pcm_base
  use pcm, only: pcm_nlo_t, pcm_instance_nlo_t
  !  TODO: (bcn 2016-09-13) details of process config should not be necessary here
  use process_config, only: COMP_REAL_FIN
  use process
  use instances
  use event_base
  use events
  use event_transforms
  use shower
  use eio_data
  use eio_base
  use rt_data

  use dispatch_beams, only: dispatch_qcd
  use dispatch_rng, only: dispatch_rng_factory
  use dispatch_me_methods, only: dispatch_core_update, dispatch_core_restore
  use dispatch_transforms, only: dispatch_evt_decay
  use dispatch_transforms, only: dispatch_evt_shower
  use dispatch_transforms, only: dispatch_evt_hadrons
  use dispatch_transforms, only: dispatch_evt_nlo

  use integrations
```

```
  use event_streams

  use evt_nlo
```

⟨*Standard module head*⟩

⟨*Simulations: public*⟩

⟨*Simulations: types*⟩

⟨*Simulations: interfaces*⟩

```
contains
```

⟨*Simulations: procedures*⟩

```
end module simulations
```

33.8.1 Event counting

In this object we collect statistical information about an event sample or subsample.

⟨*Simulations: types*⟩≡

```
  type :: counter_t
     integer :: total = 0
     integer :: generated = 0
     integer :: read = 0
     integer :: positive = 0
     integer :: negative = 0
     integer :: zero = 0
     integer :: excess = 0
     real(default) :: max_excess = 0
     real(default) :: sum_excess = 0
     logical :: reproduce_xsection = .false.
     real(default) :: mean = 0
     real(default) :: varsq = 0
     integer :: nlo_weight_counter = 0
   contains
   ⟨Simulations: counter: TBP⟩
  end type counter_t
```

Output.

⟨*Simulations: counter: TBP*⟩≡

```
  procedure :: write => counter_write
```

⟨*Simulations: procedures*⟩≡

```
  subroutine counter_write (counter, unit)
    class(counter_t), intent(in) :: counter
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
1   format (3x,A,I0)
2   format (5x,A,I0)
```

```
3   format (5x,A,ES19.12)
    write (u, 1)  "Events total      = ", counter%total
    write (u, 2)  "generated       = ", counter%generated
    write (u, 2)  "read            = ", counter%read
    write (u, 2)  "positive weight = ", counter%positive
    write (u, 2)  "negative weight = ", counter%negative
    write (u, 2)  "zero weight     = ", counter%zero
    write (u, 2)  "excess weight   = ", counter%excess
    if (counter%excess /= 0) then
       write (u, 3)  "max excess      = ", counter%max_excess
       write (u, 3)  "avg excess      = ", counter%sum_excess / counter%total
    end if
  end subroutine counter_write
```

This is a screen message: if there was an excess, display statistics.

⟨*Simulations: counter: TBP*⟩+≡

```
  procedure :: show_excess => counter_show_excess
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine counter_show_excess (counter)
    class(counter_t), intent(in) :: counter
    if (counter%excess > 0) then
       write (msg_buffer, "(A,1x,I0,1x,A,1x,'(',F7.3,' %)')") &
            "Encountered events with excess weight:", counter%excess, &
            "events", 100 * counter%excess / real (counter%total)
       call msg_warning ()
       write (msg_buffer, "(A,ES10.3)") &
            "Maximum excess weight =", counter%max_excess
       call msg_message ()
       write (msg_buffer, "(A,ES10.3)") &
            "Average excess weight =", counter%sum_excess / counter%total
       call msg_message ()
    end if
  end subroutine counter_show_excess
```

⟨*Simulations: counter: TBP*⟩+≡

```
  procedure :: show_mean_and_variance => counter_show_mean_and_variance
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine counter_show_mean_and_variance (counter)
    class(counter_t), intent(in) :: counter
    if (counter%reproduce_xsection .and. counter%nlo_weight_counter > 1) then
       print *,  "Reconstructed cross-section from event weights: "
       print *,  counter%mean, '+-', sqrt (counter%varsq / (counter%nlo_weight_counter - 1))
    end if
  end subroutine counter_show_mean_and_variance
```

Count an event. The weight and event source are optional; by default we assume that the event has been generated and has positive weight.

⟨*Simulations: counter: TBP*⟩+≡

```
  procedure :: record => counter_record
```

⟨*Simulations: procedures*⟩+≡

```
subroutine counter_record (counter, weight, excess, from_file)
  class(counter_t), intent(inout) :: counter
  real(default), intent(in), optional :: weight, excess
  logical, intent(in), optional :: from_file
  counter%total = counter%total + 1
  if (present (from_file)) then
     if (from_file) then
        counter%read = counter%read + 1
     else
        counter%generated = counter%generated + 1
     end if
  else
     counter%generated = counter%generated + 1
  end if
  if (present (weight)) then
     if (weight > 0) then
        counter%positive = counter%positive + 1
     else if (weight < 0) then
        counter%negative = counter%negative + 1
     else
        counter%zero = counter%zero + 1
     end if
  else
     counter%positive = counter%positive + 1
  end if
  if (present (excess)) then
     if (excess > 0) then
        counter%excess = counter%excess + 1
        counter%max_excess = max (counter%max_excess, excess)
        counter%sum_excess = counter%sum_excess + excess
     end if
  end if
end subroutine counter_record
```

⟨*Simulations: counter: TBP*⟩+≡

```
procedure :: record_mean_and_variance => &
   counter_record_mean_and_variance
```

⟨*Simulations: procedures*⟩+≡

```
subroutine counter_record_mean_and_variance (counter, weight, i_nlo)
  class(counter_t), intent(inout) :: counter
  real(default), intent(in) :: weight
  integer, intent(in) :: i_nlo
  real(default), save :: weight_buffer = 0._default
  integer, save :: nlo_count = 1
  if (.not. counter%reproduce_xsection) return
  if (i_nlo == 1) then
     call flush_weight_buffer (weight_buffer, nlo_count)
     weight_buffer = weight
     nlo_count = 1
  else
     weight_buffer = weight_buffer + weight
     nlo_count = nlo_count + 1
```

```
    end if
  contains
    subroutine flush_weight_buffer (w, n_nlo)
      real(default), intent(in) :: w
      integer, intent(in) :: n_nlo
      integer :: n
      real(default) :: mean_new
      counter%nlo_weight_counter = counter%nlo_weight_counter + 1
      !!! Minus 1 to take into account offset from initialization
      n = counter%nlo_weight_counter - 1
      if (n > 0) then
         mean_new = counter%mean + (w / n_nlo - counter%mean) / n
         if (n > 1) &
            counter%varsq = counter%varsq - counter%varsq / (n - 1) + &
               n * (mean_new - counter%mean)**2
         counter%mean = mean_new
      end if
    end subroutine flush_weight_buffer
  end subroutine counter_record_mean_and_variance
```

33.8.2 Simulation: component sets

For each set of process components that share a MCI entry in the process configuration, we keep a separate event record.

⟨*Simulations: types*⟩+≡

```
  type :: mci_set_t
     private
     integer :: n_components = 0
     integer, dimension(:), allocatable :: i_component
     type(string_t), dimension(:), allocatable :: component_id
     logical :: has_integral = .false.
     real(default) :: integral = 0
     real(default) :: error = 0
     real(default) :: weight_mci = 0
     type(counter_t) :: counter
   contains
   ⟨Simulations: mci set: TBP⟩
  end type mci_set_t
```

Output.

⟨*Simulations: mci set: TBP*⟩≡

```
  procedure :: write => mci_set_write
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine mci_set_write (object, unit)
    class(mci_set_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u, i
    u = given_output_unit (unit)
    write (u, "(3x,A)")  "Components:"
    do i = 1, object%n_components
       write (u, "(5x,I0,A,A,A)")  object%i_component(i), &
```

```
          ": '", char (object%component_id(i)), "'"
    end do
    if (object%has_integral) then
       write (u, "(3x,A," // FMT_19 // ")")  "Integral  = ", object%integral
       write (u, "(3x,A," // FMT_19 // ")")  "Error     = ", object%error
       write (u, "(3x,A,F13.10)")  "Weight    =", object%weight_mci
    else
       write (u, "(3x,A)")  "Integral  = [undefined]"
    end if
    call object%counter%write (u)
  end subroutine mci_set_write
```

Initialize: Get the indices and names for the process components that will contribute to this set.

⟨*Simulations: mci set: TBP*⟩+≡

```
  procedure :: init => mci_set_init
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine mci_set_init (object, i_mci, process)
    class(mci_set_t), intent(out) :: object
    integer, intent(in) :: i_mci
    type(process_t), intent(in), target :: process
    integer :: i
    call process%get_i_component (i_mci, object%i_component)
    object%n_components = size (object%i_component)
    allocate (object%component_id (object%n_components))
    do i = 1, size (object%component_id)
       object%component_id(i) = &
            process%get_component_id (object%i_component(i))
    end do
    if (process%has_integral (i_mci)) then
       object%integral = process%get_integral (i_mci)
       object%error = process%get_error (i_mci)
       object%has_integral = .true.
    end if
  end subroutine mci_set_init
```

33.8.3 Process-core Safe

This is an object that temporarily holds a process core object. We need this while rescanning a process with modified parameters. After the rescan, we want to restore the original state.

⟨*Simulations: types*⟩+≡

```
  type :: core_safe_t
     class(prc_core_t), allocatable :: core
  end type core_safe_t
```

33.8.4 Process Object

The simulation works on process objects. This subroutine makes a process object available for simulation. The process is in the process stack. If integration

is not yet done, do it. Any generated process object should be put on the global stack, if it is separate from the local one.

⟨*Simulations: procedures*⟩+≡

```
subroutine prepare_process &
     (process, process_id, use_process, integrate, local, global)
  type(process_t), pointer, intent(out) :: process
  type(string_t), intent(in) :: process_id
  logical, intent(in) :: use_process, integrate
  type(rt_data_t), intent(inout), target :: local
  type(rt_data_t), intent(inout), optional, target :: global
  call msg_debug (D_CORE, "prepare_process")
  call msg_debug (D_CORE, "global present", present (global))
  if (present (global)) then
     process => global%process_stack%get_process_ptr (process_id)
  else
     process => local%process_stack%get_process_ptr (process_id)
  end if
  call msg_debug (D_CORE, "use_process", use_process)
  call msg_debug (D_CORE, "associated process", associated (process))
  if (use_process .and. .not. associated (process)) then
     if (integrate) then
        call msg_message ("Simulate: process '" &
             // char (process_id) // "' needs integration")
     else
        call msg_message ("Simulate: process '" &
             // char (process_id) // "' needs initialization")
     end if
     if (present (global)) then
        call integrate_process (process_id, local, global, &
          init_only = .not. integrate)
     else
        call integrate_process (process_id, local, &
             local_stack = .true., init_only = .not. integrate)
     end if
     if (signal_is_pending ())  return
     process => global%process_stack%get_process_ptr (process_id)
     if (associated (process)) then
        if (integrate) then
           call msg_message ("Simulate: integration done")
           call global%process_stack%fill_result_vars (process_id)
        else
           call msg_message ("Simulate: process initialization done")
        end if
     else
        call msg_fatal ("Simulate: process '" &
             // char (process_id) // "' could not be initialized: aborting")
     end if
  else if (.not. associated (process)) then
     if (present (global)) then
        call integrate_process (process_id, local, global, &
             init_only = .true.)
     else
        call integrate_process (process_id, local, &
             local_stack = .true., init_only = .true.)
```

```
      end if
      process => global%process_stack%get_process_ptr (process_id)
      call msg_message &
           ("Simulate: process '" &
              // char (process_id) // "': enabled for rescan only")
  end if
end subroutine prepare_process
```

33.8.5 Simulation entry

For each process that we consider for event generation, we need a separate entry. The entry separately records the process ID and run ID. The `weight_mci` array is used for selecting a component set (which shares a MCI record inside the process container) when generating an event for the current process.

The simulation entry is an extension of the `event_t` event record. This core object contains configuration data, pointers to the process and process instance, the expressions, flags and values that are evaluated at runtime, and the resulting particle set.

The entry explicitly allocate the `process_instance`, which becomes the process-specific workspace for the event record.

If entries with differing environments are present simultaneously, we may need to switch QCD parameters and/or the model event by event. In this case, the `qcd` and/or `model` components are present.
For the puropose of NLO events, `entry_t` contains a pointer list to other simulation-entries. This is due to the fact that we have to associate an event for each component of the fixed order simulation, i.e. one N-particle event and N_α $N+1$-particle events. However, all entries share the same event transforms.

⟨*Simulations: types*⟩+≡

```
type, extends (event_t) :: entry_t
   private
   type(string_t) :: process_id
   type(string_t) :: library
   type(string_t) :: run_id
   logical :: has_integral = .false.
   real(default) :: integral = 0
   real(default) :: error = 0
   real(default) :: process_weight = 0
   logical :: valid = .false.
   type(counter_t) :: counter
   integer :: n_in = 0
   integer :: n_mci = 0
   type(mci_set_t), dimension(:), allocatable :: mci_sets
   type(selector_t) :: mci_selector
   type(core_safe_t), dimension(:), allocatable :: core_safe
   class(model_data_t), pointer :: model => null ()
   type(qcd_t) :: qcd
   type(entry_t), pointer :: first => null ()
   type(entry_t), pointer :: next => null ()
   class(evt_t), pointer :: evt_powheg => null ()
   logical :: requires_real_switch_off = .true.
 contains
```

```
  ⟨Simulations: entry: TBP⟩
end type entry_t
```

Output. Write just the configuration, the event is written by a separate routine.

The `verbose` option is unused, it is required by the interface of the base-object method.

⟨*Simulations: entry: TBP*⟩≡

```
procedure :: write_config => entry_write_config
```

⟨*Simulations: procedures*⟩+≡

```
subroutine entry_write_config (object, unit)
  class(entry_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  write (u, "(3x,A,A,A)")  "Process   = '", char (object%process_id), "'"
  write (u, "(3x,A,A,A)")  "Library   = '", char (object%library), "'"
  write (u, "(3x,A,A,A)")  "Run       = '", char (object%run_id), "'"
  write (u, "(3x,A,L1)")   "is valid  = ", object%valid
  if (object%has_integral) then
     write (u, "(3x,A," // FMT_19 // ")")  "Integral  = ", object%integral
     write (u, "(3x,A," // FMT_19 // ")")  "Error     = ", object%error
     write (u, "(3x,A,F13.10)")  "Weight    =", object%process_weight
  else
     write (u, "(3x,A)")  "Integral  = [undefined]"
  end if
  write (u, "(3x,A,I0)")   "MCI sets  = ", object%n_mci
  call object%counter%write (u)
  do i = 1, size (object%mci_sets)
     write (u, "(A)")
     write (u, "(1x,A,I0,A)")  "MCI set #", i, ":"
     call object%mci_sets(i)%write (u)
  end do
  if (allocated (object%core_safe)) then
     do i = 1, size (object%core_safe)
        write (u, "(1x,A,I0,A)")  "Saved process-component core #", i, ":"
        call object%core_safe(i)%core%write (u)
     end do
  end if
end subroutine entry_write_config
```

Finalizer. The `instance` pointer component of the `event_t` base type points to a target which we did explicitly allocate in the `entry_init` procedure. Therefore, we finalize and explicitly deallocate it here. Then we call the finalizer of the base type.

⟨*Simulations: entry: TBP*⟩+≡

```
procedure :: final => entry_final
```

⟨*Simulations: procedures*⟩+≡

```
subroutine entry_final (object)
  class(entry_t), intent(inout) :: object
  integer :: i
  if (associated (object%instance)) then
```

```
      do i = 1, object%n_mci
         call object%instance%final_simulation (i)
      end do
      call object%instance%final ()
      deallocate (object%instance)
   end if
   call object%event_t%final ()
end subroutine entry_final
```

Copy the content of an entry into another one, except for the next-pointer

⟨*Simulations: entry: TBP*⟩+≡

```
procedure :: copy_entry => entry_copy_entry
```

⟨*Simulations: procedures*⟩+≡

```
subroutine entry_copy_entry (entry1, entry2)
  class(entry_t), intent(in), target :: entry1
  type(entry_t), intent(inout), target :: entry2
  call entry1%event_t%clone (entry2%event_t)
  entry2%process_id = entry1%process_id
  entry2%library = entry1%library
  entry2%run_id = entry1%run_id
  entry2%has_integral = entry1%has_integral
  entry2%integral = entry1%integral
  entry2%error = entry1%error
  entry2%process_weight = entry1%process_weight
  entry2%valid = entry1%valid
  entry2%counter = entry1%counter
  entry2%n_in = entry1%n_in
  entry2%n_mci = entry1%n_mci
  if (allocated (entry1%mci_sets)) then
     allocate (entry2%mci_sets (size (entry1%mci_sets)))
     entry2%mci_sets = entry1%mci_sets
  end if
  entry2%mci_selector = entry1%mci_selector
  if (allocated (entry1%core_safe)) then
     allocate (entry2%core_safe (size (entry1%core_safe)))
     entry2%core_safe = entry1%core_safe
  end if
  entry2%model => entry1%model
  entry2%qcd = entry1%qcd
end subroutine entry_copy_entry
```

Initialization. Search for a process entry and allocate a process instance as an anonymous object, temporarily accessible via the `process_instance` pointer. Assign data by looking at the process object and at the environment.

If `n_alt` is set, we prepare for additional alternate sqme and weight entries.

The `compile` flag is only false if we don't need the Whizard process at all, just its definition. In that case, we skip process initialization.

Otherwise, and if the process object is not found initially: if `integrate` is set, attempt an integration pass and try again. Otherwise, just initialize the object.

If `generate` is set, prepare the MCI objects for generating new events. For pure rescanning, this is not necessary.

When done, we assign the `instance` and `process` pointers of the base type by the `connect` method, so we can reference them later.

⟨*Simulations: entry: TBP*⟩+≡

```
procedure :: init => entry_init
```

⟨*Simulations: procedures*⟩+≡

```
subroutine entry_init &
     (entry, process_id, &
     use_process, integrate, generate, update_sqme, &
     local, global, n_alt)
  class(entry_t), intent(inout), target :: entry
  type(string_t), intent(in) :: process_id
  logical, intent(in) :: use_process, integrate, generate, update_sqme
  type(rt_data_t), intent(inout), target :: local
  type(rt_data_t), intent(inout), optional, target :: global
  integer, intent(in), optional :: n_alt
  type(process_t), pointer :: process, master_process
  type(process_instance_t), pointer :: process_instance
  integer :: i
  logical :: combined_integration
  integer :: fixed_mci = 0
  call msg_debug (D_CORE, "entry_init")
  call msg_debug (D_CORE, "process_id", process_id)
  call prepare_process &
       (master_process, process_id, use_process, integrate, local, global)
  if (signal_is_pending ())  return

  if (associated (master_process)) then
     if (.not. master_process%has_matrix_element ()) then
        entry%has_integral = .true.
        entry%process_id = process_id
        entry%valid = .false.
        return
     end if
  else
     call entry%basic_init (local%var_list)
     entry%has_integral = .false.
     entry%process_id = process_id
     call entry%import_process_def_characteristics (local%prclib, process_id)
     entry%valid = .true.
     return
  end if

  call entry%basic_init (local%var_list, n_alt)

  entry%process_id = process_id
  if (generate .or. integrate) then
     entry%run_id = master_process%get_run_id ()
     process => master_process
  else
     call local%set_log (var_str ("?rebuild_phase_space"), &
          .false., is_known = .true.)
     call local%set_log (var_str ("?check_phs_file"), &
          .false., is_known = .true.)
```

```
   call local%set_log (var_str ("?rebuild_grids"), &
        .false., is_known = .true.)
   entry%run_id = &
        local%var_list%get_sval (var_str ("$run_id"))
   if (update_sqme) then
      call prepare_local_process (process, process_id, local)
   else
      process => master_process
   end if
end if

call entry%import_process_characteristics (process)

allocate (entry%mci_sets (entry%n_mci))
do i = 1, size (entry%mci_sets)
   call entry%mci_sets(i)%init (i, master_process)
end do

call entry%import_process_results (master_process)
call entry%prepare_expressions (local)

if (process%is_nlo_calculation ()) call process%init_nlo_settings (global%var_list)
combined_integration = local%get_lval (var_str ("?combined_nlo_integration"))
if (.not. combined_integration &
   .and. local%get_lval (var_str ("?fixed_order_nlo_events"))) then
   fixed_mci = process%extract_fixed_mci ()
end if
call prepare_process_instance (process_instance, process, local%model, &
     local = local)

if (generate) then
   if (fixed_mci > 0) then
      call process%prepare_simulation (fixed_mci)
      call process_instance%init_simulation &
         (fixed_mci, entry%config%safety_factor, &
          local%get_lval (var_str ("?keep_failed_events")))
   else
      do i = 1, entry%n_mci
         call process%prepare_simulation (i)
         call process_instance%init_simulation (i, entry%config%safety_factor, &
            local%get_lval (var_str ("?keep_failed_events")))
      end do
   end if
end if
call entry%setup_event_transforms (process, local)
call dispatch_qcd (entry%qcd, local%get_var_list_ptr (), local%os_data)

call entry%connect_qcd ()

select type (pcm => process_instance%pcm)
class is (pcm_instance_nlo_t)
    select type (config => pcm%config)
    type is (pcm_nlo_t)
       if (config%settings%fixed_order_nlo) &
```

```
            call pcm%set_fixed_order_event_mode ()
         end select
      end select

      if (present (global)) then
         call entry%connect (process_instance, local%model, global%process_stack)
      else
         call entry%connect (process_instance, local%model, local%process_stack)
      end if
      call entry%setup_expressions ()

      entry%model => process%get_model_ptr ()
      entry%valid = .true.

   end subroutine entry_init
```

⟨*Simulations: entry: TBP*⟩+≡

```
   procedure :: set_active_real_components => entry_set_active_real_components
```

⟨*Simulations: procedures*⟩+≡

```
   subroutine entry_set_active_real_components (entry, i_mci)
      class(entry_t), intent(inout) :: entry
      integer, intent(in) :: i_mci
      integer :: i
      if (.not. entry%requires_real_switch_off) return
      select type (pcm => entry%instance%pcm)
      class is (pcm_instance_nlo_t)
         i = pcm%active_real_component
         if (associated (entry%evt_powheg)) then
            select type (evt => entry%evt_powheg)
            type is (evt_shower_t)
               if (entry%process%get_component_type(i) == COMP_REAL_FIN) then
                  call evt%disable_powheg_matching ()
               else
                  call evt%enable_powheg_matching ()
               end if
            class default
               call msg_fatal ("powheg-evt should be evt_shower_t!")
            end select
         end if
      end select
      if (entry%is_nlo () .and. .not. entry%process%is_combined_nlo_integration ()) then
         if (entry%process%extract_fixed_mci () > 0) then
            call entry%process%deactivate_components &
                 (entry%process%extract_fixed_mci ())
         else
            call entry%process%deactivate_real_component ()
         end if
      end if
      entry%requires_real_switch_off = .false.
   end subroutine entry_set_active_real_components
```

Part of simulation-entry initialization: set up a process object for local use.

⟨*Simulations: procedures*⟩+≡

```
subroutine prepare_local_process (process, process_id, local)
  type(process_t), pointer, intent(inout) :: process
  type(string_t), intent(in) :: process_id
  type(rt_data_t), intent(inout), target :: local
  type(integration_t) :: intg
  call intg%create_process (process_id)
  call intg%init_process (local)
  call intg%setup_process (local, verbose=.false.)
  process => intg%get_process_ptr ()
end subroutine prepare_local_process
```

Part of simulation-entry initialization: set up a process instance matching the selected process object.

The model that we can provide as an extra argument can modify particle settings (polarization) in the density matrices that will be constructed. It does not affect parameters.

⟨*Simulations: procedures*⟩+≡

```
subroutine prepare_process_instance &
  (process_instance, process, model, local)
  type(process_instance_t), pointer, intent(inout) :: process_instance
  type(process_t), intent(inout), target :: process
  class(model_data_t), intent(in), optional :: model
  type(rt_data_t), intent(in), optional, target :: local
  integer :: i_component
  allocate (process_instance)
  call process_instance%init (process)
  if (process%is_nlo_calculation ()) then
     select type (pcm => process_instance%pcm)
     type is (pcm_instance_nlo_t)
        select type (config => pcm%config)
        type is (pcm_nlo_t)
           if (.not. config%settings%combined_integration) &
                call pcm%set_radiation_event ()
        end select
     end select
     if (process%needs_extra_code () .and. present (local)) then
           call process%create_and_load_extra_libraries &
                (local%beam_structure, process%get_var_list_ptr (), &
                local%os_data)
        call process_instance%setup_blha_helicities ()
     end if
  end if
  call process_instance%setup_event_data (model)
end subroutine prepare_process_instance
```

Part of simulation-entry initialization: query the process for basic information.

⟨*Simulations: entry: TBP*⟩+≡

```
procedure, private :: import_process_characteristics &
     => entry_import_process_characteristics
```

⟨*Simulations: procedures*⟩+≡

```
subroutine entry_import_process_characteristics (entry, process)
  class(entry_t), intent(inout) :: entry
```

```
    type(process_t), intent(in), target :: process
    entry%library = process%get_library_name ()
    entry%n_in = process%get_n_in ()
    entry%n_mci = process%get_n_mci ()
  end subroutine entry_import_process_characteristics
```

This is the alternative form which applies if there is no process entry, but just a process definition which we take from the provided `prclib` definition library.

⟨*Simulations: entry: TBP*⟩+≡

```
  procedure, private :: import_process_def_characteristics &
       => entry_import_process_def_characteristics
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine entry_import_process_def_characteristics (entry, prclib, id)
    class(entry_t), intent(inout) :: entry
    type(process_library_t), intent(in), target :: prclib
    type(string_t), intent(in) :: id
    entry%library = prclib%get_name ()
    entry%n_in = prclib%get_n_in (id)
  end subroutine entry_import_process_def_characteristics
```

Part of simulation-entry initialization: query the process for integration results.

⟨*Simulations: entry: TBP*⟩+≡

```
  procedure, private :: import_process_results &
       => entry_import_process_results
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine entry_import_process_results (entry, process)
    class(entry_t), intent(inout) :: entry
    type(process_t), intent(in), target :: process
    if (process%has_integral ()) then
       entry%integral = process%get_integral ()
       entry%error = process%get_error ()
       call entry%set_sigma (entry%integral)
       entry%has_integral = .true.
    end if
  end subroutine entry_import_process_results
```

Part of simulation-entry initialization: create expression factory objects and store them.

⟨*Simulations: entry: TBP*⟩+≡

```
  procedure, private :: prepare_expressions &
       => entry_prepare_expressions
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine entry_prepare_expressions (entry, local)
    class(entry_t), intent(inout) :: entry
    type(rt_data_t), intent(in), target :: local
    type(eval_tree_factory_t) :: expr_factory
    call expr_factory%init (local%pn%selection_lexpr)
    call entry%set_selection (expr_factory)
    call expr_factory%init (local%pn%reweight_expr)
    call entry%set_reweight (expr_factory)
```

```
    call expr_factory%init (local%pn%analysis_lexpr)
    call entry%set_analysis (expr_factory)
  end subroutine entry_prepare_expressions
```

Initializes the list of additional NLO entries. The routine gets the information about how many entries to associate from `region_data`.

⟨*Simulations: entry: TBP*⟩+≡

```
  procedure :: setup_additional_entries => entry_setup_additional_entries
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine entry_setup_additional_entries (entry)
    class(entry_t), intent(inout), target :: entry
    type(entry_t), pointer :: current_entry
    integer :: i, n_phs
    type(evt_nlo_t), pointer :: evt
    integer :: mode
    evt => null ()
    select type (pcm => entry%instance%pcm)
    class is (pcm_instance_nlo_t)
       select type (config => pcm%config)
       type is (pcm_nlo_t)
          n_phs = config%region_data%n_phs
       end select
    end select
    select type (entry)
    type is (entry_t)
       current_entry => entry
       current_entry%first => entry
       call get_nlo_evt_ptr (current_entry, evt, mode)
       if (mode > EVT_NLO_SEPARATE_BORNLIKE) then
          allocate (evt%particle_set_radiated (n_phs + 1))
          evt%event_deps%n_phs = n_phs
          evt%qcd => entry%qcd
          do i = 1, n_phs
             allocate (current_entry%next)
             current_entry%next%first => current_entry%first
             current_entry => current_entry%next
             call entry%copy_entry (current_entry)
             current_entry%i_event = i
          end do
       else
          allocate (evt%particle_set_radiated (1))
       end if
    end select
  contains
    subroutine get_nlo_evt_ptr (entry, evt, mode)
      type(entry_t), intent(in), target :: entry
      type(evt_nlo_t), intent(out), pointer :: evt
      integer, intent(out) :: mode
      class(evt_t), pointer :: current_evt
      evt => null ()
      current_evt => entry%transform_first
      do
         select type (current_evt)
```

```
            type is (evt_nlo_t)
               evt => current_evt
               mode = evt%mode
               exit
            end select
            if (associated (current_evt%next)) then
               current_evt => current_evt%next
            else
               call msg_fatal ("evt_nlo not in list of event transforms")
            end if
         end do
      end subroutine get_nlo_evt_ptr
   end subroutine entry_setup_additional_entries
```

⟨*Simulations: entry: TBP*⟩+≡

```
   procedure :: get_first => entry_get_first
```

⟨*Simulations: procedures*⟩+≡

```
   function entry_get_first (entry) result (entry_out)
      class(entry_t), intent(in), target :: entry
      type(entry_t), pointer :: entry_out
      entry_out => null ()
      select type (entry)
      type is (entry_t)
         if (entry%is_nlo ()) then
            entry_out => entry%first
         else
            entry_out => entry
         end if
      end select
   end function entry_get_first
```

⟨*Simulations: entry: TBP*⟩+≡

```
   procedure :: get_next => entry_get_next
```

⟨*Simulations: procedures*⟩+≡

```
   function entry_get_next (entry) result (next_entry)
      class(entry_t), intent(in) :: entry
      type(entry_t), pointer :: next_entry
      next_entry => null ()
      if (associated (entry%next)) then
         next_entry => entry%next
      else
         call msg_fatal ("Get next entry: No next entry")
      end if
   end function entry_get_next
```

⟨*Simulations: entry: TBP*⟩+≡

```
   procedure :: count_nlo_entries => entry_count_nlo_entries
```

⟨*Simulations: procedures*⟩+≡

```
   function entry_count_nlo_entries (entry) result (n)
      class(entry_t), intent(in), target :: entry
      integer :: n
```

```
    type(entry_t), pointer :: current_entry
    n = 1
    if (.not. associated (entry%next)) then
       return
    else
       current_entry => entry%next
       do
          n = n + 1
          if (.not. associated (current_entry%next)) exit
          current_entry => current_entry%next
       end do
    end if
  end function entry_count_nlo_entries
```

⟨*Simulations: entry: TBP*⟩+≡

```
  procedure :: reset_nlo_counter => entry_reset_nlo_counter
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine entry_reset_nlo_counter (entry)
    class(entry_t), intent(inout) :: entry
    class(evt_t), pointer :: evt
    evt => entry%transform_first
    do
       select type (evt)
       type is (evt_nlo_t)
          evt%i_evaluation = 0
          exit
       end select
       if (associated (evt%next)) evt => evt%next
   end do
  end subroutine entry_reset_nlo_counter
```

⟨*Simulations: entry: TBP*⟩+≡

```
  procedure :: determine_if_powheg_matching => entry_determine_if_powheg_matching
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine entry_determine_if_powheg_matching (entry)
     class(entry_t), intent(inout) :: entry
     class(evt_t), pointer :: current_transform
     if (associated (entry%transform_first)) then
        current_transform => entry%transform_first
        do
           select type (current_transform)
           type is (evt_shower_t)
              if (current_transform%contains_powheg_matching ()) &
                  entry%evt_powheg => current_transform
              exit
           end select
           if (associated (current_transform%next)) then
              current_transform => current_transform%next
           else
              exit
           end if
        end do
```

```
      end if
   end subroutine entry_determine_if_powheg_matching
```

Part of simulation-entry initialization: dispatch event transforms (decay, shower) as requested.

⟨*Simulations: entry: TBP*⟩+≡

```
   procedure, private :: setup_event_transforms &
        => entry_setup_event_transforms
```

⟨*Simulations: procedures*⟩+≡

```
   subroutine entry_setup_event_transforms (entry, process, local)
     class(entry_t), intent(inout) :: entry
     type(process_t), intent(inout), target :: process
     type(rt_data_t), intent(in), target :: local
     class(evt_t), pointer :: evt
     type(var_list_t), pointer :: var_list
     logical :: enable_fixed_order, enable_shower
     var_list => local%get_var_list_ptr ()
     if (process%contains_unstable (local%model)) then
        call dispatch_evt_decay (evt, local%var_list)
        if (associated (evt))  call entry%import_transform (evt)
     end if
     enable_fixed_order = local%get_lval (var_str ("?fixed_order_nlo_events"))
     if (enable_fixed_order) then
        if (local%get_lval (var_str ("?unweighted"))) &
           call msg_fatal ("NLO Fixed Order events have to be generated with &
                           &?unweighted = false")
        call dispatch_evt_nlo (evt, local%get_lval (var_str ("?keep_failed_events")))
        call entry%import_transform (evt)
     end if
     enable_shower = local%get_lval (var_str ("?allow_shower")) .and. &
             (local%get_lval (var_str ("?ps_isr_active")) &
             .or. local%get_lval (var_str ("?ps_fsr_active")) &
             .or. local%get_lval (var_str ("?muli_active")) &
             .or. local%get_lval (var_str ("?mlm_matching")) &
             .or. local%get_lval (var_str ("?ckkw_matching")) &
             .or. local%get_lval (var_str ("?powheg_matching")))
     if (enable_shower) then
        call dispatch_evt_shower (evt, var_list, local%model, &
             local%fallback_model, local%os_data, local%beam_structure, &
             process)
        call entry%import_transform (evt)
     end if
     if (local%get_lval (var_str ("?hadronization_active"))) then
        call dispatch_evt_hadrons (evt, var_list, local%fallback_model)
        call entry%import_transform (evt)
     end if
   end subroutine entry_setup_event_transforms
```

Compute weights. The integral in the argument is the sum of integrals for all processes in the sample. After computing the process weights, we repeat the normalization procedure for the process components.

⟨*Simulations: entry: TBP*⟩+≡

```
procedure :: init_mci_selector => entry_init_mci_selector
```

⟨*Simulations: procedures*⟩+≡

```
subroutine entry_init_mci_selector (entry, negative_weights)
  class(entry_t), intent(inout), target :: entry
  logical, intent(in), optional :: negative_weights
  type(entry_t), pointer :: current_entry
  integer :: i, j, k
  call msg_debug (D_CORE, "entry_init_mci_selector")
  if (entry%has_integral) then
     select type (entry)
     type is (entry_t)
        current_entry => entry
        do j = 1, current_entry%count_nlo_entries ()
           if (j > 1) current_entry => current_entry%get_next ()
           do k = 1, size(current_entry%mci_sets%integral)
              call msg_debug (D_CORE, "current_entry%mci_sets(k)%integral", &
                   current_entry%mci_sets(k)%integral)
           end do
           call current_entry%mci_selector%init &
                (current_entry%mci_sets%integral, negative_weights)
           do i = 1, current_entry%n_mci
              current_entry%mci_sets(i)%weight_mci = &
                 current_entry%mci_selector%get_weight (i)
           end do
        end do
     end select
  end if
end subroutine entry_init_mci_selector
```

Select a MCI entry, using the embedded random-number generator.

⟨*Simulations: entry: TBP*⟩+≡

```
procedure :: select_mci => entry_select_mci
```

⟨*Simulations: procedures*⟩+≡

```
function entry_select_mci (entry) result (i_mci)
  class(entry_t), intent(inout) :: entry
  integer :: i_mci
  call msg_debug2 (D_CORE, "entry_select_mci")
  if (entry%process%extract_fixed_mci () > 0) then
     i_mci = entry%process%extract_fixed_mci ()
  else
     call entry%mci_selector%generate (entry%rng, i_mci)
  end if
  call msg_debug2 (D_CORE, "i_mci", i_mci)
end function entry_select_mci
```

Record an event for this entry, i.e., increment the appropriate counters.

⟨*Simulations: entry: TBP*⟩+≡

```
procedure :: record => entry_record
```

⟨*Simulations: procedures*⟩+≡

```
subroutine entry_record (entry, i_mci, from_file)
  class(entry_t), intent(inout) :: entry
```

```
    integer, intent(in) :: i_mci
    logical, intent(in), optional :: from_file
    real(default) :: weight, excess
    weight = entry%get_weight_prc ()
    excess = entry%get_excess_prc ()
    call entry%counter%record (weight, excess, from_file)
    if (i_mci > 0) then
       call entry%mci_sets(i_mci)%counter%record (weight, excess)
    end if
  end subroutine entry_record
```

Update and restore the process core that this entry accesses, when parameters change. If explicit arguments `model`, `qcd`, or `helicity_selection` are provided, use those. Otherwise use the parameters stored in the process object.

⟨*Simulations: entry: TBP*⟩+≡

```
  procedure :: update_process => entry_update_process
  procedure :: restore_process => entry_restore_process
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine entry_update_process &
       (entry, model, qcd, helicity_selection)
    class(entry_t), intent(inout) :: entry
    class(model_data_t), intent(in), optional, target :: model
    type(qcd_t), intent(in), optional :: qcd
    type(helicity_selection_t), intent(in), optional :: helicity_selection
    type(process_t), pointer :: process
    class(prc_core_t), allocatable :: core
    integer :: i, n_terms
    class(model_data_t), pointer :: model_local
    type(qcd_t) :: qcd_local
    if (present (model)) then
       model_local => model
    else
       model_local => entry%model
    end if
    if (present (qcd)) then
       qcd_local = qcd
    else
       qcd_local = entry%qcd
    end if
    process => entry%get_process_ptr ()

    n_terms = process%get_n_terms ()
    allocate (entry%core_safe (n_terms))
    do i = 1, n_terms
       if (process%has_matrix_element (i, is_term_index = .true.)) then
          call process%extract_core (i, core)
          call dispatch_core_update (core, &
               model_local, helicity_selection, qcd_local, &
               entry%core_safe(i)%core)
          call process%restore_core (i, core)
       end if
    end do
  end subroutine entry_update_process
```

```
subroutine entry_restore_process (entry)
  class(entry_t), intent(inout) :: entry
  type(process_t), pointer :: process
  class(prc_core_t), allocatable :: core
  integer :: i, n_terms
  process => entry%get_process_ptr ()
  n_terms = process%get_n_terms ()
  do i = 1, n_terms
     if (process%has_matrix_element (i, is_term_index = .true.)) then
        call process%extract_core (i, core)
        call dispatch_core_restore (core, entry%core_safe(i)%core)
        call process%restore_core (i, core)
     end if
  end do
  deallocate (entry%core_safe)
end subroutine entry_restore_process
```

⟨*Simulations: entry: TBP*⟩+≡

```
procedure :: connect_qcd => entry_connect_qcd
```

⟨*Simulations: procedures*⟩+≡

```
subroutine entry_connect_qcd (entry)
  class(entry_t), intent(inout), target :: entry
  class(evt_t), pointer :: evt
  evt => entry%transform_first
  do while (associated (evt))
     select type (evt)
     type is (evt_shower_t)
        evt%qcd => entry%qcd
        if (allocated (evt%matching)) then
           evt%matching%qcd => entry%qcd
        end if
     end select
     evt => evt%next
  end do
end subroutine entry_connect_qcd
```

33.8.6 Entries for alternative environment

Entries for alternate environments. [No additional components anymore, so somewhat redundant.]

⟨*Simulations: types*⟩+≡

```
type, extends (entry_t) :: alt_entry_t
 contains
 ⟨Simulations: alt entry: TBP⟩
end type alt_entry_t
```

The alternative entries are there to re-evaluate the event, given momenta, in a different context.

Therefore, we allocate a local process object and use this as the reference for the local process instance, when initializing the entry. We temporarily im-

port the process object into an integration_t wrapper, to take advantage of the associated methods. The local process object is built in the context of the current environment, here called global. Then, we initialize the process instance.

The master_process object contains the integration results to which we refer when recalculating an event. Therefore, we use this object instead of the locally built process when we extract the integration results.

The locally built process object should be finalized when done. It remains accessible via the event_t base object of entry, which contains pointers to the process and instance.

⟨*Simulations: alt entry: TBP*⟩≡

```
procedure :: init_alt => alt_entry_init
```

⟨*Simulations: procedures*⟩+≡

```
subroutine alt_entry_init (entry, process_id, master_process, local)
  class(alt_entry_t), intent(inout), target :: entry
  type(string_t), intent(in) :: process_id
  type(process_t), intent(in), target :: master_process
  type(rt_data_t), intent(inout), target :: local
  type(process_t), pointer :: process
  type(process_instance_t), pointer :: process_instance
  type(string_t) :: run_id
  integer :: i

  call msg_message ("Simulate: initializing alternate process setup ...")

  run_id = &
       local%var_list%get_sval (var_str ("$run_id"))
  call local%set_log (var_str ("?rebuild_phase_space"), &
       .false., is_known = .true.)
  call local%set_log (var_str ("?check_phs_file"), &
       .false., is_known = .true.)
  call local%set_log (var_str ("?rebuild_grids"), &
       .false., is_known = .true.)

  call entry%basic_init (local%var_list)

  call prepare_local_process (process, process_id, local)
  entry%process_id = process_id
  entry%run_id = run_id

  call entry%import_process_characteristics (process)

  allocate (entry%mci_sets (entry%n_mci))
  do i = 1, size (entry%mci_sets)
     call entry%mci_sets(i)%init (i, master_process)
  end do

  call entry%import_process_results (master_process)
  call entry%prepare_expressions (local)

  call prepare_process_instance (process_instance, process, local%model)
  call entry%setup_event_transforms (process, local)
```

```
    call entry%connect (process_instance, local%model, local%process_stack)
    call entry%setup_expressions ()

    entry%model => process%get_model_ptr ()

    call msg_message ("...  alternate process setup complete.")

  end subroutine alt_entry_init
```

Copy the particle set from the master entry to the alternate entry. This is the particle set of the hard process.

⟨*Simulations: alt entry: TBP*⟩+≡

```
  procedure :: fill_particle_set => entry_fill_particle_set
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine entry_fill_particle_set (alt_entry, entry)
    class(alt_entry_t), intent(inout) :: alt_entry
    class(entry_t), intent(in), target :: entry
    type(particle_set_t) :: pset
    call entry%get_hard_particle_set (pset)
    call alt_entry%set_hard_particle_set (pset)
    call pset%final ()
  end subroutine entry_fill_particle_set
```

33.8.7 The simulation type

Each simulation object corresponds to an event sample, identified by the `sample_id`.

The simulation may cover several processes simultaneously. All process-specific data, including the event records, are stored in the `entry` subobjects. The `current` index indicates which record was selected last. `version` is foreseen to contain a tag on the `WHIZARD` event file version. It can be

⟨*Simulations: public*⟩≡

```
  public :: simulation_t
```

⟨*Simulations: types*⟩+≡

```
  type :: simulation_t
     private
     type(rt_data_t), pointer :: local => null ()
     type(string_t) :: sample_id
     logical :: unweighted = .true.
     logical :: negative_weights = .false.
     logical :: respect_selection = .true.
     integer :: norm_mode = NORM_UNDEFINED
     logical :: update_sqme = .false.
     logical :: update_weight = .false.
     logical :: update_event = .false.
     logical :: recover_beams = .false.
     logical :: pacify = .false.
     integer :: n_max_tries = 10000
     integer :: n_prc = 0
     integer :: n_alt = 0
     logical :: has_integral = .false.
```

```
    logical :: valid = .false.
    real(default) :: integral = 0
    real(default) :: error = 0
    integer :: version = 1
    character(32) :: md5sum_prc = ""
    character(32) :: md5sum_cfg = ""
    character(32), dimension(:), allocatable :: md5sum_alt
    type(entry_t), dimension(:), allocatable :: entry
    type(alt_entry_t), dimension(:,:), allocatable :: alt_entry
    type(selector_t) :: process_selector
    integer :: n_evt_requested = 0
    integer :: split_n_evt = 0
    integer :: split_n_kbytes = 0
    integer :: split_index = 0
    type(counter_t) :: counter
    class(rng_t), allocatable :: rng
    integer :: i_prc = 0
    integer :: i_mci = 0
    real(default) :: weight = 0
    real(default) :: excess = 0
  contains
  ⟨Simulations: simulation: TBP⟩
end type simulation_t
```

Output. `write_config` writes just the configuration. `write` as a method of the base type `event_t` writes the current event and process instance, depending on options.

⟨*Simulations: simulation: TBP*⟩≡

```
procedure :: write => simulation_write
```

⟨*Simulations: procedures*⟩+≡

```
subroutine simulation_write (object, unit)
  class(simulation_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer :: u, i
  u = given_output_unit (unit)
  call write_separator (u, 2)
  write (u, "(1x,A,A,A)")  "Event sample: '", char (object%sample_id), "'"
  write (u, "(3x,A,I0)")  "Processes    = ", object%n_prc
  if (object%n_alt > 0) then
     write (u, "(3x,A,I0)")  "Alt.wgts     = ", object%n_alt
  end if
  write (u, "(3x,A,L1)")  "Unweighted   = ", object%unweighted
  write (u, "(3x,A,A)")   "Event norm   = ", &
       char (event_normalization_string (object%norm_mode))
  write (u, "(3x,A,L1)")  "Neg. weights = ", object%negative_weights
  write (u, "(3x,A,L1)")  "Respect sel. = ", object%respect_selection
  write (u, "(3x,A,L1)")  "Update sqme  = ", object%update_sqme
  write (u, "(3x,A,L1)")  "Update wgt   = ", object%update_weight
  write (u, "(3x,A,L1)")  "Update event = ", object%update_event
  write (u, "(3x,A,L1)")  "Recov. beams = ", object%recover_beams
  write (u, "(3x,A,L1)")  "Pacify       = ", object%pacify
  write (u, "(3x,A,I0)")  "Max. tries   = ", object%n_max_tries
  if (object%has_integral) then
```

```
      write (u, "(3x,A," // FMT_19 // ")")  "Integral     = ", object%integral
      write (u, "(3x,A," // FMT_19 // ")")  "Error        = ", object%error
   else
      write (u, "(3x,A)")  "Integral     = [undefined]"
   end if
   write (u, "(3x,A,L1)")  "Sim. valid   = ", object%valid
   write (u, "(3x,A,I0)")  "Ev.file ver. = ", object%version
   if (object%md5sum_prc /= "") then
      write (u, "(3x,A,A,A)")  "MD5 sum (proc)   = '", object%md5sum_prc, "'"
   end if
   if (object%md5sum_cfg /= "") then
      write (u, "(3x,A,A,A)")  "MD5 sum (config) = '", object%md5sum_cfg, "'"
   end if
   write (u, "(3x,A,I0)")  "Events requested  = ", object%n_evt_requested
   if (object%split_n_evt > 0 .or. object%split_n_kbytes > 0) then
      write (u, "(3x,A,I0)")  "Events per file   = ", object%split_n_evt
      write (u, "(3x,A,I0)")  "KBytes per file   = ", object%split_n_kbytes
      write (u, "(3x,A,I0)")  "First file index  = ", object%split_index
   end if
   call object%counter%write (u)
   call write_separator (u)
   if (object%i_prc /= 0) then
      write (u, "(1x,A)")  "Current event:"
      write (u, "(3x,A,I0,A,A)")  "Process #", &
           object%i_prc, ": ", &
           char (object%entry(object%i_prc)%process_id)
      write (u, "(3x,A,I0)")  "MCI set #", object%i_mci
      write (u, "(3x,A," // FMT_19 // ")")  "Weight    = ", object%weight
      if (.not. vanishes (object%excess)) &
           write (u, "(3x,A," // FMT_19 // ")")  "Excess    = ", object%excess
   else
      write (u, "(1x,A,I0,A,A)")  "Current event: [undefined]"
   end if
   call write_separator (u)
   if (allocated (object%rng)) then
      call object%rng%write (u)
   else
      write (u, "(3x,A)")  "Random-number generator: [undefined]"
   end if
   if (allocated (object%entry)) then
      do i = 1, size (object%entry)
         if (i == 1) then
            call write_separator (u, 2)
         else
            call write_separator (u)
         end if
         write (u, "(1x,A,I0,A)") "Process #", i, ":"
         call object%entry(i)%write_config (u)
      end do
   end if
   call write_separator (u, 2)
end subroutine simulation_write
```

Write the current event record. If an explicit index is given, write that event

record.

We implement writing to `unit` (event contents / debugging format) and writing to an `eio` event stream (storage). We include a `testflag` in order to suppress numerical noise in the testsuite.

⟨*Simulations: simulation: TBP*⟩+≡

```
generic :: write_event => write_event_unit
procedure :: write_event_unit => simulation_write_event_unit
```

⟨*Simulations: procedures*⟩+≡

```
subroutine simulation_write_event_unit &
     (object, unit, i_prc, verbose, testflag)
  class(simulation_t), intent(in) :: object
  integer, intent(in), optional :: unit
  logical, intent(in), optional :: verbose
  integer, intent(in), optional :: i_prc
  logical, intent(in), optional :: testflag
  logical :: pacified
  integer :: current
  pacified = .false.;  if (present(testflag)) pacified = testflag
  pacified = pacified .or. object%pacify
  if (present (i_prc)) then
     current = i_prc
  else
     current = object%i_prc
  end if
  if (current > 0) then
     call object%entry(current)%write (unit, verbose = verbose, &
          testflag = pacified)
  else
     call msg_fatal ("Simulation: write event: no process selected")
  end if
end subroutine simulation_write_event_unit
```

This writes one of the alternate events, if allocated.

⟨*Simulations: simulation: TBP*⟩+≡

```
procedure :: write_alt_event => simulation_write_alt_event
```

⟨*Simulations: procedures*⟩+≡

```
subroutine simulation_write_alt_event (object, unit, j_alt, i_prc, &
     verbose, testflag)
  class(simulation_t), intent(in) :: object
  integer, intent(in), optional :: unit
  integer, intent(in), optional :: j_alt
  integer, intent(in), optional :: i_prc
  logical, intent(in), optional :: verbose
  logical, intent(in), optional :: testflag
  integer :: i, j
  if (present (j_alt)) then
     j = j_alt
  else
     j = 1
  end if
  if (present (i_prc)) then
     i = i_prc
```

```
    else
       i = object%i_prc
    end if
    if (i > 0) then
       if (j> 0 .and. j <= object%n_alt) then
          call object%alt_entry(i,j)%write (unit, verbose = verbose, &
               testflag = testflag)
       else
          call msg_fatal ("Simulation: write alternate event: out of range")
       end if
    else
       call msg_fatal ("Simulation: write alternate event: no process selected")
    end if
  end subroutine simulation_write_alt_event
```

Finalizer.

⟨*Simulations: simulation: TBP*⟩+≡

```
  procedure :: final => simulation_final
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine simulation_final (object)
    class(simulation_t), intent(inout) :: object
    integer :: i, j
    if (allocated (object%entry)) then
       do i = 1, size (object%entry)
          call object%entry(i)%final ()
       end do
    end if
    if (allocated (object%alt_entry)) then
       do j = 1, size (object%alt_entry, 2)
          do i = 1, size (object%alt_entry, 1)
             call object%alt_entry(i,j)%final ()
          end do
       end do
    end if
    if (allocated (object%rng))  call object%rng%final ()
  end subroutine simulation_final
```

Initialization. We can deduce all data from the given list of process IDs and the global data set. The process objects are taken from the stack. Once the individual integrals are known, we add them (and the errors), to get the sample integral.

If there are alternative environments, we suspend initialization for setting up alternative process objects, then restore the master process and its parameters. The generator or rescanner can then switch rapidly between processes.

If `integrate` is set, we make sure that all affected processes are integrated before simulation. This is necessary if we want to actually generate events. If `integrate` is unset, we don't need the integral because we just rescan existing events. In that case, we just need compiled matrix elements.

If `generate` is set, we prepare for actually generating events. Otherwise, we may only read and rescan events.

⟨*Simulations: simulation: TBP*⟩+≡

```
  procedure :: init => simulation_init
```

⟨*Simulations: procedures*⟩+≡

```
subroutine simulation_init (simulation, &
    process_id, integrate, generate, local, global, alt_env)
  class(simulation_t), intent(out), target :: simulation
  type(string_t), dimension(:), intent(in) :: process_id
  logical, intent(in) :: integrate, generate
  type(rt_data_t), intent(inout), target :: local
  type(rt_data_t), intent(inout), optional, target :: global
  type(rt_data_t), dimension(:), intent(inout), optional, target :: alt_env
  class(rng_factory_t), allocatable :: rng_factory
  type(string_t) :: norm_string, version_string
  logical :: use_process
  integer :: i, j
  simulation%local => local
  simulation%sample_id = &
       local%get_sval (var_str ("$sample"))
  simulation%unweighted = &
       local%get_lval (var_str ("?unweighted"))
  simulation%negative_weights = &
       local%get_lval (var_str ("?negative_weights"))
  simulation%respect_selection = &
       local%get_lval (var_str ("?sample_select"))
  version_string = &
       local%get_sval (var_str ("$event_file_version"))
  norm_string = &
       local%get_sval (var_str ("$sample_normalization"))
  simulation%norm_mode = &
       event_normalization_mode (norm_string, simulation%unweighted)
  simulation%pacify = &
       local%get_lval (var_str ("?sample_pacify"))
  simulation%n_max_tries = &
       local%get_ival (var_str ("sample_max_tries"))
  simulation%split_n_evt = &
       local%get_ival (var_str ("sample_split_n_evt"))
  simulation%split_n_kbytes = &
       local%get_ival (var_str ("sample_split_n_kbytes"))
  simulation%split_index = &
       local%get_ival (var_str ("sample_split_index"))
  simulation%update_sqme = &
       local%get_lval (var_str ("?update_sqme"))
  simulation%update_weight = &
       local%get_lval (var_str ("?update_weight"))
  simulation%update_event = &
       local%get_lval (var_str ("?update_event"))
  simulation%recover_beams = &
       local%get_lval (var_str ("?recover_beams"))
  simulation%counter%reproduce_xsection = &
       local%get_lval (var_str ("?check_event_weights_against_xsection"))
  use_process = &
       integrate .or. generate &
       .or. simulation%update_sqme &
       .or. simulation%update_weight &
       .or. simulation%update_event &
       .or. present (alt_env)
```

```
select case (size (process_id))
case (0)
   call msg_error ("Simulation: no process selected")
case (1)
   write (msg_buffer, "(A,A,A)") &
        "Starting simulation for process '", &
        char (process_id(1)), "'"
   call msg_message ()
case default
   write (msg_buffer, "(A,A,A)") &
        "Starting simulation for processes '", &
        char (process_id(1)), "' etc."
   call msg_message ()
end select
select case (char (version_string))
case ("", "2.2.4")
   simulation%version = 2
case ("2.2")
   simulation%version = 1
case default
   simulation%version = 0
end select
if (simulation%version == 0) then
   call msg_fatal ("Event file format '" &
        // char (version_string) &
        // "' is not compatible with this version.")
end if
simulation%n_prc = size (process_id)
allocate (simulation%entry (simulation%n_prc))
if (present (alt_env)) then
   simulation%n_alt = size (alt_env)
   do i = 1, simulation%n_prc
      call simulation%entry(i)%init (process_id(i), &
           use_process, integrate, generate, &
           simulation%update_sqme, &
           local, global, simulation%n_alt)
      if (signal_is_pending ())  return
   end do
   simulation%valid = any (simulation%entry%valid)
   if (.not. simulation%valid) then
      call msg_error ("Simulate: no process has a valid matrix element.")
      return
   end if
   call simulation%update_processes ()
   allocate (simulation%alt_entry (simulation%n_prc, simulation%n_alt))
   allocate (simulation%md5sum_alt (simulation%n_alt))
   simulation%md5sum_alt = ""
   do j = 1, simulation%n_alt
      do i = 1, simulation%n_prc
         call simulation%alt_entry(i,j)%init_alt (process_id(i), &
              simulation%entry(i)%get_process_ptr (), alt_env(j))
         if (signal_is_pending ())  return
      end do
   end do
```

```
      call simulation%restore_processes ()
   else
      do i = 1, simulation%n_prc
         call simulation%entry(i)%init &
              (process_id(i), &
              use_process, integrate, generate, simulation%update_sqme, &
              local, global)
         call simulation%entry(i)%determine_if_powheg_matching ()
         if (signal_is_pending ())  return
         if (simulation%entry(i)%is_nlo ()) &
              call simulation%entry(i)%setup_additional_entries ()
      end do
      simulation%valid = any (simulation%entry%valid)
      if (.not. simulation%valid) then
         call msg_error ("Simulate: " &
              // "no process has a valid matrix element.")
         return
      end if
   end if
!!! if this becomes conditional, some ref files will need update (seed change)
!    if (generate) then
      call dispatch_rng_factory (rng_factory, local%var_list)
      call rng_factory%make (simulation%rng)
!    end if
   if (all (simulation%entry%has_integral)) then
      simulation%integral = sum (simulation%entry%integral)
      simulation%error = sqrt (sum (simulation%entry%error ** 2))
      simulation%has_integral = .true.
      if (integrate .and. generate) then
         do i = 1, simulation%n_prc
            if (simulation%entry(i)%integral < 0 .and. .not. &
                 simulation%negative_weights) then
               call msg_fatal ("Integral of process '" // &
                    char (process_id (i)) // "'is negative.")
            end if
         end do
      end if
   else
      if (integrate .and. generate) &
           call msg_error ("Simulation contains undefined integrals.")
   end if
   if (simulation%integral > 0 .or. &
        (simulation%integral < 0 .and. simulation%negative_weights)) then
      simulation%valid = .true.
   else if (generate) then
      call msg_error ("Simulate: " &
           // "sum of process integrals must be positive; skipping.")
      simulation%valid = .false.
   else
      simulation%valid = .true.
   end if
   if (simulation%valid)  call simulation%compute_md5sum ()
 end subroutine simulation_init
```

The number of events that we want to simulate is determined by the settings of `n_events`, `luminosity`, and `?unweighted`. For weighted events, we take `n_events` at face value as the number of matrix element calls. For unweighted events, if the process is a decay, `n_events` is the number of unweighted events. In these cases, the luminosity setting is ignored.

For unweighted events with a scattering process, we calculate the event number that corresponds to the luminosity, given the current value of the integral. We then compare this with `n_events` and choose the larger number.

⟨*Simulations: simulation: TBP*⟩+≡

```
procedure :: compute_n_events => simulation_compute_n_events
```

⟨*Simulations: procedures*⟩+≡

```
subroutine simulation_compute_n_events (simulation, n_events, var_list)
  class(simulation_t), intent(in) :: simulation
  integer, intent(out) :: n_events
  type(var_list_t) :: var_list
  real(default) :: lumi, x_events_lumi
  integer :: n_events_lumi
  logical :: is_scattering
  n_events = &
       var_list%get_ival (var_str ("n_events"))
  lumi = &
       var_list%get_rval (var_str ("luminosity"))
  if (simulation%unweighted) then
     is_scattering = simulation%entry(1)%n_in == 2
     if (is_scattering) then
        x_events_lumi = abs (simulation%integral * lumi)
        if (x_events_lumi < huge (n_events)) then
           n_events_lumi = nint (x_events_lumi)
        else
           call msg_message ("Simulation: luminosity too large, &
                &limiting number of events")
           n_events_lumi = huge (n_events)
        end if
        if (n_events_lumi > n_events) then
           call msg_message ("Simulation: using n_events as computed from &
                &luminosity value")
           n_events = n_events_lumi
        else
           write (msg_buffer, "(A,1x,I0)") &
                "Simulation: requested number of events =", n_events
           call msg_message ()
           if (.not. vanishes (simulation%integral)) then
              write (msg_buffer, "(A,1x,ES11.4)") &
                   "            corr. to luminosity [fb-1] = ", &
                   n_events / simulation%integral
              call msg_message ()
           end if
        end if
     end if
  end if
end subroutine simulation_compute_n_events
```

Write the actual efficiency of the simulation run. We get the total number of

events stored in the simulation counter and compare this with the total number of calls stored in the event entries.

In order not to miscount samples that are partly read from file, use the `generated` counter, not the `total` counter.

⟨*Simulations: simulation: TBP*⟩+≡
```
procedure :: show_efficiency => simulation_show_efficiency
```

⟨*Simulations: procedures*⟩+≡
```
subroutine simulation_show_efficiency (simulation)
  class(simulation_t), intent(inout) :: simulation
  integer :: n_events, n_calls
  real(default) :: eff
  n_events = simulation%counter%generated
  n_calls = sum (simulation%entry%get_actual_calls_total ())
  if (n_calls > 0) then
     eff = real (n_events, kind=default) / n_calls
     write (msg_buffer, "(A,1x,F6.2,1x,A)") &
          "Events: actual unweighting efficiency =", 100 * eff, "%"
     call msg_message ()
  end if
end subroutine simulation_show_efficiency
```

⟨*Simulations: simulation: TBP*⟩+≡
```
procedure :: get_n_nlo_entries => simulation_get_n_nlo_entries
```

⟨*Simulations: procedures*⟩+≡
```
function simulation_get_n_nlo_entries (simulation, i_prc) result (n_extra)
  class(simulation_t), intent(in) :: simulation
  integer, intent(in) :: i_prc
  integer :: n_extra
  n_extra = simulation%entry(i_prc)%count_nlo_entries ()
end function simulation_get_n_nlo_entries
```

Compute the checksum of the process set. We retrieve the MD5 sums of all processes. This depends only on the process definitions, while parameters are not considered. The configuration checksum is retrieved from the MCI records in the process objects and furthermore includes beams, parameters, integration results, etc., so matching the latter should guarantee identical physics.

⟨*Simulations: simulation: TBP*⟩+≡
```
procedure :: compute_md5sum => simulation_compute_md5sum
```

⟨*Simulations: procedures*⟩+≡
```
subroutine simulation_compute_md5sum (simulation)
  class(simulation_t), intent(inout) :: simulation
  type(process_t), pointer :: process
  type(string_t) :: buffer
  integer :: j, i, n_mci, i_mci, n_component, i_component
  if (simulation%md5sum_prc == "") then
     buffer = ""
     do i = 1, simulation%n_prc
        if (.not. simulation%entry(i)%valid) cycle
        process => simulation%entry(i)%get_process_ptr ()
        if (associated (process)) then
           n_component = process%get_n_components ()
```

```
            do i_component = 1, n_component
               if (process%has_matrix_element (i_component)) then
                  buffer = buffer // process%get_md5sum_prc (i_component)
               end if
            end do
         end if
      end do
      simulation%md5sum_prc = md5sum (char (buffer))
   end if
   if (simulation%md5sum_cfg == "") then
      buffer = ""
      do i = 1, simulation%n_prc
         if (.not. simulation%entry(i)%valid) cycle
         process => simulation%entry(i)%get_process_ptr ()
         if (associated (process)) then
            n_mci = process%get_n_mci ()
            do i_mci = 1, n_mci
               buffer = buffer // process%get_md5sum_mci (i_mci)
            end do
         end if
      end do
      simulation%md5sum_cfg = md5sum (char (buffer))
   end if
   do j = 1, simulation%n_alt
      if (simulation%md5sum_alt(j) == "") then
         buffer = ""
         do i = 1, simulation%n_prc
            process => simulation%alt_entry(i,j)%get_process_ptr ()
            if (associated (process)) then
               buffer = buffer // process%get_md5sum_cfg ()
            end if
         end do
         simulation%md5sum_alt(j) = md5sum (char (buffer))
      end if
   end do
end subroutine simulation_compute_md5sum
```

Initialize the process selector, using the entry integrals as process weights.

⟨*Simulations: simulation: TBP*⟩+≡

```
procedure :: init_process_selector => simulation_init_process_selector
```

⟨*Simulations: procedures*⟩+≡

```
subroutine simulation_init_process_selector (simulation)
  class(simulation_t), intent(inout) :: simulation
  integer :: i
  if (simulation%has_integral) then
     call simulation%process_selector%init (simulation%entry%integral, &
          negative_weights = simulation%negative_weights)
     do i = 1, simulation%n_prc
        associate (entry => simulation%entry(i))
          if (.not. entry%valid) then
             call msg_warning ("Process '" // char (entry%process_id) // &
                  "': matrix element vanishes, no events can be generated.")
             cycle
```

```
            end if
            call entry%init_mci_selector (simulation%negative_weights)
            entry%process_weight = simulation%process_selector%get_weight (i)
          end associate
        end do
      end if
    end subroutine simulation_init_process_selector
```

Select a process, using the random-number generator.

⟨*Simulations: simulation: TBP*⟩+≡

```
    procedure :: select_prc => simulation_select_prc
```

⟨*Simulations: procedures*⟩+≡

```
    function simulation_select_prc (simulation) result (i_prc)
      class(simulation_t), intent(inout) :: simulation
      integer :: i_prc
      call simulation%process_selector%generate (simulation%rng, i_prc)
    end function simulation_select_prc
```

Select a MCI set for the selected process.

⟨*Simulations: simulation: TBP*⟩+≡

```
    procedure :: select_mci => simulation_select_mci
```

⟨*Simulations: procedures*⟩+≡

```
    function simulation_select_mci (simulation) result (i_mci)
      class(simulation_t), intent(inout) :: simulation
      integer :: i_mci
      i_mci = 0
      if (simulation%i_prc /= 0) then
         i_mci = simulation%entry(simulation%i_prc)%select_mci ()
      end if
    end function simulation_select_mci
```

Generate a predefined number of events. First select a process and a component set, then generate an event for that process and factorize the quantum state. The pair of random numbers can be used for factorization.

When generating events, we drop all configurations where the event is marked as incomplete. This happens if the event fails cuts. In fact, such events are dropped already by the sampler if unweighting is in effect, so this can happen only for weighted events. By setting a limit given by `sample_max_tries` (user parameter), we can avoid an endless loop.

NB: When reading from file, event transforms can't be applied because the process instance will not be complete. This should be fixed.

⟨*Simulations: simulation: TBP*⟩+≡

```
    procedure :: generate => simulation_generate
```

⟨*Simulations: procedures*⟩+≡

```
    subroutine simulation_generate (simulation, n, es_array)
      class(simulation_t), intent(inout), target :: simulation
      integer, intent(in) :: n
      type(event_stream_array_t), intent(inout), optional :: es_array
      type(string_t) :: str1, str2, str3
      logical :: generate_new, passed
```

```
integer :: i, j, k
type(entry_t), pointer :: current_entry
integer :: n_events
simulation%n_evt_requested = n
n_events = n * simulation%get_n_nlo_entries (1)
call simulation%entry%set_n (n)
if (simulation%n_alt > 0)  call simulation%alt_entry%set_n (n)
str1 = "Events: generating"
if (present (es_array)) then
   if (es_array%has_input ())  str1 = "Events: reading"
end if
if (simulation%entry(1)%config%unweighted) then
   str2 = "unweighted"
else
   str2 = "weighted"
end if
if (simulation%entry(1)%config%factorization_mode == &
     FM_IGNORE_HELICITY) then
   str3 = ", unpolarized"
else
   str3 = ", polarized"
end if
if (n_events == n) then
   write (msg_buffer, "(A,1x,I0,1x,A,1x,A)")  char (str1), n, &
        char (str2) // char(str3), "events ..."
else
   write (msg_buffer, "(A,1x,I0,1x,A,1x,A)") char (str1), n_events, &
        char (str2) // char(str3), "NLO events ..."
end if
call msg_message ()
write (msg_buffer, "(A,1x,A)") "Events: event normalization mode", &
     char (event_normalization_string (simulation%norm_mode))
call msg_message ()
do i = 1, n
   if (present (es_array)) then
      call simulation%read_event (es_array, .true., generate_new)
   else
      generate_new = .true.
   end if
   if (generate_new) then
      simulation%i_prc = simulation%select_prc ()
      simulation%i_mci = simulation%select_mci ()
      associate (entry => simulation%entry(simulation%i_prc))
        call entry%set_active_real_components (simulation%i_mci)
        current_entry => entry%get_first ()
        do k = 1, current_entry%count_nlo_entries ()
           if (k > 1) then
              current_entry => current_entry%get_next ()
              current_entry%particle_set => current_entry%first%particle_set
              current_entry%particle_set_is_valid &
                 = current_entry%first%particle_set_is_valid
           end if
           do j = 1, simulation%n_max_tries
              if (.not. current_entry%valid)  call msg_warning &
```

```
                 ("Process '" // char (current_entry%process_id) // "': " // &
                 "matrix element vanishes, no events can be generated.")
            call current_entry%generate (simulation%i_mci, i_nlo = k)
            if (signal_is_pending ()) return
            call simulation%counter%record_mean_and_variance &
                 (current_entry%weight_prc, k)
            if (current_entry%has_valid_particle_set ()) exit
         end do
      end do
      if (entry%is_nlo ()) call entry%reset_nlo_counter ()
      if (.not. entry%has_valid_particle_set ()) then
         write (msg_buffer, "(A,I0,A)")  "Simulation: failed to &
              &generate valid event after ", &
              simulation%n_max_tries, " tries (sample_max_tries)"
         call msg_fatal ()
      end if
      current_entry => entry%get_first ()
      do k = 1, current_entry%count_nlo_entries ()
         if (k > 1) current_entry => current_entry%get_next ()
         call current_entry%evaluate_expressions ()
      end do
      if (signal_is_pending ()) return
      if (entry%passed_selection ()) then
         simulation%weight = entry%get_weight_ref ()
         simulation%excess = entry%get_excess_prc ()
      end if
      call simulation%counter%record &
           (simulation%weight, simulation%excess)
      call entry%record (simulation%i_mci)
   end associate
else
   associate (entry => simulation%entry(simulation%i_prc))
      call entry%accept_sqme_ref ()
      call entry%accept_weight_ref ()
      !!! JRR: WK please check: why commented out
      ! call entry%evaluate_transforms ()  ! doesn't activate
      call entry%check ()
      call entry%evaluate_expressions ()
      if (signal_is_pending ()) return
      if (entry%passed_selection ()) then
         simulation%weight = entry%get_weight_ref ()
         simulation%excess = entry%get_excess_prc ()
      end if
      call simulation%counter%record &
           (simulation%weight, simulation%excess, from_file=.true.)
      call entry%record (simulation%i_mci, from_file=.true.)
   end associate
end if
call simulation%calculate_alt_entries ()
if (signal_is_pending ()) return
if (simulation%pacify)  call pacify (simulation)
if (simulation%respect_selection) then
   passed = simulation%entry(simulation%i_prc)%passed_selection ()
else
```

```
        passed = .true.
      end if
      if (present (es_array)) then
        call simulation%write_event (es_array, passed)
      end if
    end do
    call msg_message ("        ... event sample complete.")
    if (simulation%unweighted)  call simulation%show_efficiency ()
    call simulation%counter%show_excess ()
    call simulation%counter%show_mean_and_variance ()
  end subroutine simulation_generate
```

Compute the event matrix element and weight for all alternative environments, given the current event and selected process. We first copy the particle set, then temporarily update the process core with local parameters, recalculate everything, and restore the process core.

The event weight is obtained by rescaling the original event weight with the ratio of the new and old `sqme` values. (In particular, if the old value was zero, the weight will stay zero.)

Note: this may turn out to be inefficient because we always replace all parameters and recalculate everything, once for each event and environment. However, a more fine-grained control requires more code. In any case, while we may keep multiple process cores (which stay constant for a simulation run), we still have to update the external matrix element parameters event by event. The matrix element "object" is present only once.

⟨*Simulations: simulation: TBP*⟩+≡

```
  procedure :: calculate_alt_entries => simulation_calculate_alt_entries
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine simulation_calculate_alt_entries (simulation)
    class(simulation_t), intent(inout) :: simulation
    real(default) :: factor
    real(default), dimension(:), allocatable :: sqme_alt, weight_alt
    integer :: n_alt, i, j
    i = simulation%i_prc
    n_alt = simulation%n_alt
    if (n_alt == 0)  return
    allocate (sqme_alt (n_alt), weight_alt (n_alt))
    associate (entry => simulation%entry(i))
      do j = 1, n_alt
        if (signal_is_pending ())  return
        factor = entry%get_kinematical_weight ()
        associate (alt_entry => simulation%alt_entry(i,j))
          call alt_entry%update_process ()
          call alt_entry%select &
               (entry%get_i_mci (), entry%get_i_term (), entry%get_channel ())
          call alt_entry%fill_particle_set (entry)
          call alt_entry%recalculate &
               (update_sqme = .true., weight_factor = factor)
          if (signal_is_pending ())  return
          call alt_entry%accept_sqme_prc ()
          call alt_entry%update_normalization ()
          call alt_entry%accept_weight_prc ()
```

```
            call alt_entry%check ()
            call alt_entry%evaluate_expressions ()
            if (signal_is_pending ())  return
            call alt_entry%restore_process ()
            sqme_alt(j) = alt_entry%get_sqme_ref ()
            if (alt_entry%passed_selection ()) then
               weight_alt(j) = alt_entry%get_weight_ref ()
            end if
         end associate
      end do
      call entry%set (sqme_alt = sqme_alt, weight_alt = weight_alt)
      call entry%check ()
      call entry%store_alt_values ()
    end associate
  end subroutine simulation_calculate_alt_entries
```

Rescan an undefined number of events.

If `update_event` or `update_sqme` is set, we have to recalculate the event, starting from the particle set. If the latter is set, this includes the squared matrix element (i.e., the amplitude is evaluated). Otherwise, only kinematics and observables derived from it are recovered.

If any of the update flags is set, we will come up with separate `sqme_prc` and `weight_prc` values. (The latter is only distinct if `update_weight` is set.) Otherwise, we accept the reference values.

⟨*Simulations: simulation: TBP*⟩+≡

```
  procedure :: rescan => simulation_rescan
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine simulation_rescan (simulation, n, es_array, global)
    class(simulation_t), intent(inout) :: simulation
    integer, intent(in) :: n
    type(event_stream_array_t), intent(inout) :: es_array
    type(rt_data_t), intent(inout) :: global
    type(qcd_t) :: qcd
    type(string_t) :: str1, str2, str3
    logical :: complete
    str1 = "Rescanning"
    if (simulation%entry(1)%config%unweighted) then
       str2 = "unweighted"
    else
       str2 = "weighted"
    end if
    simulation%n_evt_requested = n
    call simulation%entry%set_n (n)
    if (simulation%update_sqme .or. simulation%update_weight) then
       call dispatch_qcd (qcd, global%get_var_list_ptr (), global%os_data)
       call simulation%update_processes &
            (global%model, qcd, global%get_helicity_selection ())
       str3 = "(process parameters updated) "
    else
       str3 = ""
    end if
    write (msg_buffer, "(A,1x,A,1x,A,A,A)")  char (str1), char (str2), &
```

```
      "events ", char (str3), "..."
  call msg_message ()
  do
     call simulation%read_event (es_array, .false., complete)
     if (complete)  exit
     if (simulation%update_event &
          .or. simulation%update_sqme &
          .or. simulation%update_weight) then
        call simulation%recalculate ()
        if (signal_is_pending ())  return
        associate (entry => simulation%entry(simulation%i_prc))
          call entry%update_normalization ()
          if (simulation%update_event) then
             call entry%evaluate_transforms ()
          end if
          call entry%check ()
          call entry%evaluate_expressions ()
          if (signal_is_pending ())  return
          simulation%weight = entry%get_weight_prc ()
          call simulation%counter%record (simulation%weight, from_file=.true.)
          call entry%record (simulation%i_mci, from_file=.true.)
        end associate
     else
        associate (entry => simulation%entry(simulation%i_prc))
          call entry%accept_sqme_ref ()
          call entry%accept_weight_ref ()
          call entry%check ()
          call entry%evaluate_expressions ()
          if (signal_is_pending ())  return
          simulation%weight = entry%get_weight_ref ()
          call simulation%counter%record (simulation%weight, from_file=.true.)
          call entry%record (simulation%i_mci, from_file=.true.)
        end associate
     end if
     call simulation%calculate_alt_entries ()
     if (signal_is_pending ())  return
     call simulation%write_event (es_array)
  end do
  if (simulation%update_sqme .or. simulation%update_weight) then
     call simulation%restore_processes ()
  end if
end subroutine simulation_rescan
```

These routines take care of temporary parameter redefinitions that we want to take effect while recalculating the matrix elements. We extract the core(s) of the processes that we are simulating, apply the changes, and make sure that the changes are actually used. This is the duty of `dispatch_core_update`. When done, we restore the original versions using `dispatch_core_restore`.

⟨*Simulations: simulation: TBP*⟩+≡

```
procedure :: update_processes => simulation_update_processes
procedure :: restore_processes => simulation_restore_processes
```

⟨*Simulations: procedures*⟩+≡

```
subroutine simulation_update_processes (simulation, &
```

```
      model, qcd, helicity_selection)
    class(simulation_t), intent(inout) :: simulation
    class(model_data_t), intent(in), optional, target :: model
    type(qcd_t), intent(in), optional :: qcd
    type(helicity_selection_t), intent(in), optional :: helicity_selection
    integer :: i
    do i = 1, simulation%n_prc
       call simulation%entry(i)%update_process &
            (model, qcd, helicity_selection)
    end do
  end subroutine simulation_update_processes

  subroutine simulation_restore_processes (simulation)
    class(simulation_t), intent(inout) :: simulation
    integer :: i
    do i = 1, simulation%n_prc
       call simulation%entry(i)%restore_process ()
    end do
  end subroutine simulation_restore_processes
```

33.8.8 Event Stream I/O

Write an event to a generic `eio` event stream. The process index must be selected, or the current index must be available.

⟨*Simulations: simulation: TBP*⟩+≡

```
  generic :: write_event => write_event_eio
  procedure :: write_event_eio => simulation_write_event_eio
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine simulation_write_event_eio (object, eio, i_prc)
    class(simulation_t), intent(in) :: object
    class(eio_t), intent(inout) :: eio
    integer, intent(in), optional :: i_prc
    logical :: increased
    integer :: current
    if (present (i_prc)) then
       current = i_prc
    else
       current = object%i_prc
    end if
    if (current > 0) then
       if (object%split_n_evt > 0 &
            .and. object%counter%total > 1 &
            .and. mod (object%counter%total, object%split_n_evt) == 1) then
          call eio%split_out ()
       else if (object%split_n_kbytes > 0) then
          call eio%update_split_count (increased)
          if (increased)  call eio%split_out ()
       end if
       call eio%output (object%entry(current)%event_t, current, pacify = object%pacify)
    else
       call msg_fatal ("Simulation: write event: no process selected")
    end if
```

```
end subroutine simulation_write_event_eio
```

Read an event from a generic `eio` event stream. The event stream element must specify the process within the sample (`i_prc`), the MC group for this process (`i_mci`), the selected term (`i_term`), the selected MC integration `channel`, and the particle set of the event.

We may encounter EOF, which we indicate by storing 0 for the process index `i_prc`. An I/O error will be reported, and we also abort reading.

⟨*Simulations: simulation: TBP*⟩+≡

```
generic :: read_event => read_event_eio
procedure :: read_event_eio => simulation_read_event_eio
```

⟨*Simulations: procedures*⟩+≡

```
subroutine simulation_read_event_eio (object, eio)
  class(simulation_t), intent(inout) :: object
  class(eio_t), intent(inout) :: eio
  integer :: iostat, current
  call eio%input_i_prc (current, iostat)
  select case (iostat)
  case (0)
     object%i_prc = current
     call eio%input_event (object%entry(current)%event_t, iostat)
  end select
  select case (iostat)
  case (:-1)
     object%i_prc = 0
     object%i_mci = 0
  case (1:)
     call msg_error ("Reading events: I/O error, aborting read")
     object%i_prc = 0
     object%i_mci = 0
  case default
     object%i_mci = object%entry(current)%get_i_mci ()
  end select
end subroutine simulation_read_event_eio
```

33.8.9 Event Stream Array

Write an event using an array of event I/O streams. The process index must be selected, or the current index must be available.

⟨*Simulations: simulation: TBP*⟩+≡

```
generic :: write_event => write_event_es_array
procedure :: write_event_es_array => simulation_write_event_es_array
```

⟨*Simulations: procedures*⟩+≡

```
subroutine simulation_write_event_es_array (object, es_array, passed)
  class(simulation_t), intent(in), target :: object
  class(event_stream_array_t), intent(inout) :: es_array
  logical, intent(in), optional :: passed
  integer :: i_prc, event_index
  integer :: i
  type(entry_t), pointer :: current_entry
```

```
    i_prc = object%i_prc
    if (i_prc > 0) then
       event_index = object%counter%total
       current_entry => object%entry(i_prc)%get_first ()
       do i = 1, current_entry%count_nlo_entries ()
          if (i > 1) current_entry => current_entry%get_next ()
          call es_array%output (current_entry%event_t, i_prc, &
             event_index, passed = passed, pacify = object%pacify)
       end do
    else
       call msg_fatal ("Simulation: write event: no process selected")
    end if
  end subroutine simulation_write_event_es_array
```

Read an event using an array of event I/O streams. Reading is successful if there is an input stream within the array, and if a valid event can be read from that stream. If there is a stream, but EOF is passed when reading the first item, we switch the channel to output and return failure but no error message, such that new events can be appended to that stream.

⟨*Simulations: simulation: TBP*⟩+≡

```
  generic :: read_event => read_event_es_array
  procedure :: read_event_es_array => simulation_read_event_es_array
```

⟨*Simulations: procedures*⟩+≡

```
  subroutine simulation_read_event_es_array (object, es_array, enable_switch, &
       fail)
    class(simulation_t), intent(inout), target :: object
    class(event_stream_array_t), intent(inout), target :: es_array
    logical, intent(in) :: enable_switch
    logical, intent(out) :: fail
    integer :: iostat, i_prc
    type(entry_t), pointer :: current_entry => null ()
    integer :: i
    if (es_array%has_input ()) then
       fail = .false.
       call es_array%input_i_prc (i_prc, iostat)
       select case (iostat)
       case (0)
          object%i_prc = i_prc
          current_entry => object%entry(i_prc)
          do i = 1, current_entry%count_nlo_entries ()
             if (i > 1) then
                call es_array%skip_eio_entry (iostat)
                current_entry => current_entry%get_next ()
             end if
             call es_array%input_event (current_entry%event_t, iostat)
          end do
       case (:-1)
          write (msg_buffer, "(A,1x,I0,1x,A)")  &
               "... event file terminates after", &
               object%counter%read, "events."
          call msg_message ()
          if (enable_switch) then
             call es_array%switch_inout ()
```

```
            write (msg_buffer, "(A,1x,I0,1x,A)")  &
                 "Generating remaining ", &
                 object%n_evt_requested - object%counter%read, "events ..."
            call msg_message ()
         end if
         fail = .true.
         return
      end select
      select case (iostat)
      case (0)
         object%i_mci = object%entry(i_prc)%get_i_mci ()
      case default
         write (msg_buffer, "(A,1x,I0,1x,A)")  &
              "Reading events: I/O error, aborting read after", &
              object%counter%read, "events."
         call msg_error ()
         object%i_prc = 0
         object%i_mci = 0
         fail = .true.
      end select
   else
      fail = .true.
   end if
end subroutine simulation_read_event_es_array
```

33.8.10 Recover event

Recalculate the process instance contents, given an event with known particle set. The indices for MC, term, and channel must be already set. The `recalculate` method of the selected entry will import the result into `sqme_prc` and `weight_prc`.

⟨*Simulations: simulation: TBP*⟩+≡

```
procedure :: recalculate => simulation_recalculate
```

⟨*Simulations: procedures*⟩+≡

```
subroutine simulation_recalculate (simulation)
  class(simulation_t), intent(inout) :: simulation
  integer :: i_prc
  i_prc = simulation%i_prc
  associate (entry => simulation%entry(i_prc))
    if (simulation%update_weight) then
       call simulation%entry(i_prc)%recalculate &
            (update_sqme = simulation%update_sqme, &
            recover_beams = simulation%recover_beams, &
            weight_factor = entry%get_kinematical_weight ())
    else
       call simulation%entry(i_prc)%recalculate &
            (update_sqme = simulation%update_sqme, &
            recover_beams = simulation%recover_beams)
    end if
  end associate
end subroutine simulation_recalculate
```

33.8.11 Extract contents

Return the MD5 sum that summarizes configuration and integration (but not the event file). Used for initializing the event streams.

⟨*Simulations: simulation: TBP*⟩+≡

```
procedure :: get_md5sum_prc => simulation_get_md5sum_prc
procedure :: get_md5sum_cfg => simulation_get_md5sum_cfg
procedure :: get_md5sum_alt => simulation_get_md5sum_alt
```

⟨*Simulations: procedures*⟩+≡

```
function simulation_get_md5sum_prc (simulation) result (md5sum)
  class(simulation_t), intent(in) :: simulation
  character(32) :: md5sum
  md5sum = simulation%md5sum_prc
end function simulation_get_md5sum_prc

function simulation_get_md5sum_cfg (simulation) result (md5sum)
  class(simulation_t), intent(in) :: simulation
  character(32) :: md5sum
  md5sum = simulation%md5sum_cfg
end function simulation_get_md5sum_cfg

function simulation_get_md5sum_alt (simulation, i) result (md5sum)
  class(simulation_t), intent(in) :: simulation
  integer, intent(in) :: i
  character(32) :: md5sum
  md5sum = simulation%md5sum_alt(i)
end function simulation_get_md5sum_alt
```

Return data that may be useful for writing event files.

Usually we can refer to a previously integrated process, for which we can fetch a process pointer. Occasionally, we don't have this because we're just rescanning an externally generated file without calculation. For that situation, we generate our local beam data object using the current enviroment, or, in simple cases, just fetch the necessary data from the process definition and environment.

⟨*Simulations: simulation: TBP*⟩+≡

```
procedure :: get_data => simulation_get_data
```

⟨*Simulations: procedures*⟩+≡

```
function simulation_get_data (simulation, alt) result (sdata)
  class(simulation_t), intent(in) :: simulation
  logical, intent(in), optional :: alt
  type(event_sample_data_t) :: sdata
  type(process_t), pointer :: process
  type(beam_data_t), pointer :: beam_data
  type(beam_structure_t), pointer :: beam_structure
  type(flavor_t), dimension(:), allocatable :: flv
  integer :: n, i
  logical :: enable_alt, construct_beam_data
  real(default) :: sqrts
  class(model_data_t), pointer :: model
  logical :: decay_rest_frame
  type(string_t) :: process_id
  enable_alt = .true.;  if (present (alt))  enable_alt = alt
```

```
call msg_debug (D_CORE, "simulation_get_data")
call msg_debug (D_CORE, "alternative setup", enable_alt)
if (enable_alt) then
   call sdata%init (simulation%n_prc, simulation%n_alt)
   do i = 1, simulation%n_alt
      sdata%md5sum_alt(i) = simulation%get_md5sum_alt (i)
   end do
else
   call sdata%init (simulation%n_prc)
end if
sdata%unweighted = simulation%unweighted
sdata%negative_weights = simulation%negative_weights
sdata%norm_mode = simulation%norm_mode
process => simulation%entry(1)%get_process_ptr ()
if (associated (process)) then
   beam_data => process%get_beam_data_ptr ()
   construct_beam_data = .false.
else
   n = simulation%entry(1)%n_in
   sqrts = simulation%local%get_sqrts ()
   beam_structure => simulation%local%beam_structure
   call beam_structure%check_against_n_in (n, construct_beam_data)
   if (construct_beam_data) then
      allocate (beam_data)
      model => simulation%local%model
      decay_rest_frame = &
           simulation%local%get_lval (var_str ("?decay_rest_frame"))
      call beam_data%init_structure (beam_structure, &
           sqrts, model, decay_rest_frame)
   else
      beam_data => null ()
   end if
end if
if (associated (beam_data)) then
   n = beam_data%get_n_in ()
   sdata%n_beam = n
   allocate (flv (n))
   flv = beam_data%get_flavor ()
   sdata%pdg_beam(:n) = flv%get_pdg ()
   sdata%energy_beam(:n) = beam_data%get_energy ()
   if (construct_beam_data)  deallocate (beam_data)
else
   n = simulation%entry(1)%n_in
   sdata%n_beam = n
   process_id = simulation%entry(1)%process_id
   call simulation%local%prclib%get_pdg_in_1 &
        (process_id, sdata%pdg_beam(:n))
   sdata%energy_beam(:n) = sqrts / n
end if
do i = 1, simulation%n_prc
   if (.not. simulation%entry(i)%valid) cycle
   process => simulation%entry(i)%get_process_ptr ()
   if (associated (process)) then
      sdata%proc_num_id(i) = process%get_num_id ()
```

```
      else
         process_id = simulation%entry(i)%process_id
         sdata%proc_num_id(i) = simulation%local%prclib%get_num_id (process_id)
      end if
      if (sdata%proc_num_id(i) == 0)  sdata%proc_num_id(i) = i
      if (simulation%entry(i)%has_integral) then
         sdata%cross_section(i) = simulation%entry(i)%integral
         sdata%error(i) = simulation%entry(i)%error
      end if
   end do
   sdata%total_cross_section = sum (sdata%cross_section)
   sdata%md5sum_prc = simulation%get_md5sum_prc ()
   sdata%md5sum_cfg = simulation%get_md5sum_cfg ()
   if (simulation%split_n_evt > 0 .or. simulation%split_n_kbytes > 0) then
      sdata%split_n_evt = simulation%split_n_evt
      sdata%split_n_kbytes = simulation%split_n_kbytes
      sdata%split_index = simulation%split_index
   end if
end function simulation_get_data
```

Return a default name for the current event sample. This is the process ID of the first process.

⟨*Simulations: simulation: TBP*⟩+≡

```
procedure :: get_default_sample_name => simulation_get_default_sample_name
```

⟨*Simulations: procedures*⟩+≡

```
function simulation_get_default_sample_name (simulation) result (sample)
  class(simulation_t), intent(in) :: simulation
  type(string_t) :: sample
  type(process_t), pointer :: process
  sample = "whizard"
  if (simulation%n_prc > 0) then
     process => simulation%entry(1)%get_process_ptr ()
     if (associated (process)) then
        sample = process%get_id ()
     end if
  end if
end function simulation_get_default_sample_name
```

⟨*Simulations: simulation: TBP*⟩+≡

```
procedure :: is_valid => simulation_is_valid
```

⟨*Simulations: procedures*⟩+≡

```
function simulation_is_valid (simulation) result (valid)
  class(simulation_t), intent(inout) :: simulation
  logical :: valid
  valid = simulation%valid
end function simulation_is_valid
```

33.8.12 Auxiliary

Call pacify: eliminate numerical noise.

⟨*Simulations: public*⟩+≡

```
    public :: pacify
```

⟨*Simulations: interfaces*⟩≡

```
    interface pacify
       module procedure pacify_simulation
    end interface
```

⟨*Simulations: procedures*⟩+≡

```
    subroutine pacify_simulation (simulation)
      class(simulation_t), intent(inout) :: simulation
      integer :: i, j
      i = simulation%i_prc
      if (i > 0) then
         call pacify (simulation%entry(i))
         do j = 1, simulation%n_alt
            call pacify (simulation%alt_entry(i,j))
         end do
      end if
    end subroutine pacify_simulation
```

Evaluate expressions for the currently selected process. (This is used only in the unit tests.)

⟨*Simulations: simulation: TBP*⟩+≡

```
    procedure :: evaluate_expressions => simulation_evaluate_expressions
```

⟨*Simulations: procedures*⟩+≡

```
    subroutine simulation_evaluate_expressions (simulation)
      class(simulation_t), intent(inout) :: simulation
      call simulation%entry(simulation%i_prc)%evaluate_expressions ()
    end subroutine simulation_evaluate_expressions
```

33.8.13 Unit tests

Test module, followed by the stand-alone unit-test procedures.

⟨`simulations_ut.f90`⟩≡

```
  ⟨File header⟩

  module simulations_ut
    use unit_tests
    use simulations_uti

  ⟨Standard module head⟩

  ⟨Simulations: public test⟩

  contains

  ⟨Simulations: test driver⟩

  end module simulations_ut
```

⟨simulations_uti.f90⟩≡

```
⟨File header⟩

module simulations_uti

  ⟨Use kinds⟩
    use kinds, only: i64
  ⟨Use strings⟩
    use io_units
    use ifiles
    use lexers
    use parser
    use flavors
    use interactions, only: reset_interaction_counter
    use prclib_stacks
    use phs_forests
    use event_base, only: generic_event_t
    use event_base, only: event_callback_t
    use eio_data
    use eio_base
    use eio_raw
    use eio_ascii
    use eio_callback
    use eval_trees
    use models
    use rt_data
    use event_streams
    use decays_ut, only: prepare_testbed
    use process_configurations_ut, only: prepare_test_library
    use compilations, only: compile_library
    use integrations, only: integrate_process

    use simulations

⟨Standard module head⟩

⟨Simulations: test declarations⟩

⟨Simulations: test auxiliary types⟩

contains

⟨Simulations: tests⟩

⟨Simulations: test auxiliary⟩

end module simulations_uti
```

API: driver for the unit tests below.

⟨*Simulations: public test*⟩≡

```
public :: simulations_test
```

⟨*Simulations: test driver*⟩≡

```
subroutine simulations_test (u, results)
```

```
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨Simulations: execute tests⟩
  end subroutine simulations_test
```

Initialization

Initialize a `simulation_t` object, including the embedded event records.

⟨*Simulations: execute tests*⟩≡

```
  call test (simulations_1, "simulations_1", &
       "initialization", &
       u, results)
```

⟨*Simulations: test declarations*⟩≡

```
  public :: simulations_1
```

⟨*Simulations: tests*⟩≡

```
  subroutine simulations_1 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname1, procname2
    type(rt_data_t), target :: global
    type(simulation_t), target :: simulation

    write (u, "(A)")  "* Test output: simulations_1"
    write (u, "(A)")  "*   Purpose: initialize simulation"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize processes"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()
    call global%set_log (var_str ("?omega_openmp"), &
         .false., is_known = .true.)
    call global%set_int (var_str ("seed"), &
         0, is_known = .true.)

    libname = "simulation_1a"
    procname1 = "simulation_1p"

    call prepare_test_library (global, libname, 1, [procname1])
    call compile_library (libname, global)

    call global%set_string (var_str ("$method"), &
         var_str ("unit_test"), is_known = .true.)
    call global%set_string (var_str ("$phs_method"), &
         var_str ("single"), is_known = .true.)
    call global%set_string (var_str ("$integration_method"),&
         var_str ("midpoint"), is_known = .true.)
    call global%set_log (var_str ("?vis_history"),&
         .false., is_known = .true.)
    call global%set_log (var_str ("?integration_timer"),&
         .false., is_known = .true.)
```

```
    call global%set_log (var_str ("?recover_beams"), &
         .false., is_known = .true.)

    call global%set_real (var_str ("sqrts"),&
         1000._default, is_known = .true.)

    call global%it_list%init ([1], [1000])

    call global%set_string (var_str ("$run_id"), &
         var_str ("simulations1"), is_known = .true.)
    call integrate_process (procname1, global, local_stack=.true.)

    procname2 = "sim_extra"

    call prepare_test_library (global, libname, 1, [procname2])
    call compile_library (libname, global)
    call global%set_string (var_str ("$run_id"), &
         var_str ("simulations2"), is_known = .true.)


    write (u, "(A)")  "* Initialize event generation"
    write (u, "(A)")

    call global%set_string (var_str ("$sample"), &
         var_str ("sim1"), is_known = .true.)
    call integrate_process (procname2, global, local_stack=.true.)

    call simulation%init ([procname1, procname2], .false., .true., global)
    call simulation%init_process_selector ()
    call simulation%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Write the event record for the first process"
    write (u, "(A)")

    call simulation%write_event (u, i_prc = 1)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call simulation%final ()
    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: simulations_1"

  end subroutine simulations_1
```

Weighted events

Generate events for a single process.

⟨*Simulations: execute tests*⟩+≡

```
    call test (simulations_2, "simulations_2", &
         "weighted events", &
         u, results)
```

⟨*Simulations: test declarations*⟩+≡

```
    public :: simulations_2
```

⟨*Simulations: tests*⟩+≡

```
    subroutine simulations_2 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname1
      type(rt_data_t), target :: global
      type(simulation_t), target :: simulation
      type(event_sample_data_t) :: data

      write (u, "(A)")  "* Test output: simulations_2"
      write (u, "(A)")  "*   Purpose: generate events for a single process"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize processes"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()
      call global%set_log (var_str ("?omega_openmp"), &
           .false., is_known = .true.)
      call global%set_int (var_str ("seed"), &
           0, is_known = .true.)

      libname = "simulation_2a"
      procname1 = "simulation_2p"

      call prepare_test_library (global, libname, 1, [procname1])
      call compile_library (libname, global)

      call global%append_log (&
           var_str ("?rebuild_events"), .true., intrinsic = .true.)

      call global%set_string (var_str ("$method"), &
           var_str ("unit_test"), is_known = .true.)
      call global%set_string (var_str ("$phs_method"), &
           var_str ("single"), is_known = .true.)
      call global%set_string (var_str ("$integration_method"),&
           var_str ("midpoint"), is_known = .true.)
      call global%set_log (var_str ("?vis_history"),&
           .false., is_known = .true.)
      call global%set_log (var_str ("?integration_timer"),&
           .false., is_known = .true.)
      call global%set_log (var_str ("?recover_beams"), &
           .false., is_known = .true.)

      call global%set_real (var_str ("sqrts"),&
           1000._default, is_known = .true.)
```

```
    call global%it_list%init ([1], [1000])

    call global%set_string (var_str ("$run_id"), &
         var_str ("simulations1"), is_known = .true.)
    call integrate_process (procname1, global, local_stack=.true.)

    write (u, "(A)")  "* Initialize event generation"
    write (u, "(A)")

    call global%set_log (var_str ("?unweighted"), &
         .false., is_known = .true.)
    call simulation%init ([procname1], .true., .true., global)
    call simulation%init_process_selector ()

    data = simulation%get_data ()
    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Generate three events"
    write (u, "(A)")

    call simulation%generate (3)
    call simulation%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Write the event record for the last event"
    write (u, "(A)")

    call simulation%write_event (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call simulation%final ()
    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: simulations_2"

  end subroutine simulations_2
```

Unweighted events

Generate events for a single process.

⟨*Simulations: execute tests*⟩+≡

```
  call test (simulations_3, "simulations_3", &
       "unweighted events", &
       u, results)
```

⟨*Simulations: test declarations*⟩+≡

```
  public :: simulations_3
```

⟨*Simulations: tests*⟩+≡

```
  subroutine simulations_3 (u)
```

```
integer, intent(in) :: u
type(string_t) :: libname, procname1
type(rt_data_t), target :: global
type(simulation_t), target :: simulation
type(event_sample_data_t) :: data

write (u, "(A)")  "* Test output: simulations_3"
write (u, "(A)")  "*   Purpose: generate unweighted events &
     &for a single process"
write (u, "(A)")

write (u, "(A)")  "* Initialize processes"
write (u, "(A)")

call syntax_model_file_init ()

call global%global_init ()
call global%set_log (var_str ("?omega_openmp"), &
     .false., is_known = .true.)
call global%set_int (var_str ("seed"), &
     0, is_known = .true.)

libname = "simulation_3a"
procname1 = "simulation_3p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call global%append_log (&
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call global%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call global%set_string (var_str ("$phs_method"), &
     var_str ("single"), is_known = .true.)
call global%set_string (var_str ("$integration_method"),&
     var_str ("midpoint"), is_known = .true.)
call global%set_log (var_str ("?vis_history"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?recover_beams"), &
     .false., is_known = .true.)

call global%set_real (var_str ("sqrts"),&
     1000._default, is_known = .true.)

call global%it_list%init ([1], [1000])

call global%set_string (var_str ("$run_id"), &
     var_str ("simulations1"), is_known = .true.)
call integrate_process (procname1, global, local_stack=.true.)

write (u, "(A)")  "* Initialize event generation"
```

```
    write (u, "(A)")

    call simulation%init ([procname1], .true., .true., global)
    call simulation%init_process_selector ()

    data = simulation%get_data ()
    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Generate three events"
    write (u, "(A)")

    call simulation%generate (3)
    call simulation%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Write the event record for the last event"
    write (u, "(A)")

    call simulation%write_event (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call simulation%final ()
    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: simulations_3"

  end subroutine simulations_3
```

Simulating process with structure functions

Generate events for a single process.

⟨*Simulations: execute tests*⟩+≡

```
  call test (simulations_4, "simulations_4", &
       "process with structure functions", &
       u, results)
```

⟨*Simulations: test declarations*⟩+≡

```
  public :: simulations_4
```

⟨*Simulations: tests*⟩+≡

```
  subroutine simulations_4 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname1
    type(rt_data_t), target :: global
    type(flavor_t) :: flv
    type(string_t) :: name
    type(simulation_t), target :: simulation
    type(event_sample_data_t) :: data

    write (u, "(A)")  "* Test output: simulations_4"
```

```
write (u, "(A)")  "*   Purpose: generate events for a single process &
     &with structure functions"
write (u, "(A)")

write (u, "(A)")  "* Initialize processes"
write (u, "(A)")

call syntax_model_file_init ()
call syntax_phs_forest_init ()

call global%global_init ()
call global%set_log (var_str ("?omega_openmp"), &
     .false., is_known = .true.)
call global%set_int (var_str ("seed"), &
     0, is_known = .true.)

libname = "simulation_4a"
procname1 = "simulation_4p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call global%append_log (&
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call global%set_string (var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call global%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call global%set_string (var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call global%set_string (var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call global%set_log (var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call global%set_real (var_str ("sqrts"),&
     1000._default, is_known = .true.)
call global%model_set_real (var_str ("ms"), &
     0._default)
call global%set_log (var_str ("?vis_history"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?recover_beams"), &
     .false., is_known = .true.)

call reset_interaction_counter ()

call flv%init (25, global%model)
name = flv%get_name ()
```

```
    call global%beam_structure%init_sf ([name, name], [1])
    call global%beam_structure%set_sf (1, 1, var_str ("sf_test_1"))

    write (u, "(A)")  "* Integrate"
    write (u, "(A)")

    call global%it_list%init ([1], [1000])

    call global%set_string (var_str ("$run_id"), &
         var_str ("r1"), is_known = .true.)
    call integrate_process (procname1, global, local_stack=.true.)

    write (u, "(A)")  "* Initialize event generation"
    write (u, "(A)")

    call global%set_log (var_str ("?unweighted"), &
         .false., is_known = .true.)
    call global%set_string (var_str ("$sample"), &
         var_str ("simulations4"), is_known = .true.)
    call simulation%init ([procname1], .true., .true., global)
    call simulation%init_process_selector ()

    data = simulation%get_data ()
    call data%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Generate three events"
    write (u, "(A)")

    call simulation%generate (3)
    call simulation%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Write the event record for the last event"
    write (u, "(A)")

    call simulation%write_event (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call simulation%final ()
    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: simulations_4"

  end subroutine simulations_4
```

Event I/O

Generate event for a test process, write to file and reread.

⟨*Simulations: execute tests*⟩+≡

```
call test (simulations_5, "simulations_5", &
     "raw event I/O", &
     u, results)
```

⟨*Simulations: test declarations*⟩+≡

```
public :: simulations_5
```

⟨*Simulations: tests*⟩+≡

```
subroutine simulations_5 (u)
  integer, intent(in) :: u
  type(string_t) :: libname, procname1, sample
  type(rt_data_t), target :: global
  class(eio_t), allocatable :: eio
  type(simulation_t), allocatable, target :: simulation

  write (u, "(A)")  "* Test output: simulations_5"
  write (u, "(A)")  "*   Purpose: generate events for a single process"
  write (u, "(A)")  "*            write to file and reread"
  write (u, "(A)")

  write (u, "(A)")  "* Initialize processes"
  write (u, "(A)")

  call syntax_model_file_init ()

  call global%global_init ()
  call global%set_log (var_str ("?omega_openmp"), &
       .false., is_known = .true.)
  call global%set_int (var_str ("seed"), &
       0, is_known = .true.)

  libname = "simulation_5a"
  procname1 = "simulation_5p"

  call prepare_test_library (global, libname, 1, [procname1])
  call compile_library (libname, global)

  call global%append_log (&
       var_str ("?rebuild_events"), .true., intrinsic = .true.)

  call global%set_string (var_str ("$method"), &
       var_str ("unit_test"), is_known = .true.)
  call global%set_string (var_str ("$phs_method"), &
       var_str ("single"), is_known = .true.)
  call global%set_string (var_str ("$integration_method"),&
       var_str ("midpoint"), is_known = .true.)
  call global%set_log (var_str ("?vis_history"),&
       .false., is_known = .true.)
  call global%set_log (var_str ("?integration_timer"),&
       .false., is_known = .true.)
  call global%set_log (var_str ("?recover_beams"), &
       .false., is_known = .true.)

  call global%set_real (var_str ("sqrts"),&
```

```
     1000._default, is_known = .true.)

call global%it_list%init ([1], [1000])

call global%set_string (var_str ("$run_id"), &
     var_str ("simulations5"), is_known = .true.)
call integrate_process (procname1, global, local_stack=.true.)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call global%set_log (var_str ("?unweighted"), &
     .false., is_known = .true.)
sample = "simulations5"
call global%set_string (var_str ("$sample"), &
     sample, is_known = .true.)
allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

allocate (eio_raw_t :: eio)
call eio%init_out (sample)

write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1)
call simulation%write_event (u)
call simulation%write_event (eio)

call eio%final ()
deallocate (eio)
call simulation%final ()
deallocate (simulation)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event from file"
write (u, "(A)")

call global%set_log (var_str ("?update_sqme"), &
     .true., is_known = .true.)
call global%set_log (var_str ("?update_weight"), &
     .true., is_known = .true.)
call global%set_log (var_str ("?recover_beams"), &
     .false., is_known = .true.)

allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()
allocate (eio_raw_t :: eio)
call eio%init_in (sample)
```

```
    call simulation%read_event (eio)
    call simulation%write_event (u)

    write (u, "(A)")
    write (u, "(A)")  "* Recalculate process instance"
    write (u, "(A)")

    call simulation%recalculate ()
    call simulation%evaluate_expressions ()
    call simulation%write_event (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio%final ()
    call simulation%final ()
    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: simulations_5"

  end subroutine simulations_5
```

Event I/O

Generate event for a real process with structure functions, write to file and reread.

⟨*Simulations: execute tests*⟩+≡

```
  call test (simulations_6, "simulations_6", &
       "raw event I/O with structure functions", &
       u, results)
```

⟨*Simulations: test declarations*⟩+≡

```
  public :: simulations_6
```

⟨*Simulations: tests*⟩+≡

```
  subroutine simulations_6 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname1, sample
    type(rt_data_t), target :: global
    class(eio_t), allocatable :: eio
    type(simulation_t), allocatable, target :: simulation
    type(flavor_t) :: flv
    type(string_t) :: name

    write (u, "(A)")  "* Test output: simulations_6"
    write (u, "(A)")  "*   Purpose: generate events for a single process"
    write (u, "(A)")  "*            write to file and reread"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize process and integrate"
    write (u, "(A)")
```

```
call syntax_model_file_init ()

call global%global_init ()
call global%set_log (var_str ("?omega_openmp"), &
     .false., is_known = .true.)
call global%set_int (var_str ("seed"), &
     0, is_known = .true.)

libname = "simulation_6"
procname1 = "simulation_6p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call global%append_log (&
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call global%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call global%set_string (var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call global%set_string (var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call global%set_log (var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call global%set_log (var_str ("?vis_history"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?recover_beams"), &
     .false., is_known = .true.)

call global%set_real (var_str ("sqrts"),&
     1000._default, is_known = .true.)
call global%model_set_real (var_str ("ms"), &
     0._default)

call flv%init (25, global%model)
name = flv%get_name ()

call global%beam_structure%init_sf ([name, name], [1])
call global%beam_structure%set_sf (1, 1, var_str ("sf_test_1"))

call global%it_list%init ([1], [1000])

call global%set_string (var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global, local_stack=.true.)
```

```
write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call reset_interaction_counter ()

call global%set_log (var_str ("?unweighted"), &
     .false., is_known = .true.)
sample = "simulations6"
call global%set_string (var_str ("$sample"), &
     sample, is_known = .true.)
allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

allocate (eio_raw_t :: eio)
call eio%init_out (sample)

write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1)
call pacify (simulation)
call simulation%write_event (u, verbose = .true., testflag = .true.)
call simulation%write_event (eio)

call eio%final ()
deallocate (eio)
call simulation%final ()
deallocate (simulation)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event from file"
write (u, "(A)")

call reset_interaction_counter ()

call global%set_log (var_str ("?update_sqme"), &
     .true., is_known = .true.)
call global%set_log (var_str ("?update_weight"), &
     .true., is_known = .true.)

allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()
allocate (eio_raw_t :: eio)
call eio%init_in (sample)

call simulation%read_event (eio)
call simulation%write_event (u, verbose = .true., testflag = .true.)

write (u, "(A)")
```

```
    write (u, "(A)")  "* Recalculate process instance"
    write (u, "(A)")

    call simulation%recalculate ()
    call simulation%evaluate_expressions ()
    call simulation%write_event (u, verbose = .true., testflag = .true.)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call eio%final ()
    call simulation%final ()
    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: simulations_6"

  end subroutine simulations_6
```

Automatic Event I/O

Generate events with raw-format event file as cache: generate, reread, append.

⟨*Simulations: execute tests*⟩+≡

```
  call test (simulations_7, "simulations_7", &
       "automatic raw event I/O", &
       u, results)
```

⟨*Simulations: test declarations*⟩+≡

```
  public :: simulations_7
```

⟨*Simulations: tests*⟩+≡

```
  subroutine simulations_7 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname1, sample
    type(rt_data_t), target :: global
    type(string_t), dimension(0) :: empty_string_array
    type(event_sample_data_t) :: data
    type(event_stream_array_t) :: es_array
    type(simulation_t), allocatable, target :: simulation
    type(flavor_t) :: flv
    type(string_t) :: name

    write (u, "(A)")  "* Test output: simulations_7"
    write (u, "(A)")  "*   Purpose: generate events for a single process"
    write (u, "(A)")  "*            write to file and reread"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize process and integrate"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()
    call global%init_fallback_model &
```

```
     (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"))

call global%set_log (var_str ("?omega_openmp"), &
     .false., is_known = .true.)
call global%set_int (var_str ("seed"), &
     0, is_known = .true.)

libname = "simulation_7"
procname1 = "simulation_7p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call global%append_log (&
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call global%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call global%set_string (var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call global%set_string (var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call global%set_log (var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call global%set_log (var_str ("?vis_history"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?recover_beams"), &
     .false., is_known = .true.)

call global%set_real (var_str ("sqrts"),&
     1000._default, is_known = .true.)
call global%model_set_real (var_str ("ms"), &
     0._default)

call flv%init (25, global%model)
name = flv%get_name ()

call global%beam_structure%init_sf ([name, name], [1])
call global%beam_structure%set_sf (1, 1, var_str ("sf_test_1"))

call global%it_list%init ([1], [1000])

call global%set_string (var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global, local_stack=.true.)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")
```

```
call reset_interaction_counter ()

call global%set_log (var_str ("?unweighted"), &
     .false., is_known = .true.)
sample = "simulations7"
call global%set_string (var_str ("$sample"), &
     sample, is_known = .true.)
allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
call es_array%init (sample, [var_str ("raw")], global, data)

write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1, es_array)

call es_array%final ()
call simulation%final ()
deallocate (simulation)

write (u, "(A)")  "* Re-read the event from file and generate another one"
write (u, "(A)")

call global%set_log (&
     var_str ("?rebuild_events"), .false., is_known = .true.)

call reset_interaction_counter ()

allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
call es_array%init (sample, empty_string_array, global, data, &
     input = var_str ("raw"))

call simulation%generate (2, es_array)

call pacify (simulation)
call simulation%write_event (u, verbose = .true.)

call es_array%final ()
call simulation%final ()
deallocate (simulation)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Re-read both events from file"
    write (u, "(A)")

    call reset_interaction_counter ()

    allocate (simulation)
    call simulation%init ([procname1], .true., .true., global)
    call simulation%init_process_selector ()

    data%md5sum_prc = simulation%get_md5sum_prc ()
    data%md5sum_cfg = simulation%get_md5sum_cfg ()
    call es_array%init (sample, empty_string_array, global, data, &
         input = var_str ("raw"))

    call simulation%generate (2, es_array)

    call pacify (simulation)
    call simulation%write_event (u, verbose = .true.)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call es_array%final ()
    call simulation%final ()
    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: simulations_7"

  end subroutine simulations_7
```

Rescanning Events

Generate events and rescan the resulting raw event file.

⟨*Simulations: execute tests*⟩+≡

```
  call test (simulations_8, "simulations_8", &
       "rescan raw event file", &
       u, results)
```

⟨*Simulations: test declarations*⟩+≡

```
  public :: simulations_8
```

⟨*Simulations: tests*⟩+≡

```
  subroutine simulations_8 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname1, sample
    type(rt_data_t), target :: global
    type(string_t), dimension(0) :: empty_string_array
    type(event_sample_data_t) :: data
    type(event_stream_array_t) :: es_array
    type(simulation_t), allocatable, target :: simulation
    type(flavor_t) :: flv
```

```
type(string_t) :: name

write (u, "(A)")  "* Test output: simulations_8"
write (u, "(A)")  "*   Purpose: generate events for a single process"
write (u, "(A)")  "*            write to file and rescan"
write (u, "(A)")

write (u, "(A)")  "* Initialize process and integrate"
write (u, "(A)")

call syntax_model_file_init ()

call global%global_init ()
call global%init_fallback_model &
     (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"))

call global%set_log (var_str ("?omega_openmp"), &
     .false., is_known = .true.)
call global%set_int (var_str ("seed"), &
     0, is_known = .true.)

libname = "simulation_8"
procname1 = "simulation_8p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call global%append_log (&
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call global%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call global%set_string (var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call global%set_string (var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call global%set_log (var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call global%set_log (var_str ("?vis_history"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?recover_beams"), &
     .false., is_known = .true.)

call global%set_real (var_str ("sqrts"),&
     1000._default, is_known = .true.)
call global%model_set_real (var_str ("ms"), &
     0._default)
```

```
call flv%init (25, global%model)
name = flv%get_name ()

call global%beam_structure%init_sf ([name, name], [1])
call global%beam_structure%set_sf (1, 1, var_str ("sf_test_1"))

call global%it_list%init ([1], [1000])

call global%set_string (var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global, local_stack=.true.)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call reset_interaction_counter ()

call global%set_log (var_str ("?unweighted"), &
     .false., is_known = .true.)
sample = "simulations8"
call global%set_string (var_str ("$sample"), &
     sample, is_known = .true.)
allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
write (u, "(1x,A,A,A)")  "MD5 sum (proc)   = '", data%md5sum_prc, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
call es_array%init (sample, [var_str ("raw")], global, &
     data)

write (u, "(A)")
write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1, es_array)

call pacify (simulation)
call simulation%write_event (u, verbose = .true., testflag = .true.)

call es_array%final ()
call simulation%final ()
deallocate (simulation)

write (u, "(A)")
write (u, "(A)")  "* Re-read the event from file"
write (u, "(A)")

call reset_interaction_counter ()
```

```
allocate (simulation)
call simulation%init ([procname1], .false., .false., global)
call simulation%init_process_selector ()

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = ""
write (u, "(1x,A,A,A)")  "MD5 sum (proc)   = '", data%md5sum_prc, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
call es_array%init (sample, empty_string_array, global, data, &
     input = var_str ("raw"), input_sample = sample, allow_switch = .false.)

call simulation%rescan (1, es_array, global = global)

write (u, "(A)")

call pacify (simulation)
call simulation%write_event (u, verbose = .true., testflag = .true.)

call es_array%final ()
call simulation%final ()
deallocate (simulation)

write (u, "(A)")
write (u, "(A)")  "* Re-read again and recalculate"
write (u, "(A)")

call reset_interaction_counter ()

call global%set_log (var_str ("?update_sqme"), &
     .true., is_known = .true.)
call global%set_log (var_str ("?update_event"), &
     .true., is_known = .true.)

allocate (simulation)
call simulation%init ([procname1], .false., .false., global)
call simulation%init_process_selector ()

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = ""
write (u, "(1x,A,A,A)")  "MD5 sum (proc)   = '", data%md5sum_prc, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
call es_array%init (sample, empty_string_array, global, data, &
     input = var_str ("raw"), input_sample = sample, allow_switch = .false.)

call simulation%rescan (1, es_array, global = global)

write (u, "(A)")

call pacify (simulation)
call simulation%write_event (u, verbose = .true., testflag = .true.)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"
```

```
    call es_array%final ()
    call simulation%final ()
    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: simulations_8"

  end subroutine simulations_8
```

Rescanning Check

Generate events and rescan with process mismatch.

⟨*Simulations: execute tests*⟩+≡

```
  call test (simulations_9, "simulations_9", &
       "rescan mismatch", &
       u, results)
```

⟨*Simulations: test declarations*⟩+≡

```
  public :: simulations_9
```

⟨*Simulations: tests*⟩+≡

```
  subroutine simulations_9 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname1, sample
    type(rt_data_t), target :: global
    type(string_t), dimension(0) :: empty_string_array
    type(event_sample_data_t) :: data
    type(event_stream_array_t) :: es_array
    type(simulation_t), allocatable, target :: simulation
    type(flavor_t) :: flv
    type(string_t) :: name
    logical :: error

    write (u, "(A)")  "* Test output: simulations_9"
    write (u, "(A)")  "*   Purpose: generate events for a single process"
    write (u, "(A)")  "*            write to file and rescan"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize process and integrate"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()
    call global%init_fallback_model &
         (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"))

    call global%set_log (var_str ("?omega_openmp"), &
         .false., is_known = .true.)
    call global%set_int (var_str ("seed"), &
         0, is_known = .true.)

    libname = "simulation_9"
```

```
procname1 = "simulation_9p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call global%append_log (&
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call global%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call global%set_string (var_str ("$phs_method"), &
     var_str ("wood"), is_known = .true.)
call global%set_string (var_str ("$integration_method"),&
     var_str ("vamp"), is_known = .true.)
call global%set_log (var_str ("?use_vamp_equivalences"),&
     .true., is_known = .true.)
call global%set_log (var_str ("?vis_history"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?recover_beams"), &
     .false., is_known = .true.)

call global%set_real (var_str ("sqrts"),&
     1000._default, is_known = .true.)
call global%model_set_real (var_str ("ms"), &
     0._default)

call flv%init (25, global%model)
name = flv%get_name ()

call global%beam_structure%init_sf ([name, name], [1])
call global%beam_structure%set_sf (1, 1, var_str ("sf_test_1"))

call global%it_list%init ([1], [1000])

call global%set_string (var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global, local_stack=.true.)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call reset_interaction_counter ()

call global%set_log (var_str ("?unweighted"), &
     .false., is_known = .true.)
sample = "simulations9"
call global%set_string (var_str ("$sample"), &
     sample, is_known = .true.)
```

```
allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

call simulation%write (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize raw event file"
write (u, "(A)")

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = simulation%get_md5sum_cfg ()
write (u, "(1x,A,A,A)")  "MD5 sum (proc)   = '", data%md5sum_prc, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
call es_array%init (sample, [var_str ("raw")], global, &
     data)

write (u, "(A)")
write (u, "(A)")  "* Generate an event"
write (u, "(A)")

call simulation%generate (1, es_array)

call es_array%final ()
call simulation%final ()
deallocate (simulation)

write (u, "(A)")  "* Initialize event generation for different parameters"
write (u, "(A)")

call reset_interaction_counter ()

allocate (simulation)
call simulation%init ([procname1, procname1], .false., .false., global)
call simulation%init_process_selector ()

call simulation%write (u)

write (u, "(A)")
write (u, "(A)")  "* Attempt to re-read the events (should fail)"
write (u, "(A)")

data%md5sum_prc = simulation%get_md5sum_prc ()
data%md5sum_cfg = ""
write (u, "(1x,A,A,A)")  "MD5 sum (proc)   = '", data%md5sum_prc, "'"
write (u, "(1x,A,A,A)")  "MD5 sum (config) = '", data%md5sum_cfg, "'"
call es_array%init (sample, empty_string_array, global, data, &
     input = var_str ("raw"), input_sample = sample, &
     allow_switch = .false., error = error)

write (u, "(1x,A,L1)")  "error = ", error

call simulation%rescan (1, es_array, global = global)
```

```
    call es_array%final ()
    call simulation%final ()
    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: simulations_9"

  end subroutine simulations_9
```

Alternative weights

Generate an event for a single process and reweight it in a simultaneous calculation.

⟨*Simulations: execute tests*⟩+≡

```
  call test (simulations_10, "simulations_10", &
       "alternative weight", &
       u, results)
```

⟨*Simulations: test declarations*⟩+≡

```
  public :: simulations_10
```

⟨*Simulations: tests*⟩+≡

```
  subroutine simulations_10 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname1, expr_text
    type(rt_data_t), target :: global
    type(rt_data_t), dimension(1), target :: alt_env
    type(ifile_t) :: ifile
    type(stream_t) :: stream
    type(parse_tree_t) :: pt_weight
    type(simulation_t), target :: simulation
    type(event_sample_data_t) :: data

    write (u, "(A)")  "* Test output: simulations_10"
    write (u, "(A)")  "*   Purpose: reweight event"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize processes"
    write (u, "(A)")

    call syntax_model_file_init ()
    call syntax_pexpr_init ()

    call global%global_init ()
    call global%set_log (var_str ("?omega_openmp"), &
         .false., is_known = .true.)
    call global%set_int (var_str ("seed"), &
         0, is_known = .true.)

    libname = "simulation_10a"
    procname1 = "simulation_10p"

    call prepare_test_library (global, libname, 1, [procname1])
```

```
call compile_library (libname, global)

call global%append_log (&
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call global%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call global%set_string (var_str ("$phs_method"), &
     var_str ("single"), is_known = .true.)
call global%set_string (var_str ("$integration_method"),&
     var_str ("midpoint"), is_known = .true.)
call global%set_log (var_str ("?vis_history"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?recover_beams"), &
     .false., is_known = .true.)

call global%set_real (var_str ("sqrts"),&
     1000._default, is_known = .true.)

call global%it_list%init ([1], [1000])

call global%set_string (var_str ("$run_id"), &
     var_str ("simulations1"), is_known = .true.)
call integrate_process (procname1, global, local_stack=.true.)

write (u, "(A)")  "* Initialize alternative environment with custom weight"
write (u, "(A)")

call alt_env(1)%local_init (global)
call alt_env(1)%activate ()

expr_text = "2"
write (u, "(A,A)")  "weight = ", char (expr_text)
write (u, *)

call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_weight, stream, .true.)
call stream_final (stream)
alt_env(1)%pn%weight_expr => pt_weight%get_root_ptr ()
call alt_env(1)%write_expr (u)

write (u, "(A)")
write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call global%set_log (var_str ("?unweighted"), &
```

```
      .false., is_known = .true.)
  call simulation%init ([procname1], .true., .true., global, alt_env=alt_env)
  call simulation%init_process_selector ()

  data = simulation%get_data ()
  call data%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Generate an event"
  write (u, "(A)")

  call simulation%generate (1)
  call simulation%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Write the event record for the last event"
  write (u, "(A)")

  call simulation%write_event (u)

  write (u, "(A)")
  write (u, "(A)")  "* Write the event record for the alternative setup"
  write (u, "(A)")

  call simulation%write_alt_event (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call simulation%final ()
  call global%final ()

  call syntax_model_file_final ()
  call syntax_pexpr_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: simulations_10"

end subroutine simulations_10
```

Decays

Generate an event with subsequent partonic decays.

⟨*Simulations: execute tests*⟩+≡

```
call test (simulations_11, "simulations_11", &
     "decay", &
     u, results)
```

⟨*Simulations: test declarations*⟩+≡

```
public :: simulations_11
```

⟨*Simulations: tests*⟩+≡

```
subroutine simulations_11 (u)
  integer, intent(in) :: u
```

```
type(rt_data_t), target :: global
type(prclib_entry_t), pointer :: lib
type(string_t) :: prefix, procname1, procname2
type(simulation_t), target :: simulation

write (u, "(A)")  "* Test output: simulations_11"
write (u, "(A)")  "*   Purpose: apply decay"
write (u, "(A)")

write (u, "(A)")  "* Initialize processes"
write (u, "(A)")

call syntax_model_file_init ()

call global%global_init ()
allocate (lib)
call global%add_prclib (lib)

call global%set_int (var_str ("seed"), &
     0, is_known = .true.)
call global%set_log (var_str ("?recover_beams"), &
     .false., is_known = .true.)

prefix = "simulation_11"
procname1 = prefix // "_p"
procname2 = prefix // "_d"
call prepare_testbed &
     (global%prclib, global%process_stack, &
     prefix, global%os_data, &
     scattering=.true., decay=.true.)

call global%select_model (var_str ("Test"))
call global%model%set_par (var_str ("ff"), 0.4_default)
call global%model%set_par (var_str ("mf"), &
     global%model%get_real (var_str ("ff")) &
     * global%model%get_real (var_str ("ms")))
call global%model%set_unstable (25, [procname2])

write (u, "(A)")  "* Initialize simulation object"
write (u, "(A)")

call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

write (u, "(A)")  "* Generate event"
write (u, "(A)")

call simulation%generate (1)
call simulation%write (u)

write (u, *)

call simulation%write_event (u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"
    write (u, "(A)")

    call simulation%final ()
    call global%final ()

    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: simulations_11"

  end subroutine simulations_11
```

Split Event Files

Generate event for a real process with structure functions and write to file, accepting a limit for the number of events per file.

⟨*Simulations: execute tests*⟩+≡

```
  call test (simulations_12, "simulations_12", &
       "split event files", &
       u, results)
```

⟨*Simulations: test declarations*⟩+≡

```
  public :: simulations_12
```

⟨*Simulations: tests*⟩+≡

```
  subroutine simulations_12 (u)
    integer, intent(in) :: u
    type(string_t) :: libname, procname1, sample
    type(rt_data_t), target :: global
    class(eio_t), allocatable :: eio
    type(simulation_t), allocatable, target :: simulation
    type(flavor_t) :: flv
    integer :: i_evt

    write (u, "(A)")  "* Test output: simulations_12"
    write (u, "(A)")  "*   Purpose: generate events for a single process"
    write (u, "(A)")  "*            and write to split event files"
    write (u, "(A)")

    write (u, "(A)")  "* Initialize process and integrate"
    write (u, "(A)")

    call syntax_model_file_init ()

    call global%global_init ()
    call global%set_log (var_str ("?omega_openmp"), &
         .false., is_known = .true.)
    call global%set_int (var_str ("seed"), &
         0, is_known = .true.)

    libname = "simulation_12"
```

```
procname1 = "simulation_12p"

call prepare_test_library (global, libname, 1, [procname1])
call compile_library (libname, global)

call global%append_log (&
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call global%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call global%set_string (var_str ("$phs_method"), &
     var_str ("single"), is_known = .true.)
call global%set_string (var_str ("$integration_method"),&
     var_str ("midpoint"), is_known = .true.)
call global%set_log (var_str ("?vis_history"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?recover_beams"), &
     .false., is_known = .true.)

call global%set_real (var_str ("sqrts"),&
     1000._default, is_known = .true.)
call global%model_set_real (var_str ("ms"), &
     0._default)

call flv%init (25, global%model)

call global%it_list%init ([1], [1000])

call global%set_string (var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global, local_stack=.true.)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call global%set_log (var_str ("?unweighted"), &
     .false., is_known = .true.)
sample = "simulations_12"
call global%set_string (var_str ("$sample"), &
     sample, is_known = .true.)
call global%set_int (var_str ("sample_split_n_evt"), &
     2, is_known = .true.)
call global%set_int (var_str ("sample_split_index"), &
     42, is_known = .true.)
allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()
```

```
    call simulation%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Initialize ASCII event file"
    write (u, "(A)")

    allocate (eio_ascii_short_t :: eio)
    select type (eio)
    class is (eio_ascii_t);  call eio%set_parameters ()
    end select
    call eio%init_out (sample, data = simulation%get_data ())

    write (u, "(A)")  "* Generate 5 events, distributed among three files"

    do i_evt = 1, 5
       call simulation%generate (1)
       call simulation%write_event (eio)
    end do

    call eio%final ()
    deallocate (eio)
    call simulation%final ()
    deallocate (simulation)

    write (u, *)
    call display_file ("simulations_12.42.short.evt", u)
    write (u, *)
    call display_file ("simulations_12.43.short.evt", u)
    write (u, *)
    call display_file ("simulations_12.44.short.evt", u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: simulations_12"

  end subroutine simulations_12
```

Auxiliary: display file contents.

⟨*Simulations: public test auxiliary*⟩≡

```
  public :: display_file
```

⟨*Simulations: test auxiliary*⟩≡

```
  subroutine display_file (file, u)
    use io_units, only: free_unit
    character(*), intent(in) :: file
    integer, intent(in) :: u
    character(256) :: buffer
    integer :: u_file
    write (u, "(3A)")  "* Contents of file '", file, "':"
    write (u, *)
```

```
      u_file = free_unit ()
      open (u_file, file = file, action = "read", status = "old")
      do
         read (u_file, "(A)", end = 1)  buffer
         write (u, "(A)")  trim (buffer)
      end do
1     continue
    end subroutine display_file
```

Callback

Generate events and execute a callback in place of event I/O.

⟨*Simulations: execute tests*⟩+≡

```
    call test (simulations_13, "simulations_13", &
         "callback", &
         u, results)
```

⟨*Simulations: test declarations*⟩+≡

```
    public :: simulations_13
```

⟨*Simulations: tests*⟩+≡

```
    subroutine simulations_13 (u)
      integer, intent(in) :: u
      type(string_t) :: libname, procname1, sample
      type(rt_data_t), target :: global
      class(eio_t), allocatable :: eio
      type(simulation_t), allocatable, target :: simulation
      type(flavor_t) :: flv
      integer :: i_evt
      type(simulations_13_callback_t) :: event_callback

      write (u, "(A)")  "* Test output: simulations_13"
      write (u, "(A)")  "*   Purpose: generate events for a single process"
      write (u, "(A)")  "*            and execute callback"
      write (u, "(A)")

      write (u, "(A)")  "* Initialize process and integrate"
      write (u, "(A)")

      call syntax_model_file_init ()

      call global%global_init ()
      call global%set_log (var_str ("?omega_openmp"), &
           .false., is_known = .true.)
      call global%set_int (var_str ("seed"), &
           0, is_known = .true.)

      libname = "simulation_13"
      procname1 = "simulation_13p"

      call prepare_test_library (global, libname, 1, [procname1])
      call compile_library (libname, global)

      call global%append_log (&
```

```
     var_str ("?rebuild_phase_space"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_grids"), .true., intrinsic = .true.)
call global%append_log (&
     var_str ("?rebuild_events"), .true., intrinsic = .true.)

call global%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known = .true.)
call global%set_string (var_str ("$phs_method"), &
     var_str ("single"), is_known = .true.)
call global%set_string (var_str ("$integration_method"),&
     var_str ("midpoint"), is_known = .true.)
call global%set_log (var_str ("?vis_history"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)
call global%set_log (var_str ("?recover_beams"), &
     .false., is_known = .true.)

call global%set_real (var_str ("sqrts"),&
     1000._default, is_known = .true.)

call flv%init (25, global%model)

call global%it_list%init ([1], [1000])

call global%set_string (var_str ("$run_id"), &
     var_str ("r1"), is_known = .true.)
call integrate_process (procname1, global, local_stack=.true.)

write (u, "(A)")  "* Initialize event generation"
write (u, "(A)")

call global%set_log (var_str ("?unweighted"), &
     .false., is_known = .true.)
sample = "simulations_13"
call global%set_string (var_str ("$sample"), &
     sample, is_known = .true.)

allocate (simulation)
call simulation%init ([procname1], .true., .true., global)
call simulation%init_process_selector ()

write (u, "(A)")  "* Prepare callback object"
write (u, "(A)")

event_callback%u = u
call global%set_event_callback (event_callback)

write (u, "(A)")  "* Initialize callback I/O object"
write (u, "(A)")

allocate (eio_callback_t :: eio)
select type (eio)
```

```
      class is (eio_callback_t)
         call eio%set_parameters (callback = event_callback, &
              count_interval = 3)
      end select
      call eio%init_out (sample, data = simulation%get_data ())

      write (u, "(A)")  "* Generate 7 events, with callback every 3 events"
      write (u, "(A)")

      do i_evt = 1, 7
         call simulation%generate (1)
         call simulation%write_event (eio)
      end do

      call eio%final ()
      deallocate (eio)
      call simulation%final ()
      deallocate (simulation)

      write (u, "(A)")
      write (u, "(A)")  "* Cleanup"

      call global%final ()

      write (u, "(A)")
      write (u, "(A)")  "* Test output end: simulations_13"

    end subroutine simulations_13
```

The callback object and procedure. In the type extension, we can store the output channel u so we know where to write into.

⟨*Simulations: test auxiliary types*⟩≡

```
    type, extends (event_callback_t) :: simulations_13_callback_t
       integer :: u
     contains
       procedure :: write => simulations_13_callback_write
       procedure :: proc => simulations_13_callback
    end type simulations_13_callback_t
```

⟨*Simulations: test auxiliary*⟩+≡

```
    subroutine simulations_13_callback_write (event_callback, unit)
      class(simulations_13_callback_t), intent(in) :: event_callback
      integer, intent(in), optional :: unit
      integer :: u
      u = given_output_unit (unit)
      write (u, "(1x,A)")  "Hello"
    end subroutine simulations_13_callback_write

    subroutine simulations_13_callback (event_callback, i, event)
      class(simulations_13_callback_t), intent(in) :: event_callback
      integer(i64), intent(in) :: i
      class(generic_event_t), intent(in) :: event
      write (event_callback%u, "(A,I0)")  "hello event #", i
```

```
end subroutine simulations_13_callback
```

Chapter 34

More Unit Tests

This chapter collects some procedures for testing that can't be provided at the point where the corresponding modules are defined, because they use other modules of a different level.

(We should move them back, collecting the high-level functionality in init/final hooks that we can set at runtime.)

34.1 Expression Testing

Expression objects are part of process and event objects, but the process and event object modules should not depend on the implementation of expressions. Here, we collect unit tests that depend on expression implementation.

⟨expr_tests_ut.f90⟩≡

```
⟨File header⟩
module expr_tests_ut

  use unit_tests
  use expr_tests_uti

⟨Standard module head⟩

⟨Expr tests: public test⟩

contains

⟨Expr tests: test driver⟩

end module expr_tests_ut
```

⟨expr_tests_uti.f90⟩≡

```
⟨File header⟩

module expr_tests_uti

  ⟨Use kinds⟩
  ⟨Use strings⟩
    use format_defs, only: FMT_12
    use format_utils, only: write_separator
```

```
    use os_interface
    use sm_qcd
    use lorentz
    use ifiles
    use lexers
    use parser
    use model_data
    use interactions, only: reset_interaction_counter
    use process_libraries
    use subevents
    use subevt_expr
    use rng_base
    use mci_base
    use phs_base
    use variables, only: var_list_t
    use eval_trees
    use models
    use prc_core
    use prc_test
    use process, only: process_t
    use instances, only: process_instance_t
    use events

    use rng_base_ut, only: rng_test_factory_t
    use phs_base_ut, only: phs_test_config_t
```

⟨*Standard module head*⟩

⟨*Expr tests: test declarations*⟩

```
contains
```

⟨*Expr tests: tests*⟩

```
end module expr_tests_uti
```

34.1.1 Test

This is the master for calling self-test procedures.

⟨*Expr tests: public test*⟩≡

```
    public :: subevt_expr_test
```

⟨*Expr tests: test driver*⟩≡

```
    subroutine subevt_expr_test (u, results)
      integer, intent(in) :: u
      type(test_results_t), intent(inout) :: results
    ⟨Expr tests: execute tests⟩
  end subroutine subevt_expr_test
```

Parton-event expressions

⟨*Expr tests: execute tests*⟩≡

```
call test (subevt_expr_1, "subevt_expr_1", &
     "parton-event expressions", &
     u, results)
```

⟨*Expr tests: test declarations*⟩≡

```
public :: subevt_expr_1
```

⟨*Expr tests: tests*⟩≡

```
subroutine subevt_expr_1 (u)
  integer, intent(in) :: u
  type(string_t) :: expr_text
  type(ifile_t) :: ifile
  type(stream_t) :: stream
  type(parse_tree_t) :: pt_cuts, pt_scale, pt_fac_scale, pt_ren_scale
  type(parse_tree_t) :: pt_weight
  type(parse_node_t), pointer :: pn_cuts, pn_scale, pn_fac_scale, pn_ren_scale
  type(parse_node_t), pointer :: pn_weight
  type(eval_tree_factory_t) :: expr_factory
  type(os_data_t) :: os_data
  type(model_t), target :: model
  type(parton_expr_t), target :: expr
  real(default) :: E, Ex, m
  type(vector4_t), dimension(6) :: p
  integer :: i, pdg
  logical :: passed
  real(default) :: scale, fac_scale, ren_scale, weight

  write (u, "(A)")  "* Test output: subevt_expr_1"
  write (u, "(A)")  "*   Purpose: Set up a subevt and associated &
       &process-specific expressions"
  write (u, "(A)")

  call syntax_pexpr_init ()

  call syntax_model_file_init ()
  call os_data_init (os_data)
  call model%read (var_str ("Test.mdl"), os_data)

  write (u, "(A)")  "* Expression texts"
  write (u, "(A)")


  expr_text = "all Pt > 100 [s]"
  write (u, "(A,A)")  "cuts = ", char (expr_text)
  call ifile_clear (ifile)
  call ifile_append (ifile, expr_text)
  call stream_init (stream, ifile)
  call parse_tree_init_lexpr (pt_cuts, stream, .true.)
  call stream_final (stream)
  pn_cuts => pt_cuts%get_root_ptr ()

  expr_text = "sqrts"
  write (u, "(A,A)")  "scale = ", char (expr_text)
  call ifile_clear (ifile)
  call ifile_append (ifile, expr_text)
```

```
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_scale, stream, .true.)
call stream_final (stream)
pn_scale => pt_scale%get_root_ptr ()

expr_text = "sqrts_hat"
write (u, "(A,A)")  "fac_scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_fac_scale, stream, .true.)
call stream_final (stream)
pn_fac_scale => pt_fac_scale%get_root_ptr ()

expr_text = "100"
write (u, "(A,A)")  "ren_scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_ren_scale, stream, .true.)
call stream_final (stream)
pn_ren_scale => pt_ren_scale%get_root_ptr ()

expr_text = "n_tot - n_in - n_out"
write (u, "(A,A)")  "weight = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_weight, stream, .true.)
call stream_final (stream)
pn_weight => pt_weight%get_root_ptr ()

call ifile_final (ifile)

write (u, "(A)")
write (u, "(A)")  "* Initialize process expr"
write (u, "(A)")

call expr%setup_vars (1000._default)
call expr%var_list%append_real (var_str ("tolerance"), 0._default)
call expr%link_var_list (model%get_var_list_ptr ())

call expr_factory%init (pn_cuts)
call expr%setup_selection (expr_factory)
call expr_factory%init (pn_scale)
call expr%setup_scale (expr_factory)
call expr_factory%init (pn_fac_scale)
call expr%setup_fac_scale (expr_factory)
call expr_factory%init (pn_ren_scale)
call expr%setup_ren_scale (expr_factory)
call expr_factory%init (pn_weight)
call expr%setup_weight (expr_factory)

call write_separator (u)
```

```
call expr%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Fill subevt and evaluate expressions"
write (u, "(A)")

call subevt_init (expr%subevt_t, 6)
E = 500._default
Ex = 400._default
m = 125._default
pdg = 25
p(1) = vector4_moving (E, sqrt (E**2 - m**2), 3)
p(2) = vector4_moving (E, -sqrt (E**2 - m**2), 3)
p(3) = vector4_moving (Ex, sqrt (Ex**2 - m**2), 3)
p(4) = vector4_moving (Ex, -sqrt (Ex**2 - m**2), 3)
p(5) = vector4_moving (Ex, sqrt (Ex**2 - m**2), 1)
p(6) = vector4_moving (Ex, -sqrt (Ex**2 - m**2), 1)

call expr%reset ()
do i = 1, 2
   call subevt_set_beam (expr%subevt_t, i, pdg, p(i), m**2)
end do
do i = 3, 4
   call subevt_set_incoming (expr%subevt_t, i, pdg, p(i), m**2)
end do
do i = 5, 6
   call subevt_set_outgoing (expr%subevt_t, i, pdg, p(i), m**2)
end do
expr%sqrts_hat = subevt_get_sqrts_hat (expr%subevt_t)
expr%n_in = 2
expr%n_out = 2
expr%n_tot = 4
expr%subevt_filled = .true.

call expr%evaluate (passed, scale, fac_scale, ren_scale, weight)

write (u, "(A,L1)")      "Event has passed      = ", passed
write (u, "(A," // FMT_12 // ")")  "Scale                 = ", scale
write (u, "(A," // FMT_12 // ")")  "Factorization scale   = ", fac_scale
write (u, "(A," // FMT_12 // ")")  "Renormalization scale = ", ren_scale
write (u, "(A," // FMT_12 // ")")  "Weight                = ", weight
write (u, "(A)")

call write_separator (u)
call expr%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call expr%final ()

call model%final ()
```

```
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: subevt_expr_1"

  end subroutine subevt_expr_1
```

Parton-event expressions

⟨*Expr tests: execute tests*⟩+≡

```
  call test (subevt_expr_2, "subevt_expr_2", &
      "parton-event expressions", &
      u, results)
```

⟨*Expr tests: test declarations*⟩+≡

```
  public :: subevt_expr_2
```

⟨*Expr tests: tests*⟩+≡

```
  subroutine subevt_expr_2 (u)
    integer, intent(in) :: u
    type(string_t) :: expr_text
    type(ifile_t) :: ifile
    type(stream_t) :: stream
    type(parse_tree_t) :: pt_selection
    type(parse_tree_t) :: pt_reweight, pt_analysis
    type(parse_node_t), pointer :: pn_selection
    type(parse_node_t), pointer :: pn_reweight, pn_analysis
    type(os_data_t) :: os_data
    type(model_t), target :: model
    type(eval_tree_factory_t) :: expr_factory
    type(event_expr_t), target :: expr
    real(default) :: E, Ex, m
    type(vector4_t), dimension(6) :: p
    integer :: i, pdg
    logical :: passed
    real(default) :: reweight
    logical :: analysis_flag

    write (u, "(A)")  "* Test output: subevt_expr_2"
    write (u, "(A)")  "*   Purpose: Set up a subevt and associated &
        &process-specific expressions"
    write (u, "(A)")

    call syntax_pexpr_init ()

    call syntax_model_file_init ()
    call os_data_init (os_data)
    call model%read (var_str ("Test.mdl"), os_data)

    write (u, "(A)")  "* Expression texts"
    write (u, "(A)")


    expr_text = "all Pt > 100 [s]"
```

```
write (u, "(A,A)")  "selection = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (pt_selection, stream, .true.)
call stream_final (stream)
pn_selection => pt_selection%get_root_ptr ()

expr_text = "n_tot - n_in - n_out"
write (u, "(A,A)")  "reweight = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_reweight, stream, .true.)
call stream_final (stream)
pn_reweight => pt_reweight%get_root_ptr ()

expr_text = "true"
write (u, "(A,A)")  "analysis = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (pt_analysis, stream, .true.)
call stream_final (stream)
pn_analysis => pt_analysis%get_root_ptr ()

call ifile_final (ifile)

write (u, "(A)")
write (u, "(A)")  "* Initialize process expr"
write (u, "(A)")

call expr%setup_vars (1000._default)
call expr%link_var_list (model%get_var_list_ptr ())
call expr%var_list%append_real (var_str ("tolerance"), 0._default)

call expr_factory%init (pn_selection)
call expr%setup_selection (expr_factory)
call expr_factory%init (pn_analysis)
call expr%setup_analysis (expr_factory)
call expr_factory%init (pn_reweight)
call expr%setup_reweight (expr_factory)

call write_separator (u)
call expr%write (u)
call write_separator (u)

write (u, "(A)")
write (u, "(A)")  "* Fill subevt and evaluate expressions"
write (u, "(A)")

call subevt_init (expr%subevt_t, 6)
E = 500._default
Ex = 400._default
```

```
    m = 125._default
    pdg = 25
    p(1) = vector4_moving (E, sqrt (E**2 - m**2), 3)
    p(2) = vector4_moving (E, -sqrt (E**2 - m**2), 3)
    p(3) = vector4_moving (Ex, sqrt (Ex**2 - m**2), 3)
    p(4) = vector4_moving (Ex, -sqrt (Ex**2 - m**2), 3)
    p(5) = vector4_moving (Ex, sqrt (Ex**2 - m**2), 1)
    p(6) = vector4_moving (Ex, -sqrt (Ex**2 - m**2), 1)

    call expr%reset ()
    do i = 1, 2
       call subevt_set_beam (expr%subevt_t, i, pdg, p(i), m**2)
    end do
    do i = 3, 4
       call subevt_set_incoming (expr%subevt_t, i, pdg, p(i), m**2)
    end do
    do i = 5, 6
       call subevt_set_outgoing (expr%subevt_t, i, pdg, p(i), m**2)
    end do
    expr%sqrts_hat = subevt_get_sqrts_hat (expr%subevt_t)
    expr%n_in = 2
    expr%n_out = 2
    expr%n_tot = 4
    expr%subevt_filled = .true.

    call expr%evaluate (passed, reweight, analysis_flag)

    write (u, "(A,L1)")      "Event has passed      = ", passed
    write (u, "(A," // FMT_12 // ")")  "Reweighting factor    = ", reweight
    write (u, "(A,L1)")      "Analysis flag         = ", analysis_flag
    write (u, "(A)")

    call write_separator (u)
    call expr%write (u)
    call write_separator (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call expr%final ()

    call model%final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: subevt_expr_2"

  end subroutine subevt_expr_2
```

Processes: handle partonic cuts

Initialize a process and process instance, choose a sampling point and fill the process instance, evaluating a given cut configuration.

We use the same trivial process as for the previous test. All momentum and state dependence is trivial, so we just test basic functionality.

⟨*Expr tests: execute tests*⟩+≡

```
call test (processes_5, "processes_5", &
     "handle cuts (partonic event)", &
     u, results)
```

⟨*Expr tests: test declarations*⟩+≡

```
public :: processes_5
```

⟨*Expr tests: tests*⟩+≡

```
subroutine processes_5 (u)
  integer, intent(in) :: u
  type(string_t) :: cut_expr_text
  type(ifile_t) :: ifile
  type(stream_t) :: stream
  type(parse_tree_t) :: parse_tree
  type(eval_tree_factory_t) :: expr_factory
  type(process_library_t), target :: lib
  type(string_t) :: libname
  type(string_t) :: procname
  type(string_t) :: run_id
  type(os_data_t) :: os_data
  type(qcd_t) :: qcd
  class(rng_factory_t), allocatable :: rng_factory
  type(model_t), pointer :: model_tmp
  class(model_data_t), pointer :: model
  type(var_list_t), target :: var_list
  type(process_t), allocatable, target :: process
  class(mci_t), allocatable :: mci_template
  class(phs_config_t), allocatable :: phs_config_template
  real(default) :: sqrts
  type(process_instance_t), allocatable, target :: process_instance

  write (u, "(A)")  "* Test output: processes_5"
  write (u, "(A)")  "*   Purpose: create a process &
       &and fill a process instance"
  write (u, "(A)")

  write (u, "(A)")  "* Prepare a cut expression"
  write (u, "(A)")

  call syntax_pexpr_init ()
  cut_expr_text = "all Pt > 100 [s]"
  call ifile_append (ifile, cut_expr_text)
  call stream_init (stream, ifile)
  call parse_tree_init_lexpr (parse_tree, stream, .true.)

  write (u, "(A)")  "* Build and initialize a test process"
  write (u, "(A)")

  libname = "processes5"
  procname = libname
  run_id = "run5"
  call os_data_init (os_data)
```

```
allocate (rng_test_factory_t :: rng_factory)
call prc_test_create_library (libname, lib)

call syntax_model_file_init ()
allocate (model_tmp)
call model_tmp%read (var_str ("Test.mdl"), os_data)
call var_list%init_snapshot (model_tmp%get_var_list_ptr ())
model => model_tmp

call reset_interaction_counter ()

allocate (process)
call process%init (procname, run_id, &
     lib, os_data, qcd, rng_factory, model)

call var_list%append_real (var_str ("tolerance"), 0._default)
call process%set_var_list (var_list)
call var_list%final ()

allocate (phs_test_config_t :: phs_config_template)
call process%setup_test_cores ()
call process%init_component &
   (1, .true., mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

sqrts = 1000
call process%setup_beams_sqrts (sqrts, i_core = 1)
call process%configure_phs ()
call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization and set cuts"
write (u, "(A)")

call process%setup_terms ()
call expr_factory%init (parse_tree%get_root_ptr ())
call process%set_cuts (expr_factory)
call process%write (.false., u, show_var_list=.true., show_expressions=.true.)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)

write (u, "(A)")
write (u, "(A)")  "* Inject a set of random numbers"
write (u, "(A)")

call process_instance%choose_mci (1)
call process_instance%set_mcpar ([0._default, 0._default])
```

```
write (u, "(A)")
write (u, "(A)")  "* Set up kinematics and subevt, check cuts (should fail)"
write (u, "(A)")

call process_instance%select_channel (1)
call process_instance%compute_seed_kinematics ()
call process_instance%compute_hard_kinematics ()
call process_instance%compute_eff_kinematics ()
call process_instance%evaluate_expressions ()
call process_instance%compute_other_channels ()

call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for another set (should succeed)"
write (u, "(A)")

call process_instance%reset ()
call process_instance%set_mcpar ([0.5_default, 0.125_default])
call process_instance%select_channel (1)
call process_instance%compute_seed_kinematics ()
call process_instance%compute_hard_kinematics ()
call process_instance%compute_eff_kinematics ()
call process_instance%evaluate_expressions ()
call process_instance%compute_other_channels ()
call process_instance%evaluate_trace ()

call process_instance%write (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for another set using convenience procedure &
     &(failure)"
write (u, "(A)")

call process_instance%evaluate_sqme (1, [0.0_default, 0.2_default])

call process_instance%write_header (u)

write (u, "(A)")
write (u, "(A)")  "* Evaluate for another set using convenience procedure &
     &(success)"
write (u, "(A)")

call process_instance%evaluate_sqme (1, [0.1_default, 0.2_default])

call process_instance%write_header (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call process_instance%final ()
deallocate (process_instance)

call process%final ()
```

```
    deallocate (process)

    call parse_tree_final (parse_tree)
    call stream_final (stream)
    call ifile_final (ifile)
    call syntax_pexpr_final ()

    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_5"

  end subroutine processes_5
```

Processes: scales and such

Initialize a process and process instance, choose a sampling point and fill the process instance, evaluating a given cut configuration.

We use the same trivial process as for the previous test. All momentum and state dependence is trivial, so we just test basic functionality.

⟨*Expr tests: execute tests*⟩+≡

```
  call test (processes_6, "processes_6", &
      "handle scales and weight (partonic event)", &
      u, results)
```

⟨*Expr tests: test declarations*⟩+≡

```
  public :: processes_6
```

⟨*Expr tests: tests*⟩+≡

```
  subroutine processes_6 (u)
    integer, intent(in) :: u
    type(string_t) :: expr_text
    type(ifile_t) :: ifile
    type(stream_t) :: stream
    type(parse_tree_t) :: pt_scale, pt_fac_scale, pt_ren_scale, pt_weight
    type(process_library_t), target :: lib
    type(string_t) :: libname
    type(string_t) :: procname
    type(string_t) :: run_id
    type(os_data_t) :: os_data
    type(qcd_t) :: qcd
    class(rng_factory_t), allocatable :: rng_factory
    type(model_t), pointer :: model_tmp
    class(model_data_t), pointer :: model
    type(var_list_t), target :: var_list
    type(process_t), allocatable, target :: process
    class(mci_t), allocatable :: mci_template
    class(phs_config_t), allocatable :: phs_config_template
    real(default) :: sqrts
    type(process_instance_t), allocatable, target :: process_instance
    type(eval_tree_factory_t) :: expr_factory

    write (u, "(A)")  "* Test output: processes_6"
```

```
write (u, "(A)")  "*   Purpose: create a process &
     &and fill a process instance"
write (u, "(A)")

write (u, "(A)")  "* Prepare expressions"
write (u, "(A)")

call syntax_pexpr_init ()

expr_text = "sqrts - 100 GeV"
write (u, "(A,A)")  "scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_scale, stream, .true.)
call stream_final (stream)

expr_text = "sqrts_hat"
write (u, "(A,A)")  "fac_scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_fac_scale, stream, .true.)
call stream_final (stream)

expr_text = "eval sqrt (M2) [collect [s]]"
write (u, "(A,A)")  "ren_scale = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_ren_scale, stream, .true.)
call stream_final (stream)

expr_text = "n_tot * n_in * n_out * (eval Phi / pi [s])"
write (u, "(A,A)")  "weight = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_weight, stream, .true.)
call stream_final (stream)

call ifile_final (ifile)

write (u, "(A)")
write (u, "(A)")  "* Build and initialize a test process"
write (u, "(A)")

libname = "processes4"
procname = libname
run_id = "run4"
call os_data_init (os_data)
allocate (rng_test_factory_t :: rng_factory)
call prc_test_create_library (libname, lib)
```

```
call syntax_model_file_init ()
allocate (model_tmp)
call model_tmp%read (var_str ("Test.mdl"), os_data)
call var_list%init_snapshot (model_tmp%get_var_list_ptr ())
model => model_tmp

call reset_interaction_counter ()

allocate (process)
call process%init (procname, run_id, &
     lib, os_data, qcd, rng_factory, model)

call process%set_var_list (var_list)
call var_list%final ()

call process%setup_test_cores ()
allocate (phs_test_config_t :: phs_config_template)
call process%init_component &
   (1, .true., mci_template, phs_config_template)

write (u, "(A)")  "* Prepare a trivial beam setup"
write (u, "(A)")

sqrts = 1000
call process%setup_beams_sqrts (sqrts, i_core = 1)
call process%configure_phs ()
call process%setup_mci ()

write (u, "(A)")  "* Complete process initialization and set cuts"
write (u, "(A)")

call process%setup_terms ()
call expr_factory%init (pt_scale%get_root_ptr ())
call process%set_scale (expr_factory)
call expr_factory%init (pt_fac_scale%get_root_ptr ())
call process%set_fac_scale (expr_factory)
call expr_factory%init (pt_ren_scale%get_root_ptr ())
call process%set_ren_scale (expr_factory)
call expr_factory%init (pt_weight%get_root_ptr ())
call process%set_weight (expr_factory)
call process%write (.false., u, show_expressions=.true.)

write (u, "(A)")
write (u, "(A)")  "* Create a process instance and evaluate"
write (u, "(A)")

allocate (process_instance)
call process_instance%init (process)
call process_instance%choose_mci (1)
call process_instance%evaluate_sqme (1, [0.5_default, 0.125_default])

call process_instance%write (u)

write (u, "(A)")
```

```
    write (u, "(A)")  "* Cleanup"

    call process_instance%final ()
    deallocate (process_instance)

    call process%final ()
    deallocate (process)

    call parse_tree_final (pt_scale)
    call parse_tree_final (pt_fac_scale)
    call parse_tree_final (pt_ren_scale)
    call parse_tree_final (pt_weight)
    call syntax_pexpr_final ()

    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: processes_6"

  end subroutine processes_6
```

Event expressions

After generating an event, fill the `subevt` and evaluate expressions for selection, reweighting, and analysis.

⟨*Expr tests: execute tests*⟩+≡

```
  call test (events_3, "events_3", &
       "expression evaluation", &
       u, results)
```

⟨*Expr tests: test declarations*⟩+≡

```
  public :: events_3
```

⟨*Expr tests: tests*⟩+≡

```
  subroutine events_3 (u)
    use processes_ut, only: prepare_test_process, cleanup_test_process
    integer, intent(in) :: u
    type(string_t) :: expr_text
    type(ifile_t) :: ifile
    type(stream_t) :: stream
    type(parse_tree_t) :: pt_selection, pt_reweight, pt_analysis
    type(eval_tree_factory_t) :: expr_factory
    type(event_t), allocatable, target :: event
    type(process_t), allocatable, target :: process
    type(process_instance_t), allocatable, target :: process_instance
    type(os_data_t) :: os_data
    type(model_t), pointer :: model
    type(var_list_t), target :: var_list

    write (u, "(A)")  "* Test output: events_3"
    write (u, "(A)")  "*   Purpose: generate an event and evaluate expressions"
    write (u, "(A)")
```

```
call syntax_pexpr_init ()

write (u, "(A)")  "* Expression texts"
write (u, "(A)")

expr_text = "all Pt > 100 [s]"
write (u, "(A,A)")  "selection = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (pt_selection, stream, .true.)
call stream_final (stream)

expr_text = "1 + sqrts_hat / sqrts"
write (u, "(A,A)")  "reweight = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_expr (pt_reweight, stream, .true.)
call stream_final (stream)

expr_text = "true"
write (u, "(A,A)")  "analysis = ", char (expr_text)
call ifile_clear (ifile)
call ifile_append (ifile, expr_text)
call stream_init (stream, ifile)
call parse_tree_init_lexpr (pt_analysis, stream, .true.)
call stream_final (stream)

call ifile_final (ifile)

write (u, "(A)")
write (u, "(A)")  "* Initialize test process event"

call os_data_init (os_data)

call syntax_model_file_init ()
allocate (model)
call model%read (var_str ("Test.mdl"), os_data)
call var_list%init_snapshot (model%get_var_list_ptr ())

allocate (process)
allocate (process_instance)
call prepare_test_process (process, process_instance, model)

call process%set_var_list (var_list)
call var_list%final ()

call process_instance%setup_event_data ()

write (u, "(A)")
write (u, "(A)")  "* Initialize event object and set expressions"

allocate (event)
```

```
  call event%basic_init ()

  call expr_factory%init (pt_selection%get_root_ptr ())
  call event%set_selection (expr_factory)
  call expr_factory%init (pt_reweight%get_root_ptr ())
  call event%set_reweight (expr_factory)
  call expr_factory%init (pt_analysis%get_root_ptr ())
  call event%set_analysis (expr_factory)

  call event%connect (process_instance, process%get_model_ptr ())
  call event%expr%var_list%append_real (var_str ("tolerance"), 0._default)
  call event%setup_expressions ()

  write (u, "(A)")
  write (u, "(A)")  "* Generate test process event"

  call process_instance%generate_weighted_event (1)

  write (u, "(A)")
  write (u, "(A)")  "* Fill event object and evaluate expressions"
  write (u, "(A)")

  call event%generate (1, [0.4_default, 0.4_default])
  call event%evaluate_expressions ()
  call event%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call event%final ()
  deallocate (event)

  call cleanup_test_process (process, process_instance)
  deallocate (process_instance)
  deallocate (process)

  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: events_3"

end subroutine events_3
```

Chapter 35

Top Level

The top level consists of

commands Defines generic command-list and command objects, and all specific implementations. Each command type provides a specific functionality. Together with the modules that provide expressions and variables, this module defines the Sindarin language.

whizard This module interprets streams of various kind in terms of the command language. It also contains the unit-test feature. We also define the externally visible procedures here, for the `WHIZARD` as a library.

main The driver for `WHIZARD` as a stand-alone program. Contains the command-line interpreter.

whizard_c_interface Alternative top-level procedures, for use in the context of a C-compatible caller program.

35.1 Commands

This module defines the command language of the main input file.

⟨commands.f90⟩≡

```
⟨File header⟩

module commands

⟨Use kinds⟩
⟨Use strings⟩
  use io_units
  use string_utils, only: lower_case, split_string
  use format_utils, only: write_indent
  use format_defs, only: FMT_14, FMT_19
  use diagnostics

  use physics_defs
  use sorting
  use sf_lhapdf, only: lhapdf_global_reset
  use os_interface
  use ifiles
  use lexers
  use syntax_rules
  use parser
  use analysis
  use pdg_arrays
  use variables, only: var_list_t, V_NONE, V_LOG, V_INT, V_REAL, V_CMPLX, V_STR, V_PDG
  use observables, only: var_list_check_observable
  use observables, only: var_list_check_result_var
  use eval_trees
  use models
  use auto_components
  use flavors
  use polarizations
  use particle_specifiers
  use process_libraries
  use process
  use instances
  use prclib_stacks
  use slha_interface
  use user_files
  use eio_data
  use rt_data

  use process_configurations
  use compilations, only: compile_library, compile_executable
  use integrations, only: integrate_process
  use event_streams
  use simulations

  use radiation_generator

⟨Standard module head⟩
```

⟨*Commands: public*⟩

⟨*Commands: types*⟩

⟨*Commands: variables*⟩

⟨*Commands: parameters*⟩

⟨*Commands: interfaces*⟩

```
contains
```

⟨*Commands: procedures*⟩

```
end module commands
```

35.1.1 The command type

The command type is a generic type that holds any command, compiled for execution.

Each command may come with its own local environment. The command list that determines this environment is allocated as `options`, if necessary. (It has to be allocated as a pointer because the type definition is recursive.) The local environment is available as a pointer which either points to the global environment, or is explicitly allocated and initialized.

⟨*Commands: types*⟩≡

```
type, abstract :: command_t
   type(parse_node_t), pointer :: pn => null ()
   class(command_t), pointer :: next => null ()
   type(parse_node_t), pointer :: pn_opt => null ()
   type(command_list_t), pointer :: options => null ()
   type(rt_data_t), pointer :: local => null ()
 contains
 ⟨Commands: command: TBP⟩
end type command_t
```

Finalizer: If there is an option list, finalize the option list and deallocate. If not, the local environment is just a pointer.

⟨*Commands: command: TBP*⟩≡

```
procedure :: final => command_final
```

⟨*Commands: procedures*⟩≡

```
recursive subroutine command_final (cmd)
  class(command_t), intent(inout) :: cmd
  if (associated (cmd%options)) then
     call cmd%options%final ()
     deallocate (cmd%options)
     call cmd%local%local_final ()
     deallocate (cmd%local)
  else
     cmd%local => null ()
  end if
end subroutine command_final
```

Allocate a command with the appropriate concrete type. Store the parse node pointer in the command object, so we can reference to it when compiling.

⟨*Commands: procedures*⟩+≡

```
subroutine dispatch_command (command, pn)
  class(command_t), intent(inout), pointer :: command
  type(parse_node_t), intent(in), target :: pn
  select case (char (parse_node_get_rule_key (pn)))
  case ("cmd_model")
     allocate (cmd_model_t :: command)
  case ("cmd_library")
     allocate (cmd_library_t :: command)
  case ("cmd_process")
     allocate (cmd_process_t :: command)
  case ("cmd_nlo")
     allocate (cmd_nlo_t :: command)
  case ("cmd_compile")
     allocate (cmd_compile_t :: command)
  case ("cmd_exec")
     allocate (cmd_exec_t :: command)
   case ("cmd_num", "cmd_complex", "cmd_real", "cmd_int", &
         "cmd_log_decl", "cmd_log", "cmd_string", "cmd_string_decl", &
         "cmd_alias", "cmd_result")
     allocate (cmd_var_t :: command)
  case ("cmd_slha")
     allocate (cmd_slha_t :: command)
  case ("cmd_show")
     allocate (cmd_show_t :: command)
  case ("cmd_clear")
     allocate (cmd_clear_t :: command)
  case ("cmd_expect")
     allocate (cmd_expect_t :: command)
  case ("cmd_beams")
     allocate (cmd_beams_t :: command)
  case ("cmd_beams_pol_density")
     allocate (cmd_beams_pol_density_t :: command)
  case ("cmd_beams_pol_fraction")
     allocate (cmd_beams_pol_fraction_t :: command)
  case ("cmd_beams_momentum")
     allocate (cmd_beams_momentum_t :: command)
  case ("cmd_beams_theta")
     allocate (cmd_beams_theta_t :: command)
  case ("cmd_beams_phi")
     allocate (cmd_beams_phi_t :: command)
  case ("cmd_cuts")
     allocate (cmd_cuts_t :: command)
  case ("cmd_scale")
     allocate (cmd_scale_t :: command)
  case ("cmd_fac_scale")
     allocate (cmd_fac_scale_t :: command)
  case ("cmd_ren_scale")
     allocate (cmd_ren_scale_t :: command)
  case ("cmd_weight")
```

```
   allocate (cmd_weight_t :: command)
case ("cmd_selection")
   allocate (cmd_selection_t :: command)
case ("cmd_reweight")
   allocate (cmd_reweight_t :: command)
case ("cmd_iterations")
   allocate (cmd_iterations_t :: command)
case ("cmd_integrate")
   allocate (cmd_integrate_t :: command)
case ("cmd_observable")
   allocate (cmd_observable_t :: command)
case ("cmd_histogram")
   allocate (cmd_histogram_t :: command)
case ("cmd_plot")
   allocate (cmd_plot_t :: command)
case ("cmd_graph")
   allocate (cmd_graph_t :: command)
case ("cmd_record")
   allocate (cmd_record_t :: command)
case ("cmd_analysis")
   allocate (cmd_analysis_t :: command)
case ("cmd_alt_setup")
   allocate (cmd_alt_setup_t :: command)
case ("cmd_unstable")
   allocate (cmd_unstable_t :: command)
case ("cmd_stable")
   allocate (cmd_stable_t :: command)
case ("cmd_polarized")
   allocate (cmd_polarized_t :: command)
case ("cmd_unpolarized")
   allocate (cmd_unpolarized_t :: command)
case ("cmd_sample_format")
   allocate (cmd_sample_format_t :: command)
case ("cmd_simulate")
   allocate (cmd_simulate_t :: command)
case ("cmd_rescan")
   allocate (cmd_rescan_t :: command)
case ("cmd_write_analysis")
   allocate (cmd_write_analysis_t :: command)
case ("cmd_compile_analysis")
   allocate (cmd_compile_analysis_t :: command)
case ("cmd_open_out")
   allocate (cmd_open_out_t :: command)
case ("cmd_close_out")
   allocate (cmd_close_out_t :: command)
case ("cmd_printf")
   allocate (cmd_printf_t :: command)
case ("cmd_scan")
   allocate (cmd_scan_t :: command)
case ("cmd_if")
   allocate (cmd_if_t :: command)
case ("cmd_include")
   allocate (cmd_include_t :: command)
case ("cmd_quit")
```

```
      allocate (cmd_quit_t :: command)
    case default
      print *, char (parse_node_get_rule_key (pn))
      call msg_bug ("Command not implemented")
    end select
    command%pn => pn
  end subroutine dispatch_command
```

Output. We allow for indentation so we can display a command tree.

⟨*Commands: command: TBP*⟩+≡

```
  procedure (command_write), deferred :: write
```

⟨*Commands: interfaces*⟩≡

```
  abstract interface
    subroutine command_write (cmd, unit, indent)
      import
      class(command_t), intent(in) :: cmd
      integer, intent(in), optional :: unit, indent
    end subroutine command_write
  end interface
```

Compile a command. The command type is already fixed, so this is a deferred type-bound procedure.

⟨*Commands: command: TBP*⟩+≡

```
  procedure (command_compile), deferred :: compile
```

⟨*Commands: interfaces*⟩+≡

```
  abstract interface
    subroutine command_compile (cmd, global)
      import
      class(command_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
    end subroutine command_compile
  end interface
```

Execute a command. This will use and/or modify the runtime data set. If the `quit` flag is set, the caller should terminate command execution.

⟨*Commands: command: TBP*⟩+≡

```
  procedure (command_execute), deferred :: execute
```

⟨*Commands: interfaces*⟩+≡

```
  abstract interface
    subroutine command_execute (cmd, global)
      import
      class(command_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
    end subroutine command_execute
  end interface
```

35.1.2 Options

The `options` command list is allocated, initialized, and executed, if the command is associated with an option text in curly braces. If present, a separate

local runtime data set `local` will be allocated and initialized; otherwise, `local` becomes a pointer to the global dataset.

For output, we indent the options list.

⟨*Commands: command: TBP*⟩+≡
```
procedure :: write_options => command_write_options
```

⟨*Commands: procedures*⟩+≡
```
recursive subroutine command_write_options (cmd, unit, indent)
  class(command_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: ind
  ind = 1;  if (present (indent))  ind = indent + 1
  if (associated (cmd%options))  call cmd%options%write (unit, ind)
end subroutine command_write_options
```

Compile the options list, if any. This implies initialization of the local environment. Should be done once the `pn_opt` node has been assigned (if applicable), but before the actual command compilation.

⟨*Commands: command: TBP*⟩+≡
```
procedure :: compile_options => command_compile_options
```

⟨*Commands: procedures*⟩+≡
```
recursive subroutine command_compile_options (cmd, global)
  class(command_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  if (associated (cmd%pn_opt)) then
     allocate (cmd%local)
     call cmd%local%local_init (global)
     call global%copy_globals (cmd%local)
     allocate (cmd%options)
     call cmd%options%compile (cmd%pn_opt, cmd%local)
     call global%restore_globals (cmd%local)
     call cmd%local%deactivate ()
  else
     cmd%local => global
  end if
end subroutine command_compile_options
```

Execute options. First prepare the local environment, then execute the command list.

⟨*Commands: command: TBP*⟩+≡
```
procedure :: execute_options => cmd_execute_options
```

⟨*Commands: procedures*⟩+≡
```
recursive subroutine cmd_execute_options (cmd, global)
  class(command_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  if (associated (cmd%options)) then
     call cmd%local%activate ()
     call cmd%options%execute (cmd%local)
  end if
end subroutine cmd_execute_options
```

This must be called after the parent command has been executed, to undo temporary modifications to the environment. Note that some modifications to `global` can become permanent.

⟨*Commands: command: TBP*⟩+≡
```
procedure :: reset_options => cmd_reset_options
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_reset_options (cmd, global)
  class(command_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  if (associated (cmd%options)) then
     call cmd%local%deactivate (global)
  end if
end subroutine cmd_reset_options
```

35.1.3 Specific command types

Model configuration

The command declares a model, looks for the specified file and loads it.

⟨*Commands: types*⟩+≡
```
type, extends (command_t) :: cmd_model_t
   private
   type(string_t) :: name
   type(string_t) :: scheme
 contains
 ⟨Commands: cmd model: TBP⟩
end type cmd_model_t
```

Output

⟨*Commands: cmd model: TBP*⟩≡
```
procedure :: write => cmd_model_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_model_write (cmd, unit, indent)
  class(cmd_model_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,1x,'""',A,'"')")  "model =", char (cmd%name)
end subroutine cmd_model_write
```

Compile. Get the model name and read the model from file, so it is readily available when the command list is executed. If the model has a scheme argument, take this into account.

Assign the model pointer in the `global` record, so it can be used for (read-only) variable lookup while compiling further commands.

⟨*Commands: cmd model: TBP*⟩+≡
```
procedure :: compile => cmd_model_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_model_compile (cmd, global)
  class(cmd_model_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_name, pn_arg, pn_scheme
  type(model_t), pointer :: model
  pn_name => cmd%pn%get_sub_ptr (3)
  pn_arg => pn_name%get_next_ptr ()
  if (associated (pn_arg)) then
     pn_scheme => pn_arg%get_sub_ptr ()
  else
     pn_scheme => null ()
  end if
  cmd%name = pn_name%get_string ()
  if (associated (pn_scheme)) then
     cmd%scheme = pn_scheme%get_string ()
     call preload_model (model, cmd%name, cmd%scheme)
  else
     cmd%scheme = ""
     call preload_model (model, cmd%name)
  end if
  global%model => model
  if (associated (global%model)) then
     call global%model%link_var_list (global%var_list)
  end if
contains
  subroutine preload_model (model, name, scheme)
    type(model_t), pointer, intent(out) :: model
    type(string_t), intent(in) :: name
    type(string_t), intent(in), optional :: scheme
    model => null ()
    if (associated (global%model)) then
       if (global%model%matches (name, scheme)) then
          model => global%model
       end if
    end if
    if (.not. associated (model)) then
       if (global%model_list%model_exists (name, scheme)) then
          model => global%model_list%get_model_ptr (name, scheme)
       else
          call global%read_model (name, model, scheme)
       end if
    end if
  end subroutine preload_model
end subroutine cmd_model_compile
```

Execute: Insert a pointer into the global data record and reassign the variable list.

⟨*Commands: cmd model: TBP*⟩+≡

```
procedure :: execute => cmd_model_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_model_execute (cmd, global)
  class(cmd_model_t), intent(inout) :: cmd
```

```
    type(rt_data_t), intent(inout), target :: global
    if (cmd%scheme /= "") then
       call global%select_model (cmd%name, cmd%scheme)
    else
       call global%select_model (cmd%name)
    end if
    if (.not. associated (global%model)) &
         call msg_fatal ("Switching to model '" &
         // char (cmd%name) // "': model not found")
  end subroutine cmd_model_execute
```

Library configuration

We configure a process library that should hold the subsequently defined processes. If the referenced library exists already, just make it the currently active one.

⟨*Commands: types*⟩+≡

```
  type, extends (command_t) :: cmd_library_t
     private
     type(string_t) :: name
   contains
   ⟨Commands: cmd library: TBP⟩
  end type cmd_library_t
```

Output.

⟨*Commands: cmd library: TBP*⟩≡

```
  procedure :: write => cmd_library_write
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_library_write (cmd, unit, indent)
    class(cmd_library_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u
    u = given_output_unit (unit)
    call write_indent (u, indent)
    write (u, "(1x,A,1x,'""',A,'"')")  "library =", char (cmd%name)
  end subroutine cmd_library_write
```

Compile. Get the library name.

⟨*Commands: cmd library: TBP*⟩+≡

```
  procedure :: compile => cmd_library_compile
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_library_compile (cmd, global)
    class(cmd_library_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_name
    pn_name => parse_node_get_sub_ptr (cmd%pn, 3)
    cmd%name = parse_node_get_string (pn_name)
  end subroutine cmd_library_compile
```

Execute: Initialize a new library and push it on the library stack (if it does not yet exist). Insert a pointer to the library into the global data record. Then, try to load the library unless the `rebuild` flag set.

⟨*Commands: cmd library: TBP*⟩+≡

```
procedure :: execute => cmd_library_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_library_execute (cmd, global)
  class(cmd_library_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(prclib_entry_t), pointer :: lib_entry
  type(process_library_t), pointer :: lib
  logical :: rebuild_library
  lib => global%prclib_stack%get_library_ptr (cmd%name)
  rebuild_library = &
       global%var_list%get_lval (var_str ("?rebuild_library"))
  if (.not. (associated (lib))) then
     allocate (lib_entry)
     call lib_entry%init (cmd%name)
     lib => lib_entry%process_library_t
     call global%add_prclib (lib_entry)
  else
     call global%update_prclib (lib)
  end if
  if (associated (lib) .and. .not. rebuild_library) then
     call lib%update_status (global%os_data)
  end if
end subroutine cmd_library_execute
```

Process configuration

We define a process-configuration command as a specific type. The incoming and outgoing particles are given evaluation-trees which we transform to PDG-code arrays. For transferring to `O'MEGA`, they are reconverted to strings.

For the incoming particles, we store parse nodes individually. We do not yet resolve the outgoing state, so we store just a single parse node.

This also includes the choice of method for the corresponding process: `omega` for `O'MEGA` matrix elements as Fortran code, `ovm` for `O'MEGA` matrix elements as a bytecode virtual machine, `test` for special processes, `unit_test` for internal test matrix elements generated by `WHIZARD`, `template` and `template_unity` for test matrix elements generated by `WHIZARD` as Fortran code similar to the `O'MEGA` code. If the one-loop program (OLP) `GoSam` is linked, also matrix elements from there (at leading and next-to-leading order) can be generated via `gosam`.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_process_t
   private
   type(string_t) :: id
   integer :: n_in  = 0
   type(parse_node_p), dimension(:), allocatable :: pn_pdg_in
   type(parse_node_t), pointer :: pn_out => null ()
 contains
 ⟨Commands: cmd process: TBP⟩
```

```
end type cmd_process_t
```

Output. The particle expressions are not resolved, so we just list the number of incoming particles.

⟨*Commands: cmd process: TBP*⟩≡

```
procedure :: write => cmd_process_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_process_write (cmd, unit, indent)
  class(cmd_process_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,A,A,I0,A)")  "process: ", char (cmd%id), " (", &
       size (cmd%pn_pdg_in), " -> X)"
  call cmd%write_options (u, indent)
end subroutine cmd_process_write
```

Compile. Find and assign the parse nodes.

⟨*Commands: cmd process: TBP*⟩+≡

```
procedure :: compile => cmd_process_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_process_compile (cmd, global)
  class(cmd_process_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_id, pn_in, pn_codes
  integer :: i
  pn_id => parse_node_get_sub_ptr (cmd%pn, 2)
  pn_in  => parse_node_get_next_ptr (pn_id, 2)
  cmd%pn_out => parse_node_get_next_ptr (pn_in, 2)
  cmd%pn_opt => parse_node_get_next_ptr (cmd%pn_out)
  call cmd%compile_options (global)
  cmd%id = parse_node_get_string (pn_id)
  cmd%n_in  = parse_node_get_n_sub (pn_in)
  pn_codes => parse_node_get_sub_ptr (pn_in)
  allocate (cmd%pn_pdg_in (cmd%n_in))
  do i = 1, cmd%n_in
     cmd%pn_pdg_in(i)%ptr => pn_codes
     pn_codes => parse_node_get_next_ptr (pn_codes)
  end do
end subroutine cmd_process_compile
```

Command execution. Evaluate the subevents, transform PDG codes into strings, and add the current process configuration to the process library.

The initial state will be unique (one or two particles). For the final state, we allow for expressions. The expressions will be expanded until we have a sum of final states. Each distinct final state will get its own process component.

To identify equivalent final states, we transform the final state into an array of PDG codes, which we sort and compare. If a particle entry is actually a PDG array, only the first entry in the array is used for the comparison. The

user should make sure that there is no overlap between different particles or arrays which would make the expansion ambiguous.

⟨*Commands: cmd process: TBP*⟩+≡

```
procedure :: execute => cmd_process_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_process_execute (cmd, global)
  class(cmd_process_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(pdg_array_t) :: pdg_in, pdg_out
  type(pdg_array_t), dimension(:), allocatable :: pdg_out_tab
  type(string_t), dimension(:), allocatable :: prt_in
  type(string_t) :: prt_out, prt_out1
  type(process_configuration_t) :: prc_config
  type(prt_expr_t) :: prt_expr_out
  type(prt_spec_t), dimension(:), allocatable :: prt_spec_in
  type(prt_spec_t), dimension(:), allocatable :: prt_spec_out
  type(var_list_t), pointer :: var_list
  integer, dimension(:), allocatable :: pdg
  integer, dimension(:), allocatable :: i_term
  integer :: i, j, n_in, n_out, n_terms, n_components
  logical :: nlo_fixed_order
  logical :: qcd_corr, qed_corr
  type(string_t), dimension(:), allocatable :: prt_in_nlo, prt_out_nlo
  type(radiation_generator_t) :: radiation_generator
  type(pdg_list_t) :: pl_in, pl_out, pl_excluded_gauge_splittings
  type(string_t) :: born_me_method
  type(string_t) :: real_tree_me_method
  type(string_t) :: loop_me_method
  type(string_t) :: correlation_me_method
  type(string_t) :: soft_mismatch_me_method
  type(string_t) :: dglap_me_method
  integer, dimension(:), allocatable :: i_list
  logical :: use_real_finite
  logical :: gks_active
  logical :: initial_state_colored
  logical :: has_structure_functions
  integer :: n_components_extra
  integer :: gks_multiplicity
  integer :: n_emitters
  integer, dimension(:), allocatable :: emitters
  integer :: n_components_init
  integer :: alpha_power, alphas_power
  logical :: requires_soft_mismatch, requires_dglap_remnants
  call msg_debug (D_CORE, "cmd_process_execute")
  var_list => cmd%local%get_var_list_ptr ()

  n_in = size (cmd%pn_pdg_in)
  allocate (prt_in (n_in), prt_spec_in (n_in))
  do i = 1, n_in
     pdg_in = &
          eval_pdg_array (cmd%pn_pdg_in(i)%ptr, var_list)
     prt_in(i) = make_flavor_string (pdg_in, cmd%local%model)
     prt_spec_in(i) = new_prt_spec (prt_in(i))
```

```
    end do
    call compile_prt_expr &
         (prt_expr_out, cmd%pn_out, var_list, cmd%local%model)
    call prt_expr_out%expand ()
    call scan_components ()

    nlo_fixed_order = cmd%local%nlo_fixed_order
    gks_multiplicity = var_list%get_ival (var_str ('gks_multiplicity'))
    gks_active = gks_multiplicity > 2
    call check_for_nlo_corrections ()

    born_me_method = var_list%get_sval (var_str ("$born_me_method"))
    use_real_finite = var_list%get_lval (var_str ('?nlo_use_real_partition'))
    if (nlo_fixed_order) then
       real_tree_me_method = var_list%get_sval (var_str ("$real_tree_me_method"))
       loop_me_method = var_list%get_sval (var_str ("$loop_me_method"))
       correlation_me_method = var_list%get_sval (var_str ("$correlation_me_method"))
       soft_mismatch_me_method = var_list%get_sval (var_str ("$soft_mismatch_me_method"))
       dglap_me_method = var_list%get_sval (var_str ("$dglap_me_method"))
       call check_nlo_options (cmd%local)
    end if
    if (any (cmd%local%selected_nlo_parts))  &
         call override_local_me_method (born_me_method)

    call determine_needed_components ()
    call prc_config%init (cmd%id, n_in, n_components_init, cmd%local, nlo_fixed_order)

    alpha_power = var_list%get_ival (var_str ("alpha_power"))
    alphas_power = var_list%get_ival (var_str ("alphas_power"))
    call prc_config%set_coupling_powers (alpha_power, alphas_power)

    call setup_components ()
    call prc_config%record (cmd%local)

  contains

  ⟨Commands: cmd process execute procedures⟩

  end subroutine cmd_process_execute
```

⟨*Commands: cmd process execute procedures*⟩≡

```
  subroutine override_local_me_method (me_method)
    type(string_t), intent(in) :: me_method
    call var_list%set_string (var_str ("$method"), me_method, is_known=.true.)
  end subroutine override_local_me_method
```

⟨*Commands: cmd process execute procedures*⟩+≡

```
  elemental function is_threshold (method)
    logical :: is_threshold
    type(string_t), intent(in) :: method
    is_threshold = method == var_str ("threshold")
  end function is_threshold
```

```
subroutine check_threshold_consistency ()
  if (nlo_fixed_order .and. is_threshold (born_me_method)) then
     if (.not. (is_threshold (real_tree_me_method) .and. is_threshold (loop_me_method) &
          .and. is_threshold (correlation_me_method))) then
          print *, 'born: ', char (born_me_method)
          print *, 'real: ', char (real_tree_me_method)
          print *, 'loop: ', char (loop_me_method)
          print *, 'correlation: ', char (correlation_me_method)
          call msg_fatal ("Inconsistent methods: All components need to be threshold")
     end if
  end if
end subroutine check_threshold_consistency
```

⟨*Commands: cmd process execute procedures*⟩+≡

```
subroutine check_for_nlo_corrections ()
  type(string_t) :: nlo_correction_type
  if (nlo_fixed_order .or. gks_active) then
     nlo_correction_type = var_list%get_sval (var_str ('$nlo_correction_type'))
     select case (char(nlo_correction_type))
     case ("QCD")
        qcd_corr = .true.; qed_corr = .false.
     case ("QED")
        qcd_corr = .false.; qed_corr = .true.
     case ("Full")
        qcd_corr =.true.; qed_corr = .true.
     case default
        call msg_fatal ("Invalid NLO correction type! Valid inputs are: QCD, QED, Full (default:
     end select
     call check_for_excluded_gauge_boson_splitting_partners ()
     call setup_radiation_generator ()
  end if
  if (nlo_fixed_order) then
     call radiation_generator%find_splittings ()
     nlo_fixed_order = radiation_generator%contains_emissions ()
     if (.not. nlo_fixed_order) call msg_warning &
          (arr = [var_str ("No NLO corrections found for process ") // cmd%id // var_str("."), &
          var_str ("Proceed with usual leading-order integration and simulation")])
  end if
end subroutine check_for_nlo_corrections
```

⟨*Commands: cmd process execute procedures*⟩+≡

```
subroutine check_for_excluded_gauge_boson_splitting_partners ()
  type(string_t) :: str_excluded_partners
  type(string_t), dimension(:), allocatable :: excluded_partners
  type(pdg_list_t) :: pl_tmp, pl_anti
  integer :: i, n_anti
  str_excluded_partners = var_list%get_sval &
       (var_str ("$exclude_gauge_splittings"))
  if (str_excluded_partners == "") then
     return
  else
     call split_string (str_excluded_partners, &
          var_str (":"), excluded_partners)
```

```
      call pl_tmp%init (size (excluded_partners))
      do i = 1, size (excluded_partners)
         call pl_tmp%set (i, &
              cmd%local%model%get_pdg (excluded_partners(i), .true.))
      end do
      call pl_tmp%create_antiparticles (pl_anti, n_anti)
      call pl_excluded_gauge_splittings%init (pl_tmp%get_size () + n_anti)
      do i = 1, pl_tmp%get_size ()
         call pl_excluded_gauge_splittings%set (i, pl_tmp%get(i))
      end do
      do i = 1, n_anti
         j = i + pl_tmp%get_size ()
         call pl_excluded_gauge_splittings%set (j, pl_anti%get(i))
      end do
   end if
end subroutine check_for_excluded_gauge_boson_splitting_partners
```

⟨*Commands: cmd process execute procedures*⟩+≡

```
subroutine determine_needed_components ()
   type(string_t) :: fks_method

   if (use_real_finite) then
      call radiation_generator%get_emitter_indices (emitters)
      n_emitters = size (emitters)
   end if

   if (nlo_fixed_order) then
      fks_method = var_list%get_sval (var_str ('$fks_mapping_type'))
      call check_threshold_consistency ()
      requires_soft_mismatch = fks_method == var_str ('resonances')
      n_components_extra = needed_extra_components (initial_state_colored, &
           has_structure_functions, requires_soft_mismatch, &
           use_real_finite, n_emitters)
      allocate (i_list (n_components_extra))
   else if (gks_active) then
      call radiation_generator%generate_multiple (gks_multiplicity, cmd%local%model)
      n_components_extra = radiation_generator%get_n_gks_states ()
   end if

   if (nlo_fixed_order .and. .not. use_real_finite) then
      n_components_init = n_components * n_components_extra
   else if (nlo_fixed_order .and. use_real_finite) then
      n_components_init = n_components * n_components_extra
   else if (gks_active) then
      n_components_init = n_components * (n_components_extra + 1)
   else
      n_components_init = n_components
   end if
end subroutine determine_needed_components
```

⟨*Commands: cmd process execute procedures*⟩+≡

```
subroutine setup_radiation_generator ()
   call split_prt (prt_spec_in, n_in, pl_in)
```

```
    call split_prt (prt_spec_out, n_out, pl_out)
    call radiation_generator%init (pl_in, pl_out, &
         pl_excluded_gauge_splittings, qcd = qcd_corr, qed = qed_corr)
    call radiation_generator%set_n (n_in, n_out, 0)
    initial_state_colored = pdg_in%has_colored_particles()
    has_structure_functions = global%beam_structure%get_n_record () > 0
    requires_dglap_remnants = initial_state_colored .and. has_structure_functions
    if (requires_dglap_remnants)  call radiation_generator%set_initial_state_emissions ()
    call radiation_generator%set_constraints (.false., .false., .true., .true.)
    call radiation_generator%setup_if_table (cmd%local%model)
  end subroutine setup_radiation_generator
```

⟨*Commands: cmd process execute procedures*⟩+≡

```
  subroutine scan_components ()
    n_terms = prt_expr_out%get_n_terms ()
    allocate (pdg_out_tab (n_terms))
    allocate (i_term (n_terms), source = 0)
    n_components = 0
    SCAN: do i = 1, n_terms
       if (allocated (pdg))  deallocate (pdg)
       call prt_expr_out%term_to_array (prt_spec_out, i)
       n_out = size (prt_spec_out)
       allocate (pdg (n_out))
       do j = 1, n_out
          prt_out = prt_spec_out(j)%to_string ()
          call split (prt_out, prt_out1, ":")
          pdg(j) = cmd%local%model%get_pdg (prt_out1)
       end do
       pdg_out = sort (pdg)
       do j = 1, n_components
          if (pdg_out == pdg_out_tab(j))  cycle SCAN
       end do
       n_components = n_components + 1
       i_term(n_components) = i
       pdg_out_tab(n_components) = pdg_out
    end do SCAN
  end subroutine scan_components
```

⟨*Commands: cmd process execute procedures*⟩+≡

```
  subroutine split_prt (prt, n_out, pl)
    type(prt_spec_t), intent(in), dimension(:), allocatable :: prt
    integer, intent(in) :: n_out
    type(pdg_list_t), intent(out) :: pl
    type(pdg_array_t) :: pdg
    type(string_t) :: prt_string, prt_tmp
    integer, parameter :: max_particle_number = 25
    integer, dimension(max_particle_number) :: i_particle
    integer :: i, j, n
    i_particle = 0
    call pl%init (n_out)
    do i = 1, n_out
       n = 1
       prt_string = prt(i)%to_string ()
```

```
      do
        call split (prt_string, prt_tmp, ":")
        if (prt_tmp /= "") then
          i_particle(n) = cmd%local%model%get_pdg (prt_tmp)
          n = n + 1
        else
          exit
        end if
      end do
      call pdg_array_init (pdg, n - 1)
      do j = 1, n - 1
        call pdg%set (j, i_particle(j))
      end do
      call pl%set (i, pdg)
      call pdg_array_delete (pdg)
    end do
  end subroutine split_prt
```

⟨*Commands: cmd process execute procedures*⟩+≡

```
  subroutine setup_components()
    integer :: i_comp
    call msg_debug (D_CORE, "setup_components")
    do i = 1, n_components
      call prt_expr_out%term_to_array (prt_spec_out, i_term(i))
      if (nlo_fixed_order) then
        associate (selected_nlo_parts => cmd%local%selected_nlo_parts)
          call set_component_list (i_list, i, n_components, &
               requires_dglap_remnants, requires_soft_mismatch, &
               use_real_finite, n_emitters)

          call override_local_me_method (born_me_method)
          i_comp = i
          call msg_debug (D_CORE, "Setting up this NLO component:", i_comp)
          call prc_config%setup_component (i_comp, &
               prt_spec_in, prt_spec_out, &
               cmd%local%model, var_list, BORN, &
               can_be_integrated = selected_nlo_parts (BORN))

          call radiation_generator%generate_real_particle_strings &
               (prt_in_nlo, prt_out_nlo)
          call override_local_me_method (real_tree_me_method)
          i_comp = n_components + i
          call msg_debug (D_CORE, "Setting up this NLO component:", i_comp)
          call prc_config%setup_component (i_comp, &
               new_prt_spec (prt_in_nlo), new_prt_spec (prt_out_nlo), &
               cmd%local%model, var_list, NLO_REAL, &
               can_be_integrated = selected_nlo_parts (NLO_REAL))

          call override_local_me_method (loop_me_method)
          i_comp = n_components * 2 + i
          call msg_debug (D_CORE, "Setting up this NLO component:", i_comp)
          call prc_config%setup_component (i_comp, &
               prt_spec_in, prt_spec_out, &
               cmd%local%model, var_list, NLO_VIRTUAL, &
```

```
            can_be_integrated = selected_nlo_parts (NLO_VIRTUAL))

         call override_local_me_method (correlation_me_method)
         i_comp = n_components * 3 + i
         call msg_debug (D_CORE, "Setting up this NLO component:", i_comp)
         call prc_config%setup_component (i_comp, &
              prt_spec_in, prt_spec_out, &
              cmd%local%model, var_list, NLO_SUBTRACTION, &
              can_be_integrated = selected_nlo_parts (NLO_SUBTRACTION))

         if (use_real_finite) then
            call override_local_me_method (real_tree_me_method)
            i_comp = n_components * 4 + i
            call msg_debug (D_CORE, "Setting up this NLO component:", i_comp)
            call prc_config%setup_component (i_comp, &
                 new_prt_spec (prt_in_nlo), new_prt_spec (prt_out_nlo), &
                 cmd%local%model, var_list, NLO_REAL, &
                 can_be_integrated = selected_nlo_parts (NLO_REAL))
         end if

         if (requires_dglap_remnants) then
            call override_local_me_method (dglap_me_method)
            i_comp = n_components * 4 + i
            call msg_debug (D_CORE, "Setting up this NLO component:", i_comp)
            call prc_config%setup_component (i_comp, &
                 prt_spec_in, prt_spec_out, &
                 cmd%local%model, var_list, NLO_DGLAP, &
                 can_be_integrated = selected_nlo_parts (NLO_DGLAP))
         end if

         if (requires_soft_mismatch) then
            call override_local_me_method (soft_mismatch_me_method)
            i_comp = n_components * 4 + i
            call msg_debug (D_CORE, "Setting up this NLO component:", i_comp)
            call prc_config%setup_component (i_comp, &
               prt_spec_in, prt_spec_out, &
               cmd%local%model, var_list, NLO_MISMATCH, &
               can_be_integrated = selected_nlo_parts (NLO_MISMATCH))
         end if

         call prc_config%set_component_associations (i_list, &
              requires_dglap_remnants, use_real_finite, requires_soft_mismatch)
      end associate
   else if (gks_active) then
      call override_local_me_method (var_str ("omega"))
      call prc_config%setup_component (i, prt_spec_in, prt_spec_out, &
           cmd%local%model, var_list, BORN, can_be_integrated = .true.)
      call radiation_generator%reset_queue ()
      do j = 1, n_components_extra
         prt_out_nlo =  radiation_generator%get_next_state ()
         call prc_config%setup_component (i + j, &
            new_prt_spec (prt_in), new_prt_spec (prt_out_nlo), &
            cmd%local%model, var_list, GKS, can_be_integrated = .false.)
      end do
```

```
      else
         call prc_config%setup_component (i, &
              prt_spec_in, prt_spec_out, &
              cmd%local%model, var_list, can_be_integrated = .true.)
      end if
    end do
  end subroutine setup_components
```

These three functions should be bundled with the logicals they depend on into an object (the pcm?).

⟨*Commands: procedures*⟩+≡

```
  subroutine check_nlo_options (local)
    type(rt_data_t), intent(in) :: local
    type(var_list_t), pointer :: var_list => null ()
    logical :: nlo, combined, powheg
    logical :: case_lo_but_any_other
    logical :: case_nlo_powheg_but_not_combined
    logical :: vamp_equivalences_enabled
    logical :: fixed_order_nlo_events
    var_list => local%get_var_list_ptr ()
    nlo = local%nlo_fixed_order
    combined = var_list%get_lval (var_str ('?combined_nlo_integration'))
    powheg = var_list%get_lval (var_str ('?powheg_matching'))
    case_lo_but_any_other = .not. nlo .and. any ([combined, powheg])
    case_nlo_powheg_but_not_combined = &
         nlo .and. powheg .and. .not. combined
    if (case_lo_but_any_other) then
       call msg_fatal ("Option mismatch: Leading order process is selected &
            &but either powheg_matching or combined_nlo_integration &
            &is set to true.")
    else if (case_nlo_powheg_but_not_combined) then
       call msg_fatal ("POWHEG requires the 'combined_nlo_integration'-option &
            &to be set to true.")
    end if
    fixed_order_nlo_events = var_list%get_lval (var_str ('?fixed_order_nlo_events'))
    if (fixed_order_nlo_events .and. .not. combined .and. &
         all (local%selected_nlo_parts)) &
       call msg_fatal ("Option mismatch: Fixed order NLO events of the full ", &
            [var_str ("process are requested, but ?combined_nlo_integration"), &
            var_str ("is false. You can either switch to the combined NLO"), &
            var_str ("integration mode or choose one individual NLO component"), &
            var_str ("to generate events with.")])
    vamp_equivalences_enabled = var_list%get_lval &
         (var_str ('?use_vamp_equivalences'))
    if (nlo .and. vamp_equivalences_enabled) &
         call msg_warning ("You have not disabled VAMP equivalences. ", &
              [var_str ("   Note that they are automatically switched off "), &
               var_str ("   for NLO calculations.")])
  end subroutine check_nlo_options
```

For a given process, there can be multiple Born, Real, Virtual, Special components. The layout is a grouped one, i.e. at first there are all Borns, then all Reals, etc.

⟨*Commands: procedures*⟩+≡

```
pure subroutine set_component_list (i_list, i, n_components, &
       requires_dglap_remnants, requires_soft_mismatch, &
       use_real_finite, n_emitters)
  integer, dimension(:), intent(out) :: i_list
  integer, intent(in) :: i, n_components, n_emitters
  logical, intent(in) :: requires_dglap_remnants
  logical, intent(in) :: use_real_finite, requires_soft_mismatch
  i_list(1) = i
  i_list(2) = i + n_components
  i_list(3) = i + 2 * n_components
  i_list(4) = i + 3 * n_components
  if (requires_dglap_remnants .or. requires_soft_mismatch .or. use_real_finite) then
     i_list(5) = i + 4 * n_components
  end if
end subroutine set_component_list
```

⟨*Commands: procedures*⟩+≡

```
pure function needed_extra_components (initial_state_colored, &
       has_structure_functions, requires_soft_mismatch, &
       use_real_finite, n_emitters) result (n)
  integer :: n
  logical, intent(in) :: initial_state_colored, &
       has_structure_functions, use_real_finite, &
       requires_soft_mismatch
  integer, intent(in) :: n_emitters
  if (initial_state_colored) then
     if (has_structure_functions) then
        n = 5
     else
        n = 4
     end if
  else if (use_real_finite) then
     n = 5
  else if (requires_soft_mismatch) then
     n = 5
  else
     n = 4
  end if
end function needed_extra_components
```

This is a method of the eval tree, but cannot be coded inside the `expressions` module since it uses the `model` and `flv` types which are not available there.

⟨*Commands: procedures*⟩+≡

```
function make_flavor_string (aval, model) result (prt)
  type(string_t) :: prt
  type(pdg_array_t), intent(in) :: aval
  type(model_t), intent(in), target :: model
  integer, dimension(:), allocatable :: pdg
  type(flavor_t), dimension(:), allocatable :: flv
  integer :: i
  pdg = aval
  allocate (flv (size (pdg)))
```

```
    call flv%init (pdg, model)
    if (size (pdg) /= 0) then
       prt = flv(1)%get_name ()
       do i = 2, size (flv)
          prt = prt // ":" // flv(i)%get_name ()
       end do
    else
       prt = "?"
    end if
  end function make_flavor_string
```

Create a pdg array from a particle-specification array

⟨*Commands: procedures*⟩+≡

```
  function make_pdg_array (prt, model) result (pdg_array)
    type(prt_spec_t), intent(in), dimension(:) :: prt
    type(model_t), intent(in) :: model
    integer, dimension(:), allocatable :: aval
    type(pdg_array_t) :: pdg_array
    type(flavor_t) :: flv
    integer :: k
    allocate (aval (size (prt)))
    do k = 1, size (prt)
      call flv%init (prt(k)%to_string (), model)
      aval (k) = flv%get_pdg ()
    end do
    pdg_array = aval
  end function make_pdg_array
```

Compile a (possible nested) expression, to obtain a particle-specifier expression which we can process further.

⟨*Commands: procedures*⟩+≡

```
  recursive subroutine compile_prt_expr (prt_expr, pn, var_list, model)
    type(prt_expr_t), intent(out) :: prt_expr
    type(parse_node_t), intent(in), target :: pn
    type(var_list_t), intent(in), target :: var_list
    type(model_t), intent(in), target :: model
    type(parse_node_t), pointer :: pn_entry, pn_term, pn_addition
    type(pdg_array_t) :: pdg
    type(string_t) :: prt_string
    integer :: n_entry, n_term, i
    select case (char (parse_node_get_rule_key (pn)))
    case ("prt_state_list")
       n_entry = parse_node_get_n_sub (pn)
       pn_entry => parse_node_get_sub_ptr (pn)
       if (n_entry == 1) then
          call compile_prt_expr (prt_expr, pn_entry, var_list, model)
       else
          call prt_expr%init_list (n_entry)
          select type (x => prt_expr%x)
          type is (prt_spec_list_t)
             do i = 1, n_entry
                call compile_prt_expr (x%expr(i), pn_entry, var_list, model)
                pn_entry => parse_node_get_next_ptr (pn_entry)
```

```
            end do
         end select
      end if
   case ("prt_state_sum")
      n_term = parse_node_get_n_sub (pn)
      pn_term => parse_node_get_sub_ptr (pn)
      pn_addition => pn_term
      if (n_term == 1) then
         call compile_prt_expr (prt_expr, pn_term, var_list, model)
      else
         call prt_expr%init_sum (n_term)
         select type (x => prt_expr%x)
         type is (prt_spec_sum_t)
            do i = 1, n_term
               call compile_prt_expr (x%expr(i), pn_term, var_list, model)
               pn_addition => parse_node_get_next_ptr (pn_addition)
               if (associated (pn_addition)) &
                    pn_term => parse_node_get_sub_ptr (pn_addition, 2)
            end do
         end select
      end if
   case ("cexpr")
      pdg = eval_pdg_array (pn, var_list)
      prt_string = make_flavor_string (pdg, model)
      call prt_expr%init_spec (new_prt_spec (prt_string))
   case default
      call parse_node_write_rec (pn)
      call msg_bug ("compile prt expr: impossible syntax rule")
   end select
end subroutine compile_prt_expr
```

Initiating a NLO calculation

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_nlo_t
  private
  integer, dimension(:), allocatable :: nlo_component
contains
  ⟨Commands: cmd nlo: TBP⟩
end type cmd_nlo_t
```

⟨*Commands: cmd nlo: TBP*⟩≡

```
procedure :: write => cmd_nlo_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_nlo_write (cmd, unit, indent)
  class(cmd_nlo_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
end subroutine cmd_nlo_write
```

As it is, the NLO calculation is switched on by putting nlo behind the process definition. This should be made nicer in the future.

⟨*Commands: cmd nlo: TBP*⟩+≡

```
procedure :: compile => cmd_nlo_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_nlo_compile (cmd, global)
  class(cmd_nlo_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_arg, pn_comp
  integer :: i, n_comp
  pn_arg => parse_node_get_sub_ptr (cmd%pn, 3)
  if (associated (pn_arg)) then
     n_comp = parse_node_get_n_sub (pn_arg)
  else
     n_comp = 0
  end if
  if (n_comp > 0) then
     pn_comp => parse_node_get_sub_ptr (pn_arg)
     allocate (cmd%nlo_component (n_comp))
     do i = 1, n_comp
        cmd%nlo_component(i) = component_status &
             (eval_string (pn_comp, global%var_list))
        pn_comp => parse_node_get_next_ptr (pn_comp)
     end do
  end if
end subroutine cmd_nlo_compile
```

⟨*Commands: cmd nlo: TBP*⟩+≡

```
procedure :: execute => cmd_nlo_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_nlo_execute (cmd, global)
  class(cmd_nlo_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: current_component
  type(string_t) :: component_type
  type(string_t) :: string
  integer :: n, i, j
  logical, dimension(0:5) :: selected_nlo_parts
  selected_nlo_parts = .false.
  if (allocated (cmd%nlo_component)) then
     n = size (cmd%nlo_component)
  else
     n = 0
  end if
  do i = 1, n
     select case (cmd%nlo_component (i))
     case (BORN, NLO_VIRTUAL, NLO_MISMATCH, NLO_DGLAP, NLO_REAL)
        selected_nlo_parts(cmd%nlo_component (i)) = .true.
     case (NLO_FULL)
        selected_nlo_parts = .true.
        selected_nlo_parts (NLO_SUBTRACTION) = .false.
     case default
        string = var_str ("")
        do j = BORN, NLO_DGLAP
           string = string // component_status (j) // ", "
        end do
```

```
            string = string // component_status (NLO_FULL)
            call msg_fatal ("Invalid NLO mode. Valid modes are: " // &
                 char (string))
         end select
      end do
      global%nlo_fixed_order = any (selected_nlo_parts(1:5))
      global%selected_nlo_parts = selected_nlo_parts
   end subroutine cmd_nlo_execute
```

Process compilation

⟨*Commands: types*⟩+≡

```
   type, extends (command_t) :: cmd_compile_t
      private
      type(string_t), dimension(:), allocatable :: libname
      logical :: make_executable = .false.
      type(string_t) :: exec_name
    contains
    ⟨Commands: cmd compile: TBP⟩
   end type cmd_compile_t
```

Output: list all libraries to be compiled.

⟨*Commands: cmd compile: TBP*⟩≡

```
   procedure :: write => cmd_compile_write
```

⟨*Commands: procedures*⟩+≡

```
   subroutine cmd_compile_write (cmd, unit, indent)
     class(cmd_compile_t), intent(in) :: cmd
     integer, intent(in), optional :: unit, indent
     integer :: u, i
     u = given_output_unit (unit);  if (u < 0)  return
     call write_indent (u, indent)
     write (u, "(1x,A)", advance="no")  "compile ("
     if (allocated (cmd%libname)) then
        do i = 1, size (cmd%libname)
           if (i > 1)  write (u, "(A,1x)", advance="no")  ","
           write (u, "('""',A,'""')", advance="no")  char (cmd%libname(i))
        end do
     end if
     write (u, "(A)")  ")"
   end subroutine cmd_compile_write
```

Compile the libraries specified in the argument. If the argument is empty, compile all libraries which can be found in the process library stack.

⟨*Commands: cmd compile: TBP*⟩+≡

```
   procedure :: compile => cmd_compile_compile
```

⟨*Commands: procedures*⟩+≡

```
   subroutine cmd_compile_compile (cmd, global)
     class(cmd_compile_t), intent(inout) :: cmd
     type(rt_data_t), intent(inout), target :: global
     type(parse_node_t), pointer :: pn_cmd, pn_clause, pn_arg, pn_lib
```

```
    type(parse_node_t), pointer :: pn_exec_name_spec, pn_exec_name
    integer :: n_lib, i
    pn_cmd => parse_node_get_sub_ptr (cmd%pn)
    pn_clause => parse_node_get_sub_ptr (pn_cmd)
    pn_exec_name_spec => parse_node_get_sub_ptr (pn_clause, 2)
    if (associated (pn_exec_name_spec)) then
       pn_exec_name => parse_node_get_sub_ptr (pn_exec_name_spec, 2)
    else
       pn_exec_name => null ()
    end if
    pn_arg => parse_node_get_next_ptr (pn_clause)
    cmd%pn_opt => parse_node_get_next_ptr (pn_cmd)
    call cmd%compile_options (global)
    if (associated (pn_arg)) then
       n_lib = parse_node_get_n_sub (pn_arg)
    else
       n_lib = 0
    end if
    if (n_lib > 0) then
       allocate (cmd%libname (n_lib))
       pn_lib => parse_node_get_sub_ptr (pn_arg)
       do i = 1, n_lib
          cmd%libname(i) = parse_node_get_string (pn_lib)
          pn_lib => parse_node_get_next_ptr (pn_lib)
       end do
    end if
    if (associated (pn_exec_name)) then
       cmd%make_executable = .true.
       cmd%exec_name = parse_node_get_string (pn_exec_name)
    end if
  end subroutine cmd_compile_compile
```

Command execution. Generate code, write driver, compile and link. Do this for all libraries in the list.

If no library names have been given and stored while compiling this command, we collect all libraries from the current stack and compile those.

We can compile static libraries (which actually just loads them). However, we can't incorporate in a generated executable.

⟨*Commands: cmd compile: TBP*⟩+≡

```
  procedure :: execute => cmd_compile_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_compile_execute (cmd, global)
    class(cmd_compile_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(string_t), dimension(:), allocatable :: libname, libname_static
    integer :: i
    if (allocated (cmd%libname)) then
       allocate (libname (size (cmd%libname)))
       libname = cmd%libname
    else
       call cmd%local%prclib_stack%get_names (libname)
    end if
    if (cmd%make_executable) then
```

```
      call get_prclib_static (libname_static)
      do i = 1, size (libname)
         if (any (libname_static == libname(i))) then
            call msg_fatal ("Compile: can't include static library '" &
                 // char (libname(i)) // "'")
         end if
      end do
      call compile_executable (cmd%exec_name, libname, cmd%local)
   else
      do i = 1, size (libname)
         call compile_library (libname(i), cmd%local)
      end do
   end if
end subroutine cmd_compile_execute
```

This is the interface to the external procedure which returns the names of all static libraries which are part of the executable. (The default is none.) The routine must allocate the array.

⟨*Commands: public*⟩≡

```
public :: get_prclib_static
```

⟨*Commands: interfaces*⟩+≡

```
interface
   subroutine get_prclib_static (libname)
     import
     type(string_t), dimension(:), intent(inout), allocatable :: libname
   end subroutine get_prclib_static
end interface
```

Execute a shell command

The argument is a string expression.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_exec_t
   private
   type(parse_node_t), pointer :: pn_command => null ()
 contains
 ⟨Commands: cmd exec: TBP⟩
end type cmd_exec_t
```

Simply tell the status.

⟨*Commands: cmd exec: TBP*⟩≡

```
procedure :: write => cmd_exec_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_exec_write (cmd, unit, indent)
  class(cmd_exec_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  if (associated (cmd%pn_command)) then
```

```
      write (u, "(1x,A)")  "exec: [command associated]"
    else
      write (u, "(1x,A)")  "exec: [undefined]"
    end if
  end subroutine cmd_exec_write
```

Compile the exec command.

⟨*Commands: cmd exec: TBP*⟩+≡

```
  procedure :: compile => cmd_exec_compile
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_exec_compile (cmd, global)
    class(cmd_exec_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_arg, pn_command
    pn_arg => parse_node_get_sub_ptr (cmd%pn, 2)
    pn_command => parse_node_get_sub_ptr (pn_arg)
    cmd%pn_command => pn_command
  end subroutine cmd_exec_compile
```

Execute the specified shell command.

⟨*Commands: cmd exec: TBP*⟩+≡

```
  procedure :: execute => cmd_exec_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_exec_execute (cmd, global)
    class(cmd_exec_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(string_t) :: command
    logical :: is_known
    integer :: status
    command = eval_string (cmd%pn_command, global%var_list, is_known=is_known)
    if (is_known) then
      if (command /= "") then
        call os_system_call (command, status, verbose=.true.)
        if (status /= 0) then
          write (msg_buffer, "(A,I0)")  "Return code = ", status
          call msg_message ()
          call msg_error ("System command returned with nonzero status code")
        end if
      end if
    end if
  end subroutine cmd_exec_execute
```

Variable declaration

A variable can have various types. Hold the definition as an eval tree.

There are intrinsic variables, user variables, and model variables. The latter are further divided in independent variables and dependent variables.

Regarding model variables: When dealing with them, we always look at two variable lists in parallel. The global (or local) variable list contains the user-visible values. It includes variables that correspond to variables in the

current model's list. These, in turn, are pointers to the model's parameter list, so the model is always in sync, internally. To keep the global variable list in sync with the model, the global variables carry the `is_copy` property and contain a separate pointer to the model variable. (The pointer is reassigned whenever the model changes.) Modifying the global variable changes two values simultaneously: the visible value and the model variable, via this extra pointer. After each modification, we update dependent parameters in the model variable list and re-synchronize the global variable list (again, using these pointers) with the model variable this. In the last step, modifications in the derived parameters become visible.

When we integrate a process, we capture the current variable list of the current model in a separate model instance, which is stored in the process object. Thus, the model parameters associated to this process at this time are preserved for the lifetime of the process object.

When we generate or rescan events, we can again capture a local model variable list in a model instance. This allows us to reweight event by event with different parameter sets simultaneously.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_var_t
   private
   type(string_t) :: name
   integer :: type = V_NONE
   type(parse_node_t), pointer :: pn_value => null ()
   logical :: is_intrinsic = .false.
   logical :: is_model_var = .false.
 contains
 ⟨Commands: cmd var: TBP⟩
end type cmd_var_t
```

Output. We know name, type, and properties, but not the value.

⟨*Commands: cmd var: TBP*⟩≡

```
procedure :: write => cmd_var_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_var_write (cmd, unit, indent)
  class(cmd_var_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,A,A)", advance="no")  "var: ", char (cmd%name), " ("
  select case (cmd%type)
  case (V_NONE)
     write (u, "(A)", advance="no")  "[unknown]"
  case (V_LOG)
     write (u, "(A)", advance="no")  "logical"
  case (V_INT)
     write (u, "(A)", advance="no")  "int"
  case (V_REAL)
     write (u, "(A)", advance="no")  "real"
  case (V_CMPLX)
     write (u, "(A)", advance="no")  "complex"
```

```
    case (V_STR)
       write (u, "(A)", advance="no")  "string"
    case (V_PDG)
       write (u, "(A)", advance="no")  "alias"
    end select
    if (cmd%is_intrinsic) then
       write (u, "(A)", advance="no")  ", intrinsic"
    end if
    if (cmd%is_model_var) then
       write (u, "(A)", advance="no")  ", model"
    end if
    write (u, "(A)")  ")"
  end subroutine cmd_var_write
```

Compile the lhs and determine the variable name and type. Check whether this variable can be created or modified as requested, and append the value to the variable list, if appropriate. The value is initially undefined. The rhs is assigned to a pointer, to be compiled and evaluated when the command is executed.

⟨*Commands: cmd var: TBP*⟩+≡

```
  procedure :: compile => cmd_var_compile
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_var_compile (cmd, global)
    class(cmd_var_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_var, pn_name
    type(parse_node_t), pointer :: pn_result, pn_proc
    type(string_t) :: var_name
    type(var_list_t), pointer :: model_vars
    integer :: type
    logical :: new
    pn_result => null ()
    new = .false.
    select case (char (parse_node_get_rule_key (cmd%pn)))
    case ("cmd_log_decl");    type = V_LOG
       pn_var => parse_node_get_sub_ptr (cmd%pn, 2)
       if (.not. associated (pn_var)) then   ! handle masked syntax error
          cmd%type = V_NONE; return
       end if
       pn_name => parse_node_get_sub_ptr (pn_var, 2)
       new = .true.
    case ("cmd_log");         type = V_LOG
       pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
    case ("cmd_int");         type = V_INT
       pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
       new = .true.
    case ("cmd_real");        type = V_REAL
       pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
       new = .true.
    case ("cmd_complex");       type = V_CMPLX
       pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
       new = .true.
    case ("cmd_num");         type = V_NONE
       pn_name => parse_node_get_sub_ptr (cmd%pn)
```

```
case ("cmd_string_decl"); type = V_STR
   pn_var => parse_node_get_sub_ptr (cmd%pn, 2)
   if (.not. associated (pn_var)) then   ! handle masked syntax error
      cmd%type = V_NONE; return
   end if
   pn_name => parse_node_get_sub_ptr (pn_var, 2)
   new = .true.
case ("cmd_string");      type = V_STR
   pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
case ("cmd_alias");       type = V_PDG
   pn_name => parse_node_get_sub_ptr (cmd%pn, 2)
   new = .true.
case ("cmd_result");      type = V_REAL
   pn_name => parse_node_get_sub_ptr (cmd%pn)
   pn_result => parse_node_get_sub_ptr (pn_name)
   pn_proc => parse_node_get_next_ptr (pn_result)
case default
   call parse_node_mismatch &
        ("logical|int|real|complex|?|$|alias|var_name", cmd%pn)  ! $
end select
if (.not. associated (pn_name)) then   ! handle masked syntax error
   cmd%type = V_NONE; return
end if
if (.not. associated (pn_result)) then
   var_name = parse_node_get_string (pn_name)
else
   var_name = parse_node_get_key (pn_result) &
        // "(" // parse_node_get_string (pn_proc) // ")"
end if
select case (type)
case (V_LOG);  var_name = "?" // var_name
case (V_STR);  var_name = "$" // var_name    ! $
end select
if (associated (global%model)) then
   model_vars => global%model%get_var_list_ptr ()
else
   model_vars => null ()
end if
call var_list_check_observable (global%var_list, var_name, type)
call var_list_check_result_var (global%var_list, var_name, type)
call global%var_list%check_user_var (var_name, type, new)
cmd%name = var_name
cmd%pn_value => parse_node_get_next_ptr (pn_name, 2)
if (global%var_list%contains (cmd%name, follow_link = .false.)) then
   ! local variable
   cmd%is_intrinsic = &
        global%var_list%is_intrinsic (cmd%name, follow_link = .false.)
   cmd%type = &
        global%var_list%get_type (cmd%name, follow_link = .false.)
else
   if (new)  cmd%type = type
   if (global%var_list%contains (cmd%name, follow_link = .true.)) then
      ! global variable
      cmd%is_intrinsic = &
```

```
            global%var_list%is_intrinsic (cmd%name, follow_link = .true.)
         if (cmd%type == V_NONE) then
            cmd%type = &
                 global%var_list%get_type (cmd%name, follow_link = .true.)
         end if
      else if (associated (model_vars)) then  ! check model variable
         cmd%is_model_var = &
              model_vars%contains (cmd%name)
         if (cmd%type == V_NONE) then
            cmd%type = &
                 model_vars%get_type (cmd%name)
         end if
      end if
      if (cmd%type == V_NONE) then
         call msg_fatal ("Variable '" // char (cmd%name) // "' " &
              // "set without declaration")
         cmd%type = V_NONE;  return
      end if
      if (cmd%is_model_var) then
         if (new) then
            call msg_fatal ("Model variable '" // char (cmd%name) // "' " &
                 // "redeclared")
         else if (model_vars%is_locked (cmd%name)) then
            call msg_fatal ("Model variable '" // char (cmd%name) // "' " &
                 // "is locked")
         end if
      else
         select case (cmd%type)
         case (V_LOG)
            call global%var_list%append_log (cmd%name, &
                 intrinsic=cmd%is_intrinsic, user=.true.)
         case (V_INT)
            call global%var_list%append_int (cmd%name, &
                 intrinsic=cmd%is_intrinsic, user=.true.)
         case (V_REAL)
            call global%var_list%append_real (cmd%name, &
                 intrinsic=cmd%is_intrinsic, user=.true.)
         case (V_CMPLX)
            call global%var_list%append_cmplx (cmd%name, &
                 intrinsic=cmd%is_intrinsic, user=.true.)
         case (V_PDG)
            call global%var_list%append_pdg_array (cmd%name, &
                 intrinsic=cmd%is_intrinsic, user=.true.)
         case (V_STR)
            call global%var_list%append_string (cmd%name, &
                 intrinsic=cmd%is_intrinsic, user=.true.)
         end select
      end if
   end if
end subroutine cmd_var_compile
```

Execute. Evaluate the definition and assign the variable value. If the variable is a model variable, take a snapshot of the model if necessary and set the variable

in the local model.

⟨*Commands: cmd var: TBP*⟩+≡

```
procedure :: execute => cmd_var_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_var_execute (cmd, global)
  class(cmd_var_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(var_list_t), pointer :: var_list
  real(default) :: rval
  logical :: is_known, pacified
  var_list => global%get_var_list_ptr ()
  if (cmd%is_model_var) then
     pacified = var_list%get_lval (var_str ("?pacify"))
     rval = eval_real (cmd%pn_value, var_list, is_known=is_known)
     call global%model_set_real &
          (cmd%name, rval, verbose=.true., pacified=pacified)
  else if (cmd%type /= V_NONE) then
     call cmd%set_value (var_list, verbose=.true.)
  end if
end subroutine cmd_var_execute
```

Copy the value to the variable list, where the variable should already exist.

⟨*Commands: cmd var: TBP*⟩+≡

```
procedure :: set_value => cmd_var_set_value
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_var_set_value (var, var_list, verbose, model_name)
  class(cmd_var_t), intent(inout) :: var
  type(var_list_t), intent(inout), target :: var_list
  logical, intent(in), optional :: verbose
  type(string_t), intent(in), optional :: model_name
  logical :: lval, pacified
  integer :: ival
  real(default) :: rval
  complex(default) :: cval
  type(pdg_array_t) :: aval
  type(string_t) :: sval
  logical :: is_known
  pacified = var_list%get_lval (var_str ("?pacify"))
  select case (var%type)
  case (V_LOG)
     lval = eval_log (var%pn_value, var_list, is_known=is_known)
     call var_list%set_log (var%name, &
          lval, is_known, verbose=verbose, model_name=model_name)
  case (V_INT)
     ival = eval_int (var%pn_value, var_list, is_known=is_known)
     call var_list%set_int (var%name, &
          ival, is_known, verbose=verbose, model_name=model_name)
  case (V_REAL)
     rval = eval_real (var%pn_value, var_list, is_known=is_known)
     call var_list%set_real (var%name, &
          rval, is_known, verbose=verbose, &
          model_name=model_name, pacified = pacified)
```

```
    case (V_CMPLX)
       cval = eval_cmplx (var%pn_value, var_list, is_known=is_known)
       call var_list%set_cmplx (var%name, &
            cval, is_known, verbose=verbose, &
            model_name=model_name, pacified = pacified)
    case (V_PDG)
       aval = eval_pdg_array (var%pn_value, var_list, is_known=is_known)
       call var_list%set_pdg_array (var%name, &
            aval, is_known, verbose=verbose, model_name=model_name)
    case (V_STR)
       sval = eval_string (var%pn_value, var_list, is_known=is_known)
       call var_list%set_string (var%name, &
            sval, is_known, verbose=verbose, model_name=model_name)
    end select
  end subroutine cmd_var_set_value
```

SLHA

Read a SLHA (SUSY Les Houches Accord) file to fill the appropriate model parameters. We do not access the current variable record, but directly work on the appropriate SUSY model, which is loaded if necessary.

We may be in read or write mode. In the latter case, we may write just input parameters, or the complete spectrum, or the spectrum with all decays.

⟨*Commands: types*⟩+≡

```
  type, extends (command_t) :: cmd_slha_t
     private
     type(string_t) :: file
     logical :: write_mode = .false.
   contains
   ⟨Commands: cmd slha: TBP⟩
  end type cmd_slha_t
```

Output.

⟨*Commands: cmd slha: TBP*⟩≡

```
  procedure :: write => cmd_slha_write
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_slha_write (cmd, unit, indent)
    class(cmd_slha_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    call write_indent (u, indent)
    write (u, "(1x,A,A)")  "slha: file name  = ", char (cmd%file)
    write (u, "(1x,A,L1)") "slha: write mode = ", cmd%write_mode
  end subroutine cmd_slha_write
```

Compile. Read the filename and store it.

⟨*Commands: cmd slha: TBP*⟩+≡

```
  procedure :: compile => cmd_slha_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_slha_compile (cmd, global)
  class(cmd_slha_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_key, pn_arg, pn_file
  pn_key => parse_node_get_sub_ptr (cmd%pn)
  pn_arg => parse_node_get_next_ptr (pn_key)
  pn_file => parse_node_get_sub_ptr (pn_arg)
  call cmd%compile_options (global)
  cmd%pn_opt => parse_node_get_next_ptr (pn_arg)
  select case (char (parse_node_get_key (pn_key)))
  case ("read_slha")
     cmd%write_mode = .false.
  case ("write_slha")
     cmd%write_mode = .true.
  case default
     call parse_node_mismatch ("read_slha|write_slha",  cmd%pn)
  end select
  cmd%file = parse_node_get_string (pn_file)
end subroutine cmd_slha_compile
```

Execute. Read or write the specified SLHA file. Behind the scenes, this will first read the WHIZARD model file, then read the SLHA file and assign the SLHA parameters as far as determined by `dispatch_slha`. Finally, the global variables are synchronized with the model. This is similar to executing `cmd_model`.

⟨*Commands: cmd slha: TBP*⟩+≡

```
procedure :: execute => cmd_slha_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_slha_execute (cmd, global)
  class(cmd_slha_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  logical :: input, spectrum, decays
  if (cmd%write_mode) then
     input = .true.
     spectrum = .false.
     decays = .false.
     if (.not. associated (cmd%local%model)) then
        call msg_fatal ("SLHA: local model not associated")
        return
     end if
     call slha_write_file &
          (cmd%file, cmd%local%model, &
           input = input, spectrum = spectrum, decays = decays)
  else
     if (.not. associated (global%model)) then
        call msg_fatal ("SLHA: global model not associated")
        return
     end if
     call dispatch_slha (cmd%local%var_list, &
          input = input, spectrum = spectrum, decays = decays)
     call global%ensure_model_copy ()
     call slha_read_file &
          (cmd%file, cmd%local%os_data, global%model, &
```

```
            input = input, spectrum = spectrum, decays = decays)
    end if
  end subroutine cmd_slha_execute
```

Show values

This command shows the current values of variables or other objects, in a suitably condensed form.

⟨*Commands: types*⟩+≡

```
  type, extends (command_t) :: cmd_show_t
     private
     type(string_t), dimension(:), allocatable :: name
   contains
   ⟨Commands: cmd show: TBP⟩
  end type cmd_show_t
```

Output: list the object names, not values.

⟨*Commands: cmd show: TBP*⟩≡

```
  procedure :: write => cmd_show_write
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_show_write (cmd, unit, indent)
    class(cmd_show_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u, i
    u = given_output_unit (unit);  if (u < 0)  return
    call write_indent (u, indent)
    write (u, "(1x,A)", advance="no")  "show: "
    if (allocated (cmd%name)) then
       do i = 1, size (cmd%name)
          write (u, "(1x,A)", advance="no")  char (cmd%name(i))
       end do
       write (u, *)
    else
       write (u, "(5x,A)")  "[undefined]"
    end if
  end subroutine cmd_show_write
```

Compile. Allocate an array which is filled with the names of the variables to show.

⟨*Commands: cmd show: TBP*⟩+≡

```
  procedure :: compile => cmd_show_compile
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_show_compile (cmd, global)
    class(cmd_show_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_arg, pn_var, pn_prefix, pn_name
    type(string_t) :: key
    integer :: i, n_args
    pn_arg => parse_node_get_sub_ptr (cmd%pn, 2)
    if (associated (pn_arg)) then
```

```
   select case (char (parse_node_get_rule_key (pn_arg)))
   case ("show_arg")
      cmd%pn_opt => parse_node_get_next_ptr (pn_arg)
   case default
      cmd%pn_opt => pn_arg
      pn_arg => null ()
   end select
end if
call cmd%compile_options (global)
if (associated (pn_arg)) then
   n_args = parse_node_get_n_sub (pn_arg)
   allocate (cmd%name (n_args))
   pn_var => parse_node_get_sub_ptr (pn_arg)
   i = 0
   do while (associated (pn_var))
      i = i + 1
      select case (char (parse_node_get_rule_key (pn_var)))
      case ("model", "library", "beams", "iterations", &
            "cuts", "weight", "int", "real", "complex", &
            "scale", "factorization_scale", "renormalization_scale", &
            "selection", "reweight", "analysis", "pdg", &
            "stable", "unstable", "polarized", "unpolarized", &
            "results", "expect", "intrinsic", "string", "logical")
         cmd%name(i) = parse_node_get_key (pn_var)
      case ("result_var")
         pn_prefix => parse_node_get_sub_ptr (pn_var)
         pn_name => parse_node_get_next_ptr (pn_prefix)
         if (associated (pn_name)) then
            cmd%name(i) = parse_node_get_key (pn_prefix) &
                 // "(" // parse_node_get_string (pn_name) // ")"
         else
            cmd%name(i) = parse_node_get_key (pn_prefix)
         end if
      case ("log_var", "string_var", "alias_var")
         pn_prefix => parse_node_get_sub_ptr (pn_var)
         pn_name => parse_node_get_next_ptr (pn_prefix)
         key = parse_node_get_key (pn_prefix)
         if (associated (pn_name)) then
            select case (char (parse_node_get_rule_key (pn_name)))
            case ("var_name")
               select case (char (key))
               case ("?", "$")  ! $ sign
                  cmd%name(i) = key // parse_node_get_string (pn_name)
               case ("alias")
                  cmd%name(i) = parse_node_get_string (pn_name)
               end select
            case default
               call parse_node_mismatch &
                     ("var_name",  pn_name)
            end select
         else
            cmd%name(i) = key
         end if
      case default
```

```
            cmd%name(i) = parse_node_get_string (pn_var)
         end select
         pn_var => parse_node_get_next_ptr (pn_var)
      end do
   else
      allocate (cmd%name (0))
   end if
end subroutine cmd_show_compile
```

Execute. Scan the list of objects to show.

⟨*Commands: parameters*⟩≡

```
integer, parameter, public :: SHOW_BUFFER_SIZE = 4096
```

⟨*Commands: cmd show: TBP*⟩+≡

```
procedure :: execute => cmd_show_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_show_execute (cmd, global)
  class(cmd_show_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(var_list_t), pointer :: var_list, model_vars
  type(model_t), pointer :: model
  type(string_t) :: name
  integer :: n, pdg
  type(flavor_t) :: flv
  type(process_library_t), pointer :: prc_lib
  type(process_t), pointer :: process
  logical :: pacified
  character(SHOW_BUFFER_SIZE) :: buffer
  integer :: i, j, u, u_log, u_out
  u = free_unit ()
  var_list => cmd%local%var_list
  if (associated (cmd%local%model)) then
     model_vars => cmd%local%model%get_var_list_ptr ()
  else
     model_vars => null ()
  end if
  pacified = var_list%get_lval (var_str ("?pacify"))
  open (u, status = "scratch", action = "readwrite")
  if (associated (cmd%local%model)) then
     name = cmd%local%model%get_name ()
  end if
  if (size (cmd%name) == 0) then
     if (associated (model_vars)) then
        call model_vars%write (model_name = name, &
             unit = u, pacified = pacified, follow_link = .false.)
     end if
     call var_list%write (unit = u, pacified = pacified)
  else
     do i = 1, size (cmd%name)
        select case (char (cmd%name(i)))
        case ("model")
           if (associated (cmd%local%model)) then
              call cmd%local%model%show (u)
           else
```

```
      write (u, "(A)")  "Model: [undefined]"
   end if
case ("library")
   if (associated (cmd%local%prclib)) then
      call cmd%local%prclib%show (u)
   else
      write (u, "(A)")  "Process library: [undefined]"
   end if
case ("beams")
   call cmd%local%show_beams (u)
case ("iterations")
   call cmd%local%it_list%write (u)
case ("results")
   call cmd%local%process_stack%show (u)
case ("stable")
   call cmd%local%model%show_stable (u)
case ("polarized")
   call cmd%local%model%show_polarized (u)
case ("unpolarized")
   call cmd%local%model%show_unpolarized (u)
case ("unstable")
   model => cmd%local%model
   call model%show_unstable (u)
   n = model%get_n_field ()
   do j = 1, n
      pdg = model%get_pdg (j)
      call flv%init (pdg, model)
      if (.not. flv%is_stable ()) &
           call show_unstable (cmd%local, pdg, u)
      if (flv%has_antiparticle ()) then
         associate (anti => flv%anti ())
           if (.not. anti%is_stable ()) &
                call show_unstable (cmd%local, -pdg, u)
         end associate
      end if
   end do
case ("cuts", "weight", "scale", &
     "factorization_scale", "renormalization_scale", &
     "selection", "reweight", "analysis")
   call cmd%local%pn%show (cmd%name(i), u)
case ("expect")
   call expect_summary (force = .true.)
case ("intrinsic")
   call var_list%write (intrinsic=.true., unit=u, &
        pacified = pacified)
case ("logical")
   if (associated (model_vars)) then
      call model_vars%write (only_type=V_LOG, &
           model_name = name, unit=u, pacified = pacified, &
           follow_link=.false.)
   end if
   call var_list%write (&
        only_type=V_LOG, unit=u, pacified = pacified)
case ("int")
```

```
   if (associated (model_vars)) then
      call model_vars%write (only_type=V_INT, &
           model_name = name, unit=u, pacified = pacified, &
           follow_link=.false.)
   end if
   call var_list%write (only_type=V_INT, &
        unit=u, pacified = pacified)
case ("real")
   if (associated (model_vars)) then
      call model_vars%write (only_type=V_REAL, &
           model_name = name, unit=u, pacified = pacified, &
           follow_link=.false.)
   end if
   call var_list%write (only_type=V_REAL, &
        unit=u, pacified = pacified)
case ("complex")
   if (associated (model_vars)) then
      call model_vars%write (only_type=V_CMPLX, &
           model_name = name, unit=u, pacified = pacified, &
           follow_link=.false.)
   end if
   call var_list%write (only_type=V_CMPLX, &
        unit=u, pacified = pacified)
case ("pdg")
   if (associated (model_vars)) then
      call model_vars%write (only_type=V_PDG, &
           model_name = name, unit=u, pacified = pacified, &
           follow_link=.false.)
   end if
   call var_list%write (only_type=V_PDG, &
        unit=u, pacified = pacified)
case ("string")
   if (associated (model_vars)) then
      call model_vars%write (only_type=V_STR, &
           model_name = name, unit=u, pacified = pacified, &
           follow_link=.false.)
   end if
   call var_list%write (only_type=V_STR, &
        unit=u, pacified = pacified)
case default
   if (analysis_exists (cmd%name(i))) then
      call analysis_write (cmd%name(i), u)
   else if (cmd%local%process_stack%exists (cmd%name(i))) then
      process => cmd%local%process_stack%get_process_ptr (cmd%name(i))
      call process%show (u)
   else if (associated (cmd%local%prclib_stack%get_library_ptr &
        (cmd%name(i)))) then
      prc_lib => cmd%local%prclib_stack%get_library_ptr (cmd%name(i))
      call prc_lib%show (u)
   else if (associated (model_vars)) then
      if (model_vars%contains (cmd%name(i), follow_link=.false.)) then
         call model_vars%write_var (cmd%name(i), &
              unit = u, model_name = name, pacified = pacified)
      else if (var_list%contains (cmd%name(i))) then
```

```
                 call var_list%write_var (cmd%name(i), &
                      unit = u, pacified = pacified)
              else
                 call msg_error ("show: object '" // char (cmd%name(i)) &
                      // "' not found")
              end if
           else if (var_list%contains (cmd%name(i))) then
              call var_list%write_var (cmd%name(i), &
                   unit = u, pacified = pacified)
           else
              call msg_error ("show: object '" // char (cmd%name(i)) &
                   // "' not found")
           end if
        end select
     end do
  end if
  rewind (u)
  u_log = logfile_unit ()
  u_out = given_output_unit ()
  do
     read (u, "(A)", end = 1)  buffer
     if (u_log > 0)  write (u_log, "(A)")  trim (buffer)
     if (u_out > 0)  write (u_out, "(A)")  trim (buffer)
  end do
1 close (u)
  if (u_log > 0)  flush (u_log)
  if (u_out > 0)  flush (u_out)
end subroutine cmd_show_execute
```

Clear values

This command clears the current values of variables or other objects, where this makes sense. It parallels the `show` command. The objects are cleared, but not deleted.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_clear_t
   private
   type(string_t), dimension(:), allocatable :: name
 contains
 ⟨Commands: cmd clear: TBP⟩
end type cmd_clear_t
```

Output: list the names of the objects to be cleared.

⟨*Commands: cmd clear: TBP*⟩≡

```
procedure :: write => cmd_clear_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_clear_write (cmd, unit, indent)
  class(cmd_clear_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
```

```
    call write_indent (u, indent)
    write (u, "(1x,A)", advance="no")  "clear: "
    if (allocated (cmd%name)) then
       do i = 1, size (cmd%name)
          write (u, "(1x,A)", advance="no")  char (cmd%name(i))
       end do
       write (u, *)
    else
       write (u, "(5x,A)")  "[undefined]"
    end if
  end subroutine cmd_clear_write
```

Compile. Allocate an array which is filled with the names of the objects to be cleared.

Note: there is currently no need to account for options, but we prepare for that possibility.

⟨*Commands: cmd clear: TBP*⟩+≡

```
  procedure :: compile => cmd_clear_compile
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_clear_compile (cmd, global)
    class(cmd_clear_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_arg, pn_var, pn_prefix, pn_name
    type(string_t) :: key
    integer :: i, n_args
    pn_arg => parse_node_get_sub_ptr (cmd%pn, 2)
    if (associated (pn_arg)) then
       select case (char (parse_node_get_rule_key (pn_arg)))
       case ("clear_arg")
          cmd%pn_opt => parse_node_get_next_ptr (pn_arg)
       case default
          cmd%pn_opt => pn_arg
          pn_arg => null ()
       end select
    end if
    call cmd%compile_options (global)
    if (associated (pn_arg)) then
       n_args = parse_node_get_n_sub (pn_arg)
       allocate (cmd%name (n_args))
       pn_var => parse_node_get_sub_ptr (pn_arg)
       i = 0
       do while (associated (pn_var))
          i = i + 1
          select case (char (parse_node_get_rule_key (pn_var)))
          case ("beams", "iterations", &
                "cuts", "weight", &
                "scale", "factorization_scale", "renormalization_scale", &
                "selection", "reweight", "analysis", &
                "unstable", "polarized", &
                "expect")
             cmd%name(i) = parse_node_get_key (pn_var)
          case ("log_var", "string_var")
             pn_prefix => parse_node_get_sub_ptr (pn_var)
```

```
            pn_name => parse_node_get_next_ptr (pn_prefix)
            key = parse_node_get_key (pn_prefix)
            if (associated (pn_name)) then
               select case (char (parse_node_get_rule_key (pn_name)))
               case ("var_name")
                  select case (char (key))
                  case ("?", "$")  ! $ sign
                     cmd%name(i) = key // parse_node_get_string (pn_name)
                  end select
               case default
                  call parse_node_mismatch &
                       ("var_name",  pn_name)
               end select
            else
               cmd%name(i) = key
            end if
         case default
            cmd%name(i) = parse_node_get_string (pn_var)
         end select
         pn_var => parse_node_get_next_ptr (pn_var)
      end do
   else
      allocate (cmd%name (0))
   end if
end subroutine cmd_clear_compile
```

Execute. Scan the list of objects to clear.

Objects that can be shown but not cleared: model, library, results

⟨*Commands: cmd clear: TBP*⟩+≡

```
procedure :: execute => cmd_clear_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_clear_execute (cmd, global)
  class(cmd_clear_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  integer :: i
  logical :: success
  type(var_list_t), pointer :: model_vars
  if (size (cmd%name) == 0) then
     call msg_warning ("clear: no object specified")
  else
     do i = 1, size (cmd%name)
        success = .true.
        select case (char (cmd%name(i)))
        case ("beams")
           call cmd%local%clear_beams ()
        case ("iterations")
           call cmd%local%it_list%clear ()
        case ("polarized")
           call cmd%local%model%clear_polarized ()
        case ("unstable")
           call cmd%local%model%clear_unstable ()
        case ("cuts", "weight", "scale", &
             "factorization_scale", "renormalization_scale", &
```

```
            "selection", "reweight", "analysis")
         call cmd%local%pn%clear (cmd%name(i))
      case ("expect")
         call expect_clear ()
      case default
         if (analysis_exists (cmd%name(i))) then
            call analysis_clear (cmd%name(i))
         else if (cmd%local%var_list%contains (cmd%name(i))) then
            if (.not. cmd%local%var_list%is_locked (cmd%name(i))) then
               call cmd%local%var_list%unset (cmd%name(i))
            else
               call msg_error ("clear: variable '" // char (cmd%name(i)) &
                    // "' is locked and can't be cleared")
               success = .false.
            end if
         else if (associated (cmd%local%model)) then
            model_vars => cmd%local%model%get_var_list_ptr ()
            if (model_vars%contains (cmd%name(i), follow_link=.false.)) then
               call msg_error ("clear: variable '" // char (cmd%name(i)) &
                    // "' is a model variable and can't be cleared")
            else
               call msg_error ("clear: object '" // char (cmd%name(i)) &
                    // "' not found")
            end if
            success = .false.
         else
            call msg_error ("clear: object '" // char (cmd%name(i)) &
                 // "' not found")
            success = .false.
         end if
      end select
      if (success)  call msg_message ("cleared: " // char (cmd%name(i)))
   end do
  end if
end subroutine cmd_clear_execute
```

Compare values of variables to expectation

The implementation is similar to the `show` command. There are just two arguments: two values that should be compared. For providing local values for the numerical tolerance, the command has a local argument list.

If the expectation fails, an error condition is recorded.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_expect_t
   private
   type(parse_node_t), pointer :: pn_lexpr => null ()
 contains
 ⟨Commands: cmd expect: TBP⟩
end type cmd_expect_t
```

Simply tell the status.

⟨*Commands: cmd expect: TBP*⟩≡

```
procedure :: write => cmd_expect_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_expect_write (cmd, unit, indent)
  class(cmd_expect_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  if (associated (cmd%pn_lexpr)) then
     write (u, "(1x,A)")  "expect: [expression associated]"
  else
     write (u, "(1x,A)")  "expect: [undefined]"
  end if
end subroutine cmd_expect_write
```

Compile. This merely assigns the parse node, the actual compilation is done at execution. This is necessary because the origin of variables (local/global) may change during execution.

⟨*Commands: cmd expect: TBP*⟩+≡

```
procedure :: compile => cmd_expect_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_expect_compile (cmd, global)
  class(cmd_expect_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_arg
  pn_arg => parse_node_get_sub_ptr (cmd%pn, 2)
  cmd%pn_opt => parse_node_get_next_ptr (pn_arg)
  cmd%pn_lexpr => parse_node_get_sub_ptr (pn_arg)
  call cmd%compile_options (global)
end subroutine cmd_expect_compile
```

Execute. Evaluate both arguments, print them and their difference (if numerical), and whether they agree. Record the result.

⟨*Commands: cmd expect: TBP*⟩+≡

```
procedure :: execute => cmd_expect_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_expect_execute (cmd, global)
  class(cmd_expect_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(var_list_t), pointer :: var_list
  logical :: success, is_known
  var_list => cmd%local%get_var_list_ptr ()
  success = eval_log (cmd%pn_lexpr, var_list, is_known=is_known)
  if (is_known) then
     if (success) then
        call msg_message ("expect: success")
     else
        call msg_error ("expect: failure")
     end if
  else
     call msg_error ("expect: undefined result")
```

```
      success = .false.
    end if
    call expect_record (success)
  end subroutine cmd_expect_execute
```

Beams

The beam command includes both beam and structure-function definition.

⟨*Commands: types*⟩+≡

```
  type, extends (command_t) :: cmd_beams_t
     private
     integer :: n_in = 0
     type(parse_node_p), dimension(:), allocatable :: pn_pdg
     integer :: n_sf_record = 0
     integer, dimension(:), allocatable :: n_entry
     type(parse_node_p), dimension(:,:), allocatable :: pn_sf_entry
   contains
   ⟨Commands: cmd beams: TBP⟩
  end type cmd_beams_t
```

Output. The particle expressions are not resolved.

⟨*Commands: cmd beams: TBP*⟩≡

```
  procedure :: write => cmd_beams_write
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_beams_write (cmd, unit, indent)
    class(cmd_beams_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    call write_indent (u, indent)
    select case (cmd%n_in)
    case (1)
       write (u, "(1x,A)")  "beams: 1 [decay]"
    case (2)
       write (u, "(1x,A)")  "beams: 2 [scattering]"
    case default
       write (u, "(1x,A)")  "beams: [undefined]"
    end select
    if (allocated (cmd%n_entry)) then
       if (cmd%n_sf_record > 0) then
          write (u, "(1x,A,99(1x,I0))")  "structure function entries:", &
               cmd%n_entry
       end if
    end if
  end subroutine cmd_beams_write
```

Compile. Find and assign the parse nodes.

Note: local environments are not yet supported.

⟨*Commands: cmd beams: TBP*⟩+≡

```
  procedure :: compile => cmd_beams_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_beams_compile (cmd, global)
  class(cmd_beams_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_beam_def, pn_beam_spec
  type(parse_node_t), pointer :: pn_beam_list
  type(parse_node_t), pointer :: pn_codes
  type(parse_node_t), pointer :: pn_strfun_seq, pn_strfun_pair
  type(parse_node_t), pointer :: pn_strfun_def
  integer :: i
  pn_beam_def => parse_node_get_sub_ptr (cmd%pn, 3)
  pn_beam_spec => parse_node_get_sub_ptr (pn_beam_def)
  pn_strfun_seq => parse_node_get_next_ptr (pn_beam_spec)
  pn_beam_list => parse_node_get_sub_ptr (pn_beam_spec)
  call cmd%compile_options (global)
  cmd%n_in = parse_node_get_n_sub (pn_beam_list)
  allocate (cmd%pn_pdg (cmd%n_in))
  pn_codes => parse_node_get_sub_ptr (pn_beam_list)
  do i = 1, cmd%n_in
     cmd%pn_pdg(i)%ptr => pn_codes
     pn_codes => parse_node_get_next_ptr (pn_codes)
  end do
  if (associated (pn_strfun_seq)) then
     cmd%n_sf_record = parse_node_get_n_sub (pn_beam_def) - 1
     allocate (cmd%n_entry (cmd%n_sf_record), source = 1)
     allocate (cmd%pn_sf_entry (2, cmd%n_sf_record))
     do i = 1, cmd%n_sf_record
        pn_strfun_pair => parse_node_get_sub_ptr (pn_strfun_seq, 2)
        pn_strfun_def => parse_node_get_sub_ptr (pn_strfun_pair)
        cmd%pn_sf_entry(1,i)%ptr => pn_strfun_def
        pn_strfun_def => parse_node_get_next_ptr (pn_strfun_def)
        cmd%pn_sf_entry(2,i)%ptr => pn_strfun_def
        if (associated (pn_strfun_def))  cmd%n_entry(i) = 2
        pn_strfun_seq => parse_node_get_next_ptr (pn_strfun_seq)
     end do
  else
     allocate (cmd%n_entry (0))
     allocate (cmd%pn_sf_entry (0, 0))
  end if
end subroutine cmd_beams_compile
```

Command execution: Determine beam particles and structure-function names, if any. The results are stored in the `beam_structure` component of the `global` data block.

⟨*Commands: cmd beams: TBP*⟩+≡

```
procedure :: execute => cmd_beams_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_beams_execute (cmd, global)
  class(cmd_beams_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(var_list_t), pointer :: var_list
  type(pdg_array_t) :: pdg_array
  integer, dimension(:), allocatable :: pdg
```

```
    type(flavor_t), dimension(:), allocatable :: flv
    type(parse_node_t), pointer :: pn_key
    type(string_t) :: sf_name
    integer :: i, j
    call lhapdf_global_reset ()
    var_list => cmd%local%get_var_list_ptr ()
    allocate (flv (cmd%n_in))
    do i = 1, cmd%n_in
       pdg_array = eval_pdg_array (cmd%pn_pdg(i)%ptr, var_list)
       pdg = pdg_array
       select case (size (pdg))
       case (1)
          call flv(i)%init ( pdg(1), cmd%local%model)
       case default
          call msg_fatal ("Beams: beam particles must be unique")
       end select
    end do
    select case (cmd%n_in)
    case (1)
       if (cmd%n_sf_record > 0) then
          call msg_fatal ("Beam setup: no structure functions allowed &
               &for decay")
       end if
       call global%beam_structure%init_sf (flv%get_name ())
    case (2)
       call global%beam_structure%init_sf (flv%get_name (), cmd%n_entry)
       do i = 1, cmd%n_sf_record
          do j = 1, cmd%n_entry(i)
             pn_key => parse_node_get_sub_ptr (cmd%pn_sf_entry(j,i)%ptr)
             sf_name = parse_node_get_key (pn_key)
             call global%beam_structure%set_sf (i, j, sf_name)
          end do
       end do
    end select
  end subroutine cmd_beams_execute
```

Density matrices for beam polarization

For holding beam polarization, we define a notation and a data structure for sparse matrices. The entries (and the index expressions) are numerical expressions, so we use evaluation trees.

Each entry in the sparse matrix is an n-tuple of expressions. The first tuple elements represent index values, the last one is an arbitrary (complex) number. Absent expressions are replaced by default-value rules.

Note: Here, and in some other commands, we would like to store an evaluation tree, not just a parse node pointer. However, the current expression handler wants all variables defined, so the evaluation tree can only be built by `evaluate`, i.e., compiled just-in-time and evaluated immediately.

⟨*Commands: types*⟩+≡

```
  type :: sentry_expr_t
     type(parse_node_p), dimension(:), allocatable :: expr
   contains
```

```
  ⟨Commands: sentry expr: TBP⟩
end type sentry_expr_t
```

Compile parse nodes into evaluation trees.

⟨*Commands: sentry expr: TBP*⟩≡

```
procedure :: compile => sentry_expr_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine sentry_expr_compile (sentry, pn)
  class(sentry_expr_t), intent(out) :: sentry
  type(parse_node_t), intent(in), target :: pn
  type(parse_node_t), pointer :: pn_expr, pn_extra
  integer :: n_expr, i
  n_expr = parse_node_get_n_sub (pn)
  allocate (sentry%expr (n_expr))
  if (n_expr > 0) then
     i = 0
     pn_expr => parse_node_get_sub_ptr (pn)
     pn_extra => parse_node_get_next_ptr (pn_expr)
     do i = 1, n_expr
        sentry%expr(i)%ptr => pn_expr
        if (associated (pn_extra)) then
           pn_expr => parse_node_get_sub_ptr (pn_extra, 2)
           pn_extra => parse_node_get_next_ptr (pn_extra)
        end if
     end do
  end if
end subroutine sentry_expr_compile
```

Evaluate the expressions and return an index array of predefined length together with a complex value. If the value (as the last expression) is undefined, set it to unity. If index values are undefined, repeat the previous index value.

⟨*Commands: sentry expr: TBP*⟩+≡

```
procedure :: evaluate => sentry_expr_evaluate
```

⟨*Commands: procedures*⟩+≡

```
subroutine sentry_expr_evaluate (sentry, index, value, global)
  class(sentry_expr_t), intent(inout) :: sentry
  integer, dimension(:), intent(out) :: index
  complex(default), intent(out) :: value
  type(rt_data_t), intent(in), target :: global
  type(var_list_t), pointer :: var_list
  integer :: i, n_expr, n_index
  type(eval_tree_t) :: eval_tree
  var_list => global%get_var_list_ptr ()
  n_expr = size (sentry%expr)
  n_index = size (index)
  if (n_expr <= n_index + 1) then
     do i = 1, min (n_expr, n_index)
        associate (expr => sentry%expr(i))
          call eval_tree%init_expr (expr%ptr, var_list)
          call eval_tree%evaluate ()
          if (eval_tree%is_known ()) then
             index(i) = eval_tree%get_int ()
```

```
            else
               call msg_fatal ("Evaluating density matrix: undefined index")
            end if
         end associate
      end do
      do i = n_expr + 1, n_index
         index(i) = index(n_expr)
      end do
      if (n_expr == n_index + 1) then
         associate (expr => sentry%expr(n_expr))
            call eval_tree%init_expr (expr%ptr, var_list)
            call eval_tree%evaluate ()
            if (eval_tree%is_known ()) then
               value = eval_tree%get_cmplx ()
            else
               call msg_fatal ("Evaluating density matrix: undefined index")
            end if
            call eval_tree%final ()
         end associate
      else
         value = 1
      end if
   else
      call msg_fatal ("Evaluating density matrix: index expression too long")
   end if
end subroutine sentry_expr_evaluate
```

The sparse matrix itself consists of an arbitrary number of entries.

⟨*Commands: types*⟩+≡

```
type :: smatrix_expr_t
   type(sentry_expr_t), dimension(:), allocatable :: entry
 contains
 ⟨Commands: smatrix expr: TBP⟩
end type smatrix_expr_t
```

Compile: assign sub-nodes to sentry-expressions and compile those.

⟨*Commands: smatrix expr: TBP*⟩≡

```
procedure :: compile => smatrix_expr_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine smatrix_expr_compile (smatrix_expr, pn)
  class(smatrix_expr_t), intent(out) :: smatrix_expr
  type(parse_node_t), intent(in), target :: pn
  type(parse_node_t), pointer :: pn_arg, pn_entry
  integer :: n_entry, i
  pn_arg => parse_node_get_sub_ptr (pn, 2)
  if (associated (pn_arg)) then
     n_entry = parse_node_get_n_sub (pn_arg)
     allocate (smatrix_expr%entry (n_entry))
     pn_entry => parse_node_get_sub_ptr (pn_arg)
     do i = 1, n_entry
        call smatrix_expr%entry(i)%compile (pn_entry)
        pn_entry => parse_node_get_next_ptr (pn_entry)
```

```
      end do
    else
      allocate (smatrix_expr%entry (0))
    end if
  end subroutine smatrix_expr_compile
```

Evaluate the entries and build a new `smatrix` object, which contains just the numerical results.

⟨*Commands: smatrix expr: TBP*⟩+≡

```
  procedure :: evaluate => smatrix_expr_evaluate
```

⟨*Commands: procedures*⟩+≡

```
  subroutine smatrix_expr_evaluate (smatrix_expr, smatrix, global)
    class(smatrix_expr_t), intent(inout) :: smatrix_expr
    type(smatrix_t), intent(out) :: smatrix
    type(rt_data_t), intent(in), target :: global
    integer, dimension(2) :: idx
    complex(default) :: value
    integer :: i, n_entry
    n_entry = size (smatrix_expr%entry)
    call smatrix%init (2, n_entry)
    do i = 1, n_entry
      call smatrix_expr%entry(i)%evaluate (idx, value, global)
      call smatrix%set_entry (i, idx, value)
    end do
  end subroutine smatrix_expr_evaluate
```

Beam polarization density

The beam polarization command defines spin density matrix for one or two beams (scattering or decay).

⟨*Commands: types*⟩+≡

```
  type, extends (command_t) :: cmd_beams_pol_density_t
     private
     integer :: n_in = 0
     type(smatrix_expr_t), dimension(:), allocatable :: smatrix
   contains
   ⟨Commands: cmd beams pol density: TBP⟩
  end type cmd_beams_pol_density_t
```

Output.

⟨*Commands: cmd beams pol density: TBP*⟩≡

```
  procedure :: write => cmd_beams_pol_density_write
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_beams_pol_density_write (cmd, unit, indent)
    class(cmd_beams_pol_density_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    call write_indent (u, indent)
    select case (cmd%n_in)
```

```
    case (1)
       write (u, "(1x,A)")  "beams polarization setup: 1 [decay]"
    case (2)
       write (u, "(1x,A)")  "beams polarization setup: 2 [scattering]"
    case default
       write (u, "(1x,A)")  "beams polarization setup: [undefined]"
    end select
  end subroutine cmd_beams_pol_density_write
```

Compile. Find and assign the parse nodes.

Note: local environments are not yet supported.

⟨*Commands: cmd beams pol density: TBP*⟩+≡

```
  procedure :: compile => cmd_beams_pol_density_compile
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_beams_pol_density_compile (cmd, global)
    class(cmd_beams_pol_density_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_pol_spec, pn_smatrix
    integer :: i
    pn_pol_spec => parse_node_get_sub_ptr (cmd%pn, 3)
    call cmd%compile_options (global)
    cmd%n_in = parse_node_get_n_sub (pn_pol_spec)
    allocate (cmd%smatrix (cmd%n_in))
    pn_smatrix => parse_node_get_sub_ptr (pn_pol_spec)
    do i = 1, cmd%n_in
       call cmd%smatrix(i)%compile (pn_smatrix)
       pn_smatrix => parse_node_get_next_ptr (pn_smatrix)
    end do
  end subroutine cmd_beams_pol_density_compile
```

Command execution: Fill polarization density matrices. No check yet, the matrices are checked and normalized when the actual beam object is created, just before integration. For intermediate storage, we use the `beam_structure` object in the `global` data set.

⟨*Commands: cmd beams pol density: TBP*⟩+≡

```
  procedure :: execute => cmd_beams_pol_density_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_beams_pol_density_execute (cmd, global)
    class(cmd_beams_pol_density_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(smatrix_t) :: smatrix
    integer :: i
    call global%beam_structure%init_pol (cmd%n_in)
    do i = 1, cmd%n_in
       call cmd%smatrix(i)%evaluate (smatrix, global)
       call global%beam_structure%set_smatrix (i, smatrix)
    end do
  end subroutine cmd_beams_pol_density_execute
```

Beam polarization fraction

In addition to the polarization density matrix, we can independently specify the polarization fraction for one or both beams.

⟨*Commands: types*⟩+≡
```
type, extends (command_t) :: cmd_beams_pol_fraction_t
   private
   integer :: n_in = 0
   type(parse_node_p), dimension(:), allocatable :: expr
 contains
 ⟨Commands: cmd beams pol fraction: TBP⟩
end type cmd_beams_pol_fraction_t
```

Output.

⟨*Commands: cmd beams pol fraction: TBP*⟩≡
```
procedure :: write => cmd_beams_pol_fraction_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_beams_pol_fraction_write (cmd, unit, indent)
  class(cmd_beams_pol_fraction_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  select case (cmd%n_in)
  case (1)
     write (u, "(1x,A)")  "beams polarization fraction: 1 [decay]"
  case (2)
     write (u, "(1x,A)")  "beams polarization fraction: 2 [scattering]"
  case default
     write (u, "(1x,A)")  "beams polarization fraction: [undefined]"
  end select
end subroutine cmd_beams_pol_fraction_write
```

Compile. Find and assign the parse nodes.

Note: local environments are not yet supported.

⟨*Commands: cmd beams pol fraction: TBP*⟩+≡
```
procedure :: compile => cmd_beams_pol_fraction_compile
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_beams_pol_fraction_compile (cmd, global)
  class(cmd_beams_pol_fraction_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_frac_spec, pn_expr
  integer :: i
  pn_frac_spec => parse_node_get_sub_ptr (cmd%pn, 3)
  call cmd%compile_options (global)
  cmd%n_in = parse_node_get_n_sub (pn_frac_spec)
  allocate (cmd%expr (cmd%n_in))
  pn_expr => parse_node_get_sub_ptr (pn_frac_spec)
  do i = 1, cmd%n_in
     cmd%expr(i)%ptr => pn_expr
     pn_expr => parse_node_get_next_ptr (pn_expr)
  end do
```

```
end subroutine cmd_beams_pol_fraction_compile
```

Command execution: Retrieve the numerical values of the beam polarization fractions. The results are stored in the `beam_structure` component of the `global` data block.

⟨*Commands: cmd beams pol fraction: TBP*⟩+≡

```
procedure :: execute => cmd_beams_pol_fraction_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_beams_pol_fraction_execute (cmd, global)
  class(cmd_beams_pol_fraction_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(var_list_t), pointer :: var_list
  real(default), dimension(:), allocatable :: pol_f
  type(eval_tree_t) :: expr
  integer :: i
  var_list => global%get_var_list_ptr ()
  allocate (pol_f (cmd%n_in))
  do i = 1, cmd%n_in
     call expr%init_expr (cmd%expr(i)%ptr, var_list)
     call expr%evaluate ()
     if (expr%is_known ()) then
        pol_f(i) = expr%get_real ()
     else
        call msg_fatal ("beams polarization fraction: undefined value")
     end if
     call expr%final ()
  end do
  call global%beam_structure%set_pol_f (pol_f)
end subroutine cmd_beams_pol_fraction_execute
```

Beam momentum

This is completely analogous to the previous command, hence we can use inheritance.

⟨*Commands: types*⟩+≡

```
type, extends (cmd_beams_pol_fraction_t) :: cmd_beams_momentum_t
 contains
 ⟨Commands: cmd beams momentum: TBP⟩
end type cmd_beams_momentum_t
```

Output.

⟨*Commands: cmd beams momentum: TBP*⟩≡

```
procedure :: write => cmd_beams_momentum_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_beams_momentum_write (cmd, unit, indent)
  class(cmd_beams_momentum_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
```

```
    select case (cmd%n_in)
    case (1)
       write (u, "(1x,A)")  "beams momentum: 1 [decay]"
    case (2)
       write (u, "(1x,A)")  "beams momentum: 2 [scattering]"
    case default
       write (u, "(1x,A)")  "beams momentum: [undefined]"
    end select
  end subroutine cmd_beams_momentum_write
```

Compile: inherited.

Command execution: Not inherited, but just the error string and the final command are changed.

⟨*Commands: cmd beams momentum: TBP*⟩+≡

```
  procedure :: execute => cmd_beams_momentum_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_beams_momentum_execute (cmd, global)
    class(cmd_beams_momentum_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(var_list_t), pointer :: var_list
    real(default), dimension(:), allocatable :: p
    type(eval_tree_t) :: expr
    integer :: i
    var_list => global%get_var_list_ptr ()
    allocate (p (cmd%n_in))
    do i = 1, cmd%n_in
       call expr%init_expr (cmd%expr(i)%ptr, var_list)
       call expr%evaluate ()
       if (expr%is_known ()) then
          p(i) = expr%get_real ()
       else
          call msg_fatal ("beams momentum: undefined value")
       end if
       call expr%final ()
    end do
    call global%beam_structure%set_momentum (p)
  end subroutine cmd_beams_momentum_execute
```

Beam angles

Again, this is analogous. There are two angles, polar angle θ and azimuthal angle ϕ, which can be set independently for both beams.

⟨*Commands: types*⟩+≡

```
  type, extends (cmd_beams_pol_fraction_t) :: cmd_beams_theta_t
   contains
   ⟨Commands: cmd beams theta: TBP⟩
  end type cmd_beams_theta_t

  type, extends (cmd_beams_pol_fraction_t) :: cmd_beams_phi_t
   contains
   ⟨Commands: cmd beams phi: TBP⟩
```

```
end type cmd_beams_phi_t
```

Output.

⟨*Commands: cmd beams theta: TBP*⟩≡

```
procedure :: write => cmd_beams_theta_write
```

⟨*Commands: cmd beams phi: TBP*⟩≡

```
procedure :: write => cmd_beams_phi_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_beams_theta_write (cmd, unit, indent)
  class(cmd_beams_theta_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  select case (cmd%n_in)
  case (1)
     write (u, "(1x,A)")  "beams theta: 1 [decay]"
  case (2)
     write (u, "(1x,A)")  "beams theta: 2 [scattering]"
  case default
     write (u, "(1x,A)")  "beams theta: [undefined]"
  end select
end subroutine cmd_beams_theta_write

subroutine cmd_beams_phi_write (cmd, unit, indent)
  class(cmd_beams_phi_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  select case (cmd%n_in)
  case (1)
     write (u, "(1x,A)")  "beams phi: 1 [decay]"
  case (2)
     write (u, "(1x,A)")  "beams phi: 2 [scattering]"
  case default
     write (u, "(1x,A)")  "beams phi: [undefined]"
  end select
end subroutine cmd_beams_phi_write
```

Compile: inherited.

Command execution: Not inherited, but just the error string and the final command are changed.

⟨*Commands: cmd beams theta: TBP*⟩+≡

```
procedure :: execute => cmd_beams_theta_execute
```

⟨*Commands: cmd beams phi: TBP*⟩+≡

```
procedure :: execute => cmd_beams_phi_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_beams_theta_execute (cmd, global)
  class(cmd_beams_theta_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
```

```
  type(var_list_t), pointer :: var_list
  real(default), dimension(:), allocatable :: theta
  type(eval_tree_t) :: expr
  integer :: i
  var_list => global%get_var_list_ptr ()
  allocate (theta (cmd%n_in))
  do i = 1, cmd%n_in
     call expr%init_expr (cmd%expr(i)%ptr, var_list)
     call expr%evaluate ()
     if (expr%is_known ()) then
        theta(i) = expr%get_real ()
     else
        call msg_fatal ("beams theta: undefined value")
     end if
     call expr%final ()
  end do
  call global%beam_structure%set_theta (theta)
end subroutine cmd_beams_theta_execute

subroutine cmd_beams_phi_execute (cmd, global)
  class(cmd_beams_phi_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(var_list_t), pointer :: var_list
  real(default), dimension(:), allocatable :: phi
  type(eval_tree_t) :: expr
  integer :: i
  var_list => global%get_var_list_ptr ()
  allocate (phi (cmd%n_in))
  do i = 1, cmd%n_in
     call expr%init_expr (cmd%expr(i)%ptr, var_list)
     call expr%evaluate ()
     if (expr%is_known ()) then
        phi(i) = expr%get_real ()
     else
        call msg_fatal ("beams phi: undefined value")
     end if
     call expr%final ()
  end do
  call global%beam_structure%set_phi (phi)
end subroutine cmd_beams_phi_execute
```

Cuts

Define a cut expression. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the process environment where the cut expression is used.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_cuts_t
   private
   type(parse_node_t), pointer :: pn_lexpr => null ()
 contains
 ⟨Commands: cmd cuts: TBP⟩
```

```
end type cmd_cuts_t
```

Output. Do not print the parse tree, since this may get cluttered. Just a message that cuts have been defined.

⟨*Commands: cmd cuts: TBP*⟩≡

```
procedure :: write => cmd_cuts_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_cuts_write (cmd, unit, indent)
  class(cmd_cuts_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "cuts: [defined]"
end subroutine cmd_cuts_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: cmd cuts: TBP*⟩+≡

```
procedure :: compile => cmd_cuts_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_cuts_compile (cmd, global)
  class(cmd_cuts_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  cmd%pn_lexpr => parse_node_get_sub_ptr (cmd%pn, 3)
end subroutine cmd_cuts_compile
```

Instead of evaluating the cut expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: cmd cuts: TBP*⟩+≡

```
procedure :: execute => cmd_cuts_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_cuts_execute (cmd, global)
  class(cmd_cuts_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  global%pn%cuts_lexpr => cmd%pn_lexpr
end subroutine cmd_cuts_execute
```

General, Factorization and Renormalization Scales

Define a scale expression for either the renormalization or the factorization scale. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the process environment where the expression is used.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_scale_t
   private
   type(parse_node_t), pointer :: pn_expr => null ()
 contains
```

```
  ⟨Commands: cmd scale: TBP⟩
end type cmd_scale_t
```

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_fac_scale_t
   private
   type(parse_node_t), pointer :: pn_expr => null ()
 contains
  ⟨Commands: cmd fac scale: TBP⟩
end type cmd_fac_scale_t
```

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_ren_scale_t
   private
   type(parse_node_t), pointer :: pn_expr => null ()
 contains
  ⟨Commands: cmd ren scale: TBP⟩
end type cmd_ren_scale_t
```

Output. Do not print the parse tree, since this may get cluttered. Just a message that scale, renormalization and factorization have been defined, respectively.

⟨*Commands: cmd scale: TBP*⟩≡

```
procedure :: write => cmd_scale_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_scale_write (cmd, unit, indent)
  class(cmd_scale_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "scale: [defined]"
end subroutine cmd_scale_write
```

⟨*Commands: cmd fac scale: TBP*⟩≡

```
procedure :: write => cmd_fac_scale_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_fac_scale_write (cmd, unit, indent)
  class(cmd_fac_scale_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "factorization scale: [defined]"
end subroutine cmd_fac_scale_write
```

⟨*Commands: cmd ren scale: TBP*⟩≡

```
procedure :: write => cmd_ren_scale_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_ren_scale_write (cmd, unit, indent)
  class(cmd_ren_scale_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "renormalization scale: [defined]"
end subroutine cmd_ren_scale_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: cmd scale: TBP*⟩+≡
```
procedure :: compile => cmd_scale_compile
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_scale_compile (cmd, global)
  class(cmd_scale_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  cmd%pn_expr => parse_node_get_sub_ptr (cmd%pn, 3)
end subroutine cmd_scale_compile
```

⟨*Commands: cmd fac scale: TBP*⟩+≡
```
procedure :: compile => cmd_fac_scale_compile
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_fac_scale_compile (cmd, global)
  class(cmd_fac_scale_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  cmd%pn_expr => parse_node_get_sub_ptr (cmd%pn, 3)
end subroutine cmd_fac_scale_compile
```

⟨*Commands: cmd ren scale: TBP*⟩+≡
```
procedure :: compile => cmd_ren_scale_compile
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_ren_scale_compile (cmd, global)
  class(cmd_ren_scale_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  cmd%pn_expr => parse_node_get_sub_ptr (cmd%pn, 3)
end subroutine cmd_ren_scale_compile
```

Instead of evaluating the scale expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: cmd scale: TBP*⟩+≡
```
procedure :: execute => cmd_scale_execute
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_scale_execute (cmd, global)
  class(cmd_scale_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  global%pn%scale_expr => cmd%pn_expr
end subroutine cmd_scale_execute
```

⟨*Commands: cmd fac scale: TBP*⟩+≡
```
procedure :: execute => cmd_fac_scale_execute
```
⟨*Commands: procedures*⟩+≡
```
subroutine cmd_fac_scale_execute (cmd, global)
  class(cmd_fac_scale_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  global%pn%fac_scale_expr => cmd%pn_expr
end subroutine cmd_fac_scale_execute
```

⟨*Commands: cmd ren scale: TBP*⟩+≡
```
procedure :: execute => cmd_ren_scale_execute
```
⟨*Commands: procedures*⟩+≡
```
subroutine cmd_ren_scale_execute (cmd, global)
  class(cmd_ren_scale_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  global%pn%ren_scale_expr => cmd%pn_expr
end subroutine cmd_ren_scale_execute
```

Weight

Define a weight expression. The weight is applied to a process to be integrated, event by event. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the process environment where the expression is used.

⟨*Commands: types*⟩+≡
```
type, extends (command_t) :: cmd_weight_t
   private
   type(parse_node_t), pointer :: pn_expr => null ()
 contains
 ⟨Commands: cmd weight: TBP⟩
end type cmd_weight_t
```

Output. Do not print the parse tree, since this may get cluttered. Just a message that scale, renormalization and factorization have been defined, respectively.

⟨*Commands: cmd weight: TBP*⟩≡
```
procedure :: write => cmd_weight_write
```
⟨*Commands: procedures*⟩+≡
```
subroutine cmd_weight_write (cmd, unit, indent)
  class(cmd_weight_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "weight expression: [defined]"
end subroutine cmd_weight_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: cmd weight: TBP*⟩+≡
```
procedure :: compile => cmd_weight_compile
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_weight_compile (cmd, global)
  class(cmd_weight_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  cmd%pn_expr => parse_node_get_sub_ptr (cmd%pn, 3)
end subroutine cmd_weight_compile
```

Instead of evaluating the expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: cmd weight: TBP*⟩+≡
```
procedure :: execute => cmd_weight_execute
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_weight_execute (cmd, global)
  class(cmd_weight_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  global%pn%weight_expr => cmd%pn_expr
end subroutine cmd_weight_execute
```

Selection

Define a selection expression. This is to be applied upon simulation or event-file rescanning, event by event. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the environment where the expression is used.

⟨*Commands: types*⟩+≡
```
type, extends (command_t) :: cmd_selection_t
   private
   type(parse_node_t), pointer :: pn_expr => null ()
 contains
 ⟨Commands: cmd selection: TBP⟩
end type cmd_selection_t
```

Output. Do not print the parse tree, since this may get cluttered. Just a message that scale, renormalization and factorization have been defined, respectively.

⟨*Commands: cmd selection: TBP*⟩≡
```
procedure :: write => cmd_selection_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_selection_write (cmd, unit, indent)
  class(cmd_selection_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "selection expression: [defined]"
end subroutine cmd_selection_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: cmd selection: TBP*⟩+≡
```
procedure :: compile => cmd_selection_compile
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_selection_compile (cmd, global)
  class(cmd_selection_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  cmd%pn_expr => parse_node_get_sub_ptr (cmd%pn, 3)
end subroutine cmd_selection_compile
```

Instead of evaluating the expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: cmd selection: TBP*⟩+≡
```
procedure :: execute => cmd_selection_execute
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_selection_execute (cmd, global)
  class(cmd_selection_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  global%pn%selection_lexpr => cmd%pn_expr
end subroutine cmd_selection_execute
```

Reweight

Define a reweight expression. This is to be applied upon simulation or event-file rescanning, event by event. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the environment where the expression is used.

⟨*Commands: types*⟩+≡
```
type, extends (command_t) :: cmd_reweight_t
   private
   type(parse_node_t), pointer :: pn_expr => null ()
 contains
 ⟨Commands: cmd reweight: TBP⟩
end type cmd_reweight_t
```

Output. Do not print the parse tree, since this may get cluttered. Just a message that scale, renormalization and factorization have been defined, respectively.

⟨*Commands: cmd reweight: TBP*⟩≡
```
procedure :: write => cmd_reweight_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_reweight_write (cmd, unit, indent)
  class(cmd_reweight_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "reweight expression: [defined]"
end subroutine cmd_reweight_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: cmd reweight: TBP*⟩+≡
```
procedure :: compile => cmd_reweight_compile
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_reweight_compile (cmd, global)
  class(cmd_reweight_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  cmd%pn_expr => parse_node_get_sub_ptr (cmd%pn, 3)
end subroutine cmd_reweight_compile
```

Instead of evaluating the expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: cmd reweight: TBP*⟩+≡
```
procedure :: execute => cmd_reweight_execute
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_reweight_execute (cmd, global)
  class(cmd_reweight_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  global%pn%reweight_expr => cmd%pn_expr
end subroutine cmd_reweight_execute
```

Alternative Simulation Setups

Together with simulation, we can re-evaluate event weights in the context of alternative setups. The `cmd_alt_setup_t` object is designed to hold these setups, which are brace-enclosed command lists. Compilation is deferred to the simulation environment where the setup expression is used.

⟨*Commands: types*⟩+≡
```
type, extends (command_t) :: cmd_alt_setup_t
   private
   type(parse_node_p), dimension(:), allocatable :: setup
 contains
 ⟨Commands: cmd alt setup: TBP⟩
end type cmd_alt_setup_t
```

Output. Print just a message that the alternative setup list has been defined.

⟨*Commands: cmd alt setup: TBP*⟩≡
```
procedure :: write => cmd_alt_setup_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_alt_setup_write (cmd, unit, indent)
  class(cmd_alt_setup_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,I0,A)")  "alt_setup: ", size (cmd%setup), " entries"
end subroutine cmd_alt_setup_write
```

Compile. Store the parse sub-trees in an array.

⟨*Commands: cmd alt setup: TBP*⟩+≡
```
procedure :: compile => cmd_alt_setup_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_alt_setup_compile (cmd, global)
  class(cmd_alt_setup_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_list, pn_setup
  integer :: i
  pn_list => parse_node_get_sub_ptr (cmd%pn, 3)
  if (associated (pn_list)) then
     allocate (cmd%setup (parse_node_get_n_sub (pn_list)))
     i = 1
     pn_setup => parse_node_get_sub_ptr (pn_list)
     do while (associated (pn_setup))
        cmd%setup(i)%ptr => pn_setup
        i = i + 1
        pn_setup => parse_node_get_next_ptr (pn_setup)
     end do
  else
     allocate (cmd%setup (0))
  end if
end subroutine cmd_alt_setup_compile
```

Execute. Transfer the array of command lists to the global environment.

⟨*Commands: cmd alt setup: TBP*⟩+≡

```
procedure :: execute => cmd_alt_setup_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_alt_setup_execute (cmd, global)
  class(cmd_alt_setup_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  if (allocated (global%pn%alt_setup))  deallocate (global%pn%alt_setup)
  allocate (global%pn%alt_setup (size (cmd%setup)))
  global%pn%alt_setup = cmd%setup
end subroutine cmd_alt_setup_execute
```

Integration

Integrate several processes, consecutively with identical parameters.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_integrate_t
   private
   integer :: n_proc = 0
   type(string_t), dimension(:), allocatable :: process_id
 contains
 ⟨Commands: cmd integrate: TBP⟩
end type cmd_integrate_t
```

Output: we know the process IDs.

⟨*Commands: cmd integrate: TBP*⟩≡

```
procedure :: write => cmd_integrate_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_integrate_write (cmd, unit, indent)
  class(cmd_integrate_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)", advance="no")  "integrate ("
  do i = 1, cmd%n_proc
     if (i > 1)  write (u, "(A,1x)", advance="no")  ","
     write (u, "(A)", advance="no")  char (cmd%process_id(i))
  end do
  write (u, "(A)")  ")"
end subroutine cmd_integrate_write
```

Compile.

⟨*Commands: cmd integrate: TBP*⟩+≡

```
procedure :: compile => cmd_integrate_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_integrate_compile (cmd, global)
  class(cmd_integrate_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_proclist, pn_proc
  integer :: i
  pn_proclist => parse_node_get_sub_ptr (cmd%pn, 2)
  cmd%pn_opt => parse_node_get_next_ptr (pn_proclist)
  call cmd%compile_options (global)
  cmd%n_proc = parse_node_get_n_sub (pn_proclist)
  allocate (cmd%process_id (cmd%n_proc))
  pn_proc => parse_node_get_sub_ptr (pn_proclist)
  do i = 1, cmd%n_proc
     cmd%process_id(i) = parse_node_get_string (pn_proc)
     call global%process_stack%init_result_vars (cmd%process_id(i))
     pn_proc => parse_node_get_next_ptr (pn_proc)
  end do
end subroutine cmd_integrate_compile
```

Command execution. Integrate the process(es) with the predefined number of passes, iterations and calls. For structure functions, cuts, weight and scale, use local definitions if present; by default, the local definitions are initialized with the global ones.

The `integrate` procedure should take its input from the currently active local environment, but produce a process record in the stack of the global environment.

Since the process acquires a snapshot of the variable list, so if the global list (or the local one) is deleted, this does no harm. This implies that later changes of the variable list do not affect the stored process.

⟨*Commands: cmd integrate: TBP*⟩+≡

```
procedure :: execute => cmd_integrate_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_integrate_execute (cmd, global)
```

```
    class(cmd_integrate_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    integer :: i
    call msg_debug (D_CORE, "cmd_integrate_execute")
    do i = 1, cmd%n_proc
       call msg_debug (D_CORE, "cmd%process_id(i) ", cmd%process_id(i))
       call integrate_process (cmd%process_id(i), cmd%local, global)
       call global%process_stack%fill_result_vars (cmd%process_id(i))
       if (signal_is_pending ())  return
    end do
  end subroutine cmd_integrate_execute
```

Observables

Declare an observable. After the declaration, it can be used to record data, and at the end one can retrieve average and error.

⟨*Commands: types*⟩+≡

```
  type, extends (command_t) :: cmd_observable_t
     private
     type(string_t) :: id
   contains
   ⟨Commands: cmd observable: TBP⟩
  end type cmd_observable_t
```

Output. We know the ID.

⟨*Commands: cmd observable: TBP*⟩≡

```
  procedure :: write => cmd_observable_write
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_observable_write (cmd, unit, indent)
    class(cmd_observable_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u
    u = given_output_unit (unit);  if (u < 0)  return
    call write_indent (u, indent)
    write (u, "(1x,A,A)")  "observable: ", char (cmd%id)
  end subroutine cmd_observable_write
```

Compile. Just record the observable ID.

⟨*Commands: cmd observable: TBP*⟩+≡

```
  procedure :: compile => cmd_observable_compile
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_observable_compile (cmd, global)
    class(cmd_observable_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_tag
    pn_tag => parse_node_get_sub_ptr (cmd%pn, 2)
    if (associated (pn_tag)) then
       cmd%pn_opt => parse_node_get_next_ptr (pn_tag)
    end if
    call cmd%compile_options (global)
```

```
    select case (char (parse_node_get_rule_key (pn_tag)))
    case ("analysis_id")
       cmd%id = parse_node_get_string (pn_tag)
    case default
       call msg_bug ("observable: name expression not implemented (yet)")
    end select
  end subroutine cmd_observable_compile
```

Command execution. This declares the observable and allocates it in the analysis store.

⟨*Commands: cmd observable: TBP*⟩+≡

```
  procedure :: execute => cmd_observable_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_observable_execute (cmd, global)
    class(cmd_observable_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(var_list_t), pointer :: var_list
    type(graph_options_t) :: graph_options
    type(string_t) :: label, unit
    var_list => cmd%local%get_var_list_ptr ()
    label = var_list%get_sval (var_str ("$obs_label"))
    unit = var_list%get_sval (var_str ("$obs_unit"))
    call graph_options_init (graph_options)
    call set_graph_options (graph_options, var_list)
    call analysis_init_observable (cmd%id, label, unit, graph_options)
  end subroutine cmd_observable_execute
```

Histograms

Declare a histogram. At minimum, we have to set lower and upper bound and bin width.

⟨*Commands: types*⟩+≡

```
  type, extends (command_t) :: cmd_histogram_t
     private
     type(string_t) :: id
     type(parse_node_t), pointer :: pn_lower_bound => null ()
     type(parse_node_t), pointer :: pn_upper_bound => null ()
     type(parse_node_t), pointer :: pn_bin_width => null ()
   contains
   ⟨Commands: cmd histogram: TBP⟩
  end type cmd_histogram_t
```

Output. Just print the ID.

⟨*Commands: cmd histogram: TBP*⟩≡

```
  procedure :: write => cmd_histogram_write
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_histogram_write (cmd, unit, indent)
    class(cmd_histogram_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u
```

```
    u = given_output_unit (unit);  if (u < 0)  return
    call write_indent (u, indent)
    write (u, "(1x,A,A)")  "histogram: ", char (cmd%id)
  end subroutine cmd_histogram_write
```

Compile. Record the histogram ID and initialize lower, upper bound and bin width.

⟨*Commands: cmd histogram: TBP*⟩+≡

```
  procedure :: compile => cmd_histogram_compile
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_histogram_compile (cmd, global)
    class(cmd_histogram_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_tag, pn_args, pn_arg1, pn_arg2, pn_arg3
    character(*), parameter :: e_illegal_use = &
       "illegal usage of 'histogram': insufficient number of arguments"
    pn_tag => parse_node_get_sub_ptr (cmd%pn, 2)
    pn_args => parse_node_get_next_ptr (pn_tag)
    if (associated (pn_args)) then
       pn_arg1 => parse_node_get_sub_ptr (pn_args)
       if (.not. associated (pn_arg1)) call msg_fatal (e_illegal_use)
       pn_arg2 => parse_node_get_next_ptr (pn_arg1)
       if (.not. associated (pn_arg2)) call msg_fatal (e_illegal_use)
       pn_arg3 => parse_node_get_next_ptr (pn_arg2)
       cmd%pn_opt => parse_node_get_next_ptr (pn_args)
    end if
    call cmd%compile_options (global)
    select case (char (parse_node_get_rule_key (pn_tag)))
    case ("analysis_id")
       cmd%id = parse_node_get_string (pn_tag)
    case default
       call msg_bug ("histogram: name expression not implemented (yet)")
    end select
    cmd%pn_lower_bound => pn_arg1
    cmd%pn_upper_bound => pn_arg2
    cmd%pn_bin_width => pn_arg3
  end subroutine cmd_histogram_compile
```

Command execution. This declares the histogram and allocates it in the analysis store.

⟨*Commands: cmd histogram: TBP*⟩+≡

```
  procedure :: execute => cmd_histogram_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_histogram_execute (cmd, global)
    class(cmd_histogram_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(var_list_t), pointer :: var_list
    real(default) :: lower_bound, upper_bound, bin_width
    integer :: bin_number
    logical :: bin_width_is_used, normalize_bins
    type(string_t) :: obs_label, obs_unit
    type(graph_options_t) :: graph_options
```

```
    type(drawing_options_t) :: drawing_options

    var_list => cmd%local%get_var_list_ptr ()
    lower_bound = eval_real (cmd%pn_lower_bound, var_list)
    upper_bound = eval_real (cmd%pn_upper_bound, var_list)
    if (associated (cmd%pn_bin_width)) then
       bin_width = eval_real (cmd%pn_bin_width, var_list)
       bin_width_is_used = .true.
    else if (var_list%is_known (var_str ("n_bins"))) then
       bin_number = &
            var_list%get_ival (var_str ("n_bins"))
       bin_width_is_used = .false.
    else
       call msg_error ("Cmd '" // char (cmd%id) // &
            "': neither bin width nor number is defined")
    end if
    normalize_bins = &
         var_list%get_lval (var_str ("?normalize_bins"))
    obs_label = &
         var_list%get_sval (var_str ("$obs_label"))
    obs_unit = &
         var_list%get_sval (var_str ("$obs_unit"))

    call graph_options_init (graph_options)
    call set_graph_options (graph_options, var_list)
    call drawing_options_init_histogram (drawing_options)
    call set_drawing_options (drawing_options, var_list)

    if (bin_width_is_used) then
       call analysis_init_histogram &
            (cmd%id, lower_bound, upper_bound, bin_width, &
             normalize_bins, &
             obs_label, obs_unit, &
             graph_options, drawing_options)
    else
       call analysis_init_histogram &
            (cmd%id, lower_bound, upper_bound, bin_number, &
             normalize_bins, &
             obs_label, obs_unit, &
             graph_options, drawing_options)
    end if
  end subroutine cmd_histogram_execute
```

Set the graph options from a variable list.

⟨*Commands: procedures*⟩+≡

```
  subroutine set_graph_options (gro, var_list)
    type(graph_options_t), intent(inout) :: gro
    type(var_list_t), intent(in) :: var_list
    call graph_options_set (gro, title = &
         var_list%get_sval (var_str ("$title")))
    call graph_options_set (gro, description = &
         var_list%get_sval (var_str ("$description")))
    call graph_options_set (gro, x_label = &
```

```
        var_list%get_sval (var_str ("$x_label")))
    call graph_options_set (gro, y_label = &
        var_list%get_sval (var_str ("$y_label")))
    call graph_options_set (gro, width_mm = &
        var_list%get_ival (var_str ("graph_width_mm")))
    call graph_options_set (gro, height_mm = &
        var_list%get_ival (var_str ("graph_height_mm")))
    call graph_options_set (gro, x_log = &
        var_list%get_lval (var_str ("?x_log")))
    call graph_options_set (gro, y_log = &
        var_list%get_lval (var_str ("?y_log")))
    if (var_list%is_known (var_str ("x_min"))) &
        call graph_options_set (gro, x_min = &
        var_list%get_rval (var_str ("x_min")))
    if (var_list%is_known (var_str ("x_max"))) &
        call graph_options_set (gro, x_max = &
        var_list%get_rval (var_str ("x_max")))
    if (var_list%is_known (var_str ("y_min"))) &
        call graph_options_set (gro, y_min = &
        var_list%get_rval (var_str ("y_min")))
    if (var_list%is_known (var_str ("y_max"))) &
        call graph_options_set (gro, y_max = &
        var_list%get_rval (var_str ("y_max")))
    call graph_options_set (gro, gmlcode_bg = &
        var_list%get_sval (var_str ("$gmlcode_bg")))
    call graph_options_set (gro, gmlcode_fg = &
        var_list%get_sval (var_str ("$gmlcode_fg")))
  end subroutine set_graph_options
```

Set the drawing options from a variable list.

⟨*Commands: procedures*⟩+≡

```
  subroutine set_drawing_options (dro, var_list)
    type(drawing_options_t), intent(inout) :: dro
    type(var_list_t), intent(in) :: var_list
    if (var_list%is_known (var_str ("?draw_histogram"))) then
       if (var_list%get_lval (var_str ("?draw_histogram"))) then
          call drawing_options_set (dro, with_hbars = .true.)
       else
          call drawing_options_set (dro, with_hbars = .false., &
               with_base = .false., fill = .false., piecewise = .false.)
       end if
    end if
    if (var_list%is_known (var_str ("?draw_base"))) then
       if (var_list%get_lval (var_str ("?draw_base"))) then
          call drawing_options_set (dro, with_base = .true.)
       else
          call drawing_options_set (dro, with_base = .false., fill = .false.)
       end if
    end if
    if (var_list%is_known (var_str ("?draw_piecewise"))) then
       if (var_list%get_lval (var_str ("?draw_piecewise"))) then
          call drawing_options_set (dro, piecewise = .true.)
       else
          call drawing_options_set (dro, piecewise = .false.)
```

```
   end if
end if
if (var_list%is_known (var_str ("?fill_curve"))) then
   if (var_list%get_lval (var_str ("?fill_curve"))) then
      call drawing_options_set (dro, fill = .true., with_base = .true.)
   else
      call drawing_options_set (dro, fill = .false.)
   end if
end if
if (var_list%is_known (var_str ("?draw_curve"))) then
   if (var_list%get_lval (var_str ("?draw_curve"))) then
      call drawing_options_set (dro, draw = .true.)
   else
      call drawing_options_set (dro, draw = .false.)
   end if
end if
if (var_list%is_known (var_str ("?draw_errors"))) then
   if (var_list%get_lval (var_str ("?draw_errors"))) then
      call drawing_options_set (dro, err = .true.)
   else
      call drawing_options_set (dro, err = .false.)
   end if
end if
if (var_list%is_known (var_str ("?draw_symbols"))) then
   if (var_list%get_lval (var_str ("?draw_symbols"))) then
      call drawing_options_set (dro, symbols = .true.)
   else
      call drawing_options_set (dro, symbols = .false.)
   end if
end if
if (var_list%is_known (var_str ("$fill_options"))) then
   call drawing_options_set (dro, fill_options = &
        var_list%get_sval (var_str ("$fill_options")))
end if
if (var_list%is_known (var_str ("$draw_options"))) then
   call drawing_options_set (dro, draw_options = &
        var_list%get_sval (var_str ("$draw_options")))
end if
if (var_list%is_known (var_str ("$err_options"))) then
   call drawing_options_set (dro, err_options = &
        var_list%get_sval (var_str ("$err_options")))
end if
if (var_list%is_known (var_str ("$symbol"))) then
   call drawing_options_set (dro, symbol = &
        var_list%get_sval (var_str ("$symbol")))
end if
if (var_list%is_known (var_str ("$gmlcode_bg"))) then
   call drawing_options_set (dro, gmlcode_bg = &
        var_list%get_sval (var_str ("$gmlcode_bg")))
end if
if (var_list%is_known (var_str ("$gmlcode_fg"))) then
   call drawing_options_set (dro, gmlcode_fg = &
        var_list%get_sval (var_str ("$gmlcode_fg")))
end if
```

```
end subroutine set_drawing_options
```

Plots

Declare a plot. No mandatory arguments, just options.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_plot_t
   private
   type(string_t) :: id
 contains
 ⟨Commands: cmd plot: TBP⟩
end type cmd_plot_t
```

Output. Just print the ID.

⟨*Commands: cmd plot: TBP*⟩≡

```
procedure :: write => cmd_plot_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_plot_write (cmd, unit, indent)
  class(cmd_plot_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,A)")  "plot: ", char (cmd%id)
end subroutine cmd_plot_write
```

Compile. Record the plot ID and initialize lower, upper bound and bin width.

⟨*Commands: cmd plot: TBP*⟩+≡

```
procedure :: compile => cmd_plot_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_plot_compile (cmd, global)
  class(cmd_plot_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_tag
  pn_tag => parse_node_get_sub_ptr (cmd%pn, 2)
  cmd%pn_opt => parse_node_get_next_ptr (pn_tag)
  call cmd%init (pn_tag, global)
end subroutine cmd_plot_compile
```

This init routine is separated because it is reused below for graph initialization.

⟨*Commands: cmd plot: TBP*⟩+≡

```
procedure :: init => cmd_plot_init
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_plot_init (plot, pn_tag, global)
  class(cmd_plot_t), intent(inout) :: plot
  type(parse_node_t), intent(in), pointer :: pn_tag
  type(rt_data_t), intent(inout), target :: global
  call plot%compile_options (global)
  select case (char (parse_node_get_rule_key (pn_tag)))
```

```
    case ("analysis_id")
       plot%id = parse_node_get_string (pn_tag)
    case default
       call msg_bug ("plot: name expression not implemented (yet)")
    end select
  end subroutine cmd_plot_init
```

Command execution. This declares the plot and allocates it in the analysis store.

⟨*Commands: cmd plot: TBP*⟩+≡

```
  procedure :: execute => cmd_plot_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_plot_execute (cmd, global)
    class(cmd_plot_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(var_list_t), pointer :: var_list
    type(graph_options_t) :: graph_options
    type(drawing_options_t) :: drawing_options

    var_list => cmd%local%get_var_list_ptr ()
    call graph_options_init (graph_options)
    call set_graph_options (graph_options, var_list)
    call drawing_options_init_plot (drawing_options)
    call set_drawing_options (drawing_options, var_list)

    call analysis_init_plot (cmd%id, graph_options, drawing_options)
  end subroutine cmd_plot_execute
```

Graphs

Declare a graph. The graph is defined in terms of its contents. Both the graph and its contents may carry options.

The graph object contains its own ID as well as the IDs of its elements. For the elements, we reuse the `cmd_plot_t` defined above.

⟨*Commands: types*⟩+≡

```
  type, extends (command_t) :: cmd_graph_t
     private
     type(string_t) :: id
     integer :: n_elements = 0
     type(cmd_plot_t), dimension(:), allocatable :: el
     type(string_t), dimension(:), allocatable :: element_id
   contains
   ⟨Commands: cmd graph: TBP⟩
  end type cmd_graph_t
```

Output. Just print the ID.

⟨*Commands: cmd graph: TBP*⟩≡

```
  procedure :: write => cmd_graph_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_graph_write (cmd, unit, indent)
  class(cmd_graph_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,A,A,I0,A)")  "graph: ", char (cmd%id), &
       " (", cmd%n_elements, " entries)"
end subroutine cmd_graph_write
```

Compile. Record the graph ID and initialize lower, upper bound and bin width. For compiling the graph element syntax, we use part of the `cmd_plot_t` compiler.

Note: currently, we do not respect options, therefore just IDs on the RHS.

⟨*Commands: cmd graph: TBP*⟩+≡

```
procedure :: compile => cmd_graph_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_graph_compile (cmd, global)
  class(cmd_graph_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_term, pn_tag, pn_def, pn_app
  integer :: i

  pn_term => parse_node_get_sub_ptr (cmd%pn, 2)
  pn_tag => parse_node_get_sub_ptr (pn_term)
  cmd%pn_opt => parse_node_get_next_ptr (pn_tag)
  call cmd%compile_options (global)
  select case (char (parse_node_get_rule_key (pn_tag)))
  case ("analysis_id")
     cmd%id = parse_node_get_string (pn_tag)
  case default
     call msg_bug ("graph: name expression not implemented (yet)")
  end select
  pn_def => parse_node_get_next_ptr (pn_term, 2)
  cmd%n_elements = parse_node_get_n_sub (pn_def)
  allocate (cmd%element_id (cmd%n_elements))
  allocate (cmd%el (cmd%n_elements))
  pn_term => parse_node_get_sub_ptr (pn_def)
  pn_tag => parse_node_get_sub_ptr (pn_term)
  cmd%el(1)%pn_opt => parse_node_get_next_ptr (pn_tag)
  call cmd%el(1)%init (pn_tag, global)
  cmd%element_id(1) = parse_node_get_string (pn_tag)
  pn_app => parse_node_get_next_ptr (pn_term)
  do i = 2, cmd%n_elements
     pn_term => parse_node_get_sub_ptr (pn_app, 2)
     pn_tag => parse_node_get_sub_ptr (pn_term)
     cmd%el(i)%pn_opt => parse_node_get_next_ptr (pn_tag)
     call cmd%el(i)%init (pn_tag, global)
     cmd%element_id(i) = parse_node_get_string (pn_tag)
     pn_app => parse_node_get_next_ptr (pn_app)
  end do

end subroutine cmd_graph_compile
```

Command execution. This declares the graph, allocates it in the analysis store, and copies the graph elements.

For the graph, we set graph and default drawing options. For the elements, we reset individual drawing options.

This accesses internals of the contained elements of type `cmd_plot_t`, see above. We might disentangle such an interdependency when this code is rewritten using proper type extension.

⟨*Commands: cmd graph: TBP*⟩+≡

```
procedure :: execute => cmd_graph_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_graph_execute (cmd, global)
  class(cmd_graph_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(var_list_t), pointer :: var_list
  type(graph_options_t) :: graph_options
  type(drawing_options_t) :: drawing_options
  integer :: i, type

  var_list => cmd%local%get_var_list_ptr ()
  call graph_options_init (graph_options)
  call set_graph_options (graph_options, var_list)
  call analysis_init_graph (cmd%id, cmd%n_elements, graph_options)

  do i = 1, cmd%n_elements
     if (associated (cmd%el(i)%options)) then
        call cmd%el(i)%options%execute (cmd%el(i)%local)
     end if
     type = analysis_store_get_object_type (cmd%element_id(i))
     select case (type)
     case (AN_HISTOGRAM)
        call drawing_options_init_histogram (drawing_options)
     case (AN_PLOT)
        call drawing_options_init_plot (drawing_options)
     end select
     call set_drawing_options (drawing_options, var_list)
     if (associated (cmd%el(i)%options)) then
        call set_drawing_options (drawing_options, cmd%el(i)%local%var_list)
     end if
     call analysis_fill_graph (cmd%id, i, cmd%element_id(i), drawing_options)
  end do
end subroutine cmd_graph_execute
```

Analysis

Hold the analysis ID either as a string or as an expression:

⟨*Commands: types*⟩+≡

```
type :: analysis_id_t
  type(string_t) :: tag
  type(parse_node_t), pointer :: pn_sexpr => null ()
end type analysis_id_t
```

Define the analysis expression. We store the parse tree for the right-hand side instead of compiling it. Compilation is deferred to the process environment where the analysis expression is used.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_analysis_t
   private
   type(parse_node_t), pointer :: pn_lexpr => null ()
 contains
 ⟨Commands: cmd analysis: TBP⟩
end type cmd_analysis_t
```

Output. Print just a message that analysis has been defined.

⟨*Commands: cmd analysis: TBP*⟩≡

```
procedure :: write => cmd_analysis_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_analysis_write (cmd, unit, indent)
  class(cmd_analysis_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "analysis: [defined]"
end subroutine cmd_analysis_write
```

Compile. Simply store the parse (sub)tree.

⟨*Commands: cmd analysis: TBP*⟩+≡

```
procedure :: compile => cmd_analysis_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_analysis_compile (cmd, global)
  class(cmd_analysis_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  cmd%pn_lexpr => parse_node_get_sub_ptr (cmd%pn, 3)
end subroutine cmd_analysis_compile
```

Instead of evaluating the cut expression, link the parse tree to the global data set, such that it is compiled and executed in the appropriate process context.

⟨*Commands: cmd analysis: TBP*⟩+≡

```
procedure :: execute => cmd_analysis_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_analysis_execute (cmd, global)
  class(cmd_analysis_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  global%pn%analysis_lexpr => cmd%pn_lexpr
end subroutine cmd_analysis_execute
```

Write histograms and plots

The data type encapsulating the command:

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_write_analysis_t
   private
   type(analysis_id_t), dimension(:), allocatable :: id
   type(string_t), dimension(:), allocatable :: tag
 contains
 ⟨Commands: cmd write analysis: TBP⟩
end type cmd_write_analysis_t
```

Output. Just the keyword.

⟨*Commands: cmd write analysis: TBP*⟩≡

```
procedure :: write => cmd_write_analysis_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_write_analysis_write (cmd, unit, indent)
  class(cmd_write_analysis_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "write_analysis"
end subroutine cmd_write_analysis_write
```

Compile.

⟨*Commands: cmd write analysis: TBP*⟩+≡

```
procedure :: compile => cmd_write_analysis_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_write_analysis_compile (cmd, global)
  class(cmd_write_analysis_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_clause, pn_args, pn_id
  integer :: n, i
  pn_clause => parse_node_get_sub_ptr (cmd%pn)
  pn_args => parse_node_get_sub_ptr (pn_clause, 2)
  cmd%pn_opt => parse_node_get_next_ptr (pn_clause)
  call cmd%compile_options (global)
  if (associated (pn_args)) then
     n = parse_node_get_n_sub (pn_args)
     allocate (cmd%id (n))
     do i = 1, n
         pn_id => parse_node_get_sub_ptr (pn_args, i)
         if (char (parse_node_get_rule_key (pn_id)) == "analysis_id") then
            cmd%id(i)%tag = parse_node_get_string (pn_id)
         else
            cmd%id(i)%pn_sexpr => pn_id
         end if
     end do
  else
     allocate (cmd%id (0))
  end if
```

```
    end subroutine cmd_write_analysis_compile
```

The output format for real data values:

⟨*Commands: parameters*⟩+≡

```
    character(*), parameter, public :: &
         DEFAULT_ANALYSIS_FILENAME = "whizard_analysis.dat"
    character(len=1), dimension(2), parameter, public :: &
         FORBIDDEN_ENDINGS1 = [ "o", "a" ]
    character(len=2), dimension(5), parameter, public :: &
         FORBIDDEN_ENDINGS2 = [ "mp", "ps", "vg", "lo", "la" ]
    character(len=3), dimension(16), parameter, public :: &
         FORBIDDEN_ENDINGS3 = [ "aux", "dvi", "evt", "evx", "f03", "f90", &
            "f95", "log", "ltp", "mpx", "olc", "olp", "pdf", "phs", "sin", "tex" ]
```

As this contains a lot of similar code to `cmd_compile_analysis_execute` we outsource the main code to a subroutine.

⟨*Commands: cmd write analysis: TBP*⟩+≡

```
    procedure :: execute => cmd_write_analysis_execute
```

⟨*Commands: procedures*⟩+≡

```
    subroutine cmd_write_analysis_execute (cmd, global)
      class(cmd_write_analysis_t), intent(inout) :: cmd
      type(rt_data_t), intent(inout), target :: global
      type(var_list_t), pointer :: var_list
      var_list => cmd%local%get_var_list_ptr ()
      call write_analysis_wrap (var_list, global%out_files, &
           cmd%id, tag = cmd%tag)
    end subroutine cmd_write_analysis_execute
```

If the `data_file` optional argument is present, this is called from `cmd_compile_analysis_execute`, which needs the file name for further processing, and requires the default format. For the moment, parameters and macros for custom data processing are disabled.

⟨*Commands: procedures*⟩+≡

```
    subroutine write_analysis_wrap (var_list, out_files, id, tag, data_file)
      type(var_list_t), intent(inout), target :: var_list
      type(file_list_t), intent(inout), target :: out_files
      type(analysis_id_t), dimension(:), intent(in), target :: id
      type(string_t), dimension(:), allocatable, intent(out) :: tag
      type(string_t), intent(out), optional :: data_file
      type(string_t) :: defaultfile, file
      integer :: i
      logical :: keep_open !, custom, header, columns
      type(string_t) :: extension !, comment_prefix, separator
  !!! JRR: WK please check (#542)
  !     integer :: type
  !     type(ifile_t) :: ifile
      logical :: one_file !, has_writer
  !     type(analysis_iterator_t) :: iterator
  !     type(rt_data_t), target :: sandbox
  !     type(command_list_t) :: writer
      defaultfile = var_list%get_sval (var_str ("$out_file"))
```

```
    if (present (data_file)) then
       if (defaultfile == "" .or. defaultfile == ".") then
          defaultfile = DEFAULT_ANALYSIS_FILENAME
       else
          if (scan (".", defaultfile) > 0) then
             call split (defaultfile, extension, ".", back=.true.)
             if (any (lower_case (char(extension)) == FORBIDDEN_ENDINGS1) .or. &
                 any (lower_case (char(extension)) == FORBIDDEN_ENDINGS2) .or. &
                 any (lower_case (char(extension)) == FORBIDDEN_ENDINGS3)) &
                 call msg_fatal ("The ending " // char(extension) // &
                 " is internal and not allowed as data file.")
             if (extension /= "") then
                if (defaultfile /= "") then
                   defaultfile = defaultfile // "." // extension
                else
                   defaultfile = "whizard_analysis." // extension
                end if
             else
                defaultfile = defaultfile // ".dat"
             endif
          else
             defaultfile = defaultfile // ".dat"
          end if
       end if
       data_file = defaultfile
    end if
    one_file = defaultfile /= ""
    if (one_file) then
       file = defaultfile
       keep_open = file_list_is_open (out_files, file, &
            action = "write")
       if (keep_open) then
          if (present (data_file)) then
             call msg_fatal ("Compiling analysis: File '" &
                  // char (data_file) &
                   // "' can't be used, it is already open.")
          else
             call msg_message ("Appending analysis data to file '" &
                  // char (file) // "'")
          end if
       else
          call file_list_open (out_files, file, &
               action = "write", status = "replace", position = "asis")
          call msg_message ("Writing analysis data to file '" &
               // char (file) // "'")
       end if
    end if

!!! JRR: WK please check. Custom data output. Ticket #542
!     if (present (data_file)) then
!        custom = .false.
!     else
!        custom = var_list%get_lval (&
!            var_str ("?out_custom"))
```

```
!     end if
!     comment_prefix = var_list%get_sval (&
!          var_str ("$out_comment"))
!     header = var_list%get_lval (&
!          var_str ("?out_header"))
!     write_yerr = var_list%get_lval (&
!          var_str ("?out_yerr"))
!     write_xerr = var_list%get_lval (&
!          var_str ("?out_xerr"))

    call get_analysis_tags (tag, id, var_list)
    do i = 1, size (tag)
       call file_list_write_analysis &
            (out_files, file, tag(i))
    end do
    if (one_file .and. .not. keep_open) then
       call file_list_close (out_files, file)
    end if

  contains

    subroutine get_analysis_tags (analysis_tag, id, var_list)
      type(string_t), dimension(:), intent(out), allocatable :: analysis_tag
      type(analysis_id_t), dimension(:), intent(in) :: id
      type(var_list_t), intent(in), target :: var_list
      if (size (id) /= 0) then
         allocate (analysis_tag (size (id)))
         do i = 1, size (id)
            if (associated (id(i)%pn_sexpr)) then
               analysis_tag(i) = eval_string (id(i)%pn_sexpr, var_list)
            else
               analysis_tag(i) = id(i)%tag
            end if
         end do
      else
         call analysis_store_get_ids (tag)
      end if
    end subroutine get_analysis_tags

  end subroutine write_analysis_wrap
```

Compile analysis results

This command writes files in a form suitable for GAMELAN and executes the appropriate commands to compile them. The first part is identical to `cmd_write_analysis`.

⟨*Commands: types*⟩+≡

```
  type, extends (command_t) :: cmd_compile_analysis_t
     private
     type(analysis_id_t), dimension(:), allocatable :: id
     type(string_t), dimension(:), allocatable :: tag
   contains
```

```
  ⟨Commands: cmd compile analysis: TBP⟩
end type cmd_compile_analysis_t
```

Output. Just the keyword.

⟨*Commands: cmd compile analysis: TBP*⟩≡

```
procedure :: write => cmd_compile_analysis_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_compile_analysis_write (cmd, unit, indent)
  class(cmd_compile_analysis_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "compile_analysis"
end subroutine cmd_compile_analysis_write
```

Compile.

⟨*Commands: cmd compile analysis: TBP*⟩+≡

```
procedure :: compile => cmd_compile_analysis_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_compile_analysis_compile (cmd, global)
  class(cmd_compile_analysis_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_clause, pn_args, pn_id
  integer :: n, i
  pn_clause => parse_node_get_sub_ptr (cmd%pn)
  pn_args => parse_node_get_sub_ptr (pn_clause, 2)
  cmd%pn_opt => parse_node_get_next_ptr (pn_clause)
  call cmd%compile_options (global)
  if (associated (pn_args)) then
     n = parse_node_get_n_sub (pn_args)
     allocate (cmd%id (n))
     do i = 1, n
         pn_id => parse_node_get_sub_ptr (pn_args, i)
         if (char (parse_node_get_rule_key (pn_id)) == "analysis_id") then
            cmd%id(i)%tag = parse_node_get_string (pn_id)
         else
            cmd%id(i)%pn_sexpr => pn_id
         end if
     end do
  else
     allocate (cmd%id (0))
  end if
end subroutine cmd_compile_analysis_compile
```

First write the analysis data to file, then write a GAMELAN driver and produce MetaPost and TEX output.

⟨*Commands: cmd compile analysis: TBP*⟩+≡

```
procedure :: execute => cmd_compile_analysis_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_compile_analysis_execute (cmd, global)
  class(cmd_compile_analysis_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(var_list_t), pointer :: var_list
  type(string_t) :: file, basename, extension, driver_file, &
       makefile
  integer :: u_driver, u_makefile
  logical :: has_gmlcode, only_file
  var_list => cmd%local%get_var_list_ptr ()
  call write_analysis_wrap (var_list, &
       global%out_files, cmd%id, tag = cmd%tag, &
          data_file = file)
  basename = file
  if (scan (".", basename) > 0) then
    call split (basename, extension, ".", back=.true.)
  else
    extension = ""
  end if
  driver_file = basename // ".tex"
  makefile = basename // "_ana.makefile"
  u_driver = free_unit ()
  open (unit=u_driver, file=char(driver_file), &
        action="write", status="replace")
  if (allocated (cmd%tag)) then
     call analysis_write_driver (file, cmd%tag, unit=u_driver)
     has_gmlcode = analysis_has_plots (cmd%tag)
  else
     call analysis_write_driver (file, unit=u_driver)
     has_gmlcode = analysis_has_plots ()
  end if
  close (u_driver)
  u_makefile = free_unit ()
  open (unit=u_makefile, file=char(makefile), &
       action="write", status="replace")
  call analysis_write_makefile (basename, u_makefile, &
       has_gmlcode, global%os_data)
  close (u_makefile)
  call msg_message ("Compiling analysis results display in '" &
       // char (driver_file) // "'")
  call msg_message ("Providing analysis steering makefile '" &
       // char (makefile) // "'")
  only_file = global%var_list%get_lval &
       (var_str ("?analysis_file_only"))
  if (.not. only_file)  call analysis_compile_tex &
       (basename, has_gmlcode, global%os_data)
end subroutine cmd_compile_analysis_execute
```

35.1.4 User-controlled output to data files

Open file (output)

Open a file for output.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_open_out_t
   private
   type(parse_node_t), pointer :: file_expr => null ()
 contains
 ⟨Commands: cmd open out: TBP⟩
end type cmd_open_out_t
```

Finalizer for the embedded eval tree.

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_open_out_final (object)
  class(cmd_open_out_t), intent(inout) :: object
end subroutine cmd_open_out_final
```

Output (trivial here).

⟨*Commands: cmd open out: TBP*⟩≡

```
procedure :: write => cmd_open_out_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_open_out_write (cmd, unit, indent)
  class(cmd_open_out_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)", advance="no")  "open_out: <filename>"
end subroutine cmd_open_out_write
```

Compile: create an eval tree for the filename expression.

⟨*Commands: cmd open out: TBP*⟩+≡

```
procedure :: compile => cmd_open_out_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_open_out_compile (cmd, global)
  class(cmd_open_out_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  cmd%file_expr => parse_node_get_sub_ptr (cmd%pn, 2)
  if (associated (cmd%file_expr)) then
     cmd%pn_opt => parse_node_get_next_ptr (cmd%file_expr)
  end if
  call cmd%compile_options (global)
end subroutine cmd_open_out_compile
```

Execute: append the file to the global list of open files.

⟨*Commands: cmd open out: TBP*⟩+≡

```
procedure :: execute => cmd_open_out_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_open_out_execute (cmd, global)
  class(cmd_open_out_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(var_list_t), pointer :: var_list
  type(eval_tree_t) :: file_expr
```

```
    type(string_t) :: file
    var_list => cmd%local%get_var_list_ptr ()
    call file_expr%init_sexpr (cmd%file_expr, var_list)
    call file_expr%evaluate ()
    if (file_expr%is_known ()) then
       file = file_expr%get_string ()
       call file_list_open (global%out_files, file, &
            action = "write", status = "replace", position = "asis")
    else
       call msg_fatal ("open_out: file name argument evaluates to unknown")
    end if
    call file_expr%final ()
  end subroutine cmd_open_out_execute
```

Open file (output)

Close an output file. Except for the `execute` method, everything is analogous to the open command, so we can just inherit.

⟨*Commands: types*⟩+≡

```
  type, extends (cmd_open_out_t) :: cmd_close_out_t
     private
   contains
   ⟨Commands: cmd close out: TBP⟩
  end type cmd_close_out_t
```

Execute: remove the file from the global list of output files.

⟨*Commands: cmd close out: TBP*⟩≡

```
  procedure :: execute => cmd_close_out_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_close_out_execute (cmd, global)
    class(cmd_close_out_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(var_list_t), pointer :: var_list
    type(eval_tree_t) :: file_expr
    type(string_t) :: file
    var_list => cmd%local%var_list
    call file_expr%init_sexpr (cmd%file_expr, var_list)
    call file_expr%evaluate ()
    if (file_expr%is_known ()) then
       file = file_expr%get_string ()
       call file_list_close (global%out_files, file)
    else
       call msg_fatal ("close_out: file name argument evaluates to unknown")
    end if
    call file_expr%final ()
  end subroutine cmd_close_out_execute
```

35.1.5 Print custom-formatted values

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_printf_t
   private
   type(parse_node_t), pointer :: sexpr => null ()
   type(parse_node_t), pointer :: sprintf_fun => null ()
   type(parse_node_t), pointer :: sprintf_clause => null ()
   type(parse_node_t), pointer :: sprintf => null ()
 contains
 ⟨Commands: cmd printf: TBP⟩
end type cmd_printf_t
```

Finalize.

⟨*Commands: cmd printf: TBP*⟩≡

```
procedure :: final => cmd_printf_final
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_printf_final (cmd)
  class(cmd_printf_t), intent(inout) :: cmd
  call parse_node_final (cmd%sexpr, recursive = .false.)
  deallocate (cmd%sexpr)
  call parse_node_final (cmd%sprintf_fun, recursive = .false.)
  deallocate (cmd%sprintf_fun)
  call parse_node_final (cmd%sprintf_clause, recursive = .false.)
  deallocate (cmd%sprintf_clause)
  call parse_node_final (cmd%sprintf, recursive = .false.)
  deallocate (cmd%sprintf)
end subroutine cmd_printf_final
```

Output. Do not print the parse tree, since this may get cluttered. Just a message that cuts have been defined.

⟨*Commands: cmd printf: TBP*⟩+≡

```
procedure :: write => cmd_printf_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_printf_write (cmd, unit, indent)
  class(cmd_printf_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "printf:"
end subroutine cmd_printf_write
```

Compile. We create a fake parse node (subtree) with a `sprintf` command with identical arguments which can then be handled by the corresponding evaluation procedure.

⟨*Commands: cmd printf: TBP*⟩+≡

```
procedure :: compile => cmd_printf_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_printf_compile (cmd, global)
  class(cmd_printf_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_cmd, pn_clause, pn_args, pn_format
```

```
    pn_cmd => parse_node_get_sub_ptr (cmd%pn)
    pn_clause => parse_node_get_sub_ptr (pn_cmd)
    pn_format => parse_node_get_sub_ptr (pn_clause, 2)
    pn_args => parse_node_get_next_ptr (pn_clause)
    cmd%pn_opt => parse_node_get_next_ptr (pn_cmd)
    call cmd%compile_options (global)
    allocate (cmd%sexpr)
    call parse_node_create_branch (cmd%sexpr, &
         syntax_get_rule_ptr (syntax_cmd_list, var_str ("sexpr")))
    allocate (cmd%sprintf_fun)
    call parse_node_create_branch (cmd%sprintf_fun, &
         syntax_get_rule_ptr (syntax_cmd_list, var_str ("sprintf_fun")))
    allocate (cmd%sprintf_clause)
    call parse_node_create_branch (cmd%sprintf_clause, &
         syntax_get_rule_ptr (syntax_cmd_list, var_str ("sprintf_clause")))
    allocate (cmd%sprintf)
    call parse_node_create_key (cmd%sprintf, &
         syntax_get_rule_ptr (syntax_cmd_list, var_str ("sprintf")))
    call parse_node_append_sub (cmd%sprintf_clause, cmd%sprintf)
    call parse_node_append_sub (cmd%sprintf_clause, pn_format)
    call parse_node_freeze_branch (cmd%sprintf_clause)
    call parse_node_append_sub (cmd%sprintf_fun, cmd%sprintf_clause)
    if (associated (pn_args)) then
       call parse_node_append_sub (cmd%sprintf_fun, pn_args)
    end if
    call parse_node_freeze_branch (cmd%sprintf_fun)
    call parse_node_append_sub (cmd%sexpr, cmd%sprintf_fun)
    call parse_node_freeze_branch (cmd%sexpr)
  end subroutine cmd_printf_compile
```

Execute. Evaluate the string (pretending this is a `sprintf` expression) and print it.

⟨*Commands: cmd printf: TBP*⟩+≡

```
  procedure :: execute => cmd_printf_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_printf_execute (cmd, global)
    class(cmd_printf_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(var_list_t), pointer :: var_list
    type(string_t) :: string, file
    type(eval_tree_t) :: sprintf_expr
    logical :: advance
    var_list => cmd%local%get_var_list_ptr ()
    advance = var_list%get_lval (&
         var_str ("?out_advance"))
    file = var_list%get_sval (&
         var_str ("$out_file"))
    call sprintf_expr%init_sexpr (cmd%sexpr, var_list)
    call sprintf_expr%evaluate ()
    if (sprintf_expr%is_known ()) then
       string = sprintf_expr%get_string ()
       if (len (file) == 0) then
          call msg_result (char (string))
```

```
      else
         call file_list_write (global%out_files, file, string, advance)
      end if
   end if
end subroutine cmd_printf_execute
```

Record data

The expression syntax already contains a `record` keyword; this evaluates to a logical which is always true, but it has the side-effect of recording data into analysis objects. Here we define a command as an interface to this construct.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_record_t
   private
   type(parse_node_t), pointer :: pn_lexpr => null ()
 contains
 ⟨Commands: cmd record: TBP⟩
end type cmd_record_t
```

Output. With the compile hack below, there is nothing of interest to print here.

⟨*Commands: cmd record: TBP*⟩≡

```
procedure :: write => cmd_record_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_record_write (cmd, unit, indent)
  class(cmd_record_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)")  "record"
end subroutine cmd_record_write
```

Compile. This is a hack which transforms the `record` command into a `record` expression, which we handle in the `expressions` module.

⟨*Commands: cmd record: TBP*⟩+≡

```
procedure :: compile => cmd_record_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_record_compile (cmd, global)
  class(cmd_record_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_lexpr, pn_lsinglet, pn_lterm, pn_record
  call parse_node_create_branch (pn_lexpr, &
       syntax_get_rule_ptr (syntax_cmd_list, var_str ("lexpr")))
  call parse_node_create_branch (pn_lsinglet, &
       syntax_get_rule_ptr (syntax_cmd_list, var_str ("lsinglet")))
  call parse_node_append_sub (pn_lexpr, pn_lsinglet)
  call parse_node_create_branch (pn_lterm, &
       syntax_get_rule_ptr (syntax_cmd_list, var_str ("lterm")))
  call parse_node_append_sub (pn_lsinglet, pn_lterm)
  pn_record => parse_node_get_sub_ptr (cmd%pn)
```

```
    call parse_node_append_sub (pn_lterm, pn_record)
    cmd%pn_lexpr => pn_lexpr
  end subroutine cmd_record_compile
```

Command execution. Again, transfer this to the embedded expression and just forget the logical result.

⟨*Commands: cmd record: TBP*⟩+≡

```
  procedure :: execute => cmd_record_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_record_execute (cmd, global)
    class(cmd_record_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(var_list_t), pointer :: var_list
    logical :: lval
    var_list => global%get_var_list_ptr ()
    lval = eval_log (cmd%pn_lexpr, var_list)
  end subroutine cmd_record_execute
```

Unstable particles

Mark a particle as unstable. For each unstable particle, we store a number of decay channels and compute their respective BRs.

⟨*Commands: types*⟩+≡

```
  type, extends (command_t) :: cmd_unstable_t
     private
     integer :: n_proc = 0
     type(string_t), dimension(:), allocatable :: process_id
     type(parse_node_t), pointer :: pn_prt_in => null ()
   contains
   ⟨Commands: cmd unstable: TBP⟩
  end type cmd_unstable_t
```

Output: we know the process IDs.

⟨*Commands: cmd unstable: TBP*⟩≡

```
  procedure :: write => cmd_unstable_write
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_unstable_write (cmd, unit, indent)
    class(cmd_unstable_t), intent(in) :: cmd
    integer, intent(in), optional :: unit, indent
    integer :: u, i
    u = given_output_unit (unit);  if (u < 0)  return
    call write_indent (u, indent)
    write (u, "(1x,A,1x,I0,1x,A)", advance="no")  &
         "unstable:", 1, "("
    do i = 1, cmd%n_proc
       if (i > 1)  write (u, "(A,1x)", advance="no")  ","
       write (u, "(A)", advance="no")  char (cmd%process_id(i))
    end do
    write (u, "(A)")  ")"
  end subroutine cmd_unstable_write
```

Compile. Initiate an eval tree for the decaying particle and determine the decay channel process IDs.

⟨*Commands: cmd unstable: TBP*⟩+≡

```
procedure :: compile => cmd_unstable_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_unstable_compile (cmd, global)
  class(cmd_unstable_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_list, pn_proc
  integer :: i
  cmd%pn_prt_in => parse_node_get_sub_ptr (cmd%pn, 2)
  pn_list => parse_node_get_next_ptr (cmd%pn_prt_in)
  if (associated (pn_list)) then
     select case (char (parse_node_get_rule_key (pn_list)))
     case ("unstable_arg")
        cmd%n_proc = parse_node_get_n_sub (pn_list)
        cmd%pn_opt => parse_node_get_next_ptr (pn_list)
     case default
        cmd%n_proc = 0
        cmd%pn_opt => pn_list
        pn_list => null ()
     end select
  end if
  call cmd%compile_options (global)
  if (associated (pn_list)) then
     allocate (cmd%process_id (cmd%n_proc))
     pn_proc => parse_node_get_sub_ptr (pn_list)
     do i = 1, cmd%n_proc
        cmd%process_id(i) = parse_node_get_string (pn_proc)
        call cmd%local%process_stack%init_result_vars (cmd%process_id(i))
        pn_proc => parse_node_get_next_ptr (pn_proc)
     end do
  else
     allocate (cmd%process_id (0))
  end if
end subroutine cmd_unstable_compile
```

Command execution. Evaluate the decaying particle and mark the decays in the current model object.

⟨*Commands: cmd unstable: TBP*⟩+≡

```
procedure :: execute => cmd_unstable_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_unstable_execute (cmd, global)
  class(cmd_unstable_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(var_list_t), pointer :: var_list
  logical :: auto_decays, auto_decays_radiative
  integer :: auto_decays_multiplicity
  logical :: isotropic_decay, diagonal_decay, polarized_decay
  integer :: decay_helicity
  type(pdg_array_t) :: pa_in
  integer :: pdg_in
```

```
type(string_t) :: libname_cur, libname_dec
type(string_t), dimension(:), allocatable :: auto_id, tmp_id
integer :: n_proc_user
integer :: i, u_tmp
character(80) :: buffer
var_list => cmd%local%get_var_list_ptr ()
auto_decays = &
     var_list%get_lval (var_str ("?auto_decays"))
if (auto_decays) then
   auto_decays_multiplicity = &
        var_list%get_ival (var_str ("auto_decays_multiplicity"))
   auto_decays_radiative = &
        var_list%get_lval (var_str ("?auto_decays_radiative"))
end if
isotropic_decay = &
     var_list%get_lval (var_str ("?isotropic_decay"))
if (isotropic_decay) then
   diagonal_decay = .false.
   polarized_decay = .false.
else
   diagonal_decay = &
        var_list%get_lval (var_str ("?diagonal_decay"))
   if (diagonal_decay) then
      polarized_decay = .false.
   else
      polarized_decay = &
           var_list%is_known (var_str ("decay_helicity"))
      if (polarized_decay) then
         decay_helicity = var_list%get_ival (var_str ("decay_helicity"))
      end if
   end if
end if
pa_in = eval_pdg_array (cmd%pn_prt_in, var_list)
if (pdg_array_get_length (pa_in) /= 1) &
     call msg_fatal ("Unstable: decaying particle must be unique")
pdg_in = pdg_array_get (pa_in, 1)
n_proc_user = cmd%n_proc
if (auto_decays) then
   call create_auto_decays (pdg_in, &
        auto_decays_multiplicity, auto_decays_radiative, &
        libname_dec, auto_id, cmd%local)
   allocate (tmp_id (cmd%n_proc + size (auto_id)))
   tmp_id(:cmd%n_proc) = cmd%process_id
   tmp_id(cmd%n_proc+1:) = auto_id
   call move_alloc (from = tmp_id, to = cmd%process_id)
   cmd%n_proc = size (cmd%process_id)
end if
libname_cur = cmd%local%prclib%get_name ()
do i = 1, cmd%n_proc
   if (i == n_proc_user + 1) then
      call cmd%local%update_prclib &
           (cmd%local%prclib_stack%get_library_ptr (libname_dec))
   end if
   if (.not. global%process_stack%exists (cmd%process_id(i))) then
```

```
            call var_list%set_log &
                 (var_str ("?decay_rest_frame"), .false., is_known = .true.)
            call integrate_process (cmd%process_id(i), cmd%local, global)
            call global%process_stack%fill_result_vars (cmd%process_id(i))
         end if
      end do
      call cmd%local%update_prclib &
           (cmd%local%prclib_stack%get_library_ptr (libname_cur))
      if (cmd%n_proc > 0) then
         if (polarized_decay) then
            call global%modify_particle (pdg_in, stable = .false., &
                 decay = cmd%process_id, &
                 isotropic_decay = .false., &
                 diagonal_decay = .false., &
                 decay_helicity = decay_helicity, &
                 polarized = .false.)
         else
            call global%modify_particle (pdg_in, stable = .false., &
                 decay = cmd%process_id, &
                 isotropic_decay = isotropic_decay, &
                 diagonal_decay = diagonal_decay, &
                 polarized = .false.)
         end if
         u_tmp = free_unit ()
         open (u_tmp, status = "scratch", action = "readwrite")
         call show_unstable (global, pdg_in, u_tmp)
         rewind (u_tmp)
         do
            read (u_tmp, "(A)", end = 1)  buffer
            write (msg_buffer, "(A)")  trim (buffer)
            call msg_message ()
         end do
1        continue
         close (u_tmp)
      else
         call err_unstable (global, pdg_in)
      end if
    end subroutine cmd_unstable_execute
```

Show data for the current unstable particle. This is called both by the `unstable` and by the `show` command.

⟨*Commands: procedures*⟩+≡

```
    subroutine show_unstable (global, pdg, u)
      type(rt_data_t), intent(in), target :: global
      integer, intent(in) :: pdg, u
      type(flavor_t) :: flv
      type(string_t), dimension(:), allocatable :: decay
      real(default), dimension(:), allocatable :: br
      real(default) :: width
      type(process_t), pointer :: process
      type(process_component_def_t), pointer :: prc_def
      type(string_t), dimension(:), allocatable :: prt_out, prt_out_str
      integer :: i, j
```

```
logical :: opened
call flv%init (pdg, global%model)
call flv%get_decays (decay)
if (.not. allocated (decay))  return
allocate (prt_out_str (size (decay)))
allocate (br (size (decay)))
do i = 1, size (br)
   process => global%process_stack%get_process_ptr (decay(i))
   prc_def => process%get_component_def_ptr (1)
   call prc_def%get_prt_out (prt_out)
   prt_out_str(i) = prt_out(1)
   do j = 2, size (prt_out)
      prt_out_str(i) = prt_out_str(i) // ", " // prt_out(j)
   end do
   br(i) = process%get_integral ()
end do
if (all (br >= 0)) then
   if (any (br > 0)) then
      width = sum (br)
      br = br / sum (br)
      write (u, "(A)") "Unstable particle " &
           // char (flv%get_name ()) &
           // ": computed branching ratios:"
      do i = 1, size (br)
         write (u, "(2x,A,':'," // FMT_14 // ",3x,A)") &
              char (decay(i)), br(i), char (prt_out_str(i))
      end do
      write (u, "(2x,'Total width =',," // FMT_14 // ",' GeV (computed)')")  width
      write (u, "(2x,'            =',," // FMT_14 // ",' GeV (preset)')") &
           flv%get_width ()
      if (flv%decays_isotropically ()) then
         write (u, "(2x,A)")  "Decay options: isotropic"
      else if (flv%decays_diagonal ()) then
         write (u, "(2x,A)")  "Decay options: &
              &projection on diagonal helicity states"
      else if (flv%has_decay_helicity ()) then
         write (u, "(2x,A,1x,I0)")  "Decay options: projection onto helicity =", &
              flv%get_decay_helicity ()
      else
         write (u, "(2x,A)")  "Decay options: helicity treated exactly"
      end if
   else
      inquire (unit = u, opened = opened)
      if (opened .and. .not. mask_fatal_errors)  close (u)
      call msg_fatal ("Unstable particle " &
           // char (flv%get_name ()) &
           // ": partial width vanishes for all decay channels")
   end if
else
   inquire (unit = u, opened = opened)
   if (opened .and. .not. mask_fatal_errors)  close (u)
   call msg_fatal ("Unstable particle " &
           // char (flv%get_name ()) &
           // ": partial width is negative")
```

```
    end if
  end subroutine show_unstable
```

If no decays have been found, issue a non-fatal error.

⟨*Commands: procedures*⟩+≡

```
  subroutine err_unstable (global, pdg)
    type(rt_data_t), intent(in), target :: global
    integer, intent(in) :: pdg
    type(flavor_t) :: flv
    call flv%init (pdg, global%model)
    call msg_error ("Unstable: no allowed decays found for particle " &
         // char (flv%get_name ()) // ", keeping as stable")
  end subroutine err_unstable
```

Auto decays: create process IDs and make up process configurations, using the PDG codes generated by the `ds_table` make method.

We allocate and use a self-contained process library that contains only the decay processes of the current particle. When done, we revert the global library pointer to the original library but return the name of the new one. The new library becomes part of the global library stack and can thus be referred to at any time.

⟨*Commands: procedures*⟩+≡

```
  subroutine create_auto_decays &
       (pdg_in, mult, rad, libname_dec, process_id, global)
    integer, intent(in) :: pdg_in
    integer, intent(in) :: mult
    logical, intent(in) :: rad
    type(string_t), intent(out) :: libname_dec
    type(string_t), dimension(:), allocatable, intent(out) :: process_id
    type(rt_data_t), intent(inout) :: global
    type(prclib_entry_t), pointer :: lib_entry
    type(process_library_t), pointer :: lib
    type(ds_table_t) :: ds_table
    type(split_constraints_t) :: constraints
    type(pdg_array_t), dimension(:), allocatable :: pa_out
    character(80) :: buffer
    character :: p_or_a
    type(string_t) :: process_string, libname_cur
    type(flavor_t) :: flv_in, flv_out
    type(string_t) :: prt_in
    type(string_t), dimension(:), allocatable :: prt_out
    type(process_configuration_t) :: prc_config
    integer :: i, j, k
    call flv_in%init (pdg_in, global%model)
    if (rad) then
       call constraints%init (2)
    else
       call constraints%init (3)
       call constraints%set (3, constrain_radiation ())
    end if
    call constraints%set (1, constrain_n_tot (mult))
    call constraints%set (2, &
```

```
     constrain_mass_sum (flv_in%get_mass (), margin = 0._default))
call ds_table%make (global%model, pdg_in, constraints)
prt_in = flv_in%get_name ()
if (pdg_in > 0) then
   p_or_a = "p"
else
   p_or_a = "a"
end if
if (ds_table%get_length () == 0) then
   call msg_warning ("Auto-decays: Particle " // char (prt_in) // ": " &
        // "no decays found")
   libname_dec = ""
   allocate (process_id (0))
else
   call msg_message ("Creating decay process library for particle " &
        // char (prt_in))
   libname_cur = global%prclib%get_name ()
   write (buffer, "(A,A,I0)")  "_d", p_or_a, abs (pdg_in)
   libname_dec = libname_cur // trim (buffer)
   lib => global%prclib_stack%get_library_ptr (libname_dec)
   if (.not. (associated (lib))) then
      allocate (lib_entry)
      call lib_entry%init (libname_dec)
      lib => lib_entry%process_library_t
      call global%add_prclib (lib_entry)
   else
      call global%update_prclib (lib)
   end if
   allocate (process_id (ds_table%get_length ()))
   do i = 1, size (process_id)
      write (buffer, "(A,'_',A,I0,'_',I0)") &
           "decay", p_or_a, abs (pdg_in), i
      process_id(i) = trim (buffer)
      process_string = process_id(i) // ": " // prt_in // " =>"
      call ds_table%get_pdg_out (i, pa_out)
      allocate (prt_out (size (pa_out)))
      do j = 1, size (pa_out)
         do k = 1, pa_out(j)%get_length ()
            call flv_out%init (pa_out(j)%get (k), global%model)
            if (k == 1) then
               prt_out(j) = flv_out%get_name ()
            else
               prt_out(j) = prt_out(j) // ":" // flv_out%get_name ()
            end if
         end do
         process_string = process_string // " " // prt_out(j)
      end do
      call msg_message (char (process_string))
      call prc_config%init (process_id(i), 1, 1, global)
      !!! Causes runtime error with gfortran 4.9.1
      ! call prc_config%setup_component (1, &
      !      new_prt_spec ([prt_in]), new_prt_spec (prt_out), global%model, global%var_list)
      !!! Workaround:
      call prc_config%setup_component (1, &
```

```
            [new_prt_spec (prt_in)], new_prt_spec (prt_out), global%model, global%var_list)
         call prc_config%record (global)
         deallocate (prt_out)
         deallocate (pa_out)
      end do
      lib => global%prclib_stack%get_library_ptr (libname_cur)
      call global%update_prclib (lib)
   end if
   call ds_table%final ()
end subroutine create_auto_decays
```

(Stable particles

Revert the unstable declaration for a list of particles.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_stable_t
   private
   type(parse_node_p), dimension(:), allocatable :: pn_pdg
 contains
 ⟨Commands: cmd stable: TBP⟩
end type cmd_stable_t
```

Output: we know only the number of particles.

⟨*Commands: cmd stable: TBP*⟩≡

```
procedure :: write => cmd_stable_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_stable_write (cmd, unit, indent)
  class(cmd_stable_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,1x,I0)")  "stable:", size (cmd%pn_pdg)
end subroutine cmd_stable_write
```

Compile. Assign parse nodes for the particle IDs.

⟨*Commands: cmd stable: TBP*⟩+≡

```
procedure :: compile => cmd_stable_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_stable_compile (cmd, global)
  class(cmd_stable_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_list, pn_prt
  integer :: n, i
  pn_list => parse_node_get_sub_ptr (cmd%pn, 2)
  cmd%pn_opt => parse_node_get_next_ptr (pn_list)
  call cmd%compile_options (global)
  n = parse_node_get_n_sub (pn_list)
  allocate (cmd%pn_pdg (n))
  pn_prt => parse_node_get_sub_ptr (pn_list)
```

```
    i = 1
    do while (associated (pn_prt))
       cmd%pn_pdg(i)%ptr => pn_prt
       pn_prt  => parse_node_get_next_ptr (pn_prt)
       i = i + 1
    end do
  end subroutine cmd_stable_compile
```

Execute: apply the modifications to the current model.

⟨*Commands: cmd stable: TBP*⟩+≡

```
  procedure :: execute => cmd_stable_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_stable_execute (cmd, global)
    class(cmd_stable_t), intent(inout) :: cmd
    type(rt_data_t), target, intent(inout) :: global
    type(var_list_t), pointer :: var_list
    type(pdg_array_t) :: pa
    integer :: pdg
    type(flavor_t) :: flv
    integer :: i
    var_list => cmd%local%get_var_list_ptr ()
    do i = 1, size (cmd%pn_pdg)
       pa = eval_pdg_array (cmd%pn_pdg(i)%ptr, var_list)
       if (pdg_array_get_length (pa) /= 1) &
            call msg_fatal ("Stable: listed particles must be unique")
       pdg = pdg_array_get (pa, 1)
       call global%modify_particle (pdg, stable = .true., &
         isotropic_decay = .false., &
         diagonal_decay = .false., &
         polarized = .false.)
       call flv%init (pdg, cmd%local%model)
       call msg_message ("Particle " &
            // char (flv%get_name ()) &
            // " declared as stable")
    end do
  end subroutine cmd_stable_execute
```

Polarized particles

These commands mark particles as (un)polarized, to be applied in subsequent simulation passes. Since this is technically the same as the `stable` command, we take a shortcut and make this an extension, just overriding methods.

⟨*Commands: types*⟩+≡

```
  type, extends (cmd_stable_t) :: cmd_polarized_t
   contains
   ⟨Commands: cmd polarized: TBP⟩
  end type cmd_polarized_t

  type, extends (cmd_stable_t) :: cmd_unpolarized_t
   contains
   ⟨Commands: cmd unpolarized: TBP⟩
```

```
end type cmd_unpolarized_t
```

Output: we know only the number of particles.

⟨*Commands: cmd polarized: TBP*⟩≡

```
procedure :: write => cmd_polarized_write
```

⟨*Commands: cmd unpolarized: TBP*⟩≡

```
procedure :: write => cmd_unpolarized_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_polarized_write (cmd, unit, indent)
  class(cmd_polarized_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,1x,I0)")  "polarized:", size (cmd%pn_pdg)
end subroutine cmd_polarized_write

subroutine cmd_unpolarized_write (cmd, unit, indent)
  class(cmd_unpolarized_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,1x,I0)")  "unpolarized:", size (cmd%pn_pdg)
end subroutine cmd_unpolarized_write
```

Compile: accounted for by the base command.

Execute: apply the modifications to the current model.

⟨*Commands: cmd polarized: TBP*⟩+≡

```
procedure :: execute => cmd_polarized_execute
```

⟨*Commands: cmd unpolarized: TBP*⟩+≡

```
procedure :: execute => cmd_unpolarized_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_polarized_execute (cmd, global)
  class(cmd_polarized_t), intent(inout) :: cmd
  type(rt_data_t), target, intent(inout) :: global
  type(var_list_t), pointer :: var_list
  type(pdg_array_t) :: pa
  integer :: pdg
  type(flavor_t) :: flv
  integer :: i
  var_list => cmd%local%get_var_list_ptr ()
  do i = 1, size (cmd%pn_pdg)
     pa = eval_pdg_array (cmd%pn_pdg(i)%ptr, var_list)
     if (pdg_array_get_length (pa) /= 1) &
          call msg_fatal ("Polarized: listed particles must be unique")
     pdg = pdg_array_get (pa, 1)
     call global%modify_particle (pdg, polarized = .true., &
          stable = .true., &
          isotropic_decay = .false., &
          diagonal_decay = .false.)
```

```
      call flv%init (pdg, cmd%local%model)
      call msg_message ("Particle " &
           // char (flv%get_name ()) &
           // " declared as polarized")
  end do
end subroutine cmd_polarized_execute

subroutine cmd_unpolarized_execute (cmd, global)
  class(cmd_unpolarized_t), intent(inout) :: cmd
  type(rt_data_t), target, intent(inout) :: global
  type(var_list_t), pointer :: var_list
  type(pdg_array_t) :: pa
  integer :: pdg
  type(flavor_t) :: flv
  integer :: i
  var_list => cmd%local%get_var_list_ptr ()
  do i = 1, size (cmd%pn_pdg)
     pa = eval_pdg_array (cmd%pn_pdg(i)%ptr, var_list)
     if (pdg_array_get_length (pa) /= 1) &
          call msg_fatal ("Unpolarized: listed particles must be unique")
     pdg = pdg_array_get (pa, 1)
     call global%modify_particle (pdg, polarized = .false., &
          stable = .true., &
          isotropic_decay = .false., &
          diagonal_decay = .false.)
     call flv%init (pdg, cmd%local%model)
     call msg_message ("Particle " &
          // char (flv%get_name ()) &
          // " declared as unpolarized")
  end do
end subroutine cmd_unpolarized_execute
```

Parameters: formats for event-sample output

Specify all event formats that are to be used for output files in the subsequent simulation run. (The raw format is on by default and can be turned off here.)

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_sample_format_t
   private
   type(string_t), dimension(:), allocatable :: format
 contains
 ⟨Commands: cmd sample format: TBP⟩
end type cmd_sample_format_t
```

Output: here, everything is known.

⟨*Commands: cmd sample format: TBP*⟩≡

```
procedure :: write => cmd_sample_format_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_sample_format_write (cmd, unit, indent)
  class(cmd_sample_format_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
```

```
    integer :: u, i
    u = given_output_unit (unit);  if (u < 0)  return
    call write_indent (u, indent)
    write (u, "(1x,A)", advance="no")  "sample_format = "
    do i = 1, size (cmd%format)
       if (i > 1)  write (u, "(A,1x)", advance="no")  ","
       write (u, "(A)", advance="no")  char (cmd%format(i))
    end do
    write (u, "(A)")
  end subroutine cmd_sample_format_write
```

Compile. Initialize evaluation trees.

⟨*Commands: cmd sample format: TBP*⟩+≡

```
  procedure :: compile => cmd_sample_format_compile
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_sample_format_compile (cmd, global)
    class(cmd_sample_format_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_arg
    type(parse_node_t), pointer :: pn_format
    integer :: i, n_format
    pn_arg => parse_node_get_sub_ptr (cmd%pn, 3)
    if (associated (pn_arg)) then
       n_format = parse_node_get_n_sub (pn_arg)
       allocate (cmd%format (n_format))
       pn_format => parse_node_get_sub_ptr (pn_arg)
       i = 0
       do while (associated (pn_format))
          i = i + 1
          cmd%format(i) = parse_node_get_string (pn_format)
          pn_format => parse_node_get_next_ptr (pn_format)
       end do
    else
       allocate (cmd%format (0))
    end if
  end subroutine cmd_sample_format_compile
```

Execute. Transfer the list of format specifications to the corresponding array in the runtime data set.

⟨*Commands: cmd sample format: TBP*⟩+≡

```
  procedure :: execute => cmd_sample_format_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_sample_format_execute (cmd, global)
    class(cmd_sample_format_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    if (allocated (global%sample_fmt))  deallocate (global%sample_fmt)
    allocate (global%sample_fmt (size (cmd%format)), source = cmd%format)
  end subroutine cmd_sample_format_execute
```

The simulate command

This is the actual SINDARIN command.

⟨*Commands: types*⟩+≡
```
type, extends (command_t) :: cmd_simulate_t
   ! not private anymore as required by the whizard-c-interface
   integer :: n_proc = 0
   type(string_t), dimension(:), allocatable :: process_id
 contains
 ⟨Commands: cmd simulate: TBP⟩
end type cmd_simulate_t
```

Output: we know the process IDs.

⟨*Commands: cmd simulate: TBP*⟩≡
```
procedure :: write => cmd_simulate_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_simulate_write (cmd, unit, indent)
  class(cmd_simulate_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A)", advance="no")  "simulate ("
  do i = 1, cmd%n_proc
     if (i > 1)  write (u, "(A,1x)", advance="no")  ","
     write (u, "(A)", advance="no")  char (cmd%process_id(i))
  end do
  write (u, "(A)")  ")"
end subroutine cmd_simulate_write
```

Compile. In contrast to WHIZARD 1 the confusing option to give the number of unweighted events for weighted events as if unweighting were to take place has been abandoned. (We both use `n_events` for weighted and unweighted events, the variable `n_calls` from WHIZARD 1 has been discarded.

⟨*Commands: cmd simulate: TBP*⟩+≡
```
procedure :: compile => cmd_simulate_compile
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_simulate_compile (cmd, global)
  class(cmd_simulate_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_proclist, pn_proc
  integer :: i
  pn_proclist => parse_node_get_sub_ptr (cmd%pn, 2)
  cmd%pn_opt => parse_node_get_next_ptr (pn_proclist)
  call cmd%compile_options (global)
  cmd%n_proc = parse_node_get_n_sub (pn_proclist)
  allocate (cmd%process_id (cmd%n_proc))
  pn_proc => parse_node_get_sub_ptr (pn_proclist)
  do i = 1, cmd%n_proc
     cmd%process_id(i) = parse_node_get_string (pn_proc)
     call global%process_stack%init_result_vars (cmd%process_id(i))
     pn_proc => parse_node_get_next_ptr (pn_proc)
```

```
    end do
  end subroutine cmd_simulate_compile
```

Execute command: Simulate events. This is done via a `simulation_t` object and its associated methods.

Signal handling: the `generate` method may exit abnormally if there is a pending signal. The current logic ensures that the `es_array` output channels are closed before the `execute` routine returns. The program will terminate then in `command_list_execute`.

⟨*Commands: cmd simulate: TBP*⟩+≡

```
  procedure :: execute => cmd_simulate_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_simulate_execute (cmd, global)
    class(cmd_simulate_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(var_list_t), pointer :: var_list
    type(rt_data_t), dimension(:), allocatable, target :: alt_env
    integer :: n_events, n_fmt
    type(string_t) :: sample
    logical :: rebuild_events, read_raw, write_raw
    type(simulation_t), target :: sim
    type(string_t), dimension(:), allocatable :: sample_fmt
    type(event_stream_array_t) :: es_array
    type(event_sample_data_t) :: data
    integer :: i, checkpoint, callback
    var_list => cmd%local%var_list
    if (allocated (cmd%local%pn%alt_setup)) then
       allocate (alt_env (size (cmd%local%pn%alt_setup)))
       do i = 1, size (alt_env)
          call build_alt_setup (alt_env(i), cmd%local, &
               cmd%local%pn%alt_setup(i)%ptr)
       end do
       call sim%init (cmd%process_id, .true., .true., cmd%local, global, &
            alt_env)
    else
       call sim%init (cmd%process_id, .true., .true., cmd%local, global)
    end if
    if (signal_is_pending ())  return
    if (sim%is_valid ()) then
       call sim%init_process_selector ()
       call openmp_set_num_threads_verbose &
            (var_list%get_ival (var_str ("openmp_num_threads")), &
            var_list%get_lval (var_str ("?openmp_logging")))
       call sim%compute_n_events (n_events, var_list)
       sample = var_list%get_sval (var_str ("$sample"))
       if (sample == "")  sample = sim%get_default_sample_name ()
       rebuild_events = &
            var_list%get_lval (var_str ("?rebuild_events"))
       read_raw = &
            var_list%get_lval (var_str ("?read_raw")) &
            .and. .not. rebuild_events
       write_raw = &
            var_list%get_lval (var_str ("?write_raw"))
```

```
checkpoint = &
     var_list%get_ival (var_str ("checkpoint"))
callback = &
     var_list%get_ival (var_str ("event_callback_interval"))
if (read_raw) then
   inquire (file = char (sample) // ".evx", exist = read_raw)
end if
if (allocated (cmd%local%sample_fmt)) then
   n_fmt = size (cmd%local%sample_fmt)
else
   n_fmt = 0
end if
data = sim%get_data ()
data%n_evt = n_events
data%nlo_multiplier = sim%get_n_nlo_entries (1)
if (read_raw) then
   allocate (sample_fmt (n_fmt))
   if (n_fmt > 0)  sample_fmt = cmd%local%sample_fmt
   call es_array%init (sample, &
        sample_fmt, cmd%local, &
        data = data, &
        input = var_str ("raw"), &
        allow_switch = write_raw, &
        checkpoint = checkpoint, &
        callback = callback)
   call sim%generate (n_events, es_array)
   call es_array%final ()
else if (write_raw) then
   allocate (sample_fmt (n_fmt + 1))
   if (n_fmt > 0)  sample_fmt(:n_fmt) = cmd%local%sample_fmt
   sample_fmt(n_fmt+1) = var_str ("raw")
   call es_array%init (sample, &
        sample_fmt, cmd%local, &
        data = data, &
        checkpoint = checkpoint, &
        callback = callback)
   call sim%generate (n_events, es_array)
   call es_array%final ()
else if (allocated (cmd%local%sample_fmt) &
     .or. checkpoint > 0 &
     .or. callback > 0) then
   allocate (sample_fmt (n_fmt))
   if (n_fmt > 0)  sample_fmt = cmd%local%sample_fmt
   call es_array%init (sample, &
        sample_fmt, cmd%local, &
        data = data, &
        checkpoint = checkpoint, &
        callback = callback)
   call sim%generate (n_events, es_array)
   call es_array%final ()
else
   call sim%generate (n_events)
end if
if (allocated (alt_env)) then
```

```
        do i = 1, size (alt_env)
           call alt_env(i)%local_final ()
        end do
     end if
  end if
  call sim%final ()
end subroutine cmd_simulate_execute
```

Build an alternative setup: the parse tree is stored in the global environment. We create a temporary command list to compile and execute this; the result is an alternative local environment `alt_env` which we can hand over to the `simulate` command.

⟨*Commands: procedures*⟩+≡

```
recursive subroutine build_alt_setup (alt_env, global, pn)
  type(rt_data_t), intent(inout), target :: alt_env
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), intent(in), target :: pn
  type(command_list_t), allocatable :: alt_options
  allocate (alt_options)
  call alt_env%local_init (global)
  call alt_env%activate ()
  call alt_options%compile (pn, alt_env)
  call alt_options%execute (alt_env)
  call alt_env%deactivate (global, keep_local = .true.)
  call alt_options%final ()
end subroutine build_alt_setup
```

The rescan command

This is the actual SINDARIN command.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_rescan_t
   ! private
   type(parse_node_t), pointer :: pn_filename => null ()
   integer :: n_proc = 0
   type(string_t), dimension(:), allocatable :: process_id
 contains
 ⟨Commands: cmd rescan: TBP⟩
end type cmd_rescan_t
```

Output: we know the process IDs.

⟨*Commands: cmd rescan: TBP*⟩≡

```
procedure :: write => cmd_rescan_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_rescan_write (cmd, unit, indent)
  class(cmd_rescan_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u, i
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
```

```
    write (u, "(1x,A)", advance="no")  "rescan ("
    do i = 1, cmd%n_proc
       if (i > 1)  write (u, "(A,1x)", advance="no")  ","
       write (u, "(A)", advance="no")  char (cmd%process_id(i))
    end do
    write (u, "(A)")  ")"
  end subroutine cmd_rescan_write
```

Compile. The command takes a suffix argument, namely the file name of requested event file.

⟨*Commands: cmd rescan: TBP*⟩+≡

```
  procedure :: compile => cmd_rescan_compile
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_rescan_compile (cmd, global)
    class(cmd_rescan_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_filename, pn_proclist, pn_proc
    integer :: i
    pn_filename => parse_node_get_sub_ptr (cmd%pn, 2)
    pn_proclist => parse_node_get_next_ptr (pn_filename)
    cmd%pn_opt => parse_node_get_next_ptr (pn_proclist)
    call cmd%compile_options (global)
    cmd%pn_filename => pn_filename
    cmd%n_proc = parse_node_get_n_sub (pn_proclist)
    allocate (cmd%process_id (cmd%n_proc))
    pn_proc => parse_node_get_sub_ptr (pn_proclist)
    do i = 1, cmd%n_proc
       cmd%process_id(i) = parse_node_get_string (pn_proc)
       pn_proc => parse_node_get_next_ptr (pn_proc)
    end do
  end subroutine cmd_rescan_compile
```

Execute command: Rescan events. This is done via a `simulation_t` object and its associated methods.

⟨*Commands: cmd rescan: TBP*⟩+≡

```
  procedure :: execute => cmd_rescan_execute
```

⟨*Commands: procedures*⟩+≡

```
  subroutine cmd_rescan_execute (cmd, global)
    class(cmd_rescan_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(var_list_t), pointer :: var_list
    type(rt_data_t), dimension(:), allocatable, target :: alt_env
    type(string_t) :: sample
    logical :: exist, write_raw, update_event, update_sqme
    type(simulation_t), target :: sim
    type(event_sample_data_t) :: input_data, data
    type(string_t) :: input_sample
    integer :: n_fmt
    type(string_t), dimension(:), allocatable :: sample_fmt
    type(string_t) :: input_format, input_ext, input_file
    type(string_t) :: lhef_extension, extension_hepmc, extension_lcio
    type(event_stream_array_t) :: es_array
```

```
integer :: i, n_events
var_list => cmd%local%var_list
if (allocated (cmd%local%pn%alt_setup)) then
   allocate (alt_env (size (cmd%local%pn%alt_setup)))
   do i = 1, size (alt_env)
      call build_alt_setup (alt_env(i), cmd%local, &
           cmd%local%pn%alt_setup(i)%ptr)
   end do
   call sim%init (cmd%process_id, .false., .false., cmd%local, global, &
        alt_env)
else
   call sim%init (cmd%process_id, .false., .false., cmd%local, global)
end if
call sim%compute_n_events (n_events, var_list)
input_sample = eval_string (cmd%pn_filename, var_list)
input_format = var_list%get_sval (&
     var_str ("$rescan_input_format"))
sample = var_list%get_sval (var_str ("$sample"))
if (sample == "")  sample = sim%get_default_sample_name ()
write_raw = var_list%get_lval (var_str ("?write_raw"))
if (allocated (cmd%local%sample_fmt)) then
   n_fmt = size (cmd%local%sample_fmt)
else
   n_fmt = 0
end if
if (write_raw) then
   if (sample == input_sample) then
      call msg_error ("Rescan: ?write_raw = true: " &
           // "suppressing raw event output (filename clashes with input)")
      allocate (sample_fmt (n_fmt))
      if (n_fmt > 0)  sample_fmt = cmd%local%sample_fmt
   else
      allocate (sample_fmt (n_fmt + 1))
      if (n_fmt > 0)  sample_fmt(:n_fmt) = cmd%local%sample_fmt
      sample_fmt(n_fmt+1) = var_str ("raw")
   end if
else
   allocate (sample_fmt (n_fmt))
   if (n_fmt > 0)  sample_fmt = cmd%local%sample_fmt
end if
update_event = &
     var_list%get_lval (var_str ("?update_event"))
update_sqme = &
     var_list%get_lval (var_str ("?update_sqme"))
if (update_event .or. update_sqme) then
   call msg_message ("Recalculating observables")
   if (update_sqme) then
      call msg_message ("Recalculating squared matrix elements")
   end if
end if
lhef_extension = &
     var_list%get_sval (var_str ("$lhef_extension"))
extension_hepmc = &
     var_list%get_sval (var_str ("$extension_hepmc"))
```

```
  extension_lcio = &
       var_list%get_sval (var_str ("$extension_lcio"))
  select case (char (input_format))
  case ("raw");  input_ext = "evx"
     call cmd%local%set_log &
          (var_str ("?recover_beams"), .false., is_known=.true.)
  case ("lhef"); input_ext = lhef_extension
  case ("hepmc"); input_ext = extension_hepmc
  case default
     call msg_fatal ("rescan: input sample format '" // char (input_format) &
          // "' not supported")
  end select
  input_file = input_sample // "." // input_ext
  inquire (file = char (input_file), exist = exist)
  if (exist) then
     input_data = sim%get_data (alt = .false.)
     input_data%n_evt = n_events
     data = sim%get_data ()
     data%n_evt = n_events
     input_data%md5sum_cfg = ""
     call es_array%init (sample, &
          sample_fmt, cmd%local, data, &
          input = input_format, input_sample = input_sample, &
          input_data = input_data, &
          allow_switch = .false.)
     call sim%rescan (n_events, es_array, global = cmd%local)
     call es_array%final ()
  else
     call msg_fatal ("Rescan: event file '" &
          // char (input_file) // "' not found")
  end if
  if (allocated (alt_env)) then
     do i = 1, size (alt_env)
        call alt_env(i)%local_final ()
     end do
  end if
  call sim%final ()
end subroutine cmd_rescan_execute
```

Parameters: number of iterations

Specify number of iterations and number of calls for one integration pass.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_iterations_t
   private
   integer :: n_pass = 0
   type(parse_node_p), dimension(:), allocatable :: pn_expr_n_it
   type(parse_node_p), dimension(:), allocatable :: pn_expr_n_calls
   type(parse_node_p), dimension(:), allocatable :: pn_sexpr_adapt
 contains
 ⟨Commands: cmd iterations: TBP⟩
end type cmd_iterations_t
```

Output. Display the number of passes, which is known after compilation.

⟨*Commands: cmd iterations: TBP*⟩≡

```
procedure :: write => cmd_iterations_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_iterations_write (cmd, unit, indent)
  class(cmd_iterations_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  select case (cmd%n_pass)
  case (0)
     write (u, "(1x,A)")  "iterations: [empty]"
  case (1)
     write (u, "(1x,A,I0,A)")  "iterations: ", cmd%n_pass, " pass"
  case default
     write (u, "(1x,A,I0,A)")  "iterations: ", cmd%n_pass, " passes"
  end select
end subroutine cmd_iterations_write
```

Compile. Initialize evaluation trees.

⟨*Commands: cmd iterations: TBP*⟩+≡

```
procedure :: compile => cmd_iterations_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_iterations_compile (cmd, global)
  class(cmd_iterations_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_arg, pn_n_it, pn_n_calls, pn_adapt
  type(parse_node_t), pointer :: pn_it_spec, pn_calls_spec, pn_adapt_spec
  integer :: i
  pn_arg => parse_node_get_sub_ptr (cmd%pn, 3)
  if (associated (pn_arg)) then
     cmd%n_pass = parse_node_get_n_sub (pn_arg)
     allocate (cmd%pn_expr_n_it (cmd%n_pass))
     allocate (cmd%pn_expr_n_calls (cmd%n_pass))
     allocate (cmd%pn_sexpr_adapt (cmd%n_pass))
     pn_it_spec => parse_node_get_sub_ptr (pn_arg)
     i = 1
     do while (associated (pn_it_spec))
        pn_n_it => parse_node_get_sub_ptr (pn_it_spec)
        pn_calls_spec => parse_node_get_next_ptr (pn_n_it)
        pn_n_calls => parse_node_get_sub_ptr (pn_calls_spec, 2)
        pn_adapt_spec => parse_node_get_next_ptr (pn_calls_spec)
        if (associated (pn_adapt_spec)) then
           pn_adapt => parse_node_get_sub_ptr (pn_adapt_spec, 2)
        else
           pn_adapt => null ()
        end if
        cmd%pn_expr_n_it(i)%ptr => pn_n_it
        cmd%pn_expr_n_calls(i)%ptr => pn_n_calls
        cmd%pn_sexpr_adapt(i)%ptr => pn_adapt
        i = i + 1
```

```
         pn_it_spec => parse_node_get_next_ptr (pn_it_spec)
      end do
   else
      allocate (cmd%pn_expr_n_it (0))
      allocate (cmd%pn_expr_n_calls (0))
   end if
end subroutine cmd_iterations_compile
```

Execute. Evaluate the trees and transfer the results to the iteration list in the runtime data set.

⟨*Commands: cmd iterations: TBP*⟩+≡

```
procedure :: execute => cmd_iterations_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_iterations_execute (cmd, global)
  class(cmd_iterations_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(var_list_t), pointer :: var_list
  integer, dimension(cmd%n_pass) :: n_it, n_calls
  logical, dimension(cmd%n_pass) :: custom_adapt
  type(string_t), dimension(cmd%n_pass) :: adapt_code
  integer :: i
  var_list => global%get_var_list_ptr ()
  do i = 1, cmd%n_pass
     n_it(i) = eval_int (cmd%pn_expr_n_it(i)%ptr, var_list)
     n_calls(i) = &
          eval_int (cmd%pn_expr_n_calls(i)%ptr, var_list)
     if (associated (cmd%pn_sexpr_adapt(i)%ptr)) then
        adapt_code(i) = &
             eval_string (cmd%pn_sexpr_adapt(i)%ptr, &
                          var_list, is_known = custom_adapt(i))
     else
        custom_adapt(i) = .false.
     end if
  end do
  call global%it_list%init (n_it, n_calls, custom_adapt, adapt_code)
end subroutine cmd_iterations_execute
```

Range expressions

We need a special type for storing and evaluating range expressions.

⟨*Commands: parameters*⟩+≡

```
integer, parameter :: STEP_NONE = 0
integer, parameter :: STEP_ADD = 1
integer, parameter :: STEP_SUB = 2
integer, parameter :: STEP_MUL = 3
integer, parameter :: STEP_DIV = 4
integer, parameter :: STEP_COMP_ADD = 11
integer, parameter :: STEP_COMP_MUL = 13
```

There is an abstract base type and two implementations: scan over integers and scan over reals.

⟨*Commands: types*⟩+≡

```
type, abstract :: range_t
   type(parse_node_t), pointer :: pn_expr => null ()
   type(parse_node_t), pointer :: pn_term => null ()
   type(parse_node_t), pointer :: pn_factor => null ()
   type(parse_node_t), pointer :: pn_value => null ()
   type(parse_node_t), pointer :: pn_literal => null ()
   type(parse_node_t), pointer :: pn_beg => null ()
   type(parse_node_t), pointer :: pn_end => null ()
   type(parse_node_t), pointer :: pn_step => null ()
   type(eval_tree_t) :: expr_beg
   type(eval_tree_t) :: expr_end
   type(eval_tree_t) :: expr_step
   integer :: step_mode = 0
   integer :: n_step = 0
 contains
 ⟨Commands: range: TBP⟩
end type range_t
```

These are the implementations:

⟨*Commands: types*⟩+≡

```
type, extends (range_t) :: range_int_t
   integer :: i_beg = 0
   integer :: i_end = 0
   integer :: i_step = 0
 contains
 ⟨Commands: range int: TBP⟩
end type range_int_t

type, extends (range_t) :: range_real_t
   real(default) :: r_beg = 0
   real(default) :: r_end = 0
   real(default) :: r_step = 0
   real(default) :: lr_beg  = 0
   real(default) :: lr_end  = 0
   real(default) :: lr_step = 0
 contains
 ⟨Commands: range real: TBP⟩
end type range_real_t
```

Finalize the allocated dummy node. The other nodes are just pointers.

⟨*Commands: range: TBP*⟩≡

```
procedure :: final => range_final
```

⟨*Commands: procedures*⟩+≡

```
subroutine range_final (object)
  class(range_t), intent(inout) :: object
  if (associated (object%pn_expr)) then
     call parse_node_final (object%pn_expr, recursive = .false.)
     call parse_node_final (object%pn_term, recursive = .false.)
     call parse_node_final (object%pn_factor, recursive = .false.)
     call parse_node_final (object%pn_value, recursive = .false.)
     call parse_node_final (object%pn_literal, recursive = .false.)
     deallocate (object%pn_expr)
     deallocate (object%pn_term)
```

```
      deallocate (object%pn_factor)
      deallocate (object%pn_value)
      deallocate (object%pn_literal)
    end if
  end subroutine range_final
```

Output.

⟨*Commands: range: TBP*⟩+≡

```
  procedure (range_write), deferred :: write
  procedure :: base_write => range_write
```

⟨*Commands: range int: TBP*⟩≡

```
  procedure :: write => range_int_write
```

⟨*Commands: range real: TBP*⟩≡

```
  procedure :: write => range_real_write
```

⟨*Commands: procedures*⟩+≡

```
  subroutine range_write (object, unit)
    class(range_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    write (u, "(1x,A)")  "Range specification:"
    if (associated (object%pn_expr)) then
       write (u, "(1x,A)")  "Dummy value:"
       call parse_node_write_rec (object%pn_expr, u)
    end if
    if (associated (object%pn_beg)) then
       write (u, "(1x,A)")  "Initial value:"
       call parse_node_write_rec (object%pn_beg, u)
       call object%expr_beg%write (u)
       if (associated (object%pn_end)) then
          write (u, "(1x,A)")  "Final value:"
          call parse_node_write_rec (object%pn_end, u)
          call object%expr_end%write (u)
          if (associated (object%pn_step)) then
             write (u, "(1x,A)")  "Step value:"
             call parse_node_write_rec (object%pn_step, u)
             select case (object%step_mode)
             case (STEP_ADD);   write (u, "(1x,A)")  "Step mode: +"
             case (STEP_SUB);   write (u, "(1x,A)")  "Step mode: -"
             case (STEP_MUL);   write (u, "(1x,A)")  "Step mode: *"
             case (STEP_DIV);   write (u, "(1x,A)")  "Step mode: /"
             case (STEP_COMP_ADD);  write (u, "(1x,A)")  "Division mode: +"
             case (STEP_COMP_MUL);  write (u, "(1x,A)")  "Division mode: *"
             end select
          end if
       end if
    else
       write (u, "(1x,A)")  "Expressions: [undefined]"
    end if
  end subroutine range_write

  subroutine range_int_write (object, unit)
```

```
    class(range_int_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    call object%base_write (unit)
    write (u, "(1x,A)")  "Range parameters:"
    write (u, "(3x,A,I0)")  "i_beg  = ", object%i_beg
    write (u, "(3x,A,I0)")  "i_end  = ", object%i_end
    write (u, "(3x,A,I0)")  "i_step = ", object%i_step
    write (u, "(3x,A,I0)")  "n_step = ", object%n_step
  end subroutine range_int_write

  subroutine range_real_write (object, unit)
    class(range_real_t), intent(in) :: object
    integer, intent(in), optional :: unit
    integer :: u
    u = given_output_unit (unit)
    call object%base_write (unit)
    write (u, "(1x,A)")  "Range parameters:"
    write (u, "(3x,A," // FMT_19 // ")")  "r_beg  = ", object%r_beg
    write (u, "(3x,A," // FMT_19 // ")")  "r_end  = ", object%r_end
    write (u, "(3x,A," // FMT_19 // ")")  "r_step = ", object%r_end
    write (u, "(3x,A,I0)")  "n_step = ", object%n_step
  end subroutine range_real_write
```

Initialize, given a range expression parse node. This is common to the implementations.

⟨*Commands: range: TBP*⟩+≡

```
  procedure :: init => range_init
```

⟨*Commands: procedures*⟩+≡

```
  subroutine range_init (range, pn)
    class(range_t), intent(out) :: range
    type(parse_node_t), intent(in), target :: pn
    type(parse_node_t), pointer :: pn_spec, pn_end, pn_step_spec, pn_op
    select case (char (parse_node_get_rule_key (pn)))
    case ("expr")
    case ("range_expr")
       range%pn_beg => parse_node_get_sub_ptr (pn)
       pn_spec => parse_node_get_next_ptr (range%pn_beg)
       if (associated (pn_spec)) then
          pn_end => parse_node_get_sub_ptr (pn_spec, 2)
          range%pn_end => pn_end
          pn_step_spec => parse_node_get_next_ptr (pn_end)
          if (associated (pn_step_spec)) then
             pn_op => parse_node_get_sub_ptr (pn_step_spec)
             range%pn_step => parse_node_get_next_ptr (pn_op)
             select case (char (parse_node_get_rule_key (pn_op)))
             case ("/+");  range%step_mode = STEP_ADD
             case ("/-");  range%step_mode = STEP_SUB
             case ("/*");  range%step_mode = STEP_MUL
             case ("//");  range%step_mode = STEP_DIV
             case ("/+/");  range%step_mode = STEP_COMP_ADD
             case ("/*/");  range%step_mode = STEP_COMP_MUL
```

```
            case default
               call range%write ()
               call msg_bug ("Range: step mode not implemented")
            end select
         else
            range%step_mode = STEP_ADD
         end if
      else
         range%step_mode = STEP_NONE
      end if
      call range%create_value_node ()
   case default
      call msg_bug ("range expression: node type '" &
           // char (parse_node_get_rule_key (pn)) &
           // "' not implemented")
   end select
end subroutine range_init
```

This method manually creates a parse node (actually, a cascade of parse nodes) that hold a constant value as a literal. The idea is that this node is inserted as the right-hand side of a fake variable assignment, which is prepended to each scan iteration. Before the variable assignment is compiled and executed, we can manually reset the value of the literal and thus pretend that the loop variable is assigned this value.

⟨*Commands: range: TBP*⟩+≡

```
procedure :: create_value_node => range_create_value_node
```

⟨*Commands: procedures*⟩+≡

```
subroutine range_create_value_node (range)
  class(range_t), intent(inout) :: range
  allocate (range%pn_literal)
  allocate (range%pn_value)
  select type (range)
  type is (range_int_t)
     call parse_node_create_value (range%pn_literal, &
          syntax_get_rule_ptr (syntax_cmd_list, var_str ("integer_literal")),&
          ival = 0)
     call parse_node_create_branch (range%pn_value, &
          syntax_get_rule_ptr (syntax_cmd_list, var_str ("integer_value")))
  type is (range_real_t)
     call parse_node_create_value (range%pn_literal, &
          syntax_get_rule_ptr (syntax_cmd_list, var_str ("real_literal")),&
          rval = 0._default)
     call parse_node_create_branch (range%pn_value, &
          syntax_get_rule_ptr (syntax_cmd_list, var_str ("real_value")))
  class default
     call msg_bug ("range: create value node: type not implemented")
  end select
  call parse_node_append_sub (range%pn_value, range%pn_literal)
  call parse_node_freeze_branch (range%pn_value)
  allocate (range%pn_factor)
  call parse_node_create_branch (range%pn_factor, &
       syntax_get_rule_ptr (syntax_cmd_list, var_str ("factor")))
  call parse_node_append_sub (range%pn_factor, range%pn_value)
```

```
    call parse_node_freeze_branch (range%pn_factor)
    allocate (range%pn_term)
    call parse_node_create_branch (range%pn_term, &
         syntax_get_rule_ptr (syntax_cmd_list, var_str ("term")))
    call parse_node_append_sub (range%pn_term, range%pn_factor)
    call parse_node_freeze_branch (range%pn_term)
    allocate (range%pn_expr)
    call parse_node_create_branch (range%pn_expr, &
         syntax_get_rule_ptr (syntax_cmd_list, var_str ("expr")))
    call parse_node_append_sub (range%pn_expr, range%pn_term)
    call parse_node_freeze_branch (range%pn_expr)
  end subroutine range_create_value_node
```

Compile, given an environment.

⟨*Commands: range: TBP*⟩+≡

```
  procedure :: compile => range_compile
```

⟨*Commands: procedures*⟩+≡

```
  subroutine range_compile (range, global)
    class(range_t), intent(inout) :: range
    type(rt_data_t), intent(in), target :: global
    type(var_list_t), pointer :: var_list
    var_list => global%get_var_list_ptr ()
    if (associated (range%pn_beg)) then
       call range%expr_beg%init_expr (range%pn_beg, var_list)
       if (associated (range%pn_end)) then
          call range%expr_end%init_expr (range%pn_end, var_list)
          if (associated (range%pn_step)) then
             call range%expr_step%init_expr (range%pn_step, var_list)
          end if
       end if
    end if
  end subroutine range_compile
```

Evaluate: compute the actual bounds and parameters that determine the values that we can iterate.

This is implementation-specific.

⟨*Commands: range: TBP*⟩+≡

```
  procedure (range_evaluate), deferred :: evaluate
```

⟨*Commands: interfaces*⟩+≡

```
  abstract interface
     subroutine range_evaluate (range)
       import
       class(range_t), intent(inout) :: range
     end subroutine range_evaluate
  end interface
```

The version for an integer variable. If the step is subtractive, we invert the sign and treat it as an additive step. For a multiplicative step, the step must be greater than one, and the initial and final values must be of same sign and strictly ordered. Analogously for a division step.

⟨*Commands: range int: TBP*⟩+≡

```
procedure :: evaluate => range_int_evaluate
```

⟨*Commands: procedures*⟩+≡

```
subroutine range_int_evaluate (range)
  class(range_int_t), intent(inout) :: range
  integer :: ival
  if (associated (range%pn_beg)) then
     call range%expr_beg%evaluate ()
     if (range%expr_beg%is_known ()) then
        range%i_beg = range%expr_beg%get_int ()
     else
        call range%write ()
        call msg_fatal &
             ("Range expression: initial value evaluates to unknown")
     end if
     if (associated (range%pn_end)) then
        call range%expr_end%evaluate ()
        if (range%expr_end%is_known ()) then
           range%i_end = range%expr_end%get_int ()
           if (associated (range%pn_step)) then
              call range%expr_step%evaluate ()
              if (range%expr_step%is_known ()) then
                 range%i_step = range%expr_step%get_int ()
                 select case (range%step_mode)
                 case (STEP_SUB);  range%i_step = - range%i_step
                 end select
              else
                 call range%write ()
                 call msg_fatal &
                      ("Range expression: step value evaluates to unknown")
              end if
           else
              range%i_step = 1
           end if
        else
           call range%write ()
           call msg_fatal &
                ("Range expression: final value evaluates to unknown")
        end if
     else
        range%i_end = range%i_beg
        range%i_step = 1
     end if
     select case (range%step_mode)
     case (STEP_NONE)
        range%n_step = 1
     case (STEP_ADD, STEP_SUB)
        if (range%i_step /= 0) then
           if (range%i_beg == range%i_end) then
              range%n_step = 1
           else if (sign (1, range%i_end - range%i_beg) &
                == sign (1, range%i_step)) then
              range%n_step = (range%i_end - range%i_beg) / range%i_step + 1
           else
              range%n_step = 0
```

```
      end if
   else
      call msg_fatal ("range evaluation (add): step value is zero")
   end if
case (STEP_MUL)
   if (range%i_step > 1) then
      if (range%i_beg == range%i_end) then
         range%n_step = 1
      else if (range%i_beg == 0) then
         call msg_fatal ("range evaluation (mul): initial value is zero")
      else if (sign (1, range%i_beg) == sign (1, range%i_end) &
           .and. abs (range%i_beg) < abs (range%i_end)) then
         range%n_step = 0
         ival = range%i_beg
         do while (abs (ival) <= abs (range%i_end))
            range%n_step = range%n_step + 1
            ival = ival * range%i_step
         end do
      else
         range%n_step = 0
      end if
   else
      call msg_fatal &
           ("range evaluation (mult): step value is one or less")
   end if
case (STEP_DIV)
   if (range%i_step > 1) then
      if (range%i_beg == range%i_end) then
         range%n_step = 1
      else if (sign (1, range%i_beg) == sign (1, range%i_end) &
           .and. abs (range%i_beg) > abs (range%i_end)) then
         range%n_step = 0
         ival = range%i_beg
         do while (abs (ival) >= abs (range%i_end))
            range%n_step = range%n_step + 1
            if (ival == 0)  exit
            ival = ival / range%i_step
         end do
      else
         range%n_step = 0
      end if
   else
      call msg_fatal &
           ("range evaluation (div): step value is one or less")
   end if
case (STEP_COMP_ADD)
   call msg_fatal ("range evaluation: &
        &step mode /+/ not allowed for integer variable")
case (STEP_COMP_MUL)
   call msg_fatal ("range evaluation: &
        &step mode /*/ not allowed for integer variable")
case default
   call range%write ()
   call msg_bug ("range evaluation: step mode not implemented")
```

```
      end select
    end if
  end subroutine range_int_evaluate
```

The version for a real variable.

⟨*Commands: range real: TBP*⟩+≡

```
  procedure :: evaluate => range_real_evaluate
```

⟨*Commands: procedures*⟩+≡

```
  subroutine range_real_evaluate (range)
    class(range_real_t), intent(inout) :: range
    if (associated (range%pn_beg)) then
       call range%expr_beg%evaluate ()
       if (range%expr_beg%is_known ()) then
          range%r_beg = range%expr_beg%get_real ()
       else
          call range%write ()
          call msg_fatal &
               ("Range expression: initial value evaluates to unknown")
       end if
       if (associated (range%pn_end)) then
          call range%expr_end%evaluate ()
          if (range%expr_end%is_known ()) then
             range%r_end = range%expr_end%get_real ()
             if (associated (range%pn_step)) then
                if (range%expr_step%is_known ()) then
                   select case (range%step_mode)
                   case (STEP_ADD, STEP_SUB, STEP_MUL, STEP_DIV)
                      call range%expr_step%evaluate ()
                      range%r_step = range%expr_step%get_real ()
                      select case (range%step_mode)
                      case (STEP_SUB);  range%r_step = - range%r_step
                      end select
                   case (STEP_COMP_ADD, STEP_COMP_MUL)
                      range%n_step = &
                           max (range%expr_step%get_int (), 0)
                   end select
                else
                   call range%write ()
                   call msg_fatal &
                        ("Range expression: step value evaluates to unknown")
                end if
             else
                call range%write ()
                call msg_fatal &
                     ("Range expression (real): step value must be provided")
             end if
          else
             call range%write ()
             call msg_fatal &
                  ("Range expression: final value evaluates to unknown")
          end if
       else
          range%r_end = range%r_beg
```

```
   range%r_step = 1
end if
select case (range%step_mode)
case (STEP_NONE)
   range%n_step = 1
case (STEP_ADD, STEP_SUB)
   if (range%r_step /= 0) then
      if (sign (1._default, range%r_end - range%r_beg) &
           == sign (1._default, range%r_step)) then
         range%n_step = &
              nint ((range%r_end - range%r_beg) / range%r_step + 1)
      else
         range%n_step = 0
      end if
   else
      call msg_fatal ("range evaluation (add): step value is zero")
   end if
case (STEP_MUL)
   if (range%r_step > 1) then
      if (range%r_beg == 0 .or. range%r_end == 0) then
         call msg_fatal ("range evaluation (mul): bound is zero")
      else if (sign (1._default, range%r_beg) &
           == sign (1._default, range%r_end) &
           .and. abs (range%r_beg) <= abs (range%r_end)) then
         range%lr_beg = log (abs (range%r_beg))
         range%lr_end = log (abs (range%r_end))
         range%lr_step = log (range%r_step)
         range%n_step = nint &
              (abs ((range%lr_end - range%lr_beg) / range%lr_step) + 1)
      else
         range%n_step = 0
      end if
   else
      call msg_fatal &
           ("range evaluation (mult): step value is one or less")
   end if
case (STEP_DIV)
   if (range%r_step > 1) then
      if (range%r_beg == 0 .or. range%r_end == 0) then
         call msg_fatal ("range evaluation (div): bound is zero")
      else if (sign (1._default, range%r_beg) &
           == sign (1._default, range%r_end) &
           .and. abs (range%r_beg) >= abs (range%r_end)) then
         range%lr_beg = log (abs (range%r_beg))
         range%lr_end = log (abs (range%r_end))
         range%lr_step = -log (range%r_step)
         range%n_step = nint &
              (abs ((range%lr_end - range%lr_beg) / range%lr_step) + 1)
      else
         range%n_step = 0
      end if
   else
      call msg_fatal &
           ("range evaluation (mult): step value is one or less")
```

```
         end if
      case (STEP_COMP_ADD)
         ! Number of steps already known
      case (STEP_COMP_MUL)
         ! Number of steps already known
         if (range%r_beg == 0 .or. range%r_end == 0) then
            call msg_fatal ("range evaluation (mul): bound is zero")
         else if (sign (1._default, range%r_beg) &
              == sign (1._default, range%r_end)) then
            range%lr_beg = log (abs (range%r_beg))
            range%lr_end = log (abs (range%r_end))
         else
            range%n_step = 0
         end if
      case default
         call range%write ()
         call msg_bug ("range evaluation: step mode not implemented")
      end select
   end if
end subroutine range_real_evaluate
```

Return the number of iterations:

⟨*Commands: range: TBP*⟩+≡

```
procedure :: get_n_iterations => range_get_n_iterations
```

⟨*Commands: procedures*⟩+≡

```
function range_get_n_iterations (range) result (n)
  class(range_t), intent(in) :: range
  integer :: n
  n = range%n_step
end function range_get_n_iterations
```

Compute the value for iteration `i` and store it in the embedded token.

⟨*Commands: range: TBP*⟩+≡

```
procedure (range_set_value), deferred :: set_value
```

⟨*Commands: interfaces*⟩+≡

```
abstract interface
   subroutine range_set_value (range, i)
     import
     class(range_t), intent(inout) :: range
     integer, intent(in) :: i
   end subroutine range_set_value
end interface
```

In the integer case, we compute the value directly for additive step. For multiplicative step, we perform a loop in the same way as above, where the number of iteration was determined.

⟨*Commands: range int: TBP*⟩+≡

```
procedure :: set_value => range_int_set_value
```

⟨*Commands: procedures*⟩+≡

```
subroutine range_int_set_value (range, i)
  class(range_int_t), intent(inout) :: range
  integer, intent(in) :: i
  integer :: k, ival
  select case (range%step_mode)
  case (STEP_NONE)
     ival = range%i_beg
  case (STEP_ADD, STEP_SUB)
     ival = range%i_beg + (i - 1) * range%i_step
  case (STEP_MUL)
     ival = range%i_beg
     do k = 1, i - 1
        ival = ival * range%i_step
     end do
  case (STEP_DIV)
     ival = range%i_beg
     do k = 1, i - 1
        ival = ival / range%i_step
     end do
  case default
     call range%write ()
     call msg_bug ("range iteration: step mode not implemented")
  end select
  call parse_node_set_value (range%pn_literal, ival = ival)
end subroutine range_int_set_value
```

In the integer case, we compute the value directly for additive step. For multiplicative step, we perform a loop in the same way as above, where the number of iteration was determined.

⟨*Commands: range real: TBP*⟩+≡

```
procedure :: set_value => range_real_set_value
```

⟨*Commands: procedures*⟩+≡

```
subroutine range_real_set_value (range, i)
  class(range_real_t), intent(inout) :: range
  integer, intent(in) :: i
  real(default) :: rval, x
  select case (range%step_mode)
  case (STEP_NONE)
     rval = range%r_beg
  case (STEP_ADD, STEP_SUB, STEP_COMP_ADD)
     if (range%n_step > 1) then
        x = real (i - 1, default) / (range%n_step - 1)
     else
        x = 1._default / 2
     end if
     rval = x * range%r_end + (1 - x) * range%r_beg
  case (STEP_MUL, STEP_DIV, STEP_COMP_MUL)
     if (range%n_step > 1) then
        x = real (i - 1, default) / (range%n_step - 1)
     else
        x = 1._default / 2
     end if
```

```
      rval = sign &
           (exp (x * range%lr_end + (1 - x) * range%lr_beg), range%r_beg)
   case default
      call range%write ()
      call msg_bug ("range iteration: step mode not implemented")
   end select
   call parse_node_set_value (range%pn_literal, rval = rval)
end subroutine range_real_set_value
```

Scan over parameters and other objects

The scan command allocates a new parse node for the variable assignment (the lhs). The rhs of this parse node is assigned from the available rhs expressions in the scan list, one at a time, so the compiled parse node can be prepended to the scan body.

Note: for the integer/real range array, the obvious implementation as a polymorphic array is suspended because in gfortran 4.7, polymorphic arrays are apparently broken.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_scan_t
   private
   type(string_t) :: name
   integer :: n_values = 0
   type(parse_node_p), dimension(:), allocatable :: scan_cmd
   !!! !!! gfortran 4.7.x memory corruption
   !!!  class(range_t), dimension(:), allocatable :: range
   type(range_int_t), dimension(:), allocatable :: range_int
   type(range_real_t), dimension(:), allocatable :: range_real
 contains
 ⟨Commands: cmd scan: TBP⟩
end type cmd_scan_t
```

Finalizer.

The auxiliary parse nodes that we have constructed have to be treated carefully: the embedded pointers all point to persistent objects somewhere else and should not be finalized, so we should not call the finalizer recursively.

⟨*Commands: cmd scan: TBP*⟩≡

```
procedure :: final => cmd_scan_final
```

⟨*Commands: procedures*⟩+≡

```
recursive subroutine cmd_scan_final (cmd)
  class(cmd_scan_t), intent(inout) :: cmd
  type(parse_node_t), pointer :: pn_var_single, pn_decl_single
  type(string_t) :: key
  integer :: i
  if (allocated (cmd%scan_cmd)) then
     do i = 1, size (cmd%scan_cmd)
        pn_var_single => parse_node_get_sub_ptr (cmd%scan_cmd(i)%ptr)
        key = parse_node_get_rule_key (pn_var_single)
        select case (char (key))
        case ("scan_string_decl", "scan_log_decl")
           pn_decl_single => parse_node_get_sub_ptr (pn_var_single, 2)
```

```
            call parse_node_final (pn_decl_single, recursive=.false.)
            deallocate (pn_decl_single)
         end select
         call parse_node_final (pn_var_single, recursive=.false.)
         deallocate (pn_var_single)
      end do
      deallocate (cmd%scan_cmd)
   end if
   !!! !!! gfortran 4.7.x memory corruption
   !!!  if (allocated (cmd%range)) then
   !!!     do i = 1, size (cmd%range)
   !!!        call cmd%range(i)%final ()
   !!!     end do
   !!!  end if
   if (allocated (cmd%range_int)) then
      do i = 1, size (cmd%range_int)
         call cmd%range_int(i)%final ()
      end do
   end if
   if (allocated (cmd%range_real)) then
      do i = 1, size (cmd%range_real)
         call cmd%range_real(i)%final ()
      end do
   end if
end subroutine cmd_scan_final
```

Output.

⟨*Commands: cmd scan: TBP*⟩+≡

```
procedure :: write => cmd_scan_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_scan_write (cmd, unit, indent)
  class(cmd_scan_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,1x,A,1x,'(',I0,')')")  "scan:", char (cmd%name), &
       cmd%n_values
end subroutine cmd_scan_write
```

Compile the scan command. We construct a new parse node that implements the variable assignment for a single element on the rhs, instead of the whole list that we get from the original parse tree. By simply copying the node, we copy all pointers and inherit the targets from the original. During execution, we should replace the rhs by the stored rhs pointers (the list elements), one by one, then (re)compile the redefined node.

⟨*Commands: cmd scan: TBP*⟩+≡

```
procedure :: compile => cmd_scan_compile
```

⟨*Commands: procedures*⟩+≡

```
recursive subroutine cmd_scan_compile (cmd, global)
  class(cmd_scan_t), intent(inout) :: cmd
```

```
type(rt_data_t), intent(inout), target :: global
type(var_list_t), pointer :: var_list
type(parse_node_t), pointer :: pn_var, pn_body, pn_body_first
type(parse_node_t), pointer :: pn_decl, pn_name
type(parse_node_t), pointer :: pn_arg, pn_scan_cmd, pn_rhs
type(parse_node_t), pointer :: pn_decl_single, pn_var_single
type(syntax_rule_t), pointer :: var_rule_decl, var_rule
type(string_t) :: key
integer :: var_type
integer :: i
logical, parameter :: debug = .false.
if (debug) then
   print *, "compile scan"
   call parse_node_write_rec (cmd%pn)
end if
pn_var => parse_node_get_sub_ptr (cmd%pn, 2)
pn_body => parse_node_get_next_ptr (pn_var)
if (associated (pn_body)) then
   pn_body_first => parse_node_get_sub_ptr (pn_body)
else
   pn_body_first => null ()
end if
key = parse_node_get_rule_key (pn_var)
select case (char (key))
case ("scan_num")
   pn_name => parse_node_get_sub_ptr (pn_var)
   cmd%name = parse_node_get_string (pn_name)
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, var_str ("cmd_num"))
   pn_arg => parse_node_get_next_ptr (pn_name, 2)
case ("scan_int")
   pn_name => parse_node_get_sub_ptr (pn_var, 2)
   cmd%name = parse_node_get_string (pn_name)
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, var_str ("cmd_int"))
   pn_arg => parse_node_get_next_ptr (pn_name, 2)
case ("scan_real")
   pn_name => parse_node_get_sub_ptr (pn_var, 2)
   cmd%name = parse_node_get_string (pn_name)
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, var_str ("cmd_real"))
   pn_arg => parse_node_get_next_ptr (pn_name, 2)
case ("scan_complex")
   pn_name => parse_node_get_sub_ptr (pn_var, 2)
   cmd%name = parse_node_get_string (pn_name)
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, var_str("cmd_complex"))
   pn_arg => parse_node_get_next_ptr (pn_name, 2)
case ("scan_alias")
   pn_name => parse_node_get_sub_ptr (pn_var, 2)
   cmd%name = parse_node_get_string (pn_name)
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, var_str ("cmd_alias"))
   pn_arg => parse_node_get_next_ptr (pn_name, 2)
case ("scan_string_decl")
   pn_decl => parse_node_get_sub_ptr (pn_var, 2)
   pn_name => parse_node_get_sub_ptr (pn_decl, 2)
   cmd%name = parse_node_get_string (pn_name)
   var_rule_decl => syntax_get_rule_ptr (syntax_cmd_list, &
```

```
        var_str ("cmd_string"))
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
        var_str ("cmd_string_decl"))
   pn_arg => parse_node_get_next_ptr (pn_name, 2)
case ("scan_log_decl")
   pn_decl => parse_node_get_sub_ptr (pn_var, 2)
   pn_name => parse_node_get_sub_ptr (pn_decl, 2)
   cmd%name = parse_node_get_string (pn_name)
   var_rule_decl => syntax_get_rule_ptr (syntax_cmd_list, &
        var_str ("cmd_log"))
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
        var_str ("cmd_log_decl"))
   pn_arg => parse_node_get_next_ptr (pn_name, 2)
case ("scan_cuts")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
        var_str ("cmd_cuts"))
   cmd%name = "cuts"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_weight")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
        var_str ("cmd_weight"))
   cmd%name = "weight"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_scale")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
        var_str ("cmd_scale"))
   cmd%name = "scale"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_ren_scale")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
        var_str ("cmd_ren_scale"))
   cmd%name = "renormalization_scale"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_fac_scale")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
        var_str ("cmd_fac_scale"))
   cmd%name = "factorization_scale"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_selection")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
        var_str ("cmd_selection"))
   cmd%name = "selection"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_reweight")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
        var_str ("cmd_reweight"))
   cmd%name = "reweight"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_analysis")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
        var_str ("cmd_analysis"))
   cmd%name = "analysis"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_model")
```

```
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
        var_str ("cmd_model"))
   cmd%name = "model"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case ("scan_library")
   var_rule => syntax_get_rule_ptr (syntax_cmd_list, &
        var_str ("cmd_library"))
   cmd%name = "library"
   pn_arg => parse_node_get_sub_ptr (pn_var, 3)
case default
   call msg_bug ("scan: case '" // char (key) // "' not implemented")
end select
if (associated (pn_arg)) then
   cmd%n_values = parse_node_get_n_sub (pn_arg)
end if
var_list => global%get_var_list_ptr ()
allocate (cmd%scan_cmd (cmd%n_values))
select case (char (key))
case ("scan_num")
   var_type = &
        var_list%get_type (cmd%name)
   select case (var_type)
   case (V_INT)
      !!! !!! gfortran 4.7.x memory corruption
      !!!  allocate (range_int_t :: cmd%range (cmd%n_values))
      allocate (cmd%range_int (cmd%n_values))
   case (V_REAL)
      !!! !!! gfortran 4.7.x memory corruption
      !!!  allocate (range_real_t :: cmd%range (cmd%n_values))
      allocate (cmd%range_real (cmd%n_values))
   case (V_CMPLX)
      call msg_fatal ("scan over complex variable not implemented")
   case (V_NONE)
      call msg_fatal ("scan: variable '" // char (cmd%name) //"' undefined")
   case default
      call msg_bug ("scan: impossible variable type")
   end select
case ("scan_int")
   !!! !!! gfortran 4.7.x memory corruption
   !!!  allocate (range_int_t :: cmd%range (cmd%n_values))
   allocate (cmd%range_int (cmd%n_values))
case ("scan_real")
   !!! !!! gfortran 4.7.x memory corruption
   !!!  allocate (range_real_t :: cmd%range (cmd%n_values))
   allocate (cmd%range_real (cmd%n_values))
case ("scan_complex")
   call msg_fatal ("scan over complex variable not implemented")
end select
i = 1
if (associated (pn_arg)) then
   pn_rhs => parse_node_get_sub_ptr (pn_arg)
else
   pn_rhs => null ()
end if
```

```
do while (associated (pn_rhs))
   allocate (pn_scan_cmd)
   call parse_node_create_branch (pn_scan_cmd, &
        syntax_get_rule_ptr (syntax_cmd_list, var_str ("command_list")))
   allocate (pn_var_single)
   pn_var_single = pn_var
   call parse_node_replace_rule (pn_var_single, var_rule)
   select case (char (key))
   case ("scan_num", "scan_int", "scan_real", &
        "scan_complex", "scan_alias", &
        "scan_cuts", "scan_weight", &
        "scan_scale", "scan_ren_scale", "scan_fac_scale", &
        "scan_selection", "scan_reweight", "scan_analysis", &
        "scan_model", "scan_library")
      if (allocated (cmd%range_int)) then
         call cmd%range_int(i)%init (pn_rhs)
         !!! !!! gfortran 4.7.x memory corruption
         !!!  call cmd%range_int(i)%compile (global)
         call parse_node_replace_last_sub &
              (pn_var_single, cmd%range_int(i)%pn_expr)
      else if (allocated (cmd%range_real)) then
         call cmd%range_real(i)%init (pn_rhs)
         !!! !!! gfortran 4.7.x memory corruption
         !!!  call cmd%range_real(i)%compile (global)
         call parse_node_replace_last_sub &
              (pn_var_single, cmd%range_real(i)%pn_expr)
      else
         call parse_node_replace_last_sub (pn_var_single, pn_rhs)
      end if
   case ("scan_string_decl", "scan_log_decl")
      allocate (pn_decl_single)
      pn_decl_single = pn_decl
      call parse_node_replace_rule (pn_decl_single, var_rule_decl)
      call parse_node_replace_last_sub (pn_decl_single, pn_rhs)
      call parse_node_freeze_branch (pn_decl_single)
      call parse_node_replace_last_sub (pn_var_single, pn_decl_single)
   case default
      call msg_bug ("scan: case '" // char (key)  &
           // "' broken")
   end select
   call parse_node_freeze_branch (pn_var_single)
   call parse_node_append_sub (pn_scan_cmd, pn_var_single)
   call parse_node_append_sub (pn_scan_cmd, pn_body_first)
   call parse_node_freeze_branch (pn_scan_cmd)
   cmd%scan_cmd(i)%ptr => pn_scan_cmd
   i = i + 1
   pn_rhs => parse_node_get_next_ptr (pn_rhs)
end do
if (debug) then
   do i = 1, cmd%n_values
      print *, "scan command ", i
      call parse_node_write_rec (cmd%scan_cmd(i)%ptr)
      if (allocated (cmd%range_int))  call cmd%range_int(i)%write ()
      if (allocated (cmd%range_real))  call cmd%range_real(i)%write ()
```

```
      end do
      print *, "original"
      call parse_node_write_rec (cmd%pn)
    end if
  end subroutine cmd_scan_compile
```

Execute the loop for all values in the step list. We use the parse trees with single variable assignment that we have stored, to iteratively create a local environment, execute the stored commands, and destroy it again. When we encounter a range object, we execute the commands for each value that this object provides. Computing this value has the side effect of modifying the rhs of the variable assignment that heads the local command list, directly in the local parse tree.

⟨*Commands: cmd scan: TBP*⟩+≡

```
  procedure :: execute => cmd_scan_execute
```

⟨*Commands: procedures*⟩+≡

```
  recursive subroutine cmd_scan_execute (cmd, global)
    class(cmd_scan_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(rt_data_t), allocatable :: local
    integer :: i, j
    do i = 1, cmd%n_values
      if (allocated (cmd%range_int)) then
        call cmd%range_int(i)%compile (global)
        call cmd%range_int(i)%evaluate ()
        do j = 1, cmd%range_int(i)%get_n_iterations ()
          call cmd%range_int(i)%set_value (j)
          allocate (local)
          call build_alt_setup (local, global, cmd%scan_cmd(i)%ptr)
          call local%local_final ()
          deallocate (local)
        end do
      else if (allocated (cmd%range_real)) then
        call cmd%range_real(i)%compile (global)
        call cmd%range_real(i)%evaluate ()
        do j = 1, cmd%range_real(i)%get_n_iterations ()
          call cmd%range_real(i)%set_value (j)
          allocate (local)
          call build_alt_setup (local, global, cmd%scan_cmd(i)%ptr)
          call local%local_final ()
          deallocate (local)
        end do
      else
        allocate (local)
        call build_alt_setup (local, global, cmd%scan_cmd(i)%ptr)
        call local%local_final ()
        deallocate (local)
      end if
    end do
  end subroutine cmd_scan_execute
```

Conditionals

Conditionals are implemented as a list that is compiled and evaluated recursively; this allows for a straightforward representation of `else if` constructs. A `cmd_if_t` object can hold either an `else_if` clause which is another object of this type, or an `else_body`, but not both.

If- or else-bodies are no scoping units, so all data remain global and no copy-in copy-out is needed.

⟨*Commands: types*⟩+≡
```
type, extends (command_t) :: cmd_if_t
   private
   type(parse_node_t), pointer :: pn_if_lexpr => null ()
   type(command_list_t), pointer :: if_body => null ()
   type(cmd_if_t), dimension(:), pointer :: elsif_cmd => null ()
   type(command_list_t), pointer :: else_body => null ()
 contains
 ⟨Commands: cmd if: TBP⟩
end type cmd_if_t
```

Finalizer. There are no local options, therefore we can simply override the default finalizer.

⟨*Commands: cmd if: TBP*⟩≡
```
procedure :: final => cmd_if_final
```

⟨*Commands: procedures*⟩+≡
```
recursive subroutine cmd_if_final (cmd)
  class(cmd_if_t), intent(inout) :: cmd
  integer :: i
  if (associated (cmd%if_body)) then
     call command_list_final (cmd%if_body)
     deallocate (cmd%if_body)
  end if
  if (associated (cmd%elsif_cmd)) then
     do i = 1, size (cmd%elsif_cmd)
        call cmd_if_final (cmd%elsif_cmd(i))
     end do
     deallocate (cmd%elsif_cmd)
  end if
  if (associated (cmd%else_body)) then
     call command_list_final (cmd%else_body)
     deallocate (cmd%else_body)
  end if
end subroutine cmd_if_final
```

Output. Recursively write the command lists.

⟨*Commands: cmd if: TBP*⟩+≡
```
procedure :: write => cmd_if_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine cmd_if_write (cmd, unit, indent)
  class(cmd_if_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u, ind, i
```

```
    u = given_output_unit (unit);  if (u < 0)  return
    ind = 0;  if (present (indent))  ind = indent
    call write_indent (u, indent)
    write (u, "(A)")  "if <expr> then"
    if (associated (cmd%if_body)) then
       call cmd%if_body%write (unit, ind + 1)
    end if
    if (associated (cmd%elsif_cmd)) then
       do i = 1, size (cmd%elsif_cmd)
          call write_indent (u, indent)
          write (u, "(A)")  "elsif <expr> then"
          if (associated (cmd%elsif_cmd(i)%if_body)) then
             call cmd%elsif_cmd(i)%if_body%write (unit, ind + 1)
          end if
       end do
    end if
    if (associated (cmd%else_body)) then
       call write_indent (u, indent)
       write (u, "(A)")  "else"
       call cmd%else_body%write (unit, ind + 1)
    end if
  end subroutine cmd_if_write
```

Compile the conditional.

⟨*Commands: cmd if: TBP*⟩+≡

```
  procedure :: compile => cmd_if_compile
```

⟨*Commands: procedures*⟩+≡

```
  recursive subroutine cmd_if_compile (cmd, global)
    class(cmd_if_t), intent(inout) :: cmd
    type(rt_data_t), intent(inout), target :: global
    type(parse_node_t), pointer :: pn_lexpr, pn_body
    type(parse_node_t), pointer :: pn_elsif_clauses, pn_cmd_elsif
    type(parse_node_t), pointer :: pn_else_clause, pn_cmd_else
    integer :: i, n_elsif
    pn_lexpr => parse_node_get_sub_ptr (cmd%pn, 2)
    cmd%pn_if_lexpr => pn_lexpr
    pn_body => parse_node_get_next_ptr (pn_lexpr, 2)
    select case (char (parse_node_get_rule_key (pn_body)))
    case ("command_list")
       allocate (cmd%if_body)
       call cmd%if_body%compile (pn_body, global)
       pn_elsif_clauses => parse_node_get_next_ptr (pn_body)
    case default
       pn_elsif_clauses => pn_body
    end select
    select case (char (parse_node_get_rule_key (pn_elsif_clauses)))
    case ("elsif_clauses")
       n_elsif = parse_node_get_n_sub (pn_elsif_clauses)
       allocate (cmd%elsif_cmd (n_elsif))
       pn_cmd_elsif => parse_node_get_sub_ptr (pn_elsif_clauses)
       do i = 1, n_elsif
          pn_lexpr => parse_node_get_sub_ptr (pn_cmd_elsif, 2)
          cmd%elsif_cmd(i)%pn_if_lexpr => pn_lexpr
```

```
         pn_body => parse_node_get_next_ptr (pn_lexpr, 2)
         if (associated (pn_body)) then
            allocate (cmd%elsif_cmd(i)%if_body)
            call cmd%elsif_cmd(i)%if_body%compile (pn_body, global)
         end if
         pn_cmd_elsif => parse_node_get_next_ptr (pn_cmd_elsif)
      end do
      pn_else_clause => parse_node_get_next_ptr (pn_elsif_clauses)
   case default
      pn_else_clause => pn_elsif_clauses
   end select
   select case (char (parse_node_get_rule_key (pn_else_clause)))
   case ("else_clause")
      pn_cmd_else => parse_node_get_sub_ptr (pn_else_clause)
      pn_body => parse_node_get_sub_ptr (pn_cmd_else, 2)
      if (associated (pn_body)) then
         allocate (cmd%else_body)
         call cmd%else_body%compile (pn_body, global)
      end if
   end select
end subroutine cmd_if_compile
```

(Recursively) execute the condition. Context remains global in all cases.

⟨*Commands: cmd if: TBP*⟩+≡

```
procedure :: execute => cmd_if_execute
```

⟨*Commands: procedures*⟩+≡

```
recursive subroutine cmd_if_execute (cmd, global)
   class(cmd_if_t), intent(inout) :: cmd
   type(rt_data_t), intent(inout), target :: global
   type(var_list_t), pointer :: var_list
   logical :: lval, is_known
   integer :: i
   var_list => global%get_var_list_ptr ()
   lval = eval_log (cmd%pn_if_lexpr, var_list, is_known=is_known)
   if (is_known) then
      if (lval) then
         if (associated (cmd%if_body)) then
            call cmd%if_body%execute (global)
         end if
         return
      end if
   else
      call error_undecided ()
      return
   end if
   if (associated (cmd%elsif_cmd)) then
      SCAN_ELSIF: do i = 1, size (cmd%elsif_cmd)
         lval = eval_log (cmd%elsif_cmd(i)%pn_if_lexpr, var_list, &
               is_known=is_known)
         if (is_known) then
            if (lval) then
               if (associated (cmd%elsif_cmd(i)%if_body)) then
                  call cmd%elsif_cmd(i)%if_body%execute (global)
```

```
            end if
            return
         end if
      else
         call error_undecided ()
         return
      end if
   end do SCAN_ELSIF
  end if
  if (associated (cmd%else_body)) then
     call cmd%else_body%execute (global)
  end if
contains
  subroutine error_undecided ()
    call msg_error ("Undefined result of cmditional expression: " &
         // "neither branch will be executed")
  end subroutine error_undecided
end subroutine cmd_if_execute
```

Include another command-list file

The include command allocates a local parse tree. This must not be deleted before the command object itself is deleted, since pointers may point to subobjects of it.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_include_t
   private
   type(string_t) :: file
   type(command_list_t), pointer :: command_list => null ()
   type(parse_tree_t) :: parse_tree
 contains
 ⟨Commands: cmd include: TBP⟩
end type cmd_include_t
```

Finalizer: delete the command list. No options, so we can simply override the default finalizer.

⟨*Commands: cmd include: TBP*⟩≡

```
procedure :: final => cmd_include_final
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_include_final (cmd)
  class(cmd_include_t), intent(inout) :: cmd
  call parse_tree_final (cmd%parse_tree)
  if (associated (cmd%command_list)) then
     call cmd%command_list%final ()
     deallocate (cmd%command_list)
  end if
end subroutine cmd_include_final
```

Write: display the command list as-is, if allocated.

⟨*Commands: cmd include: TBP*⟩+≡

```
procedure :: write => cmd_include_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_include_write (cmd, unit, indent)
  class(cmd_include_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u, ind
  u = given_output_unit (unit)
  ind = 0;  if (present (indent))  ind = indent
  call write_indent (u, indent)
  write (u, "(A,A,A,A)")  "include ", '"', char (cmd%file), '"'
  if (associated (cmd%command_list)) then
     call cmd%command_list%write (u, ind + 1)
  end if
end subroutine cmd_include_write
```

Compile file contents: First parse the file, then immediately compile its contents. Use the global data set.

⟨*Commands: cmd include: TBP*⟩+≡

```
procedure :: compile => cmd_include_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_include_compile (cmd, global)
  class(cmd_include_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_arg, pn_file
  type(string_t) :: file
  logical :: exist
  integer :: u
  type(stream_t), target :: stream
  type(lexer_t) :: lexer
  pn_arg => parse_node_get_sub_ptr (cmd%pn, 2)
  pn_file => parse_node_get_sub_ptr (pn_arg)
  file = parse_node_get_string (pn_file)
  inquire (file=char(file), exist=exist)
  if (exist) then
     cmd%file = file
  else
     cmd%file = global%os_data%whizard_cutspath // "/" // file
     inquire (file=char(cmd%file), exist=exist)
     if (.not. exist) then
        call msg_error ("Include file '" // char (file) // "' not found")
        return
     end if
  end if
  u = free_unit ()
  call lexer_init_cmd_list (lexer, global%lexer)
  call stream_init (stream, char (cmd%file))
  call lexer_assign_stream (lexer, stream)
  call parse_tree_init (cmd%parse_tree, syntax_cmd_list, lexer)
  call stream_final (stream)
  call lexer_final (lexer)
  close (u)
  allocate (cmd%command_list)
  call cmd%command_list%compile (cmd%parse_tree%get_root_ptr (), &
       global)
```

```
end subroutine cmd_include_compile
```

Execute file contents in the global context.

⟨*Commands: cmd include: TBP*⟩+≡

```
procedure :: execute => cmd_include_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_include_execute (cmd, global)
  class(cmd_include_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  if (associated (cmd%command_list)) then
     call msg_message &
          ("Including Sindarin from '" // char (cmd%file) // "'")
     call cmd%command_list%execute (global)
     call msg_message &
          ("End of included '" // char (cmd%file) // "'")
  end if
end subroutine cmd_include_execute
```

Quit command execution

The code is the return code of the whole program if it is terminated by this command.

⟨*Commands: types*⟩+≡

```
type, extends (command_t) :: cmd_quit_t
   private
   logical :: has_code = .false.
   type(parse_node_t), pointer :: pn_code_expr => null ()
 contains
 ⟨Commands: cmd quit: TBP⟩
end type cmd_quit_t
```

Output.

⟨*Commands: cmd quit: TBP*⟩≡

```
procedure :: write => cmd_quit_write
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_quit_write (cmd, unit, indent)
  class(cmd_quit_t), intent(in) :: cmd
  integer, intent(in), optional :: unit, indent
  integer :: u
  u = given_output_unit (unit);  if (u < 0)  return
  call write_indent (u, indent)
  write (u, "(1x,A,L1)")  "quit: has_code = ", cmd%has_code
end subroutine cmd_quit_write
```

Compile: allocate a `quit` object which serves as a placeholder.

⟨*Commands: cmd quit: TBP*⟩+≡

```
procedure :: compile => cmd_quit_compile
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_quit_compile (cmd, global)
  class(cmd_quit_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_arg
  pn_arg => parse_node_get_sub_ptr (cmd%pn, 2)
  if (associated (pn_arg)) then
     cmd%pn_code_expr => parse_node_get_sub_ptr (pn_arg)
     cmd%has_code = .true.
  end if
end subroutine cmd_quit_compile
```

Execute: The quit command does not execute anything, it just stops command execution. This is achieved by setting quit flag and quit code in the global variable list. However, the return code, if present, is an expression which has to be evaluated.

⟨*Commands: cmd quit: TBP*⟩+≡

```
procedure :: execute => cmd_quit_execute
```

⟨*Commands: procedures*⟩+≡

```
subroutine cmd_quit_execute (cmd, global)
  class(cmd_quit_t), intent(inout) :: cmd
  type(rt_data_t), intent(inout), target :: global
  type(var_list_t), pointer :: var_list
  logical :: is_known
  var_list => global%get_var_list_ptr ()
  if (cmd%has_code) then
     global%quit_code = eval_int (cmd%pn_code_expr, var_list, &
          is_known=is_known)
     if (.not. is_known) then
        call msg_error ("Undefined return code of quit/exit command")
     end if
  end if
  global%quit = .true.
end subroutine cmd_quit_execute
```

35.1.6 The command list

The command list holds a list of commands and relevant global data.

⟨*Commands: public*⟩+≡

```
public :: command_list_t
```

⟨*Commands: types*⟩+≡

```
type :: command_list_t
   ! not private anymore as required by the whizard-c-interface
   class(command_t), pointer :: first => null ()
   class(command_t), pointer :: last => null ()
 contains
 ⟨Commands: command list: TBP⟩
end type command_list_t
```

Output.

⟨*Commands: command list: TBP*⟩≡

```
procedure :: write => command_list_write
```

⟨*Commands: procedures*⟩+≡

```
recursive subroutine command_list_write (cmd_list, unit, indent)
  class(command_list_t), intent(in) :: cmd_list
  integer, intent(in), optional :: unit, indent
  class(command_t), pointer :: cmd
  cmd => cmd_list%first
  do while (associated (cmd))
     call cmd%write (unit, indent)
     cmd => cmd%next
  end do
end subroutine command_list_write
```

Append a new command to the list and free the original pointer.

⟨*Commands: command list: TBP*⟩+≡

```
procedure :: append => command_list_append
```

⟨*Commands: procedures*⟩+≡

```
subroutine command_list_append (cmd_list, command)
  class(command_list_t), intent(inout) :: cmd_list
  class(command_t), intent(inout), pointer :: command
  if (associated (cmd_list%last)) then
     cmd_list%last%next => command
  else
     cmd_list%first => command
  end if
  cmd_list%last => command
  command => null ()
end subroutine command_list_append
```

Finalize.

⟨*Commands: command list: TBP*⟩+≡

```
procedure :: final => command_list_final
```

⟨*Commands: procedures*⟩+≡

```
recursive subroutine command_list_final (cmd_list)
  class(command_list_t), intent(inout) :: cmd_list
  class(command_t), pointer :: command
  do while (associated (cmd_list%first))
     command => cmd_list%first
     cmd_list%first => cmd_list%first%next
     call command%final ()
     deallocate (command)
  end do
  cmd_list%last => null ()
end subroutine command_list_final
```

35.1.7 Compiling the parse tree

Transform a parse tree into a command list. Initialization is assumed to be done.

After each command, we set a breakpoint.

⟨*Commands: command list: TBP*⟩+≡

```
procedure :: compile => command_list_compile
```

⟨*Commands: procedures*⟩+≡

```
recursive subroutine command_list_compile (cmd_list, pn, global)
  class(command_list_t), intent(inout), target :: cmd_list
  type(parse_node_t), intent(in), target :: pn
  type(rt_data_t), intent(inout), target :: global
  type(parse_node_t), pointer :: pn_cmd
  class(command_t), pointer :: command
  integer :: i
  pn_cmd => parse_node_get_sub_ptr (pn)
  do i = 1, parse_node_get_n_sub (pn)
     call dispatch_command (command, pn_cmd)
     call command%compile (global)
     call cmd_list%append (command)
     call terminate_now_if_signal ()
     pn_cmd => parse_node_get_next_ptr (pn_cmd)
  end do
end subroutine command_list_compile
```

35.1.8 Executing the command list

Before executing a command we should execute its options (if any). After that, reset the options, i.e., remove temporary effects from the global state.

Also here, after each command we set a breakpoint.

⟨*Commands: command list: TBP*⟩+≡

```
procedure :: execute => command_list_execute
```

⟨*Commands: procedures*⟩+≡

```
recursive subroutine command_list_execute (cmd_list, global)
  class(command_list_t), intent(in) :: cmd_list
  type(rt_data_t), intent(inout), target :: global
  class(command_t), pointer :: command
  command => cmd_list%first
  COMMAND_COND: do while (associated (command))
     call command%execute_options (global)
     call command%execute (global)
     call command%reset_options (global)
     call terminate_now_if_signal ()
     if (global%quit)  exit COMMAND_COND
     command => command%next
  end do COMMAND_COND
end subroutine command_list_execute
```

35.1.9 Command list syntax

⟨*Commands: public*⟩+≡
```
public :: syntax_cmd_list
```

⟨*Commands: variables*⟩≡
```
type(syntax_t), target, save :: syntax_cmd_list
```

⟨*Commands: public*⟩+≡
```
public :: syntax_cmd_list_init
```

⟨*Commands: procedures*⟩+≡
```
subroutine syntax_cmd_list_init ()
  type(ifile_t) :: ifile
  call define_cmd_list_syntax (ifile)
  call syntax_init (syntax_cmd_list, ifile)
  call ifile_final (ifile)
end subroutine syntax_cmd_list_init
```

⟨*Commands: public*⟩+≡
```
public :: syntax_cmd_list_final
```

⟨*Commands: procedures*⟩+≡
```
subroutine syntax_cmd_list_final ()
  call syntax_final (syntax_cmd_list)
end subroutine syntax_cmd_list_final
```

⟨*Commands: public*⟩+≡
```
public :: syntax_cmd_list_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine syntax_cmd_list_write (unit)
  integer, intent(in), optional :: unit
  call syntax_write (syntax_cmd_list, unit)
end subroutine syntax_cmd_list_write
```

⟨*Commands: procedures*⟩+≡
```
subroutine define_cmd_list_syntax (ifile)
  type(ifile_t), intent(inout) :: ifile
  call ifile_append (ifile, "SEQ command_list = command*")
  call ifile_append (ifile, "ALT command = " &
       // "cmd_model | cmd_library | cmd_iterations | cmd_sample_format | " &
       // "cmd_var | cmd_slha | " &
       // "cmd_show | cmd_clear | " &
       // "cmd_expect | " &
       // "cmd_cuts | cmd_scale | cmd_fac_scale | cmd_ren_scale | " &
       // "cmd_weight | cmd_selection | cmd_reweight | " &
       // "cmd_beams | cmd_beams_pol_density | cmd_beams_pol_fraction | " &
       // "cmd_beams_momentum | cmd_beams_theta | cmd_beams_phi | " &
       // "cmd_integrate | " &
       // "cmd_observable | cmd_histogram | cmd_plot | cmd_graph | " &
       // "cmd_record | " &
       // "cmd_analysis | cmd_alt_setup | " &
       // "cmd_unstable | cmd_stable | cmd_simulate | cmd_rescan | " &
       // "cmd_process | cmd_compile | cmd_exec | " &
```

```
     // "cmd_scan | cmd_if | cmd_include | cmd_quit | " &
     // "cmd_polarized | cmd_unpolarized | " &
     // "cmd_open_out | cmd_close_out | cmd_printf | " &
     // "cmd_write_analysis | cmd_compile_analysis | cmd_nlo | cmd_components")
call ifile_append (ifile, "GRO options = '{' local_command_list '}'")
call ifile_append (ifile, "SEQ local_command_list = local_command*")
call ifile_append (ifile, "ALT local_command = " &
     // "cmd_model | cmd_library | cmd_iterations | cmd_sample_format | " &
     // "cmd_var | cmd_slha | " &
     // "cmd_show | " &
     // "cmd_expect | " &
     // "cmd_cuts | cmd_scale | cmd_fac_scale | cmd_ren_scale | " &
     // "cmd_weight | cmd_selection | cmd_reweight | " &
     // "cmd_beams | cmd_beams_pol_density | cmd_beams_pol_fraction | " &
     // "cmd_beams_momentum | cmd_beams_theta | cmd_beams_phi | " &
     // "cmd_observable | cmd_histogram | cmd_plot | cmd_graph | " &
     // "cmd_clear | cmd_record | " &
     // "cmd_analysis | cmd_alt_setup | " &
     // "cmd_open_out | cmd_close_out | cmd_printf | " &
     // "cmd_write_analysis | cmd_compile_analysis | cmd_nlo | cmd_components")
call ifile_append (ifile, "SEQ cmd_model = model '=' model_name model_arg?")
call ifile_append (ifile, "KEY model")
call ifile_append (ifile, "ALT model_name = model_id | string_literal")
call ifile_append (ifile, "IDE model_id")
call ifile_append (ifile, "ARG model_arg = ( model_scheme? )")
call ifile_append (ifile, "ALT model_scheme = scheme_id | string_literal")
call ifile_append (ifile, "IDE scheme_id")
call ifile_append (ifile, "SEQ cmd_library = library '=' lib_name")
call ifile_append (ifile, "KEY library")
call ifile_append (ifile, "ALT lib_name = lib_id | string_literal")
call ifile_append (ifile, "IDE lib_id")
call ifile_append (ifile, "ALT cmd_var = " &
     // "cmd_log_decl | cmd_log | " &
     // "cmd_int | cmd_real | cmd_complex | cmd_num | " &
     // "cmd_string_decl | cmd_string | cmd_alias | " &
     // "cmd_result")
call ifile_append (ifile, "SEQ cmd_log_decl = logical cmd_log")
call ifile_append (ifile, "SEQ cmd_log = '?' var_name '=' lexpr")
call ifile_append (ifile, "SEQ cmd_int = int var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_real = real var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_complex = complex var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_num = var_name '=' expr")
call ifile_append (ifile, "SEQ cmd_string_decl = string cmd_string")
call ifile_append (ifile, "SEQ cmd_string = " &
     // "'$' var_name '=' sexpr") ! $
call ifile_append (ifile, "SEQ cmd_alias = alias var_name '=' cexpr")
call ifile_append (ifile, "SEQ cmd_result = result '=' expr")
call ifile_append (ifile, "SEQ cmd_slha = slha_action slha_arg options?")
call ifile_append (ifile, "ALT slha_action = " &
     // "read_slha | write_slha")
call ifile_append (ifile, "KEY read_slha")
call ifile_append (ifile, "KEY write_slha")
call ifile_append (ifile, "ARG slha_arg = ( string_literal )")
call ifile_append (ifile, "SEQ cmd_show = show show_arg options?")
```

```
call ifile_append (ifile, "KEY show")
call ifile_append (ifile, "ARG show_arg = ( showable* )")
call ifile_append (ifile, "ALT showable = " &
     // "model | library | beams | iterations | " &
     // "cuts | weight | logical | string | pdg | " &
     // "scale | factorization_scale | renormalization_scale | " &
     // "selection | reweight | analysis | " &
     // "stable | unstable | polarized | unpolarized | " &
     // "expect | intrinsic | int | real | complex | " &
     // "alias_var | string | results | result_var | " &
     // "log_var | string_var | var_name")
call ifile_append (ifile, "KEY results")
call ifile_append (ifile, "KEY intrinsic")
call ifile_append (ifile, "SEQ alias_var = alias var_name")
call ifile_append (ifile, "SEQ result_var = result_key result_arg?")
call ifile_append (ifile, "SEQ log_var = '?' var_name")
call ifile_append (ifile, "SEQ string_var = '$' var_name")  ! $
call ifile_append (ifile, "SEQ cmd_clear = clear clear_arg options?")
call ifile_append (ifile, "KEY clear")
call ifile_append (ifile, "ARG clear_arg = ( clearable* )")
call ifile_append (ifile, "ALT clearable = " &
     // "beams | iterations | " &
     // "cuts | weight | " &
     // "scale | factorization_scale | renormalization_scale | " &
     // "selection | reweight | analysis | " &
     // "unstable | polarized | " &
     // "expect | " &
     // "log_var | string_var | var_name")
call ifile_append (ifile, "SEQ cmd_expect = expect expect_arg options?")
call ifile_append (ifile, "KEY expect")
call ifile_append (ifile, "ARG expect_arg = ( lexpr )")
call ifile_append (ifile, "SEQ cmd_cuts = cuts '=' lexpr")
call ifile_append (ifile, "SEQ cmd_scale = scale '=' expr")
call ifile_append (ifile, "SEQ cmd_fac_scale = " &
     // "factorization_scale '=' expr")
call ifile_append (ifile, "SEQ cmd_ren_scale = " &
     // "renormalization_scale '=' expr")
call ifile_append (ifile, "SEQ cmd_weight = weight '=' expr")
call ifile_append (ifile, "SEQ cmd_selection = selection '=' lexpr")
call ifile_append (ifile, "SEQ cmd_reweight = reweight '=' expr")
call ifile_append (ifile, "KEY cuts")
call ifile_append (ifile, "KEY scale")
call ifile_append (ifile, "KEY factorization_scale")
call ifile_append (ifile, "KEY renormalization_scale")
call ifile_append (ifile, "KEY weight")
call ifile_append (ifile, "KEY selection")
call ifile_append (ifile, "KEY reweight")
call ifile_append (ifile, "SEQ cmd_process = process process_id '=' " &
     // "process_prt '=>' prt_state_list options?")
call ifile_append (ifile, "KEY process")
call ifile_append (ifile, "KEY '=>'")
call ifile_append (ifile, "LIS process_prt = cexpr+")
call ifile_append (ifile, "LIS prt_state_list = prt_state_sum+")
call ifile_append (ifile, "SEQ prt_state_sum = " &
```

```
     // "prt_state prt_state_addition*")
call ifile_append (ifile, "SEQ prt_state_addition = '+' prt_state")
call ifile_append (ifile, "ALT prt_state = grouped_prt_state_list | cexpr")
call ifile_append (ifile, "GRO grouped_prt_state_list = " &
     // "( prt_state_list )")
call ifile_append (ifile, "SEQ cmd_compile = compile_cmd options?")
call ifile_append (ifile, "SEQ compile_cmd = compile_clause compile_arg?")
call ifile_append (ifile, "SEQ compile_clause = compile exec_name_spec?")
call ifile_append (ifile, "KEY compile")
call ifile_append (ifile, "SEQ exec_name_spec = as exec_name")
call ifile_append (ifile, "KEY as")
call ifile_append (ifile, "ALT exec_name = exec_id | string_literal")
call ifile_append (ifile, "IDE exec_id")
call ifile_append (ifile, "ARG compile_arg = ( lib_name* )")
call ifile_append (ifile, "SEQ cmd_exec = exec exec_arg")
call ifile_append (ifile, "KEY exec")
call ifile_append (ifile, "ARG exec_arg = ( sexpr )")
call ifile_append (ifile, "SEQ cmd_beams = beams '=' beam_def")
call ifile_append (ifile, "KEY beams")
call ifile_append (ifile, "SEQ beam_def = beam_spec strfun_seq*")
call ifile_append (ifile, "SEQ beam_spec = beam_list")
call ifile_append (ifile, "LIS beam_list = cexpr, cexpr?")
call ifile_append (ifile, "SEQ cmd_beams_pol_density = " &
     // "beams_pol_density '=' beams_pol_spec")
call ifile_append (ifile, "KEY beams_pol_density")
call ifile_append (ifile, "LIS beams_pol_spec = smatrix, smatrix?")
call ifile_append (ifile, "SEQ smatrix = '@' smatrix_arg")
! call ifile_append (ifile, "KEY '@'")     !!! Key already exists
call ifile_append (ifile, "ARG smatrix_arg = ( sentry* )")
call ifile_append (ifile, "SEQ sentry = expr extra_sentry*")
call ifile_append (ifile, "SEQ extra_sentry = ':' expr")
call ifile_append (ifile, "SEQ cmd_beams_pol_fraction = " &
     // "beams_pol_fraction '=' beams_par_spec")
call ifile_append (ifile, "KEY beams_pol_fraction")
call ifile_append (ifile, "SEQ cmd_beams_momentum = " &
     // "beams_momentum '=' beams_par_spec")
call ifile_append (ifile, "KEY beams_momentum")
call ifile_append (ifile, "SEQ cmd_beams_theta = " &
     // "beams_theta '=' beams_par_spec")
call ifile_append (ifile, "KEY beams_theta")
call ifile_append (ifile, "SEQ cmd_beams_phi = " &
     // "beams_phi '=' beams_par_spec")
call ifile_append (ifile, "KEY beams_phi")
call ifile_append (ifile, "LIS beams_par_spec = expr, expr?")
call ifile_append (ifile, "SEQ strfun_seq = '=>' strfun_pair")
call ifile_append (ifile, "LIS strfun_pair = strfun_def, strfun_def?")
call ifile_append (ifile, "SEQ strfun_def = strfun_id")
call ifile_append (ifile, "ALT strfun_id = " &
      // "none | lhapdf | lhapdf_photon | pdf_builtin | pdf_builtin_photon | " &
      // "isr | epa | ewa | circe1 | circe2 | energy_scan | " &
      // "gaussian | beam_events")
call ifile_append (ifile, "KEY none")
call ifile_append (ifile, "KEY lhapdf")
call ifile_append (ifile, "KEY lhapdf_photon")
```

```
call ifile_append (ifile, "KEY pdf_builtin")
call ifile_append (ifile, "KEY pdf_builtin_photon")
call ifile_append (ifile, "KEY isr")
call ifile_append (ifile, "KEY epa")
call ifile_append (ifile, "KEY ewa")
call ifile_append (ifile, "KEY circe1")
call ifile_append (ifile, "KEY circe2")
call ifile_append (ifile, "KEY energy_scan")
call ifile_append (ifile, "KEY gaussian")
call ifile_append (ifile, "KEY beam_events")
call ifile_append (ifile, "SEQ cmd_integrate = " &
     // "integrate proc_arg options?")
call ifile_append (ifile, "KEY integrate")
call ifile_append (ifile, "ARG proc_arg = ( proc_id* )")
call ifile_append (ifile, "IDE proc_id")
call ifile_append (ifile, "SEQ cmd_iterations = " &
     // "iterations '=' iterations_list")
call ifile_append (ifile, "KEY iterations")
call ifile_append (ifile, "LIS iterations_list = iterations_spec+")
call ifile_append (ifile, "ALT iterations_spec = it_spec")
call ifile_append (ifile, "SEQ it_spec = expr calls_spec adapt_spec?")
call ifile_append (ifile, "SEQ calls_spec = ':' expr")
call ifile_append (ifile, "SEQ adapt_spec = ':' sexpr")
call ifile_append (ifile, "SEQ cmd_components = " &
     // "active '=' component_list")
call ifile_append (ifile, "KEY active")
call ifile_append (ifile, "LIS component_list = sexpr+")
call ifile_append (ifile, "SEQ cmd_sample_format = " &
     // "sample_format '=' event_format_list")
call ifile_append (ifile, "KEY sample_format")
call ifile_append (ifile, "LIS event_format_list = event_format+")
call ifile_append (ifile, "IDE event_format")
call ifile_append (ifile, "SEQ cmd_observable = " &
     // "observable analysis_tag options?")
call ifile_append (ifile, "KEY observable")
call ifile_append (ifile, "SEQ cmd_histogram = " &
     // "histogram analysis_tag histogram_arg " &
     // "options?")
call ifile_append (ifile, "KEY histogram")
call ifile_append (ifile, "ARG histogram_arg = (expr, expr, expr?)")
call ifile_append (ifile, "SEQ cmd_plot = plot analysis_tag options?")
call ifile_append (ifile, "KEY plot")
call ifile_append (ifile, "SEQ cmd_graph = graph graph_term '=' graph_def")
call ifile_append (ifile, "KEY graph")
call ifile_append (ifile, "SEQ graph_term = analysis_tag options?")
call ifile_append (ifile, "SEQ graph_def = graph_term graph_append*")
call ifile_append (ifile, "SEQ graph_append = '&' graph_term")
call ifile_append (ifile, "SEQ cmd_analysis = analysis '=' lexpr")
call ifile_append (ifile, "KEY analysis")
call ifile_append (ifile, "SEQ cmd_alt_setup = " &
     // "alt_setup '=' option_list_expr")
call ifile_append (ifile, "KEY alt_setup")
call ifile_append (ifile, "ALT option_list_expr = " &
     // "grouped_option_list | option_list")
```

```
call ifile_append (ifile, "GRO grouped_option_list = ( option_list_expr )")
call ifile_append (ifile, "LIS option_list = options+")
call ifile_append (ifile, "SEQ cmd_open_out = open_out open_arg options?")
call ifile_append (ifile, "SEQ cmd_close_out = close_out open_arg options?")
call ifile_append (ifile, "KEY open_out")
call ifile_append (ifile, "KEY close_out")
call ifile_append (ifile, "ARG open_arg = (sexpr)")
call ifile_append (ifile, "SEQ cmd_printf = printf_cmd options?")
call ifile_append (ifile, "SEQ printf_cmd = printf_clause sprintf_args?")
call ifile_append (ifile, "SEQ printf_clause = printf sexpr")
call ifile_append (ifile, "KEY printf")
call ifile_append (ifile, "SEQ cmd_record = record_cmd")
call ifile_append (ifile, "SEQ cmd_unstable = " &
     // "unstable cexpr unstable_arg options?")
call ifile_append (ifile, "KEY unstable")
call ifile_append (ifile, "ARG unstable_arg = ( proc_id* )")
call ifile_append (ifile, "SEQ cmd_stable = stable stable_list options?")
call ifile_append (ifile, "KEY stable")
call ifile_append (ifile, "LIS stable_list = cexpr+")
call ifile_append (ifile, "KEY polarized")
call ifile_append (ifile, "SEQ cmd_polarized = polarized polarized_list options?")
call ifile_append (ifile, "LIS polarized_list = cexpr+")
call ifile_append (ifile, "KEY unpolarized")
call ifile_append (ifile, "SEQ cmd_unpolarized = unpolarized unpolarized_list options?")
call ifile_append (ifile, "LIS unpolarized_list = cexpr+")
call ifile_append (ifile, "SEQ cmd_simulate = " &
     // "simulate proc_arg options?")
call ifile_append (ifile, "KEY simulate")
call ifile_append (ifile, "SEQ cmd_rescan = " &
     // "rescan sexpr proc_arg options?")
call ifile_append (ifile, "KEY rescan")
call ifile_append (ifile, "SEQ cmd_scan = scan scan_var scan_body?")
call ifile_append (ifile, "KEY scan")
call ifile_append (ifile, "ALT scan_var = " &
     // "scan_log_decl | scan_log | " &
     // "scan_int | scan_real | scan_complex | scan_num | " &
     // "scan_string_decl | scan_string | scan_alias | " &
     // "scan_cuts | scan_weight | " &
     // "scan_scale | scan_ren_scale | scan_fac_scale | " &
     // "scan_selection | scan_reweight | scan_analysis | " &
     // "scan_model | scan_library")
call ifile_append (ifile, "SEQ scan_log_decl = logical scan_log")
call ifile_append (ifile, "SEQ scan_log = '?' var_name '=' scan_log_arg")
call ifile_append (ifile, "ARG scan_log_arg = ( lexpr* )")
call ifile_append (ifile, "SEQ scan_int = int var_name '=' scan_num_arg")
call ifile_append (ifile, "SEQ scan_real = real var_name '=' scan_num_arg")
call ifile_append (ifile, "SEQ scan_complex = " &
     // "complex var_name '=' scan_num_arg")
call ifile_append (ifile, "SEQ scan_num = var_name '=' scan_num_arg")
call ifile_append (ifile, "ARG scan_num_arg = ( range* )")
call ifile_append (ifile, "ALT range = grouped_range | range_expr")
call ifile_append (ifile, "GRO grouped_range = ( range_expr )")
call ifile_append (ifile, "SEQ range_expr = expr range_spec?")
call ifile_append (ifile, "SEQ range_spec = '=>' expr step_spec?")
```

```
call ifile_append (ifile, "SEQ step_spec = step_op expr")
call ifile_append (ifile, "ALT step_op = " &
     // "'/+' | '/-' | '/*' | '//' | '/+/' | '/*/'")
call ifile_append (ifile, "KEY '/+'")
call ifile_append (ifile, "KEY '/-'")
call ifile_append (ifile, "KEY '/*'")
call ifile_append (ifile, "KEY '//'")
call ifile_append (ifile, "KEY '/+/'")
call ifile_append (ifile, "KEY '/*/'")
call ifile_append (ifile, "SEQ scan_string_decl = string scan_string")
call ifile_append (ifile, "SEQ scan_string = " &
     // "'$' var_name '=' scan_string_arg")
call ifile_append (ifile, "ARG scan_string_arg = ( sexpr* )")
call ifile_append (ifile, "SEQ scan_alias = " &
     // "alias var_name '=' scan_alias_arg")
call ifile_append (ifile, "ARG scan_alias_arg = ( cexpr* )")
call ifile_append (ifile, "SEQ scan_cuts = cuts '=' scan_lexpr_arg")
call ifile_append (ifile, "ARG scan_lexpr_arg = ( lexpr* )")
call ifile_append (ifile, "SEQ scan_scale = scale '=' scan_expr_arg")
call ifile_append (ifile, "ARG scan_expr_arg = ( expr* )")
call ifile_append (ifile, "SEQ scan_fac_scale = " &
     // "factorization_scale '=' scan_expr_arg")
call ifile_append (ifile, "SEQ scan_ren_scale = " &
     // "renormalization_scale '=' scan_expr_arg")
call ifile_append (ifile, "SEQ scan_weight = weight '=' scan_expr_arg")
call ifile_append (ifile, "SEQ scan_selection = selection '=' scan_lexpr_arg")
call ifile_append (ifile, "SEQ scan_reweight = reweight '=' scan_expr_arg")
call ifile_append (ifile, "SEQ scan_analysis = analysis '=' scan_lexpr_arg")
call ifile_append (ifile, "SEQ scan_model = model '=' scan_model_arg")
call ifile_append (ifile, "ARG scan_model_arg = ( model_name* )")
call ifile_append (ifile, "SEQ scan_library = library '=' scan_library_arg")
call ifile_append (ifile, "ARG scan_library_arg = ( lib_name* )")
call ifile_append (ifile, "GRO scan_body = '{' command_list '}'")
call ifile_append (ifile, "SEQ cmd_if = " &
     // "if lexpr then command_list elsif_clauses else_clause endif")
call ifile_append (ifile, "SEQ elsif_clauses = cmd_elsif*")
call ifile_append (ifile, "SEQ cmd_elsif = elsif lexpr then command_list")
call ifile_append (ifile, "SEQ else_clause = cmd_else?")
call ifile_append (ifile, "SEQ cmd_else = else command_list")
call ifile_append (ifile, "SEQ cmd_include = include include_arg")
call ifile_append (ifile, "KEY include")
call ifile_append (ifile, "ARG include_arg = ( string_literal )")
call ifile_append (ifile, "SEQ cmd_quit = quit_cmd quit_arg?")
call ifile_append (ifile, "ALT quit_cmd = quit | exit")
call ifile_append (ifile, "KEY quit")
call ifile_append (ifile, "KEY exit")
call ifile_append (ifile, "ARG quit_arg = ( expr )")
call ifile_append (ifile, "SEQ cmd_write_analysis = " &
     // "write_analysis_clause options?")
call ifile_append (ifile, "SEQ cmd_compile_analysis = " &
     // "compile_analysis_clause options?")
call ifile_append (ifile, "SEQ write_analysis_clause = " &
     // "write_analysis write_analysis_arg?")
call ifile_append (ifile, "SEQ compile_analysis_clause = " &
```

```
          // "compile_analysis write_analysis_arg?")
      call ifile_append (ifile, "KEY write_analysis")
      call ifile_append (ifile, "KEY compile_analysis")
      call ifile_append (ifile, "ARG write_analysis_arg = ( analysis_tag* )")
      call ifile_append (ifile, "SEQ cmd_nlo = " &
                         // "nlo_calculation '=' nlo_calculation_list")
      call ifile_append (ifile, "KEY nlo_calculation")
      call ifile_append (ifile, "LIS nlo_calculation_list = sexpr+")
      call define_expr_syntax (ifile, particles=.true., analysis=.true.)
    end subroutine define_cmd_list_syntax
```

⟨*Commands: public*⟩+≡

```
    public :: lexer_init_cmd_list
```

⟨*Commands: procedures*⟩+≡

```
    subroutine lexer_init_cmd_list (lexer, parent_lexer)
      type(lexer_t), intent(out) :: lexer
      type(lexer_t), intent(in), optional, target :: parent_lexer
      call lexer_init (lexer, &
           comment_chars = "#!", &
           quote_chars = '"', &
           quote_match = '"', &
           single_chars = "()[]{},;:&%?$@", &
           special_class = [ "+-*/^", "<>=~ " ] , &
           keyword_list = syntax_get_keyword_list_ptr (syntax_cmd_list), &
           parent = parent_lexer)
    end subroutine lexer_init_cmd_list
```

35.1.10 Unit Tests

Test module, followed by the corresponding implementation module.

⟨`commands_ut.f90`⟩≡

```
  ⟨File header⟩

  module commands_ut
    use unit_tests
    use commands_uti

  ⟨Standard module head⟩

  ⟨Commands: public test⟩

  contains

  ⟨Commands: test driver⟩

  end module commands_ut
```

⟨`commands_uti.f90`⟩≡

```
  ⟨File header⟩

  module commands_uti
```

```
  ⟨Use kinds⟩
    use kinds, only: i64
  ⟨Use strings⟩
    use io_units
    use ifiles
    use parser
    use interactions, only: reset_interaction_counter
    use prclib_stacks
    use analysis
    use variables, only: var_list_t
    use models
    use slha_interface
    use rt_data
    use event_base, only: generic_event_t, event_callback_t

    use commands

⟨Standard module head⟩

⟨Commands: test declarations⟩

⟨Commands: test auxiliary types⟩

contains

⟨Commands: tests⟩

⟨Commands: test auxiliary⟩

end module commands_uti
```

API: driver for the unit tests below.

⟨*Commands: public test*⟩≡
```
  public :: commands_test
```

⟨*Commands: test driver*⟩≡
```
  subroutine commands_test (u, results)
    integer, intent(in) :: u
    type(test_results_t), intent(inout) :: results
  ⟨Commands: execute tests⟩
  end subroutine commands_test
```

Prepare Sindarin code

This routine parses an internal file, prints the parse tree, and returns a parse node to the root. We use the routine in the tests below.

⟨*Commands: public test auxiliary*⟩≡
```
  public :: parse_ifile
```

⟨*Commands: test auxiliary*⟩≡
```
  subroutine parse_ifile (ifile, pn_root, u)
    use ifiles
    use lexers
```

```
    use parser
    use commands
    type(ifile_t), intent(in) :: ifile
    type(parse_node_t), pointer, intent(out) :: pn_root
    integer, intent(in), optional :: u
    type(stream_t), target :: stream
    type(lexer_t), target :: lexer
    type(parse_tree_t) :: parse_tree

    call lexer_init_cmd_list (lexer)
    call stream_init (stream, ifile)
    call lexer_assign_stream (lexer, stream)

    call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
    if (present (u))  call parse_tree_write (parse_tree, u)
    pn_root => parse_tree%get_root_ptr ()

    call stream_final (stream)
    call lexer_final (lexer)
  end subroutine parse_ifile
```

Empty command list

Compile and execute an empty command list. Should do nothing but test the integrity of the workflow.

⟨*Commands: execute tests*⟩≡

```
  call test (commands_1, "commands_1", &
       "empty command list", &
       u, results)
```

⟨*Commands: test declarations*⟩≡

```
  public :: commands_1
```

⟨*Commands: tests*⟩≡

```
  subroutine commands_1 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root

    write (u, "(A)")  "* Test output: commands_1"
    write (u, "(A)")  "*   Purpose: compile and execute empty command list"
    write (u, "(A)")

    write (u, "(A)")  "*  Initialization"
    write (u, "(A)")

    call syntax_cmd_list_init ()
    call global%global_init ()

    write (u, "(A)")  "*  Parse empty file"
    write (u, "(A)")
```

```
    call parse_ifile (ifile, pn_root, u)

    write (u, "(A)")
    write (u, "(A)")  "* Compile command list"

    if (associated (pn_root)) then
       call command_list%compile (pn_root, global)
    end if

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"

    call global%activate ()
    call command_list%execute (global)
    call global%deactivate ()

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call syntax_cmd_list_final ()
    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_1"

  end subroutine commands_1
```

Read model

Execute a `model` assignment.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_2, "commands_2", &
       "model", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_2
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_2 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root

    write (u, "(A)")  "* Test output: commands_2"
    write (u, "(A)")  "*   Purpose: set model"
    write (u, "(A)")
```

```
    write (u, "(A)")  "*  Initialization"
    write (u, "(A)")

    call syntax_cmd_list_init ()
    call syntax_model_file_init ()
    call global%global_init ()

    write (u, "(A)")  "*  Input file"
    write (u, "(A)")

    call ifile_append (ifile, 'model = "Test"')

    call ifile_write (ifile, u)

    write (u, "(A)")  "*  Parse file"
    write (u, "(A)")

    call parse_ifile (ifile, pn_root, u)

    write (u, "(A)")
    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)
    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_2"

  end subroutine commands_2
```

Declare Process

Read a model, then declare a process. The process library is allocated explicitly. For the process definition, We take the default (`omega`) method. Since we do not compile, `O'MEGA` is not actually called.

⟨*Commands: execute tests*⟩+≡

```
    call test (commands_3, "commands_3", &
         "process declaration", &
         u, results)
```

⟨*Commands: test declarations*⟩+≡

```
    public :: commands_3
```

⟨*Commands: tests*⟩+≡

```
    subroutine commands_3 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root
      type(prclib_entry_t), pointer :: lib

      write (u, "(A)")  "* Test output: commands_3"
      write (u, "(A)")  "*   Purpose: define process"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call syntax_model_file_init ()
      call global%global_init ()
      call global%var_list%set_log (var_str ("?omega_openmp"), &
           .false., is_known = .true.)

      allocate (lib)
      call lib%init (var_str ("lib_cmd3"))
      call global%add_prclib (lib)

      write (u, "(A)")  "*  Input file"
      write (u, "(A)")

      call ifile_append (ifile, 'model = "Test"')
      call ifile_append (ifile, 'process t3 = s, s => s, s')

      call ifile_write (ifile, u)

      write (u, "(A)")
      write (u, "(A)")  "*  Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root, u)

      write (u, "(A)")
      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")

      call command_list%compile (pn_root, global)
      call command_list%write (u)

      write (u, "(A)")
```

```
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    call global%prclib_stack%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_3"

  end subroutine commands_3
```

Compile Process

Read a model, then declare a process and compile the library. The process library is allocated explicitly. For the process definition, We take the default (`unit_test`) method. There is no external code, so compilation of the library is merely a formal status change.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_4, "commands_4", &
       "compilation", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_4
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_4 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    type(prclib_entry_t), pointer :: lib

    write (u, "(A)")  "* Test output: commands_4"
    write (u, "(A)")  "*   Purpose: define process and compile library"
    write (u, "(A)")

    write (u, "(A)")  "*  Initialization"
    write (u, "(A)")

    call syntax_cmd_list_init ()
```

```
  call syntax_model_file_init ()
  call global%global_init ()
  call global%var_list%set_string (var_str ("$method"), &
       var_str ("unit_test"), is_known=.true.)

  allocate (lib)
  call lib%init (var_str ("lib_cmd4"))
  call global%add_prclib (lib)

  write (u, "(A)")  "*  Input file"
  write (u, "(A)")

  call ifile_append (ifile, 'model = "Test"')
  call ifile_append (ifile, 'process t4 = s, s => s, s')
  call ifile_append (ifile, 'compile ("lib_cmd4")')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "*  Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root, u)

  write (u, "(A)")
  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%compile (pn_root, global)
  call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
  write (u, "(A)")

  call command_list%execute (global)

  call global%prclib_stack%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call ifile_final (ifile)

  call command_list%final ()
  call global%final ()
  call syntax_cmd_list_final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: commands_4"

end subroutine commands_4
```

Integrate Process

Read a model, then declare a process, compile the library, and integrate over phase space. We take the default (`unit_test`) method and use the simplest methods of phase-space parameterization and integration.

⟨*Commands: execute tests*⟩+≡
```
    call test (commands_5, "commands_5", &
         "integration", &
         u, results)
```

⟨*Commands: test declarations*⟩+≡
```
    public :: commands_5
```

⟨*Commands: tests*⟩+≡
```
    subroutine commands_5 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root
      type(prclib_entry_t), pointer :: lib

      write (u, "(A)")  "* Test output: commands_5"
      write (u, "(A)")  "*   Purpose: define process, iterations, and integrate"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call syntax_model_file_init ()
      call global%global_init ()
      call global%var_list%set_string (var_str ("$method"), &
           var_str ("unit_test"), is_known=.true.)
      call global%var_list%set_string (var_str ("$phs_method"), &
           var_str ("single"), is_known=.true.)
      call global%var_list%set_string (var_str ("$integration_method"),&
           var_str ("midpoint"), is_known=.true.)
      call global%var_list%set_log (var_str ("?vis_history"),&
           .false., is_known=.true.)
      call global%var_list%set_log (var_str ("?integration_timer"),&
           .false., is_known = .true.)
      call global%var_list%set_real (var_str ("sqrts"), &
           1000._default, is_known=.true.)
      call global%var_list%set_int (var_str ("seed"), 0, is_known=.true.)

      allocate (lib)
      call lib%init (var_str ("lib_cmd5"))
      call global%add_prclib (lib)

      write (u, "(A)")  "*  Input file"
      write (u, "(A)")

      call ifile_append (ifile, 'model = "Test"')
      call ifile_append (ifile, 'process t5 = s, s => s, s')
```

```
    call ifile_append (ifile, 'compile')
    call ifile_append (ifile, 'iterations = 1:1000')
    call ifile_append (ifile, 'integrate (t5)')

    call ifile_write (ifile, u)

    write (u, "(A)")
    write (u, "(A)")  "*  Parse file"
    write (u, "(A)")

    call parse_ifile (ifile, pn_root, u)

    write (u, "(A)")
    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)
    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call reset_interaction_counter ()
    call command_list%execute (global)

    call global%it_list%write (u)
    write (u, "(A)")
    call global%process_stack%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_5"

  end subroutine commands_5
```

Variables

Set intrinsic and user-defined variables.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_6, "commands_6", &
      "variables", &
      u, results)
```

⟨*Commands: test declarations*⟩+≡

```
public :: commands_6
```

⟨*Commands: tests*⟩+≡

```
subroutine commands_6 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(command_list_t), target :: command_list
  type(rt_data_t), target :: global
  type(parse_node_t), pointer :: pn_root

  write (u, "(A)")  "* Test output: commands_6"
  write (u, "(A)")  "*   Purpose: define and set variables"
  write (u, "(A)")

  write (u, "(A)")  "*  Initialization"
  write (u, "(A)")

  call syntax_cmd_list_init ()
  call global%global_init ()
  call global%write_vars (u, [ &
       var_str ("$run_id"), &
       var_str ("?unweighted"), &
       var_str ("sqrts")])

  write (u, "(A)")
  write (u, "(A)")  "*  Input file"
  write (u, "(A)")

  call ifile_append (ifile, '$run_id = "run1"')
  call ifile_append (ifile, '?unweighted = false')
  call ifile_append (ifile, 'sqrts = 1000')
  call ifile_append (ifile, 'int j = 10')
  call ifile_append (ifile, 'real x = 1000.')
  call ifile_append (ifile, 'complex z = 5')
  call ifile_append (ifile, 'string $text = "abcd"')
  call ifile_append (ifile, 'logical ?flag = true')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "*  Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root, u)

  write (u, "(A)")
  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%compile (pn_root, global)
  call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
```

```
    write (u, "(A)")

    call command_list%execute (global)

    call global%write_vars (u, [ &
         var_str ("$run_id"), &
         var_str ("?unweighted"), &
         var_str ("sqrts"), &
         var_str ("j"), &
         var_str ("x"), &
         var_str ("z"), &
         var_str ("$text"), &
         var_str ("?flag")])


    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call syntax_cmd_list_final ()
    call global%final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_6"

  end subroutine commands_6
```

Process library

Open process libraries explicitly.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_7, "commands_7", &
       "process library", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_7
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_7 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root

    write (u, "(A)")  "* Test output: commands_7"
    write (u, "(A)")  "*   Purpose: declare process libraries"
    write (u, "(A)")

    write (u, "(A)")  "*  Initialization"
    write (u, "(A)")
```

```
  call syntax_cmd_list_init ()
  call global%global_init ()
  call global%var_list%set_log (var_str ("?omega_openmp"), &
       .false., is_known = .true.)
  global%os_data%fc = "Fortran-compiler"
  global%os_data%fcflags = "Fortran-flags"

  write (u, "(A)")
  write (u, "(A)")  "*  Input file"
  write (u, "(A)")

  call ifile_append (ifile, 'library = "lib_cmd7_1"')
  call ifile_append (ifile, 'library = "lib_cmd7_2"')
  call ifile_append (ifile, 'library = "lib_cmd7_1"')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "*  Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root, u)

  write (u, "(A)")
  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%compile (pn_root, global)
  call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
  write (u, "(A)")

  call command_list%execute (global)

  call global%write_libraries (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call ifile_final (ifile)

  call command_list%final ()
  call syntax_cmd_list_final ()
  call global%final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: commands_7"

end subroutine commands_7
```

Generate events

Read a model, then declare a process, compile the library, and generate weighted events. We take the default (`unit_test`) method and use the simplest methods of phase-space parameterization and integration.

⟨*Commands: execute tests*⟩+≡
```
call test (commands_8, "commands_8", &
     "event generation", &
     u, results)
```

⟨*Commands: test declarations*⟩+≡
```
public :: commands_8
```

⟨*Commands: tests*⟩+≡
```
subroutine commands_8 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(command_list_t), target :: command_list
  type(rt_data_t), target :: global
  type(parse_node_t), pointer :: pn_root
  type(prclib_entry_t), pointer :: lib

  write (u, "(A)")  "* Test output: commands_8"
  write (u, "(A)")  "*   Purpose: define process, integrate, generate events"
  write (u, "(A)")

  write (u, "(A)")  "*  Initialization"
  write (u, "(A)")

  call syntax_cmd_list_init ()
  call syntax_model_file_init ()
  call global%global_init ()
  call global%init_fallback_model &
       (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"))

  call global%var_list%set_string (var_str ("$method"), &
       var_str ("unit_test"), is_known=.true.)
  call global%var_list%set_string (var_str ("$phs_method"), &
       var_str ("single"), is_known=.true.)
  call global%var_list%set_string (var_str ("$integration_method"),&
       var_str ("midpoint"), is_known=.true.)
  call global%var_list%set_log (var_str ("?vis_history"),&
       .false., is_known=.true.)
  call global%var_list%set_log (var_str ("?integration_timer"),&
       .false., is_known = .true.)
  call global%var_list%set_real (var_str ("sqrts"), &
       1000._default, is_known=.true.)

  allocate (lib)
  call lib%init (var_str ("lib_cmd8"))
  call global%add_prclib (lib)

  write (u, "(A)")  "*  Input file"
  write (u, "(A)")
```

```
    call ifile_append (ifile, 'model = "Test"')
    call ifile_append (ifile, 'process commands_8_p = s, s => s, s')
    call ifile_append (ifile, 'compile')
    call ifile_append (ifile, 'iterations = 1:1000')
    call ifile_append (ifile, 'integrate (commands_8_p)')
    call ifile_append (ifile, '?unweighted = false')
    call ifile_append (ifile, 'n_events = 3')
    call ifile_append (ifile, '?read_raw = false')
    call ifile_append (ifile, 'simulate (commands_8_p)')

    call ifile_write (ifile, u)

    write (u, "(A)")
    write (u, "(A)")  "*  Parse file"
    write (u, "(A)")

    call parse_ifile (ifile, pn_root, u)

    write (u, "(A)")
    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)
    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"

    call command_list%execute (global)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_8"

  end subroutine commands_8
```

Define cuts

Declare a cut expression.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_9, "commands_9", &
      "cuts", &
      u, results)
```

⟨*Commands: test declarations*⟩+≡

```
public :: commands_9
```

⟨*Commands: tests*⟩+≡

```
subroutine commands_9 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(command_list_t), target :: command_list
  type(rt_data_t), target :: global
  type(parse_node_t), pointer :: pn_root
  type(string_t), dimension(0) :: no_vars

  write (u, "(A)")  "* Test output: commands_9"
  write (u, "(A)")  "*   Purpose: define cuts"
  write (u, "(A)")

  write (u, "(A)")  "*  Initialization"
  write (u, "(A)")

  call syntax_cmd_list_init ()
  call global%global_init ()

  write (u, "(A)")  "*  Input file"
  write (u, "(A)")

  call ifile_append (ifile, 'cuts = all Pt > 0 [particle]')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "*  Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root, u)

  write (u, "(A)")
  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%compile (pn_root, global)
  call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
  write (u, "(A)")

  call command_list%execute (global)

  call global%write (u, vars = no_vars)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call ifile_final (ifile)
```

```
    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_9"

  end subroutine commands_9
```

Beams

Define beam setup.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_10, "commands_10", &
       "beams", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_10
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_10 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root

    write (u, "(A)")  "* Test output: commands_10"
    write (u, "(A)")  "*   Purpose: define beams"
    write (u, "(A)")

    write (u, "(A)")  "*  Initialization"
    write (u, "(A)")

    call syntax_cmd_list_init ()
    call syntax_model_file_init ()
    call global%global_init ()

    write (u, "(A)")  "*  Input file"
    write (u, "(A)")

    call ifile_append (ifile, 'model = QCD')
    call ifile_append (ifile, 'sqrts = 1000')
    call ifile_append (ifile, 'beams = p, p')

    call ifile_write (ifile, u)

    write (u, "(A)")
    write (u, "(A)")  "*  Parse file"
    write (u, "(A)")

    call parse_ifile (ifile, pn_root, u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)
    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    call global%write_beams (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_10"

  end subroutine commands_10
```

Structure functions

Define beam setup with structure functions

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_11, "commands_11", &
       "structure functions", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_11
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_11 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root

    write (u, "(A)")  "* Test output: commands_11"
    write (u, "(A)")  "*   Purpose: define beams with structure functions"
    write (u, "(A)")
```

```
  write (u, "(A)")  "*  Initialization"
  write (u, "(A)")

  call syntax_cmd_list_init ()
  call syntax_model_file_init ()
  call global%global_init ()

  write (u, "(A)")  "*  Input file"
  write (u, "(A)")

  call ifile_append (ifile, 'model = QCD')
  call ifile_append (ifile, 'sqrts = 1100')
  call ifile_append (ifile, 'beams = p, p => lhapdf => pdf_builtin, isr')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "*  Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root, u)

  write (u, "(A)")
  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%compile (pn_root, global)
  call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
  write (u, "(A)")

  call command_list%execute (global)

  call global%write_beams (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call ifile_final (ifile)

  call command_list%final ()
  call global%final ()
  call syntax_cmd_list_final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: commands_11"

end subroutine commands_11
```

Rescan events

Read a model, then declare a process, compile the library, and generate weighted events. We take the default (`unit_test`) method and use the simplest methods of phase-space parameterization and integration. Then, rescan the generated event sample.

⟨*Commands: execute tests*⟩+≡
```
call test (commands_12, "commands_12", &
     "event rescanning", &
     u, results)
```

⟨*Commands: test declarations*⟩+≡
```
public :: commands_12
```

⟨*Commands: tests*⟩+≡
```
subroutine commands_12 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(command_list_t), target :: command_list
  type(rt_data_t), target :: global
  type(parse_node_t), pointer :: pn_root
  type(prclib_entry_t), pointer :: lib

  write (u, "(A)")  "* Test output: commands_12"
  write (u, "(A)")  "*   Purpose: generate events and rescan"
  write (u, "(A)")

  write (u, "(A)")  "*  Initialization"
  write (u, "(A)")

  call syntax_cmd_list_init ()
  call syntax_model_file_init ()

  call global%global_init ()
  call global%var_list%append_log (&
       var_str ("?rebuild_phase_space"), .false., &
       intrinsic=.true.)
  call global%var_list%append_log (&
       var_str ("?rebuild_grids"), .false., &
       intrinsic=.true.)
  call global%init_fallback_model &
       (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"))

  call global%var_list%set_string (var_str ("$method"), &
       var_str ("unit_test"), is_known=.true.)
  call global%var_list%set_string (var_str ("$phs_method"), &
       var_str ("single"), is_known=.true.)
  call global%var_list%set_string (var_str ("$integration_method"),&
       var_str ("midpoint"), is_known=.true.)
  call global%var_list%set_log (var_str ("?vis_history"),&
       .false., is_known=.true.)
  call global%var_list%set_log (var_str ("?integration_timer"),&
       .false., is_known = .true.)
  call global%var_list%set_real (var_str ("sqrts"), &
       1000._default, is_known=.true.)
```

```
  allocate (lib)
  call lib%init (var_str ("lib_cmd12"))
  call global%add_prclib (lib)

  write (u, "(A)")  "*  Input file"
  write (u, "(A)")

  call ifile_append (ifile, 'model = "Test"')
  call ifile_append (ifile, 'process commands_12_p = s, s => s, s')
  call ifile_append (ifile, 'compile')
  call ifile_append (ifile, 'iterations = 1:1000')
  call ifile_append (ifile, 'integrate (commands_12_p)')
  call ifile_append (ifile, '?unweighted = false')
  call ifile_append (ifile, 'n_events = 3')
  call ifile_append (ifile, '?read_raw = false')
  call ifile_append (ifile, 'simulate (commands_12_p)')
  call ifile_append (ifile, '?write_raw = false')
  call ifile_append (ifile, 'rescan "commands_12_p" (commands_12_p)')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "*  Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root, u)

  write (u, "(A)")
  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%compile (pn_root, global)
  call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"

  call command_list%execute (global)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call ifile_final (ifile)

  call command_list%final ()
  call global%final ()
  call syntax_cmd_list_final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: commands_12"

end subroutine commands_12
```

Event Files

Set output formats for event files.

⟨*Commands: execute tests*⟩+≡

```
    call test (commands_13, "commands_13", &
         "event output formats", &
         u, results)
```

⟨*Commands: test declarations*⟩+≡

```
    public :: commands_13
```

⟨*Commands: tests*⟩+≡

```
    subroutine commands_13 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root
      type(prclib_entry_t), pointer :: lib
      logical :: exist

      write (u, "(A)")  "* Test output: commands_13"
      write (u, "(A)")  "*   Purpose: generate events and rescan"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call syntax_model_file_init ()
      call global%global_init ()
      call global%init_fallback_model &
           (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"))

      call global%var_list%set_string (var_str ("$method"), &
           var_str ("unit_test"), is_known=.true.)
      call global%var_list%set_string (var_str ("$phs_method"), &
           var_str ("single"), is_known=.true.)
      call global%var_list%set_string (var_str ("$integration_method"),&
           var_str ("midpoint"), is_known=.true.)
      call global%var_list%set_real (var_str ("sqrts"), &
           1000._default, is_known=.true.)
      call global%var_list%set_log (var_str ("?vis_history"),&
           .false., is_known=.true.)
      call global%var_list%set_log (var_str ("?integration_timer"),&
           .false., is_known = .true.)

      allocate (lib)
      call lib%init (var_str ("lib_cmd13"))
      call global%add_prclib (lib)

      write (u, "(A)")  "*  Input file"
      write (u, "(A)")
```

```
call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'process commands_13_p = s, s => s, s')
call ifile_append (ifile, 'compile')
call ifile_append (ifile, 'iterations = 1:1000')
call ifile_append (ifile, 'integrate (commands_13_p)')
call ifile_append (ifile, '?unweighted = false')
call ifile_append (ifile, 'n_events = 1')
call ifile_append (ifile, '?read_raw = false')
call ifile_append (ifile, 'sample_format = weight_stream')
call ifile_append (ifile, 'simulate (commands_13_p)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "*  Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root, u)

write (u, "(A)")
write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"

call command_list%execute (global)

write (u, "(A)")
write (u, "(A)")  "* Verify output files"
write (u, "(A)")

inquire (file = "commands_13_p.evx", exist = exist)
if (exist)  write (u, "(1x,A)")  "raw"

inquire (file = "commands_13_p.weights.dat", exist = exist)
if (exist)  write (u, "(1x,A)")  "weight_stream"

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call global%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_13"
```

```
end subroutine commands_13
```

Compile Empty Libraries

(This is a regression test:) Declare two empty libraries and compile them.

⟨*Commands: execute tests*⟩+≡

```
call test (commands_14, "commands_14", &
     "empty libraries", &
     u, results)
```

⟨*Commands: test declarations*⟩+≡

```
public :: commands_14
```

⟨*Commands: tests*⟩+≡

```
subroutine commands_14 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(command_list_t), target :: command_list
  type(rt_data_t), target :: global
  type(parse_node_t), pointer :: pn_root

  write (u, "(A)")  "* Test output: commands_14"
  write (u, "(A)")  "*   Purpose: define and compile empty libraries"
  write (u, "(A)")

  write (u, "(A)")  "* Initialization"
  write (u, "(A)")

  call syntax_model_file_init ()
  call syntax_cmd_list_init ()

  call global%global_init ()

  write (u, "(A)")  "* Input file"
  write (u, "(A)")

  call ifile_append (ifile, 'model = "Test"')
  call ifile_append (ifile, 'library = "lib1"')
  call ifile_append (ifile, 'library = "lib2"')
  call ifile_append (ifile, 'compile ()')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "* Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root)

  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%compile (pn_root, global)
```

```
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    call global%prclib_stack%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()

    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_14"

  end subroutine commands_14
```

Compile Process

Read a model, then declare a process and compile the library. The process library is allocated explicitly. For the process definition, We take the default (`unit_test`) method. There is no external code, so compilation of the library is merely a formal status change.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_15, "commands_15", &
       "compilation", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_15
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_15 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    type(prclib_entry_t), pointer :: lib

    write (u, "(A)")  "* Test output: commands_15"
    write (u, "(A)")  "*   Purpose: define process and compile library"
    write (u, "(A)")

    write (u, "(A)")  "* Initialization"
    write (u, "(A)")
```

```
call syntax_cmd_list_init ()
call syntax_model_file_init ()
call global%global_init ()
call global%var_list%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known=.true.)
call global%var_list%set_string (var_str ("$phs_method"), &
     var_str ("single"), is_known=.true.)
call global%var_list%set_string (var_str ("$integration_method"),&
     var_str ("midpoint"), is_known=.true.)
call global%var_list%set_real (var_str ("sqrts"), &
     1000._default, is_known=.true.)
call global%var_list%set_log (var_str ("?vis_history"),&
     .false., is_known=.true.)
call global%var_list%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)

allocate (lib)
call lib%init (var_str ("lib_cmd15"))
call global%add_prclib (lib)

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'process t15 = s, s => s, s')
call ifile_append (ifile, 'iterations = 1:1000')
call ifile_append (ifile, 'integrate (t15)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)

write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

call global%prclib_stack%write (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)
```

```
    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_15"

  end subroutine commands_15
```

Observable

Declare an observable, fill it and display.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_16, "commands_16", &
       "observables", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_16
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_16 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root

    write (u, "(A)")  "* Test output: commands_16"
    write (u, "(A)")  "*   Purpose: declare an observable"
    write (u, "(A)")

    write (u, "(A)")  "* Initialization"
    write (u, "(A)")

    call syntax_cmd_list_init ()
    call global%global_init ()

    write (u, "(A)")  "* Input file"
    write (u, "(A)")

    call ifile_append (ifile, '$obs_label = "foo"')
    call ifile_append (ifile, '$obs_unit = "cm"')
    call ifile_append (ifile, '$title = "Observable foo"')
    call ifile_append (ifile, '$description = "This is observable foo"')
    call ifile_append (ifile, 'observable foo')

    call ifile_write (ifile, u)

    write (u, "(A)")
    write (u, "(A)")  "* Parse file"
    write (u, "(A)")
```

```
    call parse_ifile (ifile, pn_root)

    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)

    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    write (u, "(A)")  "* Record two data items"
    write (u, "(A)")

    call analysis_record_data (var_str ("foo"), 1._default)
    call analysis_record_data (var_str ("foo"), 3._default)

    write (u, "(A)")  "* Display analysis store"
    write (u, "(A)")

    call analysis_write (u, verbose=.true.)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call analysis_final ()
    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_16"

  end subroutine commands_16
```

Histogram

Declare a histogram, fill it and display.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_17, "commands_17", &
      "histograms", &
      u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_17
```

⟨*Commands: tests*⟩+≡

```
subroutine commands_17 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(command_list_t), target :: command_list
  type(rt_data_t), target :: global
  type(parse_node_t), pointer :: pn_root
  type(string_t), dimension(3) :: name
  integer :: i

  write (u, "(A)")  "* Test output: commands_17"
  write (u, "(A)")  "*   Purpose: declare histograms"
  write (u, "(A)")

  write (u, "(A)")  "* Initialization"
  write (u, "(A)")

  call syntax_cmd_list_init ()
  call global%global_init ()

  write (u, "(A)")  "* Input file"
  write (u, "(A)")

  call ifile_append (ifile, '$obs_label = "foo"')
  call ifile_append (ifile, '$obs_unit = "cm"')
  call ifile_append (ifile, '$title = "Histogram foo"')
  call ifile_append (ifile, '$description = "This is histogram foo"')
  call ifile_append (ifile, 'histogram foo (0,5,1)')
  call ifile_append (ifile, '$title = "Histogram bar"')
  call ifile_append (ifile, '$description = "This is histogram bar"')
  call ifile_append (ifile, 'n_bins = 2')
  call ifile_append (ifile, 'histogram bar (0,5)')
  call ifile_append (ifile, '$title = "Histogram gee"')
  call ifile_append (ifile, '$description = "This is histogram gee"')
  call ifile_append (ifile, '?normalize_bins = true')
  call ifile_append (ifile, 'histogram gee (0,5)')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "* Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root)

  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%compile (pn_root, global)

  call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
```

```
    write (u, "(A)")

    call command_list%execute (global)

    write (u, "(A)")  "* Record two data items"
    write (u, "(A)")

    name(1) = "foo"
    name(2) = "bar"
    name(3) = "gee"

    do i = 1, 3
       call analysis_record_data (name(i), 0.1_default, &
            weight = 0.25_default)
       call analysis_record_data (name(i), 3.1_default)
       call analysis_record_data (name(i), 4.1_default, &
            excess = 0.5_default)
       call analysis_record_data (name(i), 7.1_default)
    end do

    write (u, "(A)")  "* Display analysis store"
    write (u, "(A)")

    call analysis_write (u, verbose=.true.)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call analysis_final ()
    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_17"

  end subroutine commands_17
```

Plot

Declare a plot, fill it and display contents.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_18, "commands_18", &
       "plots", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_18
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_18 (u)
```

```
integer, intent(in) :: u
type(ifile_t) :: ifile
type(command_list_t), target :: command_list
type(rt_data_t), target :: global
type(parse_node_t), pointer :: pn_root

write (u, "(A)")  "* Test output: commands_18"
write (u, "(A)")  "*   Purpose: declare a plot"
write (u, "(A)")

write (u, "(A)")  "* Initialization"
write (u, "(A)")

call syntax_cmd_list_init ()
call global%global_init ()

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, '$obs_label = "foo"')
call ifile_append (ifile, '$obs_unit = "cm"')
call ifile_append (ifile, '$title = "Plot foo"')
call ifile_append (ifile, '$description = "This is plot foo"')
call ifile_append (ifile, '$x_label = "x axis"')
call ifile_append (ifile, '$y_label = "y axis"')
call ifile_append (ifile, '?x_log = false')
call ifile_append (ifile, '?y_log = true')
call ifile_append (ifile, 'x_min = -1')
call ifile_append (ifile, 'x_max = 1')
call ifile_append (ifile, 'y_min = 0.1')
call ifile_append (ifile, 'y_max = 1000')
call ifile_append (ifile, 'plot foo')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)

call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)
```

```
    write (u, "(A)")  "* Record two data items"
    write (u, "(A)")

    call analysis_record_data (var_str ("foo"), 0._default, 20._default, &
         xerr = 0.25_default)
    call analysis_record_data (var_str ("foo"), 0.5_default, 0.2_default, &
         yerr = 0.07_default)
    call analysis_record_data (var_str ("foo"), 3._default, 2._default)

    write (u, "(A)")  "* Display analysis store"
    write (u, "(A)")

    call analysis_write (u, verbose=.true.)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call analysis_final ()
    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_18"

  end subroutine commands_18
```

Graph

Combine two (empty) plots to a graph.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_19, "commands_19", &
       "graphs", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_19
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_19 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root

    write (u, "(A)")  "* Test output: commands_19"
    write (u, "(A)")  "*   Purpose: combine two plots to a graph"
    write (u, "(A)")

    write (u, "(A)")  "* Initialization"
```

```
write (u, "(A)")

call syntax_cmd_list_init ()
call global%global_init ()

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'plot a')
call ifile_append (ifile, 'plot b')
call ifile_append (ifile, '$title = "Graph foo"')
call ifile_append (ifile, '$description = "This is graph foo"')
call ifile_append (ifile, 'graph foo = a & b')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)

call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

write (u, "(A)")  "* Display analysis object"
write (u, "(A)")

call analysis_write (var_str ("foo"), u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call analysis_final ()
call command_list%final ()
call global%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()

write (u, "(A)")
write (u, "(A)")  "* Test output end: commands_19"
```

```
end subroutine commands_19
```

Record Data

Record data in previously allocated analysis objects.

⟨*Commands: execute tests*⟩+≡

```
call test (commands_20, "commands_20", &
     "record data", &
     u, results)
```

⟨*Commands: test declarations*⟩+≡

```
public :: commands_20
```

⟨*Commands: tests*⟩+≡

```
subroutine commands_20 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(command_list_t), target :: command_list
  type(rt_data_t), target :: global
  type(parse_node_t), pointer :: pn_root

  write (u, "(A)")  "* Test output: commands_20"
  write (u, "(A)")  "*   Purpose: record data"
  write (u, "(A)")

  write (u, "(A)")  "* Initialization: create observable, histogram, plot"
  write (u, "(A)")

  call syntax_cmd_list_init ()
  call global%global_init ()

  call analysis_init_observable (var_str ("o"))
  call analysis_init_histogram (var_str ("h"), 0._default, 1._default, 3, &
       normalize_bins = .false.)
  call analysis_init_plot (var_str ("p"))

  write (u, "(A)")  "* Input file"
  write (u, "(A)")

  call ifile_append (ifile, 'record o (1.234)')
  call ifile_append (ifile, 'record h (0.5)')
  call ifile_append (ifile, 'record p (1, 2)')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "* Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root)

  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")
```

```
    call command_list%compile (pn_root, global)

    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    write (u, "(A)")  "* Display analysis object"
    write (u, "(A)")

    call analysis_write (u, verbose = .true.)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call analysis_final ()
    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_20"

  end subroutine commands_20
```

Analysis

Declare an analysis expression and use it to fill an observable during event generation.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_21, "commands_21", &
      "analysis expression", &
      u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_21
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_21 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    type(prclib_entry_t), pointer :: lib

    write (u, "(A)")  "* Test output: commands_21"
```

```
write (u, "(A)")  "*   Purpose: create and use analysis expression"
write (u, "(A)")

write (u, "(A)")  "* Initialization: create observable"
write (u, "(A)")

call syntax_cmd_list_init ()
call syntax_model_file_init ()
call global%global_init ()
call global%init_fallback_model &
     (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"))

call global%var_list%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known=.true.)
call global%var_list%set_string (var_str ("$phs_method"), &
     var_str ("single"), is_known=.true.)
call global%var_list%set_string (var_str ("$integration_method"),&
     var_str ("midpoint"), is_known=.true.)
call global%var_list%set_log (var_str ("?vis_history"),&
     .false., is_known=.true.)
call global%var_list%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)
call global%var_list%set_real (var_str ("sqrts"), &
     1000._default, is_known=.true.)

allocate (lib)
call lib%init (var_str ("lib_cmd8"))
call global%add_prclib (lib)

call analysis_init_observable (var_str ("m"))

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'process commands_21_p = s, s => s, s')
call ifile_append (ifile, 'compile')
call ifile_append (ifile, 'iterations = 1:100')
call ifile_append (ifile, 'integrate (commands_21_p)')
call ifile_append (ifile, '?unweighted = true')
call ifile_append (ifile, 'n_events = 3')
call ifile_append (ifile, '?read_raw = false')
call ifile_append (ifile, 'observable m')
call ifile_append (ifile, 'analysis = record m (eval M [s])')
call ifile_append (ifile, 'simulate (commands_21_p)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)
```

```
    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)

    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    write (u, "(A)")  "* Display analysis object"
    write (u, "(A)")

    call analysis_write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call analysis_final ()
    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_21"

  end subroutine commands_21
```

Write Analysis

Write accumulated analysis data to file.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_22, "commands_22", &
       "write analysis", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_22
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_22 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    integer :: u_file, iostat
    logical :: exist
```

```
character(80) :: buffer

write (u, "(A)")  "* Test output: commands_22"
write (u, "(A)")  "*   Purpose: write analysis data"
write (u, "(A)")

write (u, "(A)")  "* Initialization: create observable"
write (u, "(A)")

call syntax_cmd_list_init ()
call global%global_init ()

call analysis_init_observable (var_str ("m"))
call analysis_record_data (var_str ("m"), 125._default)

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, '$out_file = "commands_22.dat"')
call ifile_append (ifile, 'write_analysis')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)

call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

write (u, "(A)")  "* Display analysis data"
write (u, "(A)")

inquire (file = "commands_22.dat", exist = exist)
if (.not. exist) then
   write (u, "(A)")  "ERROR: File commands_22.dat not found"
   return
end if

u_file = free_unit ()
open (u_file, file = "commands_22.dat", &
     action = "read", status = "old")
```

```
    do
       read (u_file, "(A)", iostat = iostat)  buffer
       if (iostat /= 0)  exit
       write (u, "(A)") trim (buffer)
    end do
    close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call analysis_final ()
    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_22"

  end subroutine commands_22
```

Compile Analysis

Write accumulated analysis data to file and compile.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_23, "commands_23", &
       "compile analysis", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_23
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_23 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    integer :: u_file, iostat
    character(256) :: buffer
    logical :: exist
    type(graph_options_t) :: graph_options

    write (u, "(A)")  "* Test output: commands_23"
    write (u, "(A)")  "*   Purpose: write and compile analysis data"
    write (u, "(A)")

    write (u, "(A)")  "* Initialization: create and fill histogram"
    write (u, "(A)")

    call syntax_cmd_list_init ()
```

```
call global%global_init ()

call graph_options_init (graph_options)
call graph_options_set (graph_options, &
     title = var_str ("Histogram for test: commands 23"), &
     description = var_str ("This is a test."), &
     width_mm = 125, height_mm = 85)
call analysis_init_histogram (var_str ("h"), &
     0._default, 10._default, 2._default, .false., &
     graph_options = graph_options)
call analysis_record_data (var_str ("h"), 1._default)
call analysis_record_data (var_str ("h"), 1._default)
call analysis_record_data (var_str ("h"), 1._default)
call analysis_record_data (var_str ("h"), 1._default)
call analysis_record_data (var_str ("h"), 3._default)
call analysis_record_data (var_str ("h"), 3._default)
call analysis_record_data (var_str ("h"), 3._default)
call analysis_record_data (var_str ("h"), 5._default)
call analysis_record_data (var_str ("h"), 7._default)
call analysis_record_data (var_str ("h"), 7._default)
call analysis_record_data (var_str ("h"), 7._default)
call analysis_record_data (var_str ("h"), 7._default)
call analysis_record_data (var_str ("h"), 9._default)
call analysis_record_data (var_str ("h"), 9._default)
call analysis_record_data (var_str ("h"), 9._default)
call analysis_record_data (var_str ("h"), 9._default)
call analysis_record_data (var_str ("h"), 9._default)
call analysis_record_data (var_str ("h"), 9._default)
call analysis_record_data (var_str ("h"), 9._default)

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, '$out_file = "commands_23.dat"')
call ifile_append (ifile, 'compile_analysis')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)

call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Delete Postscript output"
write (u, "(A)")
```

```
  inquire (file = "commands_23.ps", exist = exist)
  if (exist) then
     u_file = free_unit ()
     open (u_file, file = "commands_23.ps", action = "write", status = "old")
     close (u_file, status = "delete")
  end if
  inquire (file = "commands_23.ps", exist = exist)
  write (u, "(1x,A,L1)")  "Postcript output exists = ", exist

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
  write (u, "(A)")

  call command_list%execute (global)

  write (u, "(A)")  "* TeX file"
  write (u, "(A)")

  inquire (file = "commands_23.tex", exist = exist)
  if (.not. exist) then
     write (u, "(A)")  "ERROR: File commands_23.tex not found"
     return
  end if

  u_file = free_unit ()
  open (u_file, file = "commands_23.tex", &
       action = "read", status = "old")
  do
     read (u_file, "(A)", iostat = iostat)  buffer
     if (iostat /= 0)  exit
     write (u, "(A)") trim (buffer)
  end do
  close (u_file)
  write (u, *)

  inquire (file = "commands_23.ps", exist = exist)
  write (u, "(1x,A,L1)")  "Postcript output exists = ", exist

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call ifile_final (ifile)

  call analysis_final ()
  call command_list%final ()
  call global%final ()
  call syntax_cmd_list_final ()
  call syntax_model_file_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: commands_23"

end subroutine commands_23
```

Histogram

Declare a histogram, fill it and display.

⟨*Commands: execute tests*⟩+≡

```
    call test (commands_24, "commands_24", &
         "drawing options", &
         u, results)
```

⟨*Commands: test declarations*⟩+≡

```
    public :: commands_24
```

⟨*Commands: tests*⟩+≡

```
    subroutine commands_24 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_24"
      write (u, "(A)")  "*   Purpose: check graph and drawing options"
      write (u, "(A)")

      write (u, "(A)")  "* Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()

      write (u, "(A)")  "* Input file"
      write (u, "(A)")

      call ifile_append (ifile, '$title = "Title"')
      call ifile_append (ifile, '$description = "Description"')
      call ifile_append (ifile, '$x_label = "X Label"')
      call ifile_append (ifile, '$y_label = "Y Label"')
      call ifile_append (ifile, 'graph_width_mm = 111')
      call ifile_append (ifile, 'graph_height_mm = 222')
      call ifile_append (ifile, 'x_min = -11')
      call ifile_append (ifile, 'x_max = 22')
      call ifile_append (ifile, 'y_min = -33')
      call ifile_append (ifile, 'y_max = 44')
      call ifile_append (ifile, '$gmlcode_bg = "GML Code BG"')
      call ifile_append (ifile, '$gmlcode_fg = "GML Code FG"')
      call ifile_append (ifile, '$fill_options = "Fill Options"')
      call ifile_append (ifile, '$draw_options = "Draw Options"')
      call ifile_append (ifile, '$err_options = "Error Options"')
      call ifile_append (ifile, '$symbol = "Symbol"')
      call ifile_append (ifile, 'histogram foo (0,1)')
      call ifile_append (ifile, 'plot bar')

      call ifile_write (ifile, u)
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Parse file"
    write (u, "(A)")

    call parse_ifile (ifile, pn_root)

    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)

    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    write (u, "(A)")  "* Display analysis store"
    write (u, "(A)")

    call analysis_write (u, verbose=.true.)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call analysis_final ()
    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_24"

  end subroutine commands_24
```

Local Environment

Declare a local environment.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_25, "commands_25", &
       "local process environment", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_25
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_25 (u)
    integer, intent(in) :: u
```

```
type(ifile_t) :: ifile
type(command_list_t), target :: command_list
type(rt_data_t), target :: global
type(parse_node_t), pointer :: pn_root

write (u, "(A)")  "* Test output: commands_25"
write (u, "(A)")  "*   Purpose: declare local environment for process"
write (u, "(A)")

call syntax_model_file_init ()
call syntax_cmd_list_init ()
call global%global_init ()
call global%var_list%set_log (var_str ("?omega_openmp"), &
     .false., is_known = .true.)

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'library = "commands_25_lib"')
call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'process commands_25_p1 = g, g => g, g &
     &{ model = "QCD" }')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)
call global%write_libraries (u)

write (u, "(A)")
write (u, "(A)")  "* Cleanup"

call ifile_final (ifile)

call command_list%final ()
call global%final ()
call syntax_cmd_list_final ()
call syntax_model_file_final ()
```

```
    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_25"

  end subroutine commands_25
```

Alternative Setups

Declare a list of alternative setups.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_26, "commands_26", &
       "alternative setups", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_26
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_26 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root

    write (u, "(A)")  "* Test output: commands_26"
    write (u, "(A)")  "*   Purpose: declare alternative setups for simulation"
    write (u, "(A)")

    call syntax_cmd_list_init ()
    call global%global_init ()

    write (u, "(A)")  "* Input file"
    write (u, "(A)")

    call ifile_append (ifile, 'int i = 0')
    call ifile_append (ifile, 'alt_setup = ({ i = 1 }, { i = 2 })')

    call ifile_write (ifile, u)

    write (u, "(A)")
    write (u, "(A)")  "* Parse file"
    write (u, "(A)")

    call parse_ifile (ifile, pn_root)

    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)
    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")
```

```
    call command_list%execute (global)

    call global%write_expr (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_26"

  end subroutine commands_26
```

Unstable Particle

Define decay processes and declare a particle as unstable. Also check the commands stable, polarized, unpolarized.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_27, "commands_27", &
       "unstable and polarized particles", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_27
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_27 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    type(prclib_entry_t), pointer :: lib

    write (u, "(A)")  "* Test output: commands_27"
    write (u, "(A)")  "*   Purpose: modify particle properties"
    write (u, "(A)")

    call syntax_cmd_list_init ()
    call syntax_model_file_init ()
    call global%global_init ()
    call global%var_list%set_string (var_str ("$method"), &
         var_str ("unit_test"), is_known=.true.)
    call global%var_list%set_string (var_str ("$phs_method"), &
         var_str ("single"), is_known=.true.)
    call global%var_list%set_string (var_str ("$integration_method"),&
         var_str ("midpoint"), is_known=.true.)
```

```
call global%var_list%set_log (var_str ("?vis_history"),&
     .false., is_known=.true.)
call global%var_list%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)

allocate (lib)
call lib%init (var_str ("commands_27_lib"))
call global%add_prclib (lib)

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'ff = 0.4')
call ifile_append (ifile, 'process d1 = s => f, fbar')
call ifile_append (ifile, 'unstable s (d1)')
call ifile_append (ifile, 'polarized f, fbar')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

write (u, "(A)")  "* Show model"
write (u, "(A)")

call global%model%write (u)

write (u, "(A)")
write (u, "(A)")  "* Extra Input"
write (u, "(A)")

call ifile_final (ifile)
call ifile_append (ifile, '?diagonal_decay = true')
call ifile_append (ifile, 'unstable s (d1)')

call ifile_write (ifile, u)

write (u, "(A)")
```

```
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%final ()
call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

write (u, "(A)")  "* Show model"
write (u, "(A)")

call global%model%write (u)

write (u, "(A)")
write (u, "(A)")  "* Extra Input"
write (u, "(A)")

call ifile_final (ifile)
call ifile_append (ifile, '?isotropic_decay = true')
call ifile_append (ifile, 'unstable s (d1)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)

write (u, "(A)")  "* Compile command list"
write (u, "(A)")

call command_list%final ()
call command_list%compile (pn_root, global)
call command_list%write (u)

write (u, "(A)")
write (u, "(A)")  "* Execute command list"
write (u, "(A)")

call command_list%execute (global)

write (u, "(A)")  "* Show model"
write (u, "(A)")
```

```
  call global%model%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Extra Input"
  write (u, "(A)")

  call ifile_final (ifile)
  call ifile_append (ifile, 'stable s')
  call ifile_append (ifile, 'unpolarized f')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "* Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root)

  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%final ()
  call command_list%compile (pn_root, global)
  call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
  write (u, "(A)")

  call command_list%execute (global)

  write (u, "(A)")  "* Show model"
  write (u, "(A)")

  call global%model%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call ifile_final (ifile)

  call command_list%final ()
  call global%final ()
  call syntax_model_file_init ()
  call syntax_cmd_list_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: commands_27"

end subroutine commands_27
```

Quit the program

Quit the program.

⟨*Commands: execute tests*⟩+≡

```
call test (commands_28, "commands_28", &
     "quit", &
     u, results)
```

⟨*Commands: test declarations*⟩+≡

```
public :: commands_28
```

⟨*Commands: tests*⟩+≡

```
subroutine commands_28 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(command_list_t), target :: command_list
  type(rt_data_t), target :: global
  type(parse_node_t), pointer :: pn_root1, pn_root2
  type(string_t), dimension(0) :: no_vars

  write (u, "(A)")  "* Test output: commands_28"
  write (u, "(A)")  "*   Purpose: quit the program"
  write (u, "(A)")

  write (u, "(A)")  "*  Initialization"
  write (u, "(A)")

  call syntax_cmd_list_init ()
  call global%global_init ()

  write (u, "(A)")  "*  Input file: quit without code"
  write (u, "(A)")

  call ifile_append (ifile, 'quit')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "*  Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root1, u)

  write (u, "(A)")
  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%compile (pn_root1, global)
  call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
  write (u, "(A)")

  call command_list%execute (global)
```

```
    call global%write (u, vars = no_vars)

    write (u, "(A)")
    write (u, "(A)")  "*  Input file: quit with code"
    write (u, "(A)")

    call ifile_final (ifile)
    call command_list%final ()
    call ifile_append (ifile, 'quit ( 3 + 4 )')

    call ifile_write (ifile, u)

    write (u, "(A)")
    write (u, "(A)")  "*  Parse file"
    write (u, "(A)")

    call parse_ifile (ifile, pn_root2, u)

    write (u, "(A)")
    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root2, global)
    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    call global%write (u, vars = no_vars)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_28"

  end subroutine commands_28
```

SLHA interface

Testing commands steering the SLHA interface.

⟨*Commands: execute tests*⟩+≡

```
    call test (commands_29, "commands_29", &
         "SLHA interface", &
         u, results)
```

⟨*Commands: test declarations*⟩+≡

```
    public :: commands_29
```

⟨*Commands: tests*⟩+≡

```
    subroutine commands_29 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(var_list_t), pointer :: model_vars
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_29"
      write (u, "(A)")  "*   Purpose: test SLHA interface"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call syntax_model_file_init ()
      call syntax_slha_init ()
      call global%global_init ()

      write (u, "(A)")  "*  Model MSSM, read SLHA file"
      write (u, "(A)")

      call ifile_append (ifile, 'model = "MSSM"')
      call ifile_append (ifile, '?slha_read_decays = true')
      call ifile_append (ifile, 'read_slha ("sps1ap_decays.slha")')

      call ifile_write (ifile, u)

      write (u, "(A)")
      write (u, "(A)")  "*  Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root, u)

      write (u, "(A)")
      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")

      call command_list%compile (pn_root, global)
      call command_list%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Model MSSM, default values:"
      write (u, "(A)")

      call global%model%write (u, verbose = .false., &
```

```
        show_vertices = .false., show_particles = .false.)

    write (u, "(A)")
    write (u, "(A)")  "* Selected global variables"
    write (u, "(A)")

    model_vars => global%model%get_var_list_ptr ()

    call model_vars%write_var (var_str ("mch1"), u)
    call model_vars%write_var (var_str ("wch1"), u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    write (u, "(A)")  "* Model MSSM, values from SLHA file"
    write (u, "(A)")

    call global%model%write (u, verbose = .false., &
        show_vertices = .false., show_particles = .false.)

    write (u, "(A)")
    write (u, "(A)")  "* Selected global variables"
    write (u, "(A)")

    model_vars => global%model%get_var_list_ptr ()

    call model_vars%write_var (var_str ("mch1"), u)
    call model_vars%write_var (var_str ("wch1"), u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_slha_final ()
    call syntax_model_file_final ()
    call syntax_cmd_list_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_29"

  end subroutine commands_29
```

Expressions for scales

Declare a scale, factorization scale or factorization scale expression.
⟨*Commands: execute tests*⟩+≡

```
    call test (commands_30, "commands_30", &
         "scales", &
         u, results)
```

⟨*Commands: test declarations*⟩+≡

```
    public :: commands_30
```

⟨*Commands: tests*⟩+≡

```
    subroutine commands_30 (u)
      integer, intent(in) :: u
      type(ifile_t) :: ifile
      type(command_list_t), target :: command_list
      type(rt_data_t), target :: global
      type(parse_node_t), pointer :: pn_root

      write (u, "(A)")  "* Test output: commands_30"
      write (u, "(A)")  "*   Purpose: define scales"
      write (u, "(A)")

      write (u, "(A)")  "*  Initialization"
      write (u, "(A)")

      call syntax_cmd_list_init ()
      call global%global_init ()

      write (u, "(A)")  "*  Input file"
      write (u, "(A)")

      call ifile_append (ifile, 'scale = 200 GeV')
      call ifile_append (ifile, &
           'factorization_scale = eval Pt [particle]')
      call ifile_append (ifile, &
           'renormalization_scale = eval E [particle]')

      call ifile_write (ifile, u)

      write (u, "(A)")
      write (u, "(A)")  "*  Parse file"
      write (u, "(A)")

      call parse_ifile (ifile, pn_root, u)

      write (u, "(A)")
      write (u, "(A)")  "* Compile command list"
      write (u, "(A)")

      call command_list%compile (pn_root, global)
      call command_list%write (u)

      write (u, "(A)")
      write (u, "(A)")  "* Execute command list"
      write (u, "(A)")

      call command_list%execute (global)
```

```
    call global%write_expr (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_30"

  end subroutine commands_30
```

Weight and reweight expressions

Declare an expression for event weights and reweighting.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_31, "commands_31", &
       "event weights/reweighting", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_31
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_31 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root

    write (u, "(A)")  "* Test output: commands_31"
    write (u, "(A)")  "*   Purpose: define weight/reweight"
    write (u, "(A)")

    write (u, "(A)")  "*  Initialization"
    write (u, "(A)")

    call syntax_cmd_list_init ()
    call global%global_init ()

    write (u, "(A)")  "*  Input file"
    write (u, "(A)")

    call ifile_append (ifile, 'weight = eval Pz [particle]')
    call ifile_append (ifile, 'reweight = eval M2 [particle]')

    call ifile_write (ifile, u)

    write (u, "(A)")
```

```
    write (u, "(A)")  "*  Parse file"
    write (u, "(A)")

    call parse_ifile (ifile, pn_root, u)

    write (u, "(A)")
    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)
    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    call global%write_expr (u)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_31"

  end subroutine commands_31
```

Selecting events

Declare an expression for selecting events in an analysis.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_32, "commands_32", &
      "event selection", &
      u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_32
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_32 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
```

```
  write (u, "(A)")  "* Test output: commands_32"
  write (u, "(A)")  "*   Purpose: define selection"
  write (u, "(A)")

  write (u, "(A)")  "*  Initialization"
  write (u, "(A)")

  call syntax_cmd_list_init ()
  call global%global_init ()

  write (u, "(A)")  "*  Input file"
  write (u, "(A)")

  call ifile_append (ifile, 'selection = any PDG == 13 [particle]')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "*  Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root, u)

  write (u, "(A)")
  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%compile (pn_root, global)
  call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
  write (u, "(A)")

  call command_list%execute (global)

  call global%write_expr (u)

  write (u, "(A)")
  write (u, "(A)")  "* Cleanup"

  call ifile_final (ifile)

  call command_list%final ()
  call global%final ()
  call syntax_cmd_list_final ()

  write (u, "(A)")
  write (u, "(A)")  "* Test output end: commands_32"

end subroutine commands_32
```

Executing shell commands

Execute a shell command.

⟨*Commands: execute tests*⟩+≡

```
call test (commands_33, "commands_33", &
     "execute shell command", &
     u, results)
```

⟨*Commands: test declarations*⟩+≡

```
public :: commands_33
```

⟨*Commands: tests*⟩+≡

```
subroutine commands_33 (u)
  integer, intent(in) :: u
  type(ifile_t) :: ifile
  type(command_list_t), target :: command_list
  type(rt_data_t), target :: global
  type(parse_node_t), pointer :: pn_root
  integer :: u_file, iostat
  character(3) :: buffer

  write (u, "(A)")  "* Test output: commands_33"
  write (u, "(A)")  "*   Purpose: execute shell command"
  write (u, "(A)")

  write (u, "(A)")  "*  Initialization"
  write (u, "(A)")

  call syntax_cmd_list_init ()
  call global%global_init ()

  write (u, "(A)")  "*  Input file"
  write (u, "(A)")

  call ifile_append (ifile, 'exec ("echo foo >> bar")')

  call ifile_write (ifile, u)

  write (u, "(A)")
  write (u, "(A)")  "*  Parse file"
  write (u, "(A)")

  call parse_ifile (ifile, pn_root, u)

  write (u, "(A)")
  write (u, "(A)")  "* Compile command list"
  write (u, "(A)")

  call command_list%compile (pn_root, global)
  call command_list%write (u)

  write (u, "(A)")
  write (u, "(A)")  "* Execute command list"
  write (u, "(A)")
```

```
    call command_list%execute (global)
    u_file = free_unit ()
    open (u_file, file = "bar", &
         action = "read", status = "old")
    do
       read (u_file, "(A)", iostat = iostat)  buffer
       if (iostat /= 0) exit
    end do
    write (u, "(A,A)")  "should be 'foo': ", trim (buffer)
    close (u_file)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_33"

  end subroutine commands_33
```

Callback

Instead of an explicit write, use the callback feature to write the analysis file during event generation. We generate 4 events and arrange that the callback is executed while writing the 3rd event.

⟨*Commands: execute tests*⟩+≡

```
  call test (commands_34, "commands_34", &
       "analysis via callback", &
       u, results)
```

⟨*Commands: test declarations*⟩+≡

```
  public :: commands_34
```

⟨*Commands: tests*⟩+≡

```
  subroutine commands_34 (u)
    integer, intent(in) :: u
    type(ifile_t) :: ifile
    type(command_list_t), target :: command_list
    type(rt_data_t), target :: global
    type(parse_node_t), pointer :: pn_root
    type(prclib_entry_t), pointer :: lib
    type(event_callback_34_t) :: event_callback

    write (u, "(A)")  "* Test output: commands_34"
    write (u, "(A)")  "*   Purpose: write analysis data"
    write (u, "(A)")

    write (u, "(A)")  "* Initialization: create observable"
```

```
write (u, "(A)")

call syntax_cmd_list_init ()
call global%global_init ()

call syntax_model_file_init ()
call global%global_init ()
call global%init_fallback_model &
     (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"))

call global%var_list%set_string (var_str ("$method"), &
     var_str ("unit_test"), is_known=.true.)
call global%var_list%set_string (var_str ("$phs_method"), &
     var_str ("single"), is_known=.true.)
call global%var_list%set_string (var_str ("$integration_method"),&
     var_str ("midpoint"), is_known=.true.)
call global%var_list%set_real (var_str ("sqrts"), &
     1000._default, is_known=.true.)
call global%var_list%set_log (var_str ("?vis_history"),&
     .false., is_known=.true.)
call global%var_list%set_log (var_str ("?integration_timer"),&
     .false., is_known = .true.)

allocate (lib)
call lib%init (var_str ("lib_cmd34"))
call global%add_prclib (lib)

write (u, "(A)")  "* Prepare callback for writing analysis to I/O unit"
write (u, "(A)")

event_callback%u = u
call global%set_event_callback (event_callback)

write (u, "(A)")  "* Input file"
write (u, "(A)")

call ifile_append (ifile, 'model = "Test"')
call ifile_append (ifile, 'process commands_34_p = s, s => s, s')
call ifile_append (ifile, 'compile')
call ifile_append (ifile, 'iterations = 1:1000')
call ifile_append (ifile, 'integrate (commands_34_p)')
call ifile_append (ifile, 'observable sq')
call ifile_append (ifile, 'analysis = record sq (sqrts)')
call ifile_append (ifile, 'n_events = 4')
call ifile_append (ifile, 'event_callback_interval = 3')
call ifile_append (ifile, 'simulate (commands_34_p)')

call ifile_write (ifile, u)

write (u, "(A)")
write (u, "(A)")  "* Parse file"
write (u, "(A)")

call parse_ifile (ifile, pn_root)
```

```
    write (u, "(A)")  "* Compile command list"
    write (u, "(A)")

    call command_list%compile (pn_root, global)

    call command_list%write (u)

    write (u, "(A)")
    write (u, "(A)")  "* Execute command list"
    write (u, "(A)")

    call command_list%execute (global)

    write (u, "(A)")
    write (u, "(A)")  "* Cleanup"

    call ifile_final (ifile)

    call analysis_final ()
    call command_list%final ()
    call global%final ()
    call syntax_cmd_list_final ()
    call syntax_model_file_final ()

    write (u, "(A)")
    write (u, "(A)")  "* Test output end: commands_34"

  end subroutine commands_34
```

For this test, we invent a callback object which simply writes the analysis file, using the standard call for this. Here we rely on the fact that the analysis data are stored as a global entity, otherwise we would have to access them via the event object.

⟨*Commands: test auxiliary types*⟩≡

```
  type, extends (event_callback_t) :: event_callback_34_t
     private
     integer :: u = 0
   contains
     procedure :: write => event_callback_34_write
     procedure :: proc => event_callback_34
  end type event_callback_34_t
```

The output routine is unused. The actual callback should write the analysis data to the output unit that we have injected into the callback object.

⟨*Commands: test auxiliary*⟩+≡

```
  subroutine event_callback_34_write (event_callback, unit)
    class(event_callback_34_t), intent(in) :: event_callback
    integer, intent(in), optional :: unit
  end subroutine event_callback_34_write

  subroutine event_callback_34 (event_callback, i, event)
```

```
  class(event_callback_34_t), intent(in) :: event_callback
  integer(i64), intent(in) :: i
  class(generic_event_t), intent(in) :: event
  call analysis_write (event_callback%u)
end subroutine event_callback_34
```

35.2 Toplevel module WHIZARD

⟨whizard.f90⟩≡
⟨*File header*⟩

```
module whizard

  use io_units
⟨Use strings⟩
  use system_defs, only: VERSION_STRING
  use system_defs, only: EOF, BACKSLASH
  use diagnostics
  use os_interface
  use ifiles
  use lexers
  use parser
  use variables, only: var_list_t
  use eval_trees
  use models
  use phs_forests
  use prclib_stacks
  use slha_interface
  use blha_config
  use rt_data
  use commands

⟨Standard module head⟩

⟨WHIZARD: public⟩

⟨WHIZARD: types⟩

  save

contains

⟨WHIZARD: procedures⟩

end module whizard
```

35.2.1 Options

Here we introduce a wrapper that holds various user options, so they can transparently be passed from the main program to the `whizard` object. Most parameters are used for initializing the `global` state.

⟨*WHIZARD: public*⟩≡

```
  public :: whizard_options_t
```

⟨*WHIZARD: types*⟩≡

```
  type :: whizard_options_t
     type(string_t) :: preload_model
     type(string_t) :: default_lib
     type(string_t) :: preload_libraries
     logical :: rebuild_library = .false.
```

```
    logical :: recompile_library = .false.
    logical :: rebuild_user
    logical :: rebuild_phs = .false.
    logical :: rebuild_grids = .false.
    logical :: rebuild_events = .false.
end type whizard_options_t
```

35.2.2 Parse tree stack

We collect all parse trees that we generate in the `whizard` object. To this end, we create a stack of parse trees. They must not be finalized before the `global` object is finalized, because items such as a cut definition may contain references to the parse tree from which they were generated.

⟨*WHIZARD: types*⟩+≡

```
type, extends (parse_tree_t) :: pt_entry_t
   type(pt_entry_t), pointer :: previous => null ()
end type pt_entry_t
```

This is the stack. Since we always prepend, we just need the `last` pointer.

⟨*WHIZARD: types*⟩+≡

```
type :: pt_stack_t
   type(pt_entry_t), pointer :: last => null ()
 contains
 ⟨WHIZARD: pt stack: TBP⟩
end type pt_stack_t
```

The finalizer is called at the very end.

⟨*WHIZARD: pt stack: TBP*⟩≡

```
procedure :: final => pt_stack_final
```

⟨*WHIZARD: procedures*⟩≡

```
subroutine pt_stack_final (pt_stack)
  class(pt_stack_t), intent(inout) :: pt_stack
  type(pt_entry_t), pointer :: current
  do while (associated (pt_stack%last))
     current => pt_stack%last
     pt_stack%last => current%previous
     call parse_tree_final (current%parse_tree_t)
     deallocate (current)
  end do
end subroutine pt_stack_final
```

Create and push a new entry, keeping the previous ones.

⟨*WHIZARD: pt stack: TBP*⟩+≡

```
procedure :: push => pt_stack_push
```

⟨*WHIZARD: procedures*⟩+≡

```
subroutine pt_stack_push (pt_stack, parse_tree)
  class(pt_stack_t), intent(inout) :: pt_stack
  type(parse_tree_t), intent(out), pointer :: parse_tree
  type(pt_entry_t), pointer :: current
```

```
    allocate (current)
    parse_tree => current%parse_tree_t
    current%previous => pt_stack%last
    pt_stack%last => current
  end subroutine pt_stack_push
```

35.2.3 The whizard object

An object of type whizard_t is the top-level wrapper for a WHIZARD instance. The object holds various default settings and the current state of the generator, the global object of type rt_data_t. This object contains, for instance, the list of variables and the process libraries.

Since components of the global subobject are frequently used as targets, the whizard object should also consistently carry the target attribute.

The various self-tests do no not use this object. They initialize only specific subsets of the system, according to their needs.

Note: we intend to allow several concurrent instances. In the current implementation, there are still a few obstacles to this: the model library and the syntax tables are global variables, and the error handling uses global state. This should be improved.

⟨*WHIZARD: public*⟩+≡

```
  public :: whizard_t
```

⟨*WHIZARD: types*⟩+≡

```
  type :: whizard_t
     type(whizard_options_t) :: options
     type(rt_data_t) :: global
     type(pt_stack_t) :: pt_stack
   contains
   ⟨WHIZARD: whizard: TBP⟩
  end type whizard_t
```

35.2.4 Initialization and finalization

⟨*WHIZARD: whizard: TBP*⟩≡

```
  procedure :: init => whizard_init
```

⟨*WHIZARD: procedures*⟩+≡

```
  subroutine whizard_init (whizard, options, paths, logfile)
    class(whizard_t), intent(out), target :: whizard
    type(whizard_options_t), intent(in) :: options
    type(paths_t), intent(in), optional :: paths
    type(string_t), intent(in), optional :: logfile
    call init_syntax_tables ()
    whizard%options = options
    call whizard%global%global_init (paths, logfile)
    call whizard%init_rebuild_flags ()
    call whizard%preload_model ()
    call whizard%preload_library ()
    call whizard%global%init_fallback_model &
         (var_str ("SM_hadrons"), var_str ("SM_hadrons.mdl"))
```

```
    end subroutine whizard_init
```

Apart from the global data which have been initialized above, the process and model lists need to be finalized.

⟨*WHIZARD: whizard: TBP*⟩+≡
```
    procedure :: final => whizard_final
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_final (whizard)
      class(whizard_t), intent(inout), target :: whizard
      call whizard%global%final ()
      call whizard%pt_stack%final ()
  !!! JRR: WK please check (#529)
      !    call user_code_final ()
      call final_syntax_tables ()
    end subroutine whizard_final
```

Set the rebuild flags. They can be specified on the command line and set the initial value for the associated logical variables.

⟨*WHIZARD: whizard: TBP*⟩+≡
```
    procedure :: init_rebuild_flags => whizard_init_rebuild_flags
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_init_rebuild_flags (whizard)
      class(whizard_t), intent(inout), target :: whizard
      associate (var_list => whizard%global%var_list, options => whizard%options)
        call var_list%append_log (var_str ("?rebuild_library"), &
             options%rebuild_library, intrinsic=.true.)
        call var_list%append_log (var_str ("?recompile_library"), &
             options%recompile_library, intrinsic=.true.)
        call var_list%append_log (var_str ("?rebuild_phase_space"), &
             options%rebuild_phs, intrinsic=.true.)
        call var_list%append_log (var_str ("?rebuild_grids"), &
             options%rebuild_grids, intrinsic=.true.)
        call var_list%append_log (var_str ("?powheg_rebuild_grids"), &
             options%rebuild_grids, intrinsic=.true.)
        call var_list%append_log (var_str ("?rebuild_events"), &
             options%rebuild_events, intrinsic=.true.)
      end associate
    end subroutine whizard_init_rebuild_flags
```

This procedure preloads a model, if a model name is given.

⟨*WHIZARD: whizard: TBP*⟩+≡
```
    procedure :: preload_model => whizard_preload_model
```

⟨*WHIZARD: procedures*⟩+≡
```
    subroutine whizard_preload_model (whizard)
      class(whizard_t), intent(inout), target :: whizard
      type(string_t) :: model_name
      model_name = whizard%options%preload_model
      if (model_name /= "") then
         call whizard%global%read_model (model_name, whizard%global%preload_model)
         whizard%global%model => whizard%global%preload_model
```

```
      if (associated (whizard%global%model)) then
         call whizard%global%model%link_var_list (whizard%global%var_list)
         call msg_message ("Preloaded model: " &
              // char (model_name))
      else
         call msg_fatal ("Preloading model " // char (model_name) &
              // " failed")
      end if
   else
      call msg_message ("No model preloaded")
   end if
end subroutine whizard_preload_model
```

This procedure preloads a library, if a library name is given.

Note: This version just opens a new library with that name. It does not load (yet) an existing library on file, as previous WHIZARD versions would do.

⟨*WHIZARD: whizard: TBP*⟩+≡

```
procedure :: preload_library => whizard_preload_library
```

⟨*WHIZARD: procedures*⟩+≡

```
subroutine whizard_preload_library (whizard)
  class(whizard_t), intent(inout), target :: whizard
  type(string_t) :: library_name, libs
  type(string_t), dimension(:), allocatable :: libname_static
  type(prclib_entry_t), pointer :: lib_entry
  integer :: i
  call get_prclib_static (libname_static)
  do i = 1, size (libname_static)
     allocate (lib_entry)
     call lib_entry%init_static (libname_static(i))
     call whizard%global%add_prclib (lib_entry)
  end do
  libs = adjustl (whizard%options%preload_libraries)
  if (libs == "" .and. whizard%options%default_lib /= "") then
        allocate (lib_entry)
        call lib_entry%init (whizard%options%default_lib)
        call whizard%global%add_prclib (lib_entry)
        call msg_message ("Preloaded library: " // &
             char (whizard%options%default_lib))
     end if
  SCAN_LIBS: do while (libs /= "")
     call split (libs, library_name, " ")
     if (library_name /= "") then
        allocate (lib_entry)
        call lib_entry%init (library_name)
        call whizard%global%add_prclib (lib_entry)
        call msg_message ("Preloaded library: " // char (library_name))
     end if
  end do SCAN_LIBS
end subroutine whizard_preload_library
```

35.2.5 Initialization and finalization (old version)

These procedures initialize and finalize global variables. Most of them are collected in the `global` data record located here, the others are syntax tables located in various modules, which do not change during program execution. Furthermore, there is a global model list and a global process store, which get filled during program execution but are finalized here.

During initialization, we can preload a default model and initialize a default library for setting up processes. The default library is loaded if requested by the setup. Further libraries can be loaded as specified by command-line flags. Initialize/finalize the syntax tables used by WHIZARD:

⟨*WHIZARD: public*⟩+≡
```
public :: init_syntax_tables
public :: final_syntax_tables
```

⟨*WHIZARD: procedures*⟩+≡
```
subroutine init_syntax_tables ()
  call syntax_model_file_init ()
  call syntax_phs_forest_init ()
  call syntax_pexpr_init ()
  call syntax_slha_init ()
  call syntax_cmd_list_init ()
end subroutine init_syntax_tables

subroutine final_syntax_tables ()
  call syntax_model_file_final ()
  call syntax_phs_forest_final ()
  call syntax_pexpr_final ()
  call syntax_slha_final ()
  call syntax_cmd_list_final ()
end subroutine final_syntax_tables
```

Write the syntax tables to external files.

⟨*WHIZARD: public*⟩+≡
```
public :: write_syntax_tables
```

⟨*WHIZARD: procedures*⟩+≡
```
subroutine write_syntax_tables ()
  integer :: unit
  character(*), parameter :: file_model = "whizard.model_file.syntax"
  character(*), parameter :: file_phs = "whizard.phase_space_file.syntax"
  character(*), parameter :: file_pexpr = "whizard.prt_expressions.syntax"
  character(*), parameter :: file_slha = "whizard.slha.syntax"
  character(*), parameter :: file_sindarin = "whizard.sindarin.syntax"
  unit = free_unit ()
  print *, "Writing file '" // file_model // "'"
  open (unit=unit, file=file_model, status="replace", action="write")
  write (unit, "(A)")  VERSION_STRING
  write (unit, "(A)")  "Syntax definition file: " // file_model
  call syntax_model_file_write (unit)
  close (unit)
  print *, "Writing file '" // file_phs // "'"
  open (unit=unit, file=file_phs, status="replace", action="write")
  write (unit, "(A)")  VERSION_STRING
```

```
    write (unit, "(A)")  "Syntax definition file: " // file_phs
    call syntax_phs_forest_write (unit)
    close (unit)
    print *, "Writing file '" // file_pexpr // "'"
    open (unit=unit, file=file_pexpr, status="replace", action="write")
    write (unit, "(A)")  VERSION_STRING
    write (unit, "(A)")  "Syntax definition file: " // file_pexpr
    call syntax_pexpr_write (unit)
    close (unit)
    print *, "Writing file '" // file_slha // "'"
    open (unit=unit, file=file_slha, status="replace", action="write")
    write (unit, "(A)")  VERSION_STRING
    write (unit, "(A)")  "Syntax definition file: " // file_slha
    call syntax_slha_write (unit)
    close (unit)
    print *, "Writing file '" // file_sindarin // "'"
    open (unit=unit, file=file_sindarin, status="replace", action="write")
    write (unit, "(A)")  VERSION_STRING
    write (unit, "(A)")  "Syntax definition file: " // file_sindarin
    call syntax_cmd_list_write (unit)
    close (unit)
  end subroutine write_syntax_tables
```

35.2.6 Execute command lists

Process commands given on the command line, stored as an `ifile`. The whole input is read, compiled and executed as a whole.

⟨*WHIZARD: whizard: TBP*⟩+≡

```
  procedure :: process_ifile => whizard_process_ifile
```

⟨*WHIZARD: procedures*⟩+≡

```
  subroutine whizard_process_ifile (whizard, ifile, quit, quit_code)
    class(whizard_t), intent(inout), target :: whizard
    type(ifile_t), intent(in) :: ifile
    logical, intent(out) :: quit
    integer, intent(out) :: quit_code
    type(lexer_t), target :: lexer
    type(stream_t), target :: stream
    call msg_message ("Reading commands given on the command line")
    call lexer_init_cmd_list (lexer)
    call stream_init (stream, ifile)
    call whizard%process_stream (stream, lexer, quit, quit_code)
    call stream_final (stream)
    call lexer_final (lexer)
  end subroutine whizard_process_ifile
```

Process standard input as a command list. The whole input is read, compiled and executed as a whole.

⟨*WHIZARD: whizard: TBP*⟩+≡

```
  procedure :: process_stdin => whizard_process_stdin
```

⟨*WHIZARD: procedures*⟩+≡

```
subroutine whizard_process_stdin (whizard, quit, quit_code)
  class(whizard_t), intent(inout), target :: whizard
  logical, intent(out) :: quit
  integer, intent(out) :: quit_code
  type(lexer_t), target :: lexer
  type(stream_t), target :: stream
  call msg_message ("Reading commands from standard input")
  call lexer_init_cmd_list (lexer)
  call stream_init (stream, 5)
  call whizard%process_stream (stream, lexer, quit, quit_code)
  call stream_final (stream)
  call lexer_final (lexer)
end subroutine whizard_process_stdin
```

Process a file as a command list.

⟨*WHIZARD: whizard: TBP*⟩+≡

```
procedure :: process_file => whizard_process_file
```

⟨*WHIZARD: procedures*⟩+≡

```
subroutine whizard_process_file (whizard, file, quit, quit_code)
  class(whizard_t), intent(inout), target :: whizard
  type(string_t), intent(in) :: file
  logical, intent(out) :: quit
  integer, intent(out) :: quit_code
  type(lexer_t), target :: lexer
  type(stream_t), target :: stream
  logical :: exist
  call msg_message ("Reading commands from file '" // char (file) // "'")
  inquire (file=char(file), exist=exist)
  if (exist) then
     call lexer_init_cmd_list (lexer)
     call stream_init (stream, char (file))
     call whizard%process_stream (stream, lexer, quit, quit_code)
     call stream_final (stream)
     call lexer_final (lexer)
  else
     call msg_error ("File '" // char (file) // "' not found")
  end if
end subroutine whizard_process_file
```

⟨*WHIZARD: whizard: TBP*⟩+≡

```
procedure :: process_stream => whizard_process_stream
```

⟨*WHIZARD: procedures*⟩+≡

```
subroutine whizard_process_stream (whizard, stream, lexer, quit, quit_code)
  class(whizard_t), intent(inout), target :: whizard
  type(stream_t), intent(inout), target :: stream
  type(lexer_t), intent(inout), target :: lexer
  logical, intent(out) :: quit
  integer, intent(out) :: quit_code
  type(parse_tree_t), pointer :: parse_tree
  type(command_list_t), target :: command_list
  call lexer_assign_stream (lexer, stream)
```

```
    call whizard%pt_stack%push (parse_tree)
    call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
    if (associated (parse_tree%get_root_ptr ())) then
       whizard%global%lexer => lexer
       call command_list%compile (parse_tree%get_root_ptr (), &
            whizard%global)
    end if
    call whizard%global%activate ()
    call command_list%execute (whizard%global)
    call command_list%final ()
    quit = whizard%global%quit
    quit_code = whizard%global%quit_code
  end subroutine whizard_process_stream
```

35.2.7 The WHIZARD shell

This procedure implements interactive mode. One line is processed at a time.

⟨*WHIZARD: whizard: TBP*⟩+≡

```
  procedure :: shell => whizard_shell
```

⟨*WHIZARD: procedures*⟩+≡

```
  subroutine whizard_shell (whizard, quit_code)
    class(whizard_t), intent(inout), target :: whizard
    integer, intent(out) :: quit_code
    type(lexer_t), target :: lexer
    type(stream_t), target :: stream
    type(string_t) :: prompt1
    type(string_t) :: prompt2
    type(string_t) :: input
    type(string_t) :: extra
    integer :: last
    integer :: iostat
    logical :: mask_tmp
    logical :: quit
    call msg_message ("Launching interactive shell")
    call lexer_init_cmd_list (lexer)
    prompt1 = "whish? "
    prompt2 = "     > "
    COMMAND_LOOP: do
       call put (6, prompt1)
       call get (5, input, iostat=iostat)
       if (iostat > 0 .or. iostat == EOF) exit COMMAND_LOOP
       CONTINUE_INPUT: do
          last = len_trim (input)
          if (extract (input, last, last) /= BACKSLASH)  exit CONTINUE_INPUT
          call put (6, prompt2)
          call get (5, extra, iostat=iostat)
          if (iostat > 0) exit COMMAND_LOOP
          input = replace (input, last, extra)
       end do CONTINUE_INPUT
       call stream_init (stream, input)
       mask_tmp = mask_fatal_errors
       mask_fatal_errors = .true.
```

```
      call whizard%process_stream (stream, lexer, quit, quit_code)
      msg_count = 0
      mask_fatal_errors = mask_tmp
      call stream_final (stream)
      if (quit)  exit COMMAND_LOOP
   end do COMMAND_LOOP
   print *
   call lexer_final (lexer)
end subroutine whizard_shell
```

35.3 Tools for the command line

We don't intent to be very smart here, but this module provides a few small tools that simplify dealing with the command line.

⟨cmdline_options.f90⟩≡

```
⟨File header⟩

module cmdline_options

⟨Use strings⟩
  use diagnostics

⟨Standard module head⟩

  public :: init_options
  public :: no_option_value
  public :: get_option_value

⟨Main: cmdline arg len declaration⟩

  abstract interface
     subroutine msg
     end subroutine msg
  end interface

  procedure (msg), pointer :: print_usage => null ()

contains

  subroutine init_options (usage_msg)
    procedure (msg) :: usage_msg
    print_usage => usage_msg
  end subroutine init_options

  subroutine no_option_value (option, value)
    type(string_t), intent(in) :: option, value
    if (value /= "") then
       call msg_error (" Option '" // char (option) // "' should have no value")
    end if
  end subroutine no_option_value

  function get_option_value (i, option, value) result (string)
    type(string_t) :: string
    integer, intent(inout) :: i
    type(string_t), intent(in) :: option
    type(string_t), intent(in), optional :: value
    character(CMDLINE_ARG_LEN) :: arg_value
    integer :: arg_len, arg_status
    logical :: has_value
    if (present (value)) then
       has_value = value /= ""
    else
       has_value = .false.
    end if
```

```
    if (has_value) then
       string = value
    else
       i = i + 1
       call get_command_argument (i, arg_value, arg_len, arg_status)
       select case (arg_status)
       case (0)
       case (-1)
          call msg_error (" Option value truncated: '" // arg_value // "'")
       case default
          call print_usage ()
          call msg_fatal (" Option '" // char (option) // "' needs a value")
       end select
       select case (arg_value(1:1))
       case ("-")
          call print_usage ()
          call msg_fatal (" Option '" // char (option) // "' needs a value")
       end select
       string = trim (arg_value)
    end if
  end function get_option_value

end module cmdline_options
```

35.4 Query Feature Support

This module accesses the various optional features (modules) that WHIZARD can support and repors on their availability.

⟨features.f90⟩≡
```
module features

  use string_utils, only: lower_case
  use system_dependencies, only: WHIZARD_VERSION
⟨Features: dependencies⟩

⟨Standard module head⟩

⟨Features: public⟩

contains

⟨Features: procedures⟩

end module features
```

35.4.1 Output

⟨Features: public⟩≡
```
  public :: print_features
```

⟨Features: procedures⟩≡
```
  subroutine print_features ()
    print "(A)", "WHIZARD " // WHIZARD_VERSION
    print "(A)", "Build configuration:"
  ⟨Features: config⟩
    print "(A)", "Optional features available in this build:"
  ⟨Features: print⟩
  end subroutine print_features
```

35.4.2 Query function

⟨Features: procedures⟩+≡
```
  subroutine check (feature, recognized, result, help)
    character(*), intent(in) :: feature
    logical, intent(out) :: recognized
    character(*), intent(out) :: result, help
    recognized = .true.
    result = "no"
    select case (lower_case (trim (feature)))
  ⟨Features: cases⟩
    case default
       recognized = .false.
    end select
  end subroutine check
```

Print this result:

⟨*Features: procedures*⟩+≡
```
subroutine print_check (feature)
  character(*), intent(in) :: feature
  character(16) :: f
  logical :: recognized
  character(10) :: result
  character(48) :: help
  call check (feature, recognized, result, help)
  if (.not. recognized) then
     result = "unknown"
     help = ""
  end if
  f = feature
  print "(2x,A,1x,A,'(',A,')')", f, result, trim (help)
end subroutine print_check
```

35.4.3 Basic configuration

⟨*Features: config*⟩≡
```
call print_check ("precision")
```

⟨*Features: dependencies*⟩≡
```
use kinds, only: default
```

⟨*Features: cases*⟩≡
```
case ("precision")
   write (result, "(I0)")  precision (1._default)
   help = "significant decimals of real/complex numbers"
```

35.4.4 Optional features case by case

⟨*Features: print*⟩≡
```
call print_check ("OpenMP")
```

⟨*Features: dependencies*⟩+≡
```
use system_dependencies, only: openmp_is_active
```

⟨*Features: cases*⟩+≡
```
case ("openmp")
   if (openmp_is_active ()) then
      result = "yes"
   end if
   help = "OpenMP parallel execution"
```

⟨*Features: print*⟩+≡
```
call print_check ("GoSam")
```

⟨*Features: dependencies*⟩+≡
```
use system_dependencies, only: GOSAM_AVAILABLE
```

⟨*Features: cases*⟩+≡
```
case ("gosam")
   if (GOSAM_AVAILABLE) then
      result = "yes"
   end if
   help = "external NLO matrix element provider"
```

⟨*Features: print*⟩+≡
```
call print_check ("OpenLoops")
```

⟨*Features: dependencies*⟩+≡
```
use system_dependencies, only: OPENLOOPS_AVAILABLE
```

⟨*Features: cases*⟩+≡
```
case ("openloops")
   if (OPENLOOPS_AVAILABLE) then
      result = "yes"
   end if
   help = "external NLO matrix element provider"
```

⟨*Features: print*⟩+≡
```
call print_check ("Recola")
```

⟨*Features: dependencies*⟩+≡
```
use system_dependencies, only: RECOLA_AVAILABLE
```

⟨*Features: cases*⟩+≡
```
case ("recola")
   if (RECOLA_AVAILABLE) then
      result = "yes"
   end if
   help = "external NLO matrix element provider"
```

⟨*Features: print*⟩+≡
```
call print_check ("LHAPDF")
```

⟨*Features: dependencies*⟩+≡
```
use system_dependencies, only: LHAPDF5_AVAILABLE
use system_dependencies, only: LHAPDF6_AVAILABLE
```

⟨*Features: cases*⟩+≡
```
case ("lhapdf")
   if (LHAPDF5_AVAILABLE) then
      result = "v5"
   else if (LHAPDF6_AVAILABLE) then
      result = "v6"
   end if
   help = "PDF library"
```

⟨*Features: print*⟩+≡
```
call print_check ("HOPPET")
```

⟨*Features: dependencies*⟩+≡
```
use system_dependencies, only: HOPPET_AVAILABLE
```

⟨*Features: cases*⟩+≡
```
case ("hoppet")
   if (HOPPET_AVAILABLE) then
      result = "yes"
   end if
   help = "PDF evolution package"
```

⟨*Features: print*⟩+≡
```
call print_check ("fastjet")
```

⟨*Features: dependencies*⟩+≡
```
use jets, only: fastjet_available
```

⟨*Features: cases*⟩+≡
```
    case ("fastjet")
       if (fastjet_available ()) then
          result = "yes"
       end if
       help = "jet-clustering package"
```

⟨*Features: print*⟩+≡
```
    call print_check ("Pythia6")
```

⟨*Features: dependencies*⟩+≡
```
    use system_dependencies, only: PYTHIA6_AVAILABLE
```

⟨*Features: cases*⟩+≡
```
    case ("pythia6")
       if (PYTHIA6_AVAILABLE) then
          result = "yes"
       end if
       help = "direct access for shower/hadronization"
```

⟨*Features: print*⟩+≡
```
    call print_check ("Pythia8")
```

⟨*Features: dependencies*⟩+≡
```
    use system_dependencies, only: PYTHIA8_AVAILABLE
```

⟨*Features: cases*⟩+≡
```
    case ("pythia8")
       if (PYTHIA8_AVAILABLE) then
          result = "yes"
       end if
       help = "direct access for shower/hadronization"
```

⟨*Features: print*⟩+≡
```
    call print_check ("StdHEP")
```

⟨*Features: cases*⟩+≡
```
    case ("stdhep")
       result = "yes"
       help = "event I/O format"
```

⟨*Features: print*⟩+≡
```
    call print_check ("HepMC")
```

⟨*Features: dependencies*⟩+≡
```
    use hepmc_interface, only: hepmc_is_available
```

⟨*Features: cases*⟩+≡
```
    case ("hepmc")
       if (hepmc_is_available ()) then
          result = "yes"
       end if
       help = "event I/O format"
```

⟨*Features: print*⟩+≡
```
    call print_check ("LCIO")
```

⟨*Features: dependencies*⟩+≡
```
    use lcio_interface, only: lcio_is_available
```

⟨*Features: cases*⟩+≡

```
    case ("lcio")
       if (lcio_is_available ()) then
          result = "yes"
       end if
       help = "event I/O format"
```

⟨*Features: print*⟩+≡

```
    call print_check ("MetaPost")
```

⟨*Features: dependencies*⟩+≡

```
    use system_dependencies, only: EVENT_ANALYSIS
```

⟨*Features: cases*⟩+≡

```
    case ("metapost")
       result = EVENT_ANALYSIS
       help = "graphical event analysis via LaTeX/MetaPost"
```

35.5 Driver program

The main program handles command options, initializes the environment, and runs WHIZARD in a particular mode (interactive, file, standard input).

This is also used in the C interface:

⟨*Main: cmdline arg len declaration*⟩≡

```
integer, parameter :: CMDLINE_ARG_LEN = 1000
```

The actual main program:

⟨main.f90⟩≡

```
⟨File header⟩

program main

⟨Use strings⟩
  use system_dependencies
  use diagnostics
  use ifiles
  use os_interface
  use whizard

  use cmdline_options
  use features

  implicit none

⟨Main: cmdline arg len declaration⟩

!!! (WK 02/2016) Interface for the separate external routine below
  interface
     subroutine print_usage ()
     end subroutine print_usage
  end interface

! Main program variable declarations
  character(CMDLINE_ARG_LEN) :: arg
  character(2) :: option
  type(string_t) :: long_option, value
  integer :: i, j, arg_len, arg_status, area
  logical :: look_for_options
  logical :: interactive
  logical :: banner
  type(string_t) :: files, this, model, default_lib, library, libraries
  type(string_t) :: logfile
  logical :: user_code_enable = .false.
  integer :: n_user_src = 0, n_user_lib = 0
  type(string_t) :: user_src, user_lib, user_target
  type(paths_t) :: paths
  logical :: rebuild_library, rebuild_user
  logical :: rebuild_phs, rebuild_grids, rebuild_events
  logical :: recompile_library
  type(ifile_t) :: commands
  type(string_t) :: command
```

```
type(whizard_options_t), allocatable :: options
type(whizard_t), allocatable, target :: whizard_instance

! Exit status
logical :: quit = .false.
integer :: quit_code = 0

! Initial values
look_for_options = .true.
interactive = .false.
files = ""
model = "SM"
default_lib = "default_lib"
library = ""
libraries = ""
banner = .true.
logging = .true.
msg_level = RESULT
logfile = "whizard.log"
user_src = ""
user_lib = ""
user_target = ""
rebuild_library = .false.
rebuild_user = .false.
rebuild_phs = .false.
rebuild_grids = .false.
rebuild_events = .false.
recompile_library = .false.
call paths_init (paths)

! Read and process options
call init_options (print_usage)
i = 0
SCAN_CMDLINE: do
   i = i + 1
   call get_command_argument (i, arg, arg_len, arg_status)
   select case (arg_status)
   case (0)
   case (-1)
      call msg_error (" Command argument truncated: '" // arg // "'")
   case default
      exit SCAN_CMDLINE
   end select
   if (look_for_options) then
      select case (arg(1:2))
      case ("--")
         value = trim (arg)
         call split (value, long_option, "=")
         select case (char (long_option))
         case ("--version")
            call no_option_value (long_option, value)
            call print_version (); stop
         case ("--help")
            call no_option_value (long_option, value)
```

```
   call print_usage (); stop
case ("--prefix")
   paths%prefix = get_option_value (i, long_option, value)
   cycle scan_cmdline
case ("--exec-prefix")
   paths%exec_prefix = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--bindir")
   paths%bindir = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--libdir")
   paths%libdir = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--includedir")
   paths%includedir = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--datarootdir")
   paths%datarootdir = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--libtool")
   paths%libtool = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--lhapdfdir")
   paths%lhapdfdir = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--check")
   call print_usage ()
   call msg_fatal ("Option --check not supported &
        &(for unit tests, run whizard_ut instead)")
case ("--show-config")
   call no_option_value (long_option, value)
   call print_features (); stop
case ("--execute")
   command = get_option_value (i, long_option, value)
   call ifile_append (commands, command)
   cycle SCAN_CMDLINE
case ("--interactive")
   call no_option_value (long_option, value)
   interactive = .true.
   cycle SCAN_CMDLINE
case ("--library")
   library = get_option_value (i, long_option, value)
   libraries = libraries // " " // library
   cycle SCAN_CMDLINE
case ("--no-library")
   call no_option_value (long_option, value)
   default_lib = ""
   library = ""
   libraries = ""
   cycle SCAN_CMDLINE
case ("--localprefix")
   paths%localprefix = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--logfile")
```

```
   logfile = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--no-logfile")
   call no_option_value (long_option, value)
   logfile = ""
   cycle SCAN_CMDLINE
case ("--logging")
   call no_option_value (long_option, value)
   logging = .true.
   cycle SCAN_CMDLINE
case ("--no-logging")
   call no_option_value (long_option, value)
   logging = .false.
   cycle SCAN_CMDLINE
case ("--debug")
   call no_option_value (long_option, value)
   area = d_area (get_option_value (i, long_option, value))
   msg_level(area) = DEBUG
   cycle SCAN_CMDLINE
case ("--debug2")
   call no_option_value (long_option, value)
   area = d_area (get_option_value (i, long_option, value))
   msg_level(area) = DEBUG2
   cycle SCAN_CMDLINE
case ("--single-event")
   call no_option_value (long_option, value)
   single_event = .true.
   cycle SCAN_CMDLINE
case ("--banner")
   call no_option_value (long_option, value)
   banner = .true.
   cycle SCAN_CMDLINE
case ("--no-banner")
   call no_option_value (long_option, value)
   banner = .false.
   cycle SCAN_CMDLINE
case ("--model")
   model = get_option_value (i, long_option, value)
   cycle SCAN_CMDLINE
case ("--no-model")
   call no_option_value (long_option, value)
   model = ""
   cycle SCAN_CMDLINE
case ("--rebuild")
   call no_option_value (long_option, value)
   rebuild_library = .true.
   rebuild_user = .true.
   rebuild_phs = .true.
   rebuild_grids = .true.
   rebuild_events = .true.
   cycle SCAN_CMDLINE
case ("--no-rebuild")
   call no_option_value (long_option, value)
   rebuild_library = .false.
```

```
   recompile_library = .false.
   rebuild_user = .false.
   rebuild_phs = .false.
   rebuild_grids = .false.
   rebuild_events = .false.
   cycle SCAN_CMDLINE
case ("--rebuild-library")
   call no_option_value (long_option, value)
   rebuild_library = .true.
   cycle SCAN_CMDLINE
case ("--rebuild-user")
   call no_option_value (long_option, value)
   rebuild_user = .true.
   cycle SCAN_CMDLINE
case ("--rebuild-phase-space")
   call no_option_value (long_option, value)
   rebuild_phs = .true.
   cycle SCAN_CMDLINE
case ("--rebuild-grids")
   call no_option_value (long_option, value)
   rebuild_grids = .true.
   cycle SCAN_CMDLINE
case ("--rebuild-events")
   call no_option_value (long_option, value)
   rebuild_events = .true.
   cycle SCAN_CMDLINE
case ("--recompile")
   call no_option_value (long_option, value)
   recompile_library = .true.
   rebuild_grids = .true.
   cycle SCAN_CMDLINE
case ("--user")
   user_code_enable = .true.
   cycle SCAN_CMDLINE
case ("--user-src")
   if (user_src == "") then
      user_src = get_option_value (i, long_option, value)
   else
      user_src = user_src // " " &
           // get_option_value (i, long_option, value)
   end if
   n_user_src = n_user_src + 1
   cycle SCAN_CMDLINE
case ("--user-lib")
   if (user_lib == "") then
      user_lib = get_option_value (i, long_option, value)
   else
      user_lib = user_lib // " " &
           // get_option_value (i, long_option, value)
   end if
   n_user_lib = n_user_lib + 1
   cycle SCAN_CMDLINE
case ("--user-target")
   user_target = get_option_value (i, long_option, value)
```

```
      cycle SCAN_CMDLINE
   case ("--write-syntax-tables")
      call no_option_value (long_option, value)
      call init_syntax_tables ()
      call write_syntax_tables ()
      call final_syntax_tables ()
      stop
      cycle SCAN_CMDLINE
   case default
      call print_usage ()
      call msg_fatal ("Option '" // trim (arg) // "' not recognized")
   end select
end select
select case (arg(1:1))
case ("-")
   j = 1
   if (len_trim (arg) == 1) then
      look_for_options = .false.
   else
      SCAN_SHORT_OPTIONS: do
         j = j + 1
         if (j > len_trim (arg)) exit SCAN_SHORT_OPTIONS
         option = "-" // arg(j:j)
         select case (option)
         case ("-V")
            call print_version (); stop
         case ("-?", "-h")
            call print_usage (); stop
         case ("-e")
            command = get_option_value (i, var_str (option))
            call ifile_append (commands, command)
            cycle SCAN_CMDLINE
         case ("-i")
            interactive = .true.
            cycle SCAN_SHORT_OPTIONS
         case ("-l")
            if (j == len_trim (arg)) then
               library = get_option_value (i, var_str (option))
            else
               library = trim (arg(j+1:))
            end if
            libraries = libraries // " " // library
            cycle SCAN_CMDLINE
         case ("-L")
            if (j == len_trim (arg)) then
               logfile = get_option_value (i, var_str (option))
            else
               logfile = trim (arg(j+1:))
            end if
            cycle SCAN_CMDLINE
         case ("-m")
            if (j < len_trim (arg))  call msg_fatal &
                 ("Option '" // option // "' needs a value")
            model = get_option_value (i, var_str (option))
```

```
                 cycle SCAN_CMDLINE
              case ("-r")
                 rebuild_library = .true.
                 rebuild_user = .true.
                 rebuild_phs = .true.
                 rebuild_grids = .true.
                 rebuild_events = .true.
                 cycle SCAN_SHORT_OPTIONS
              case ("-u")
                 user_code_enable = .true.
                 cycle SCAN_SHORT_OPTIONS
              case default
                 call print_usage ()
                 call msg_fatal &
                      ("Option '" // option // "' not recognized")
              end select
           end do SCAN_SHORT_OPTIONS
        end if
     case default
        files = files // " " // trim (arg)
     end select
  else
     files = files // " " // trim (arg)
  end if
end do SCAN_CMDLINE

! Overall initialization
if (logfile /= "")  call logfile_init (logfile)
if (banner)  call msg_banner ()

 allocate (options)
 allocate (whizard_instance)

 if (.not. quit) then

    ! Set options and initialize the whizard object
    options%preload_model = model
    options%default_lib = default_lib
    options%preload_libraries = libraries
    options%rebuild_library = rebuild_library
    options%recompile_library = recompile_library
    options%rebuild_user = rebuild_user
    options%rebuild_phs = rebuild_phs
    options%rebuild_grids = rebuild_grids
    options%rebuild_events = rebuild_events

    call whizard_instance%init (options, paths, logfile)

    call mask_term_signals ()

 end if

 ! Run commands given on the command line
 if (.not. quit .and. ifile_get_length (commands) > 0) then
```

```
      call whizard_instance%process_ifile (commands, quit, quit_code)
   end if

   if (.not. quit) then
      ! Process commands from standard input
      if (.not. interactive .and. files == "") then
         call whizard_instance%process_stdin (quit, quit_code)

         ! ... or process commands from file
      else
         files = trim (adjustl (files))
         SCAN_FILES: do while (files /= "")
            call split (files, this, " ")
            call whizard_instance%process_file (this, quit, quit_code)
            if (quit)  exit SCAN_FILES
         end do SCAN_FILES

      end if
  end if

  ! Enter an interactive shell if requested
  if (.not. quit .and. interactive) then
     call whizard_instance%shell (quit_code)
  end if

  ! Overall finalization
  call ifile_final (commands)

  deallocate (options)

  call whizard_instance%final ()
  deallocate (whizard_instance)

  call terminate_now_if_signal ()
  call release_term_signals ()
  call msg_terminate (quit_code = quit_code)

!!!
```

```
subroutine print_usage ()
  use system_dependencies, only: WHIZARD_VERSION
  print "(A)", "WHIZARD " // WHIZARD_VERSION
  print "(A)", "Usage: whizard [OPTIONS] [FILE]"
  print "(A)", "Run WHIZARD with the command list taken from FILE(s)"
  print "(A)", "Options for resetting default directories and tools" &
          // "(GNU naming conventions):"
  print "(A)", "    --prefix DIR"
  print "(A)", "    --exec_prefix DIR"
  print "(A)", "    --bindir DIR"
  print "(A)", "    --libdir DIR"
  print "(A)", "    --includedir DIR"
  print "(A)", "    --datarootdir DIR"
  print "(A)", "    --libtool LOCAL_LIBTOOL"
  print "(A)", "    --lhapdfdir DIR   (PDF sets directory)"
  print "(A)", "Other options:"
  print "(A)", "-h, --help            display this help and exit"
  print "(A)", "    --banner          display banner at startup (default)"
  print "(A)", "    --debug AREA      switch on debug output for AREA."
  print "(A)", "                      AREA can be one of Whizard's src dirs or 'all'"
  print "(A)", "    --debug2 AREA     switch on more verbose debug output for AREA."
  print "(A)", "    --single-event    only compute one phase-space point (for debugging)"
  print "(A)", "-e, --execute CMDS    execute SINDARIN CMDS before reading FILE(s)"
  print "(A)", "-i, --interactive     run interactively after reading FILE(s)"
  print "(A)", "-l, --library         preload process library NAME"
  print "(A)", "    --localprefix DIR"
  print "(A)", "                      search in DIR for local models (default: ~/.whizard)"
  print "(A)", "-L, --logfile FILE    write log to FILE (default: 'whizard.log'"
  print "(A)", "    --logging         switch on logging at startup (default)"
  print "(A)", "-m, --model NAME      preload model NAME (default: 'SM')"
  print "(A)", "    --no-banner       do not display banner at startup"
  print "(A)", "    --no-library      do not preload process library"
  print "(A)", "    --no-logfile      do not write a logfile"
  print "(A)", "    --no-logging      switch off logging at startup"
  print "(A)", "    --no-model        do not preload a model"
  print "(A)", "    --no-rebuild      do not force rebuilding"
  print "(A)", "-r, --rebuild         rebuild all (see below)"
  print "(A)", "    --rebuild-library"
  print "(A)", "                      rebuild process code library"
  print "(A)", "    --rebuild-user    rebuild user-provided code"
  print "(A)", "    --rebuild-phase-space"
  print "(A)", "                      rebuild phase-space configuration"
  print "(A)", "    --rebuild-grids   rebuild integration grids"
  print "(A)", "    --rebuild-events  rebuild event samples"
  print "(A)", "    --recompile       recompile process code"
  print "(A)", "    --show-config     show build-time configuration"
  print "(A)", "-u  --user            enable user-provided code"
  print "(A)", "    --user-src FILE   user-provided source file"
  print "(A)", "    --user-lib FILE   user-provided library file"
  print "(A)", "    --user-target BN  basename of created user library (default: user)"
  print "(A)", "-V, --version         output version information and exit"
  print "(A)", "    --write-syntax-tables"
  print "(A)", "                      write the internal syntax tables to files and exit"
```

```
  print "(A)", "-                      further options are taken as filenames"
  print *
  print "(A)", "With no FILE, read standard input."
end subroutine print_usage
```

35.6 Driver program for the unit tests

This is a variant of the above main program that takes unit-test names as command-line options and runs those tests.

⟨main_ut.f90⟩≡

```
⟨File header⟩

program main_ut

⟨Use strings⟩
  use unit_tests
  use io_units
  use system_dependencies
  use diagnostics
  use os_interface

  use cmdline_options

  use model_testbed !NODEP!

⟨Main: use tests⟩

  implicit none

⟨Main: cmdline arg len declaration⟩

!!! (WK 02/2016) Interface for the separate external routine below
  interface
     subroutine print_usage ()
     end subroutine print_usage
  end interface

  ! Main program variable declarations
  character(CMDLINE_ARG_LEN) :: arg
  character(2) :: option
  type(string_t) :: long_option, value
  integer :: i, j, arg_len, arg_status, area
  logical :: look_for_options
  logical :: banner
  type(string_t) :: check, checks
  type(test_results_t) :: test_results
  logical :: success

  ! Exit status
  integer :: quit_code = 0

  ! Initial values
  look_for_options = .true.
  banner = .true.
  logging = .false.
  msg_level = RESULT
  check = ""
  checks = ""
```

```
! Read and process options
call init_options (print_usage)
i = 0
SCAN_CMDLINE: do
   i = i + 1
   call get_command_argument (i, arg, arg_len, arg_status)
   select case (arg_status)
   case (0)
   case (-1)
      call msg_error (" Command argument truncated: '" // arg // "'")
   case default
      exit SCAN_CMDLINE
   end select
   if (look_for_options) then
      select case (arg(1:2))
      case ("--")
         value = trim (arg)
         call split (value, long_option, "=")
         select case (char (long_option))
         case ("--version")
            call no_option_value (long_option, value)
            call print_version (); stop
         case ("--help")
            call no_option_value (long_option, value)
            call print_usage (); stop
         case ("--banner")
            call no_option_value (long_option, value)
            banner = .true.
            cycle SCAN_CMDLINE
         case ("--no-banner")
            call no_option_value (long_option, value)
            banner = .false.
            cycle SCAN_CMDLINE
         case ("--check")
            check = get_option_value (i, long_option, value)
            checks = checks // " " // check
            cycle SCAN_CMDLINE
         case ("--debug")
            call no_option_value (long_option, value)
            area = d_area (get_option_value (i, long_option, value))
            msg_level(area) = DEBUG
            cycle SCAN_CMDLINE
         case ("--debug2")
            call no_option_value (long_option, value)
            area = d_area (get_option_value (i, long_option, value))
            msg_level(area) = DEBUG
            cycle SCAN_CMDLINE
         case default
            call print_usage ()
            call msg_fatal ("Option '" // trim (arg) // "' not recognized")
         end select
      end select
      select case (arg(1:1))
      case ("-")
```

```
            j = 1
            if (len_trim (arg) == 1) then
               look_for_options = .false.
            else
               SCAN_SHORT_OPTIONS: do
                  j = j + 1
                  if (j > len_trim (arg)) exit SCAN_SHORT_OPTIONS
                  option = "-" // arg(j:j)
                  select case (option)
                  case ("-V")
                     call print_version (); stop
                  case ("-?", "-h")
                     call print_usage (); stop
                  case default
                     call print_usage ()
                     call msg_fatal &
                          ("Option '" // option // "' not recognized")
                  end select
               end do SCAN_SHORT_OPTIONS
            end if
         case default
            call print_usage ()
            call msg_fatal ("Option '" // trim (arg) // "' not recognized")
         end select
      else
         call print_usage ()
         call msg_fatal ("Option '" // trim (arg) // "' not recognized")
      end if
   end do SCAN_CMDLINE

   ! Overall initialization
   if (banner)  call msg_banner ()

    ! Run any self-checks (and no commands)
    if (checks /= "") then
       checks = trim (adjustl (checks))
       RUN_CHECKS: do while (checks /= "")
          call split (checks, check, " ")
          call whizard_check (check, test_results)
       end do RUN_CHECKS
       call test_results%wrapup (6, success)
       if (.not. success)  quit_code = 7
    end if

    call msg_terminate (quit_code = quit_code)

!!!
```

```
    print "(A)", "warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE."
    print *
  end subroutine print_version
```

⟨*Main: tests*⟩

```
end program main_ut

!!! (WK 02/2016)
!!! Separate subroutine, because this becomes a procedure pointer target
!!! Internal procedures as targets are not supported by some compilers.

  subroutine print_usage ()
    use system_dependencies, only: WHIZARD_VERSION
    print "(A)", "WHIZARD " // WHIZARD_VERSION // " (unit test driver)"
    print "(A)", "Usage: whizard_ut [OPTIONS] [FILE]"
    print "(A)", "Run WHIZARD unit tests as given on the command line"
    print "(A)", "Options:"
    print "(A)", "-h, --help            display this help and exit"
    print "(A)", "    --banner          display banner at startup (default)"
    print "(A)", "    --no-banner       do not display banner at startup"
    print "(A)", "    --debug AREA      switch on debug output for AREA."
    print "(A)", "                      AREA can be one of Whizard's src dirs or 'all'"
    print "(A)", "    --debug2 AREA     switch on more verbose debug output for AREA."
    print "(A)", "-V, --version         output version information and exit"
    print "(A)", "    --check TEST      run unit test TEST"
  end subroutine print_usage
```

35.6.1 Self-tests

For those self-tests, we need some auxiliary routines that provide an enviroment. The environment depends on things that are not available at the level of the module that we want to test.

Testbed for event I/O

This subroutine prepares a test process with a single event. All objects are allocated via anonymous pointers, because we want to recover the pointers and delete the objects in a separate procedure.

⟨*Main: tests*⟩≡

```
  subroutine prepare_eio_test (event, unweighted, n_alt)
    use variables, only: var_list_t
    use model_data
    use process, only: process_t
    use instances, only: process_instance_t
    use processes_ut, only: prepare_test_process
    use event_base
    use events

    class(generic_event_t), intent(inout), pointer :: event
    logical, intent(in), optional :: unweighted
    integer, intent(in), optional :: n_alt
    type(model_data_t), pointer :: model
```

```
    type(var_list_t) :: var_list
    type(process_t), pointer :: proc
    type(process_instance_t), pointer :: process_instance

    allocate (model)
    call model%init_test ()

    allocate (proc)
    allocate (process_instance)

    call prepare_test_process (proc, process_instance, model)
    call process_instance%setup_event_data ()

    call model%final ()
    deallocate (model)

    allocate (event_t :: event)
    select type (event)
    type is (event_t)
       if (present (unweighted)) then
          call var_list%append_log (&
               var_str ("?unweighted"), unweighted, &
               intrinsic = .true.)
       else
          call var_list%append_log (&
               var_str ("?unweighted"), .true., &
               intrinsic = .true.)
       end if
       call var_list%append_string (&
            var_str ("$sample_normalization"), &
            var_str ("auto"), intrinsic = .true.)
       call event%basic_init (var_list, n_alt)
       call event%connect (process_instance, proc%get_model_ptr ())
       call var_list%final ()
    end select

  end subroutine prepare_eio_test
```

Recover those pointers, finalize the objects and deallocate.

⟨*Main: tests*⟩+≡

```
  subroutine cleanup_eio_test (event)
    use model_data
    use process, only: process_t
    use instances, only: process_instance_t
    use processes_ut, only: cleanup_test_process
    use event_base
    use events

    class(generic_event_t), intent(inout), pointer :: event
    type(process_t), pointer :: proc
    type(process_instance_t), pointer :: process_instance

    select type (event)
    type is (event_t)
```

```
      proc => event%get_process_ptr ()
      process_instance => event%get_process_instance_ptr ()
      call cleanup_test_process (proc, process_instance)
      deallocate (process_instance)
      deallocate (proc)
      call event%final ()
    end select
    deallocate (event)

  end subroutine cleanup_eio_test
```

Assign those procedures to appropriate pointers (module variables) in the `eio_base` module, so they can be called as if they were module procedures.

⟨*Main: use tests*⟩≡

```
  use eio_base_ut, only: eio_prepare_test
  use eio_base_ut, only: eio_cleanup_test
```

⟨*Main: prepare testbed*⟩≡

```
  eio_prepare_test => prepare_eio_test
  eio_cleanup_test => cleanup_eio_test
```

Any Model

This procedure reads any model from file and, optionally, assigns a var-list pointer.

⟨*Main: tests*⟩+≡

```
  subroutine prepare_whizard_model (model, name, vars)
  ⟨Use strings⟩
    use os_interface
    use model_data
    use var_base
    use models
    class(model_data_t), intent(inout), pointer :: model
    type(string_t), intent(in) :: name
    class(vars_t), pointer, intent(out), optional :: vars
    type(os_data_t) :: os_data
    call syntax_model_file_init ()
    call os_data_init (os_data)
    allocate (model_t :: model)
    select type (model)
    type is (model_t)
      call model%read (name // ".mdl", os_data)
      if (present (vars)) then
        vars => model%get_var_list_ptr ()
      end if
    end select
  end subroutine prepare_whizard_model
```

Cleanup after use. Includes deletion of the model-file syntax.

⟨*Main: tests*⟩+≡

```
  subroutine cleanup_whizard_model (model)
    use model_data
    use models
```

```
    class(model_data_t), intent(inout), pointer :: model
    call model%final ()
    deallocate (model)
    call syntax_model_file_final ()
  end subroutine cleanup_whizard_model
```

Assign those procedures to appropriate pointers (module variables) in the `model_testbed` module, so they can be called as if they were module procedures.

⟨*Main: prepare testbed*⟩+≡

```
  prepare_model => prepare_whizard_model
  cleanup_model => cleanup_whizard_model
```

Fallback model: hadrons

Some event format tests require the hadronic SM implementation, which has to be read from file. We provide the functionality here, so the tests do not depend on model I/O.

⟨*Main: tests*⟩+≡

```
  subroutine prepare_fallback_model (model)
    use model_data
    class(model_data_t), intent(inout), pointer :: model
    call prepare_whizard_model (model, var_str ("SM_hadrons"))
  end subroutine prepare_fallback_model
```

Assign those procedures to appropriate pointers (module variables) in the `eio_base` module, so they can be called as if they were module procedures.

⟨*Main: use tests*⟩+≡

```
  use eio_base_ut, only: eio_prepare_fallback_model
  use eio_base_ut, only: eio_cleanup_fallback_model
```

⟨*Main: prepare testbed*⟩+≡

```
  eio_prepare_fallback_model => prepare_fallback_model
  eio_cleanup_fallback_model => cleanup_model
```

Access to the test random-number generator

This generator is not normally available for the dispatcher. We assign an additional dispatch routine to the hook in the `dispatch` module which will be checked before the default rule.

⟨*Main: use tests*⟩+≡

```
  use dispatch_rng, only: dispatch_rng_factory_extra
  use dispatch_rng_ut, only: dispatch_rng_factory_test
```

⟨*Main: prepare testbed*⟩+≡

```
  dispatch_rng_factory_extra => dispatch_rng_factory_test
```

Access to the test structure functions

These are not normally available for the dispatcher. We assign an additional dispatch routine to the hook in the `dispatch` module which will be checked before the default rule.

⟨*Main: use tests*⟩+≡

```
use dispatch_beams, only: dispatch_sf_data_extra
use dispatch_ut, only: dispatch_sf_data_test
```

⟨*Main: prepare testbed*⟩+≡

```
dispatch_sf_data_extra => dispatch_sf_data_test
```

Procedure for Checking

This is for developers only, but needs a well-defined interface.

⟨*Main: tests*⟩+≡

```
subroutine whizard_check (check, results)
  type(string_t), intent(in) :: check
  type(test_results_t), intent(inout) :: results
  type(os_data_t) :: os_data
  integer :: u
  call os_data_init (os_data)
  u = free_unit ()
  open (u, file="whizard_check." // char (check) // ".log", &
       action="write", status="replace")
  call msg_message (repeat ('=', 76), 0)
  call msg_message ("Running self-test: " // char (check), 0)
  call msg_message (repeat ('-', 76), 0)
⟨Main: prepare testbed⟩
  select case (char (check))
⟨Main: test cases⟩
  case ("all")
   ⟨Main: all tests⟩
  case default
     call msg_fatal ("Self-test '" // char (check) // "' not implemented.")
  end select
  close (u)
end subroutine whizard_check
```

35.6.2 Unit test references

Formats

⟨*Main: use tests*⟩+≡

```
use formats_ut, only: format_test
```

⟨*Main: test cases*⟩≡

```
case ("formats")
   call format_test (u, results)
```

⟨*Main: all tests*⟩≡

```
call format_test (u, results)
```

MD5

⟨*Main: use tests*⟩+≡

```
use md5_ut, only: md5_test
```

⟨*Main: test cases*⟩+≡

```
case ("md5")
   call md5_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call md5_test (u, results)
```

OS Interface

⟨*Main: use tests*⟩+≡
```
use os_interface_ut, only: os_interface_test
```

⟨*Main: test cases*⟩+≡
```
case ("os_interface")
   call os_interface_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call os_interface_test (u, results)
```

Sorting

⟨*Main: use tests*⟩+≡
```
use sorting_ut, only: sorting_test
```

⟨*Main: test cases*⟩+≡
```
case ("sorting")
   call sorting_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sorting_test (u, results)
```

Grids

⟨*Main: use tests*⟩+≡
```
use grids_ut, only: grids_test
```

⟨*Main: test cases*⟩+≡
```
case ("grids")
   call grids_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call grids_test (u, results)
```

Solver

⟨*Main: use tests*⟩+≡
```
use solver_ut, only: solver_test
```

⟨*Main: test cases*⟩+≡
```
case ("solver")
   call solver_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call solver_test (u, results)
```

CPU Time

⟨*Main: use tests*⟩+≡
```
use cputime_ut, only: cputime_test
```

⟨*Main: test cases*⟩+≡
```
case ("cputime")
   call cputime_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call cputime_test (u, results)
```

SM QCD

⟨*Main: use tests*⟩+≡
```
use sm_qcd_ut, only: sm_qcd_test
```
⟨*Main: test cases*⟩+≡
```
case ("sm_qcd")
   call sm_qcd_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call sm_qcd_test (u, results)
```

SM physics

⟨*Main: use tests*⟩+≡
```
use sm_physics_ut, only: sm_physics_test
```
⟨*Main: test cases*⟩+≡
```
case ("sm_physics")
   call sm_physics_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call sm_physics_test (u, results)
```

Lexers

⟨*Main: use tests*⟩+≡
```
use lexers_ut, only: lexer_test
```
⟨*Main: test cases*⟩+≡
```
case ("lexers")
   call lexer_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call lexer_test (u, results)
```

Parser

⟨*Main: use tests*⟩+≡
```
use parser_ut, only: parse_test
```
⟨*Main: test cases*⟩+≡
```
case ("parser")
   call parse_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call parse_test (u, results)
```

XML

⟨*Main: use tests*⟩+≡
```
use xml_ut, only: xml_test
```

⟨*Main: test cases*⟩+≡
```
case ("xml")
   call xml_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call xml_test (u, results)
```

Colors

⟨*Main: use tests*⟩+≡
```
use colors_ut, only: color_test
```
⟨*Main: test cases*⟩+≡
```
case ("colors")
   call color_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call color_test (u, results)
```

State matrices

⟨*Main: use tests*⟩+≡
```
use state_matrices_ut, only: state_matrix_test
```
⟨*Main: test cases*⟩+≡
```
case ("state_matrices")
   call state_matrix_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call state_matrix_test (u, results)
```

Analysis

⟨*Main: use tests*⟩+≡
```
use analysis_ut, only: analysis_test
```
⟨*Main: test cases*⟩+≡
```
case ("analysis")
   call analysis_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call analysis_test (u, results)
```

Particles

⟨*Main: use tests*⟩+≡
```
use particles_ut, only: particles_test
```
⟨*Main: test cases*⟩+≡
```
case ("particles")
   call particles_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call particles_test (u, results)
```

Models

⟨*Main: use tests*⟩+≡
```
use models_ut, only: models_test
```

⟨*Main: test cases*⟩+≡
```
case ("models")
   call models_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call models_test (u, results)
```

Auto Components

⟨*Main: use tests*⟩+≡
```
use auto_components_ut, only: auto_components_test
```

⟨*Main: test cases*⟩+≡
```
case ("auto_components")
   call auto_components_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call auto_components_test (u, results)
```

Radiation Generator

⟨*Main: use tests*⟩+≡
```
use radiation_generator_ut, only: radiation_generator_test
```

⟨*Main: test cases*⟩+≡
```
case ("radiation_generator")
   call radiation_generator_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call radiation_generator_test (u, results)
```

35.6.3 BLHA

⟨*Main: use tests*⟩+≡
```
use blha_ut, only: blha_test
```

⟨*Main: test cases*⟩+≡
```
case ("blha")
   call blha_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call blha_test (u, results)
```

Evaluators

⟨*Main: use tests*⟩+≡
```
use evaluators_ut, only: evaluator_test
```

⟨*Main: test cases*⟩+≡
```
case ("evaluators")
   call evaluator_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call evaluator_test (u, results)
```

Expressions

⟨*Main: use tests*⟩+≡
```
use eval_trees_ut, only: expressions_test
```

⟨*Main: test cases*⟩+≡
```
case ("expressions")
   call expressions_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call expressions_test (u, results)
```

Resonances

⟨*Main: use tests*⟩+≡
```
use resonances_ut, only: resonances_test
```

⟨*Main: test cases*⟩+≡
```
case ("resonances")
   call resonances_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call resonances_test (u, results)
```

PHS Trees

⟨*Main: use tests*⟩+≡
```
use phs_trees_ut, only: phs_trees_test
```

⟨*Main: test cases*⟩+≡
```
case ("phs_trees")
   call phs_trees_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call phs_trees_test (u, results)
```

PHS Forests

⟨*Main: use tests*⟩+≡
```
use phs_forests_ut, only: phs_forests_test
```

⟨*Main: test cases*⟩+≡
```
case ("phs_forests")
   call phs_forests_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call phs_forests_test (u, results)
```

Beams

⟨*Main: use tests*⟩+≡
```
use beams_ut, only: beams_test
```

⟨*Main: test cases*⟩+≡
```
case ("beams")
   call beams_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call beams_test (u, results)
```

$su(N)$ Algebra

⟨*Main: use tests*⟩+≡
```
use su_algebra_ut, only: su_algebra_test
```

⟨*Main: test cases*⟩+≡
```
case ("su_algebra")
   call su_algebra_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call su_algebra_test (u, results)
```

Bloch Vectors

⟨*Main: use tests*⟩+≡
```
use bloch_vectors_ut, only: bloch_vectors_test
```

⟨*Main: test cases*⟩+≡
```
case ("bloch_vectors")
   call bloch_vectors_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call bloch_vectors_test (u, results)
```

Polarizations

⟨*Main: use tests*⟩+≡
```
use polarizations_ut, only: polarizations_test
```

⟨*Main: test cases*⟩+≡
```
case ("polarizations")
   call polarizations_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call polarizations_test (u, results)
```

SF Aux

⟨*Main: use tests*⟩+≡
```
use sf_aux_ut, only: sf_aux_test
```

⟨*Main: test cases*⟩+≡
```
case ("sf_aux")
   call sf_aux_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sf_aux_test (u, results)
```

SF Mappings

⟨*Main: use tests*⟩+≡
```
use sf_mappings_ut, only: sf_mappings_test
```

⟨*Main: test cases*⟩+≡
```
case ("sf_mappings")
   call sf_mappings_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sf_mappings_test (u, results)
```

SF Base

⟨*Main: use tests*⟩+≡
```
use sf_base_ut, only: sf_base_test
```

⟨*Main: test cases*⟩+≡
```
case ("sf_base")
   call sf_base_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sf_base_test (u, results)
```

SF PDF Builtin

⟨*Main: use tests*⟩+≡
```
use sf_pdf_builtin_ut, only: sf_pdf_builtin_test
```

⟨*Main: test cases*⟩+≡
```
case ("sf_pdf_builtin")
   call sf_pdf_builtin_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sf_pdf_builtin_test (u, results)
```

SF LHAPDF

⟨*Main: use tests*⟩+≡
```
use sf_lhapdf_ut, only: sf_lhapdf_test
```

⟨*Main: test cases*⟩+≡
```
case ("sf_lhapdf")
   call sf_lhapdf_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sf_lhapdf_test (u, results)
```

SF ISR

⟨*Main: use tests*⟩+≡
```
use sf_isr_ut, only: sf_isr_test
```

⟨*Main: test cases*⟩+≡
```
case ("sf_isr")
   call sf_isr_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sf_isr_test (u, results)
```

SF EPA

⟨*Main: use tests*⟩+≡
```
use sf_epa_ut, only: sf_epa_test
```

⟨*Main: test cases*⟩+≡
```
case ("sf_epa")
   call sf_epa_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sf_epa_test (u, results)
```

SF EWA

⟨*Main: use tests*⟩+≡
```
use sf_ewa_ut, only: sf_ewa_test
```

⟨*Main: test cases*⟩+≡
```
case ("sf_ewa")
   call sf_ewa_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sf_ewa_test (u, results)
```

SF CIRCE1

⟨*Main: use tests*⟩+≡
```
use sf_circe1_ut, only: sf_circe1_test
```

⟨*Main: test cases*⟩+≡
```
case ("sf_circe1")
   call sf_circe1_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sf_circe1_test (u, results)
```

SF CIRCE2

⟨*Main: use tests*⟩+≡
```
use sf_circe2_ut, only: sf_circe2_test
```

⟨*Main: test cases*⟩+≡
```
case ("sf_circe2")
   call sf_circe2_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sf_circe2_test (u, results)
```

SF Gaussian

⟨*Main: use tests*⟩+≡
```
use sf_gaussian_ut, only: sf_gaussian_test
```

⟨*Main: test cases*⟩+≡
```
case ("sf_gaussian")
   call sf_gaussian_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sf_gaussian_test (u, results)
```

SF Beam Events

⟨*Main: use tests*⟩+≡
```
use sf_beam_events_ut, only: sf_beam_events_test
```

⟨*Main: test cases*⟩+≡
```
case ("sf_beam_events")
   call sf_beam_events_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sf_beam_events_test (u, results)
```

SF EScan

⟨*Main: use tests*⟩+≡
```
use sf_escan_ut, only: sf_escan_test
```

⟨*Main: test cases*⟩+≡
```
case ("sf_escan")
   call sf_escan_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call sf_escan_test (u, results)
```

PHS Base

⟨*Main: use tests*⟩+≡
```
use phs_base_ut, only: phs_base_test
```

⟨*Main: test cases*⟩+≡
```
case ("phs_base")
   call phs_base_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call phs_base_test (u, results)
```

PHS Single

⟨*Main: use tests*⟩+≡
```
use phs_single_ut, only: phs_single_test
```

⟨*Main: test cases*⟩+≡
```
case ("phs_single")
   call phs_single_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call phs_single_test (u, results)
```

PHS Wood

⟨*Main: use tests*⟩+≡
```
use phs_wood_ut, only: phs_wood_test
use phs_wood_ut, only: phs_wood_vis_test
```

⟨*Main: test cases*⟩+≡
```
case ("phs_wood")
   call phs_wood_test (u, results)
case ("phs_wood_vis")
   call phs_wood_vis_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call phs_wood_test (u, results)
call phs_wood_vis_test (u, results)
```

PHS FKS Generator

⟨*Main: use tests*⟩+≡
```
use phs_fks_ut, only: phs_fks_generator_test
```

⟨*Main: test cases*⟩+≡
```
case ("phs_fks_generator")
   call phs_fks_generator_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call phs_fks_generator_test (u, results)
```

FKS regions

⟨*Main: use tests*⟩+≡
```
use fks_regions_ut, only: fks_regions_test
```
⟨*Main: test cases*⟩+≡
```
case ("fks_regions")
   call fks_regions_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call fks_regions_test (u, results)
```

RECOLA

⟨*Main: use tests*⟩+≡
```
use prc_recola_ut, only: prc_recola_test
```
⟨*Main: test cases*⟩+≡
```
case ("prc_recola")
   call prc_recola_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call prc_recola_test (u, results)
```

NLO Color data

⟨*Main: use tests*⟩+≡
```
use nlo_color_data_ut, only: nlo_color_data_test
```
⟨*Main: test cases*⟩+≡
```
case ("nlo_color_data")
   call nlo_color_data_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call nlo_color_data_test (u, results)
```

RNG Base

⟨*Main: use tests*⟩+≡
```
use rng_base_ut, only: rng_base_test
```
⟨*Main: test cases*⟩+≡
```
case ("rng_base")
   call rng_base_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call rng_base_test (u, results)
```

RNG Tao

⟨*Main: use tests*⟩+≡
```
use rng_tao_ut, only: rng_tao_test
```

⟨*Main: test cases*⟩+≡
```
case ("rng_tao")
   call rng_tao_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call rng_tao_test (u, results)
```

RNG Stream

⟨*Main: use tests*⟩+≡
```
use rng_stream_ut, only: rng_stream_test
```
⟨*Main: test cases*⟩+≡
```
case ("rng_stream")
   call rng_stream_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call rng_stream_test (u, results)
```

Selectors

⟨*Main: use tests*⟩+≡
```
use selectors_ut, only: selectors_test
```
⟨*Main: test cases*⟩+≡
```
case ("selectors")
   call selectors_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call selectors_test (u, results)
```

VEGAS

⟨*Main: use tests*⟩+≡
```
use vegas_ut, only: vegas_test
```
⟨*Main: test cases*⟩+≡
```
case ("vegas")
   call vegas_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call vegas_test (u, results)
```

VAMP2

⟨*Main: use tests*⟩+≡
```
use vamp2_ut, only: vamp2_test
```
⟨*Main: test cases*⟩+≡
```
case ("vamp2")
   call vamp2_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call vamp2_test (u, results)
```

MCI Base

⟨*Main: use tests*⟩+≡
```
use mci_base_ut, only: mci_base_test
```

⟨*Main: test cases*⟩+≡
```
case ("mci_base")
   call mci_base_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call mci_base_test (u, results)
```

MCI Midpoint

⟨*Main: use tests*⟩+≡
```
use mci_midpoint_ut, only: mci_midpoint_test
```
⟨*Main: test cases*⟩+≡
```
case ("mci_midpoint")
   call mci_midpoint_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call mci_midpoint_test (u, results)
```

MCI VAMP

⟨*Main: use tests*⟩+≡
```
use mci_vamp_ut, only: mci_vamp_test
```
⟨*Main: test cases*⟩+≡
```
case ("mci_vamp")
   call mci_vamp_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call mci_vamp_test (u, results)
```

MCI VEGAS

⟨*Main: use tests*⟩+≡
```
use mci_vamp2_ut, only: mci_vamp2_test
```
⟨*Main: test cases*⟩+≡
```
case ("mci_vamp2")
   call mci_vamp2_test (u, results)
```
⟨*Main: all tests vamp2*⟩≡
```
call mci_vamp2_test (u, results)
```

PRCLib Interfaces

⟨*Main: use tests*⟩+≡
```
use prclib_interfaces_ut, only: prclib_interfaces_test
```
⟨*Main: test cases*⟩+≡
```
case ("prclib_interfaces")
   call prclib_interfaces_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call prclib_interfaces_test (u, results)
```

Particle Specifiers

⟨*Main: use tests*⟩+≡
```
use particle_specifiers_ut, only: particle_specifiers_test
```

⟨*Main: test cases*⟩+≡
```
case ("particle_specifiers")
   call particle_specifiers_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call particle_specifiers_test (u, results)
```

Process Libraries

⟨*Main: use tests*⟩+≡
```
use process_libraries_ut, only: process_libraries_test
```
⟨*Main: test cases*⟩+≡
```
case ("process_libraries")
   call process_libraries_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call process_libraries_test (u, results)
```

PRCLib Stacks

⟨*Main: use tests*⟩+≡
```
use prclib_stacks_ut, only: prclib_stacks_test
```
⟨*Main: test cases*⟩+≡
```
case ("prclib_stacks")
   call prclib_stacks_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call prclib_stacks_test (u, results)
```

HepMC

⟨*Main: use tests*⟩+≡
```
use hepmc_interface_ut, only: hepmc_interface_test
```
⟨*Main: test cases*⟩+≡
```
case ("hepmc")
   call hepmc_interface_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call hepmc_interface_test (u, results)
```

LCIO

⟨*Main: use tests*⟩+≡
```
use lcio_interface_ut, only: lcio_interface_test
```
⟨*Main: test cases*⟩+≡
```
case ("lcio")
   call lcio_interface_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call lcio_interface_test (u, results)
```

Jets

⟨*Main: use tests*⟩+≡
```
use jets_ut, only: jets_test
```

⟨*Main: test cases*⟩+≡
```
case ("jets")
   call jets_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call jets_test (u, results)
```

PDG Arrays

⟨*Main: use tests*⟩+≡
```
use pdg_arrays_ut, only: pdg_arrays_test
```
⟨*Main: test cases*⟩+≡
```
case ("pdg_arrays")
   call pdg_arrays_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call pdg_arrays_test (u, results)
```

interactions

⟨*Main: use tests*⟩+≡
```
use interactions_ut, only: interaction_test
```
⟨*Main: test cases*⟩+≡
```
case ("interactions")
   call interaction_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call interaction_test (u, results)
```

SLHA

⟨*Main: use tests*⟩+≡
```
use slha_interface_ut, only: slha_test
```
⟨*Main: test cases*⟩+≡
```
case ("slha_interface")
   call slha_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call slha_test (u, results)
```

Cascades

⟨*Main: use tests*⟩+≡
```
use cascades_ut, only: cascades_test
```
⟨*Main: test cases*⟩+≡
```
case ("cascades")
   call cascades_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call cascades_test (u, results)
```

PRC Test

⟨*Main: use tests*⟩+≡
```
use prc_test_ut, only: prc_test_test
```

⟨*Main: test cases*⟩+≡
```
case ("prc_test")
   call prc_test_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call prc_test_test (u, results)
```

PRC Template ME

⟨*Main: use tests*⟩+≡
```
use prc_template_me_ut, only: prc_template_me_test
```
⟨*Main: test cases*⟩+≡
```
case ("prc_template_me")
   call prc_template_me_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call prc_template_me_test (u, results)
```

PRC OMega

⟨*Main: use tests*⟩+≡
```
use prc_omega_ut, only: prc_omega_test
use prc_omega_ut, only: prc_omega_diags_test
```
⟨*Main: test cases*⟩+≡
```
case ("prc_omega")
   call prc_omega_test (u, results)
case ("prc_omega_diags")
   call prc_omega_diags_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call prc_omega_test (u, results)
call prc_omega_diags_test (u, results)
```

Parton States

⟨*Main: use tests*⟩+≡
```
use parton_states_ut, only: parton_states_test
```
⟨*Main: test cases*⟩+≡
```
case ("parton_states")
   call parton_states_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call parton_states_test (u, results)
```

Subevt Expr

⟨*Main: use tests*⟩+≡
```
use expr_tests_ut, only: subevt_expr_test
```
⟨*Main: test cases*⟩+≡
```
case ("subevt_expr")
   call subevt_expr_test (u, results)
```
⟨*Main: all tests*⟩+≡
```
call subevt_expr_test (u, results)
```

Processes

⟨*Main: use tests*⟩+≡
```
use processes_ut, only: processes_test
```

⟨*Main: test cases*⟩+≡
```
case ("processes")
   call processes_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call processes_test (u, results)
```

Process Stacks

⟨*Main: use tests*⟩+≡
```
use process_stacks_ut, only: process_stacks_test
```

⟨*Main: test cases*⟩+≡
```
case ("process_stacks")
   call process_stacks_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call process_stacks_test (u, results)
```

Event Transforms

⟨*Main: use tests*⟩+≡
```
use event_transforms_ut, only: event_transforms_test
```

⟨*Main: test cases*⟩+≡
```
case ("event_transforms")
   call event_transforms_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call event_transforms_test (u, results)
```

Decays

⟨*Main: use tests*⟩+≡
```
use decays_ut, only: decays_test
```

⟨*Main: test cases*⟩+≡
```
case ("decays")
   call decays_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call decays_test (u, results)
```

Shower

⟨*Main: use tests*⟩+≡
```
use shower_ut, only: shower_test
```

⟨*Main: test cases*⟩+≡
```
case ("shower")
   call shower_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call shower_test (u, results)
```

Events

⟨*Main: use tests*⟩+≡
```
use events_ut, only: events_test
```

⟨*Main: test cases*⟩+≡
```
case ("events")
   call events_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call events_test (u, results)
```

HEP Events

⟨*Main: use tests*⟩+≡
```
use hep_events_ut, only: hep_events_test
```

⟨*Main: test cases*⟩+≡
```
case ("hep_events")
   call hep_events_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call hep_events_test (u, results)
```

EIO Data

⟨*Main: use tests*⟩+≡
```
use eio_data_ut, only: eio_data_test
```

⟨*Main: test cases*⟩+≡
```
case ("eio_data")
   call eio_data_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call eio_data_test (u, results)
```

EIO Base

⟨*Main: use tests*⟩+≡
```
use eio_base_ut, only: eio_base_test
```

⟨*Main: test cases*⟩+≡
```
case ("eio_base")
   call eio_base_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call eio_base_test (u, results)
```

EIO Raw

⟨*Main: use tests*⟩+≡
```
use eio_raw_ut, only: eio_raw_test
```

⟨*Main: test cases*⟩+≡
```
case ("eio_raw")
   call eio_raw_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call eio_raw_test (u, results)
```

EIO Checkpoints

⟨*Main: use tests*⟩+≡
```
use eio_checkpoints_ut, only: eio_checkpoints_test
```

⟨*Main: test cases*⟩+≡
```
case ("eio_checkpoints")
   call eio_checkpoints_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call eio_checkpoints_test (u, results)
```

EIO LHEF

⟨*Main: use tests*⟩+≡
```
use eio_lhef_ut, only: eio_lhef_test
```

⟨*Main: test cases*⟩+≡
```
case ("eio_lhef")
   call eio_lhef_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call eio_lhef_test (u, results)
```

EIO HepMC

⟨*Main: use tests*⟩+≡
```
use eio_hepmc_ut, only: eio_hepmc_test
```

⟨*Main: test cases*⟩+≡
```
case ("eio_hepmc")
   call eio_hepmc_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call eio_hepmc_test (u, results)
```

EIO LCIO

⟨*Main: use tests*⟩+≡
```
use eio_lcio_ut, only: eio_lcio_test
```

⟨*Main: test cases*⟩+≡
```
case ("eio_lcio")
   call eio_lcio_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call eio_lcio_test (u, results)
```

EIO StdHEP

⟨*Main: use tests*⟩+≡
```
use eio_stdhep_ut, only: eio_stdhep_test
```

⟨*Main: test cases*⟩+≡
```
case ("eio_stdhep")
   call eio_stdhep_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call eio_stdhep_test (u, results)
```

EIO ASCII

⟨*Main: use tests*⟩+≡
```
use eio_ascii_ut, only: eio_ascii_test
```

⟨*Main: test cases*⟩+≡
```
case ("eio_ascii")
   call eio_ascii_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call eio_ascii_test (u, results)
```

EIO Weights

⟨*Main: use tests*⟩+≡
```
use eio_weights_ut, only: eio_weights_test
```

⟨*Main: test cases*⟩+≡
```
case ("eio_weights")
   call eio_weights_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call eio_weights_test (u, results)
```

EIO Dump

⟨*Main: use tests*⟩+≡
```
use eio_dump_ut, only: eio_dump_test
```

⟨*Main: test cases*⟩+≡
```
case ("eio_dump")
   call eio_dump_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call eio_dump_test (u, results)
```

Iterations

⟨*Main: use tests*⟩+≡
```
use iterations_ut, only: iterations_test
```

⟨*Main: test cases*⟩+≡
```
case ("iterations")
   call iterations_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call iterations_test (u, results)
```

Beam Structures

⟨*Main: use tests*⟩+≡
```
use beam_structures_ut, only: beam_structures_test
```

⟨*Main: test cases*⟩+≡
```
case ("beam_structures")
   call beam_structures_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call beam_structures_test (u, results)
```

RT Data

⟨*Main: use tests*⟩+≡
```
use rt_data_ut, only: rt_data_test
```

⟨*Main: test cases*⟩+≡
```
case ("rt_data")
   call rt_data_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call rt_data_test (u, results)
```

Dispatch

⟨*Main: use tests*⟩+≡
```
use dispatch_ut, only: dispatch_test
```

⟨*Main: test cases*⟩+≡
```
case ("dispatch")
   call dispatch_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call dispatch_test (u, results)
```

Dispatch RNG

⟨*Main: use tests*⟩+≡
```
use dispatch_rng_ut, only: dispatch_rng_test
```

⟨*Main: test cases*⟩+≡
```
case ("dispatch_rng")
   call dispatch_rng_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call dispatch_rng_test (u, results)
```

Dispatch MCI

⟨*Main: use tests*⟩+≡
```
use dispatch_mci_ut, only: dispatch_mci_test
```

⟨*Main: test cases*⟩+≡
```
case ("dispatch_mci")
   call dispatch_mci_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call dispatch_mci_test (u, results)
```

Dispatch PHS

⟨*Main: use tests*⟩+≡
```
use dispatch_phs_ut, only: dispatch_phs_test
```

⟨*Main: test cases*⟩+≡
```
case ("dispatch_phs")
   call dispatch_phs_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call dispatch_phs_test (u, results)
```

Dispatch transforms

⟨*Main: use tests*⟩+≡
```
use dispatch_transforms_ut, only: dispatch_transforms_test
```

⟨*Main: test cases*⟩+≡
```
case ("dispatch_transforms")
   call dispatch_transforms_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call dispatch_transforms_test (u, results)
```

Shower partons

⟨*Main: use tests*⟩+≡
```
use shower_base_ut, only: shower_base_test
```

⟨*Main: test cases*⟩+≡
```
case ("shower_base")
   call shower_base_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call shower_base_test (u, results)
```

Process Configurations

⟨*Main: use tests*⟩+≡
```
use process_configurations_ut, only: process_configurations_test
```

⟨*Main: test cases*⟩+≡
```
case ("process_configurations")
   call process_configurations_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call process_configurations_test (u, results)
```

Compilations

⟨*Main: use tests*⟩+≡
```
use compilations_ut, only: compilations_test
use compilations_ut, only: compilations_static_test
```

⟨*Main: test cases*⟩+≡
```
case ("compilations")
   call compilations_test (u, results)
case ("compilations_static")
   call compilations_static_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call compilations_test (u, results)
call compilations_static_test (u, results)
```

Integrations

⟨*Main: use tests*⟩+≡
```
use integrations_ut, only: integrations_test
use integrations_ut, only: integrations_history_test
```

⟨*Main: test cases*⟩+≡
```
case ("integrations")
   call integrations_test (u, results)
case ("integrations_history")
   call integrations_history_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call integrations_test (u, results)
call integrations_history_test (u, results)
```

Event Streams

⟨*Main: use tests*⟩+≡
```
use event_streams_ut, only: event_streams_test
```

⟨*Main: test cases*⟩+≡
```
case ("event_streams")
   call event_streams_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call event_streams_test (u, results)
```

Simulations

⟨*Main: use tests*⟩+≡
```
use simulations_ut, only: simulations_test
```

⟨*Main: test cases*⟩+≡
```
case ("simulations")
   call simulations_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call simulations_test (u, results)
```

Commands

⟨*Main: use tests*⟩+≡
```
use commands_ut, only: commands_test
```

⟨*Main: test cases*⟩+≡
```
case ("commands")
   call commands_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call commands_test (u, results)
```

ttV formfactors

⟨*Main: use tests*⟩+≡
```
use ttv_formfactors_ut, only: ttv_formfactors_test
```

⟨*Main: test cases*⟩+≡
```
case ("ttv_formfactors")
   call ttv_formfactors_test (u, results)
```

⟨*Main: all tests*⟩+≡
```
call ttv_formfactors_test (u, results)
```

35.7 Whizard-C-Interface

⟨whizard-c-interface.f90⟩≡
  ⟨*File header*⟩

  ⟨*Whizard-C-Interface: Internals*⟩
  ⟨*Whizard-C-Interface: Init and Finalize*⟩
  ⟨*Whizard-C-Interface: Interfaced Commads*⟩
  ⟨*Whizard-C-Interface: HepMC*⟩

⟨*Whizard-C-Interface: Internals*⟩≡

```
    subroutine c_whizard_convert_string (c_string, f_string)
      use, intrinsic :: iso_c_binding
      use iso_varying_string, string_t => varying_string !NODEP!

      implicit none

      character(kind=c_char), intent(in) :: c_string(*)
      type(string_t), intent(inout) :: f_string
      character(len=1) :: dummy_char
      integer :: dummy_i = 1

      f_string = ""
      do
         if (c_string(dummy_i) == c_null_char) then
            exit
         else if (c_string(dummy_i) == c_new_line) then
            dummy_char = CHAR(13)
            f_string = f_string // dummy_char
            dummy_char = CHAR(10)
         else
            dummy_char = c_string (dummy_i)
         end if
         f_string = f_string // dummy_char
         dummy_i = dummy_i + 1
      end do
      dummy_i = 1
    end subroutine c_whizard_convert_string

    subroutine c_whizard_commands (w_c_instance, cmds)
      use, intrinsic :: iso_c_binding
      use iso_varying_string, string_t => varying_string !NODEP!
      use commands
      use diagnostics
      use lexers
      use models
      use parser
      use whizard

      type(c_ptr), intent(inout) :: w_c_instance
      type(whizard_t), pointer :: whizard_instance
      type(string_t) :: cmds
      type(parse_tree_t) :: parse_tree
      type(parse_node_t), pointer :: pn_root
```

```
    type(stream_t), target :: stream
    type(lexer_t) :: lexer
    type(command_list_t), target :: cmd_list

    call c_f_pointer (w_c_instance, whizard_instance)
    call lexer_init_cmd_list (lexer)
    call syntax_cmd_list_init ()

    call stream_init (stream, cmds)
    call lexer_assign_stream (lexer, stream)
    call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
    pn_root => parse_tree%get_root_ptr ()

    if (associated (pn_root)) then
       call cmd_list%compile (pn_root, whizard_instance%global)
    end if
    call whizard_instance%global%activate ()
    call cmd_list%execute (whizard_instance%global)
    call cmd_list%final ()

    call parse_tree_final (parse_tree)
    call stream_final (stream)
    call lexer_final (lexer)
    call syntax_cmd_list_final ()
  end subroutine c_whizard_commands
```

⟨*Whizard-C-Interface: Init and Finalize*⟩≡

```
  subroutine c_whizard_init (w_c_instance) bind(C)
    use, intrinsic :: iso_c_binding
    use iso_varying_string, string_t => varying_string !NODEP!
    use system_dependencies
    use diagnostics
    use ifiles
    use os_interface
    use whizard

    implicit none
```

⟨*Main: cmdline arg len declaration*⟩

```
    type(c_ptr), intent(out) :: w_c_instance
    logical :: banner
    type(string_t) :: files, model, default_lib, library, libraries
!     type(string_t) :: check, checks
    type(string_t) :: logfile
    type(string_t) :: user_src, user_lib
    type(paths_t) :: paths
    logical :: rebuild_library, rebuild_user
    logical :: rebuild_phs, rebuild_grids, rebuild_events

    type(whizard_options_t), allocatable :: options
    type(whizard_t), pointer :: whizard_instance
```

```
    ! Initial values
    files = ""
    model = "SM"
    default_lib = "default_lib"
    library = ""
    libraries = ""
    banner = .true.
    logging = .true.
    logfile = "whizard.log"
!     check = ""
!     checks = ""
    user_src = ""
    user_lib = ""
    rebuild_library = .false.
    rebuild_user = .false.
    rebuild_phs = .false.
    rebuild_grids = .false.
    rebuild_events = .false.
    call paths_init (paths)

    ! Overall initialization
    if (logfile /= "")  call logfile_init (logfile)
    call mask_term_signals ()
    if (banner)  call msg_banner ()

    ! Set options and initialize the whizard object
    allocate (options)
    options%preload_model = model
    options%default_lib = default_lib
    options%preload_libraries = libraries
    options%rebuild_library = rebuild_library
    options%rebuild_user = rebuild_user
    options%rebuild_phs = rebuild_phs
    options%rebuild_grids = rebuild_grids
    options%rebuild_events = rebuild_events

    allocate (whizard_instance)
    call whizard_instance%init (options, paths)

!     if (checks /= "") then
!        checks = trim (adjustl (checks))
!        RUN_CHECKS: do while (checks /= "")
!           call split (checks, check, " ")
!           call whizard_check (check, test_results)
!        end do RUN_CHECKS
!        call test_results%wrapup (6, success)
!        if (.not. success)  quit_code = 7
!        quit = .true.
!     end if

    w_c_instance = c_loc (whizard_instance)

  end subroutine c_whizard_init
```

```
subroutine c_whizard_finalize (w_c_instance) bind(C)
  use, intrinsic :: iso_c_binding
  use system_dependencies
  use diagnostics
  use ifiles
  use os_interface
  use whizard

  type(c_ptr), intent(in) :: w_c_instance
  type(whizard_t), pointer :: whizard_instance
  integer :: quit_code = 0

  call c_f_pointer (w_c_instance, whizard_instance)
  call whizard_instance%final ()
  deallocate (whizard_instance)
  call terminate_now_if_signal ()
  call release_term_signals ()
  call msg_terminate (quit_code = quit_code)
end subroutine c_whizard_finalize

subroutine c_whizard_process_string (w_c_instance, c_cmds_in) bind(C)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  type(c_ptr), intent(inout) :: w_c_instance
  character(kind=c_char) :: c_cmds_in(*)
  type(string_t) :: f_cmds

  call c_whizard_convert_string (c_cmds_in, f_cmds)
  call c_whizard_commands (w_c_instance, f_cmds)
end subroutine c_whizard_process_string
```

⟨*Whizard-C-Interface: Interfaced Commads*⟩≡

```
subroutine c_whizard_model (w_c_instance, c_model) bind(C)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  type(c_ptr), intent(inout) :: w_c_instance
  character(kind=c_char) :: c_model(*)
  type(string_t) :: model, mdl_str

  call c_whizard_convert_string (c_model, model)
  mdl_str = "model = " // model
  call c_whizard_commands (w_c_instance, mdl_str)
end subroutine c_whizard_model

subroutine c_whizard_library (w_c_instance, c_library) bind(C)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!
```

```
  implicit none

  type(c_ptr), intent(inout) :: w_c_instance
  character(kind=c_char) :: c_library(*)
  type(string_t) :: library, lib_str

  call c_whizard_convert_string(c_library, library)
  lib_str = "library = " // library
  call c_whizard_commands (w_c_instance, lib_str)
end subroutine c_whizard_library

subroutine c_whizard_process (w_c_instance, c_id, c_in, c_out) bind(C)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  type(c_ptr), intent(inout) :: w_c_instance
  character(kind=c_char) :: c_id(*), c_in(*), c_out(*)
  type(string_t) :: proc_str, id, in, out

  call c_whizard_convert_string (c_id, id)
  call c_whizard_convert_string (c_in, in)
  call c_whizard_convert_string (c_out, out)
  proc_str = "process " // id // " = " // in // " => " // out
  call c_whizard_commands (w_c_instance, proc_str)
end subroutine c_whizard_process

subroutine c_whizard_compile (w_c_instance) bind(C)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  type(c_ptr), intent(inout) :: w_c_instance
  type(string_t) :: cmp_str
  cmp_str = "compile"
  call c_whizard_commands (w_c_instance, cmp_str)
end subroutine c_whizard_compile

subroutine c_whizard_beams (w_c_instance, c_specs) bind(C)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  type(c_ptr), intent(inout) :: w_c_instance
  character(kind=c_char) :: c_specs(*)
  type(string_t) :: specs, beam_str

  call c_whizard_convert_string (c_specs, specs)
  beam_str = "beams = " // specs
  call c_whizard_commands (w_c_instance, beam_str)
end subroutine c_whizard_beams

subroutine c_whizard_integrate (w_c_instance, c_process) bind(C)
```

```
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  type(c_ptr), intent(inout) :: w_c_instance
  character(kind=c_char) :: c_process(*)
  type(string_t) :: process, int_str

  call c_whizard_convert_string (c_process, process)
  int_str = "integrate (" // process //")"
  call c_whizard_commands (w_c_instance, int_str)
end subroutine c_whizard_integrate

subroutine c_whizard_matrix_element_test &
     (w_c_instance, c_process, n_calls) bind(C)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  type(c_ptr), intent(inout) :: w_c_instance
  integer(kind=c_int) :: n_calls
  character(kind=c_char) :: c_process(*)
  type(string_t) :: process, me_str
  character(len=8) :: buffer

  call c_whizard_convert_string (c_process, process)
  write (buffer, "(I0)")  n_calls
  me_str = "integrate (" // process // ") { ?phs_only = true" // &
       "  n_calls_test = " // trim (buffer)
  call c_whizard_commands (w_c_instance, me_str)
end subroutine c_whizard_matrix_element_test

subroutine c_whizard_simulate (w_c_instance, c_id) bind(C)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none

  type(c_ptr), intent(inout) :: w_c_instance
  character(kind=c_char) :: c_id(*)
  type(string_t) :: sim_str, id

  call c_whizard_convert_string(c_id, id)
  sim_str = "simulate (" // id // ")"
  call c_whizard_commands (w_c_instance, sim_str)
end subroutine c_whizard_simulate

subroutine c_whizard_sqrts (w_c_instance, c_value, c_unit) bind(C)
  use, intrinsic :: iso_c_binding
  use iso_varying_string, string_t => varying_string !NODEP!

  implicit none
```

```
    type(c_ptr), intent(inout) :: w_c_instance
    character(kind=c_char) :: c_unit(*)
    integer(kind=c_int) :: c_value
    integer :: f_value
    character(len=8) :: f_val
    type(string_t) :: val, unit, sqrts_str

    f_value = c_value
    write (f_val,'(i8)') f_value
    val = f_val
    call c_whizard_convert_string (c_unit, unit)
    sqrts_str = "sqrts =" // val // unit
    call c_whizard_commands (w_c_instance, sqrts_str)
  end subroutine c_whizard_sqrts
```

⟨*Whizard-C-Interface: HepMC*⟩≡

```
  type(c_ptr) function c_whizard_hepmc_test &
       (w_c_instance, c_id, c_proc_id, c_event_id) bind(C)
    use, intrinsic :: iso_c_binding
    use iso_varying_string, string_t => varying_string !NODEP!
    use commands
    use diagnostics
    use events
    use hepmc_interface
    use lexers
    use models
    use parser
    use instances
    use rt_data
    use simulations
    use whizard
    use os_interface

    implicit none

    type(c_ptr), intent(inout) :: w_c_instance
    type(string_t) :: sim_str
    type(parse_tree_t) :: parse_tree
    type(parse_node_t), pointer :: pn_root
    type(stream_t), target :: stream
    type(lexer_t) :: lexer
    type(command_list_t), pointer :: cmd_list
    type(whizard_t), pointer :: whizard_instance

    type(simulation_t), target :: sim

    character(kind=c_char), intent(in) :: c_id(*)
    type(string_t) :: id
    integer(kind=c_int), value :: c_proc_id, c_event_id
    integer :: proc_id

    type(hepmc_event_t), pointer :: hepmc_event
```

```
  call c_f_pointer (w_c_instance, whizard_instance)

  call c_whizard_convert_string (c_id, id)
  sim_str = "simulate (" // id // ")"

  proc_id = c_proc_id

  allocate (hepmc_event)
  call hepmc_event_init (hepmc_event, c_proc_id, c_event_id)

  call syntax_cmd_list_init ()
  call lexer_init_cmd_list (lexer)

  call stream_init (stream, sim_str)
  call lexer_assign_stream (lexer, stream)
  call parse_tree_init (parse_tree, syntax_cmd_list, lexer)
  pn_root => parse_tree%get_root_ptr ()

  allocate (cmd_list)
  if (associated (pn_root)) then
     call cmd_list%compile (pn_root, whizard_instance%global)
  end if

  call sim%init ([id], .true., .true., whizard_instance%global)

  !!! This should generate a HepMC event as hepmc_event_t type
  call msg_message ("Not enabled for the moment.")

  call sim%final ()

  call cmd_list%final ()

  call parse_tree_final (parse_tree)
  call stream_final (stream)
  call lexer_final (lexer)
  call syntax_cmd_list_final ()

  c_whizard_hepmc_test = c_loc(hepmc_event)
  return
end function c_whizard_hepmc_test
```

Chapter 36

Cross References

36.1 Identifiers

36.2 Chunks